Detection and Generation of Real-Word Errors in German Using Language Models

Bachelor Thesis
Faculty of Science, University of Bern

submitted by

Elias Kellenberger

from Bern, Switzerland

Supervision:

PD Dr. Kaspar Riesen
Corina Masanti
Institute of Computer Science (INF)
University of Bern, Switzerland

Abstract

This thesis addresses the detection of real-word errors in German, a challenging task in natural language processing that requires contextual understanding, as erroneous tokens remain valid words. A major limitation in this area is the scarcity of annotated data.

To overcome this difficulty, both data-centric and model-centric approaches are explored. A cleaned version of the Falko–MERLIN corpus is constructed, focusing on real-word errors. Additionally, two synthetic data generation strategies are explored: LLM-based sentence rewriting that preserves errors, and tuple-based corruption using extracted error–correction pairs.

A BERT-based token classification model serves as the baseline and is extended with custom architectures that combine representations from multiple transformer layers. The approaches are evaluated using precision, recall, and F1 score.

Results show that synthetic data significantly improves performance, particularly recall, with tuple-based augmentation providing the best balance. In contrast, architectural modifications yield only minor improvements. These findings highlight the importance of data quality and augmentation over increased model complexity for real-word error detection.

Acknowledgements

I would like to express my sincere gratitude to Corina Masanti for her continuous support throughout this thesis. Our numerous meetings and the many email exchanges provided invaluable guidance, clarity, and direction at every stage of the process. Her feedback and engagement significantly contributed to the development of this work.

I also wish to thank Prof. Dr. Riesen for his role and oversight in the course of this project.

Finally, I am very grateful to Manuel Flückiger for taking the time to proofread this thesis and for his helpful suggestions and attention to detail.

Contents

Chapter 1

Introduction

Artificial Intelligence (AI) has become a central field of research in modern computer science, enabling machines to perform tasks that traditionally require human intelligence. A major subfield of AI is Natural Language Processing (NLP), which focuses on enabling computers to understand, interpret, and generate human language.

Since early foundational work such as Turing's proposal of the Turing Test [1], NLP has evolved significantly and is now widely applied in real-world scenarios. These include applications such as machine translation [2], sentiment analysis [3], named entity recognition [4], and question answering [5]. Recent advances in model architectures, combined with improvements in hardware capabilities, have significantly accelerated progress in this field, particularly through the development of large-scale language models.

Error Detection in NLP Within NLP, one important class of problems is classification, where models assign labels to linguistic units such as tokens or sentences. A specific instance of this is Grammatical Error Detection (GED), which aims to identify incorrect or non-standard language usage in text.

Grammatical Error Detection is a foundational component in modern NLP, with wide-ranging practical impact. For example in language learning systems, it enables real-time feedback for learners by identifying mistakes in syntax and word usage. This supports a more personalized and scalable education compared to traditional methods.

Another application of GED can be found in automated proofreading systems that enhance writing quality by catching issues that go beyond simple spell-checking, such as subject–verb agreement, incorrect tense usage, or misplaced modifiers, thereby improving clarity and professionalism in both casual and formal texts.

Beyond end-user applications, GED is also an important preprocessing step in

many NLP pipelines. Tasks such as machine translation, sentiment analysis, and information extraction rely heavily on clean and well-structured input data. Detecting and potentially correcting grammatical errors before processing can reduce noise, leading to more accurate model predictions and better overall system performance.

Real-Word Errors This thesis focuses on a particularly challenging subset of grammatical errors known as *real-word errors*. Such errors occur when a word is valid in isolation but does not fit the semantic or grammatical context of the sentence. For example, using "their" instead of "there" results in a syntactically valid but contextually incorrect sentence.

Real-word errors are difficult to detect because they cannot be identified using simple dictionary-based approaches. Instead, their detection requires a deep understanding of context and semantics, making them a challenging problem for both rule-based and machine learning systems.

Motivation and Research Gap Despite advances in NLP, the detection of real-word errors remains a difficult task. A major challenge lies in the availability of high-quality annotated data. Creating such datasets is costly and time-consuming, particularly for languages other than English, such as German.

As a result, there is a growing interest in the use of synthetic data to augment existing datasets. However, the effectiveness of different synthetic data generation strategies, as well as their interaction with modern language models, is not yet fully understood. In particular, there is limited work on generating realistic real-word errors in German and evaluating their impact on model performance.

A further open question concerns the choice of model architecture for real-word error detection. While modern language models provide strong contextual representations, it remains unclear to what extent variations in the classification head influence overall performance. Systematically comparing different classifier head designs on top of a shared backbone may provide insights into whether architectural choices at this level yield measurable improvements.

Research Questions Based on the challenges and open issues outlined above, this thesis is guided by the following research questions:

- Does Synthetic Data Help?
- Which BERT Layers Are Most Informative?
- Does Layer Pooling Improve Performance?

Approach and Objectives To answer these questions, this thesis adopts both data-centric and model-centric perspectives on real-word error detection. Specifically, the work pursues the following objectives:

- Construct a comprehensive dataset for German real-word errors using rule-based extraction methods from public corpora.
- Fine-tune and evaluate language models for real-word error detection, using precision, recall, and F1 score as evaluation metrics.
- Generate synthetic data that mimics naturally occurring real-word errors using prompt-based large language models and pattern-based noise injection methods.
- Analyze the impact of different classifier head architectures on real-word error detection performance.

Scope This work focuses exclusively on the detection of real-word errors in German text. The task is formulated as a token-level classification problem. Error correction is not considered in this thesis.

Thesis Structure The remainder of this thesis is structured as follows. Chapter 2 provides the necessary background and reviews related work in GED, synthetic data generation, and transformer-based language models. Chapter 3 describes the datasets used and the methods for data preprocessing and augmentation. Chapter 4 introduces the model architectures evaluated in this work. Chapter 5 outlines the experimental setup and evaluation methodology. Chapter 6 presents and discusses the results. Finally, Chapter 7 concludes the thesis and suggests directions for future work.

Chapter 2

Background and Related Work

This chapter establishes the theoretical foundation and contextual framework for the research presented in this thesis. It begins by defining the task of GED and tracing its evolution from rule-based systems to modern data-driven approaches. Given the data-intensive nature of neural NLP, we subsequently explore the role of synthetic data generation and augmentation techniques in overcoming data scarcity.

The latter half of the chapter provides a technical deep dive into the mechanisms of representation learning and the transformer architecture, which serve as the backbone of our methodology. Finally, we discuss BERT, a specific implementation of the transformer encoder, detailing its architecture, training objectives, and its application to token-level classification tasks. Together, these sections provide the necessary technical background to understand the experimental setup and contributions described in the following chapters.

2.1 Grammatical Error Detection

Grammatical Error Detection is a subfield of NLP that focuses on identifying deviations from standard grammar usage in text. GED is typically formulated as a classification task at the token or sentence level, where the objective is to detect whether a token or span contains an error.

Early work in error detection and correction was strongly influenced by rule-based systems [6] and statistical language models [7]. With the introduction of large annotated learner corpora [8] and machine learning methods, the task has increasingly shifted toward data-driven approaches.

More recently, neural approaches based on recurrent and transformer architectures [2] have significantly improved performance. In particular, pretrained language models such as BERT [5] have become the dominant approach due to their ability to model contextual dependencies effectively.

An important distinction relevant to this thesis is the concept of *real-word errors*, which are errors where the incorrect token is still a valid word in the language (e.g., "form" instead of "from"). These errors are particularly difficult to detect because they cannot be identified using vocabulary-based heuristics alone.

2.2 Synthetic Data in NLP

High-quality annotated data is a major bottleneck in many NLP applications, especially in error detection and correction tasks. As a result, synthetic data generation has become a widely used technique to augment training datasets.

One of the earliest and most influential approaches is back-translation, introduced by Sennrich et al. [9], where monolingual target-language sentences are translated into another language and back again to generate paraphrases.

Simple augmentation techniques such as EDA (Easy Data Augmentation) [10] apply heuristic transformations like synonym replacement, random insertion, and deletion. While effective in low-resource settings, these methods often fail to preserve grammatical structure in a realistic manner.

More recently, neural and transformer-based approaches have been used for data augmentation. Unsupervised Data Augmentation (UDA) [11] leverages consistency training between original and augmented samples to improve generalization.

Large language models (LLMs) have significantly expanded the design space of synthetic data generation by enabling flexible transformations such as controlled rewriting, paraphrasing, and structured data augmentation pipelines. These capabilities allow practitioners to generate task-specific data at scale, which can help avoid data scarcity and support learning tasks [12].

Empirical evidence surveyed in prior work suggests that LLM-generated data can be effective across a variety of applications, although its utility depends critically on factors such as data quality and diversity [12].

In the context of GED, synthetic data is commonly generated through two strategies: (i) generating corrected versions of erroneous inputs, and (ii) injecting artificial errors into well-formed text. These approaches align with broader categories of LLM-driven data generation and augmentation pipelines identified in the literature [12].

2.3 Representation Learning

Representation learning is the process of training a machine learning model to automatically discover the underlying features or "representations" required for feature

detection or classification from raw data [13]. The primary goal is to transform input data into a format that preserves essential information while being computationally efficient for downstream tasks.

For example, in this work, sentences are transformed into multiple vectors containing the same information as the original sentence itself.

Historical Context Historically, feature engineering was a manual, labor-intensive process. In NLP, early methods relied on "one-hot encoding", where each word was represented as a sparse vector the size of the entire vocabulary. This approach failed to capture semantic relationships – for example, the vectors for "cat" and "dog" were as mathematically distant as "cat" and "refrigerator".

Subsequent methods like Word2Vec [14] and GloVe [15] introduced static dense embeddings, which mapped words to a vector space where proximity implied semantic similarity. However, these remained context-insensitive; the word "bank" would have the same representation whether referring to a river or a financial institution.

Current Approach In this work, representation learning is handled dynamically with an encoder through a pipeline that transforms discrete text into a continuous latent space. The process follows these stages:

1. **Tokenization:** Breaking raw text into sub-word units to handle out-of-vocabulary terms.

2. **Embedding Layer:** Mapping tokens to initial d-dimensional vectors through the use of a lookup table.

3. **Transformer Processing:** Applying self-attention to inject context into the embeddings. More in the next section [2.4].

Example Consider the sentence: *"The bank is open."* Initially, the embedding for [*bank*] is a generic vector. After passing through the transformer layers, the representation of [*bank*] is updated (contextualized) based on its relation to [*open*], changing the embedding vector toward concepts associated with "finance" rather than "river". Figure 2.1 illustrates this example.

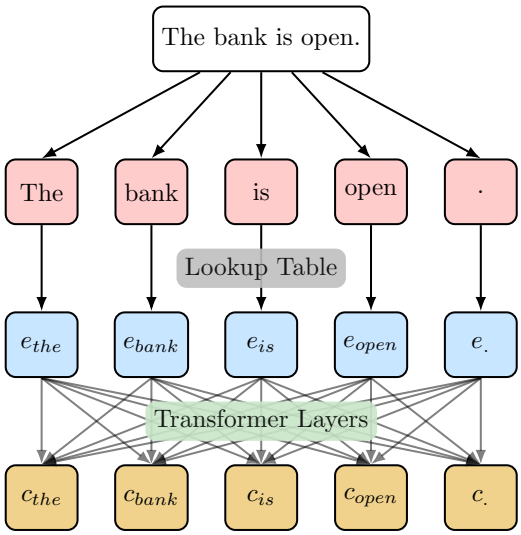

Figure 2.1: Example transformation of a sentence into contextualized embeddings, with e_i the embedding of token i and c_j the contextualized embedding of token j.

2.4 Transformer Architecture

The transformer architecture, introduced by Vaswani et al. [2], marked a fundamental shift in NLP by departing from the sequential nature of Recurrent Neural Networks (RNNs). Unlike its predecessors, the transformer relies entirely on attention mechanisms, allowing for significant parallelization during training and the ability to capture long-range dependencies regardless of their distance in the sequence.

The original transformer architecture contains an encoder (turning tokens into embeddings) and decoder (turning embeddings back into tokens). For this work, only the encoder was used, so only the encoder will be introduced here. The decoder differs only in a few places, but it would be too extensive to cover all of it.

Forward Pass Algorithm 1 shows the steps of the forward pass of the encoder. The inputs are the learned weight matrices W and the positionally encoded embeddings $X_{initial}$. The algorithm produces contextualized embeddings X as output. The individual components are described extensively below.

Algorithm 1 Transformer Forward Pass (Encoder Style).

Require: Weights W, Positional encoded embeddings X_{intial}

1: **for** each head $h \in 1, ..., H$ **do**
2: $\quad Q_h \leftarrow X \cdot W_h^Q$
3: $\quad K_h \leftarrow X \cdot W_h^K$
4: $\quad V_h \leftarrow X \cdot W_h^V$
5: $\quad A_h \leftarrow \text{softmax}\left(\frac{Q_h K_h^T}{\sqrt{d_k}}\right)$
6: $\quad \text{head}_h \leftarrow A_h \cdot V_h$
7: **end for**
8: $X_{attn} \leftarrow \text{Concatenate}(\text{head}_1, \ldots, \text{head}_H) \cdot W^O$
9: $X \leftarrow \text{LayerNorm}(X_{attn} + X_{initial})$
10: $X_{res} \leftarrow X$
11: $X_{\text{FFN}} \leftarrow \text{GELU}(X \cdot W_1^{FFN} + b_1) \cdot W_2^{FFN} + b_2$
12: $X \leftarrow \text{LayerNorm}(X_{FFN} + X_{res})$
13: **return** X

Positional Encodings (Algorithm 1, Requirements) Since transformers lack recurrence or convolution, they have no inherent sense of the order of tokens. Positional encodings are used to provide information about the relative or absolute position of tokens in a sequence. In the original transformer implementation, these positional encodings are calculated according to

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right), \qquad PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

and then added element wise to the embeddings $X_{initial} \leftarrow X + PE(X)$.

Attention Head (Algorithm 1, line 1) An individual "head" represents a single instance of the attention mechanism. By using multiple heads (Multi-Head Attention), the model can simultaneously attend to information from different representation subspaces at different positions.

For example, in a model with two heads, one head may focus on identifying whether a token belongs to a noun phrase, while another head may specialize in detecting domain-specific associations, such as whether a token is related to finance.

This allows the model to capture diverse linguistic patterns, such as syntactic dependencies (e.g., subject–verb agreement) as well as semantic relationships (e.g., topic or contextual relevance).

Importantly, these heads operate in parallel and learn different representations ("topics"), which are later combined to form a richer overall encoding.

Query, Key and Value (Algorithm 1, lines 2-4) For the attention in the next step, a transformer uses the three following linear projections of the input: Queries

(Q), keys (K), and values (V). Formally, given an input sequence X, these projections are computed as

$$Q = XW^Q, \quad K = XW^K, \quad V = XW^V$$

where W^Q, W^K, and W^V are learned parameter matrices.

Intuitively, the query represents the current token seeking relevant information, the keys represent how relevant each token in the sequence is with respect to that query, and the values contain the information that will be collected.

Continuing the previous example, consider a head that focuses on financial context. In this case, the query vector of a given token encodes the question: "which other tokens are relevant for understanding this financial concept?". The key vectors of all tokens encode how strongly they relate to financial signals (e.g., terms like "interest rate", "inflation", or "earnings" may receive higher relevance scores). The value vectors then contain the associated contextual information (such as relationships between economic indicators or company performance), which is aggregated according to these relevance scores.

Attention (Algorithm 1, lines 5-6) The attention mechanism calculates a compatibility score between a query and all keys and applies it to the values. Formally, it is computed as:

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

The scaling factor $1/\sqrt{d_k}$ prevents the dot products from growing too large in magnitude, which would push the softmax function into regions with extremely small gradients.

Continuing the financial head example, consider a token such as "inflation" in a sentence discussing economic conditions. The query vector from the inflation-embedding is used with all key vectors to calculate similarity scores. These similarity scores reflect their financial relevance. Tokens such as "interest rates", "central bank", or "consumer prices" are likely to get higher compatibility scores, as they are closely related in financial contexts.

With the softmax, these scores are transformed into a probability distribution that determines how much attention each token receives. For instance, tokens strongly associated with economics may receive higher weights, while less relevant tokens (e.g., "a", "an", "the", ...) receive lower weights.

Finally, these attention weights are used to compute a weighted sum of the value

vectors. The resulting representation emphasizes financially relevant information.

Multi-Head Attention (Algorithm 1, line 8) After all attention heads are calculated, they are concatenated and multiplied with a weight matrix W^O. This step combines the information captured by each head back into a single vector, of the same dimension as the embeddings.

Feed Forward Network (FFN) (Algorithm 1, line 11) Following the attention sub-layer, a feed-forward network is applied. It consists of two linear transformations with GELU (or sometimes ReLU) in between. This layer processes each token position independently and identically, providing the model with the capacity to transform the features extracted by the attention mechanism.

Summarizing The transformer gets positionally encoded embeddings, combines them with self attention (Algorithm 1, lines 2–8), applies some FFN with normalization steps (Algorithm 1, lines 8–12) and then returns the contextualized embeddings (Algorithm 1, lines 13). This is further displayed in Figure 2.2.

Here, self attention means that each token representation is updated by attending to all other tokens in the sequence, including itself. This mechanism allows the model to capture dependencies independent of their distance in the sequence, enabling both local and global context integration.

In the multi-head setting, several attention operations are performed in parallel, each focusing on different aspects of the input (e.g. finances and noun phrase).

Overall, this architecture enables the transformer to efficiently model complex relationships within sequences while maintaining parallelizability during training.

2.5 BERT

BERT (Bidirectional Encoder Representations from Transformers) is a pretrained language model introduced by Devlin et al. [5]. It is based on the transformer encoder architecture and is designed to learn deep bidirectional representations of text.

The original BERT was trained on the English language only, but nowadays there exist multiple BERT-models trained on different languages. In this work, only the *Bert-base-german-cased* [16] model was used.

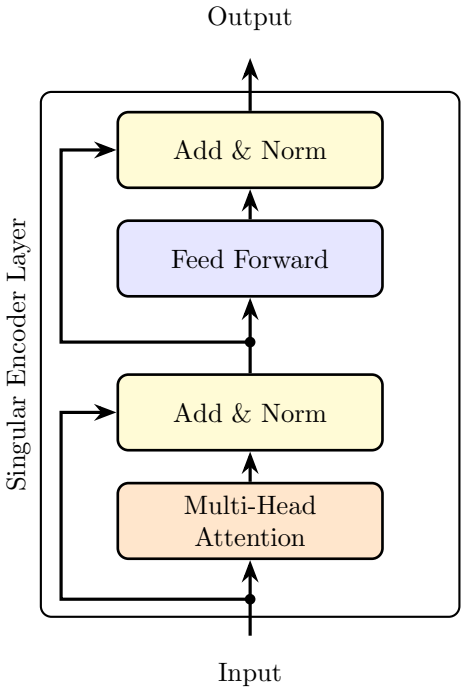

Figure 2.2: High level overview of the encoder forward pass in the original transformer architecture, adapted from Vaswani et al. [2]

2.5.1 Architecture

BERT consists of multiple stacked transformer encoder layers (12 layers for BERT-base, meaning 12 rounds of contextualization). Each layer produces contextualized token representations with an embedding-dimension of 768 and typically 12 self-attention heads.

That means, each head "works" with $768/12 = 64$ dimensional vectors (embeddings).
The model processes input text in the following steps:

1. Tokenization into subword units.
2. Mapping tokens to embeddings (token, positional embeddings, and a segment[1]).
3. Passing embeddings through multiple transformer encoder layers.
4. Producing contextualized token representations at each layer.

Unlike autoregressive models, BERT is bidirectional, meaning that each token can attend to both left and right context simultaneously.

[1] This is a dense vector representing separation of sentences.

2.5.2 Training Objectives

BERT is pretrained using two self-supervised objectives:

Masked Language Modeling (MLM): A random subset (15%) of tokens is selected, and each selected token is replaced using one of three strategies:

1. Replaced with a `[MASK]` token (80%)
2. Replaced with a random token (10%)
3. Left unchanged (10%)

Here, the model is trained to predict the original token.

Next Sentence Prediction (NSP): Here, the model is trained to predict whether two sentences appear consecutively in the original corpus.

While NSP was part of the original BERT training procedure, later work such as RoBERTa [17] demonstrated that removing NSP and increasing training data significantly improves performance.

2.5.3 Layer Representations

A key property of BERT is that different layers encode different linguistic information. Lower layers tend to capture syntactic patterns, while higher layers encode semantic and task-specific features [18]. This motivates the use of intermediate embeddings in downstream tasks.

In this thesis, embeddings that have been contextualized n times are referred to as *layer* n. For instance, referring to layers 0 and 12 indicates the same token represented twice: once without contextualization and once after 12 layers of contextualization. Without this clarification, the term "multiple embeddings" could be misinterpreted as referring to embeddings of different tokens rather than multiple representations of the same token across layers.

2.5.4 BERT for Token Classification

For sequence labeling tasks such as GED, BERT is commonly fine-tuned using a token classification head. Each token embedding is passed through a linear layer followed by a softmax function to produce class probabilities. This pipeline is further displayed in Figure 2.3.

This architecture forms the baseline model used in this thesis.

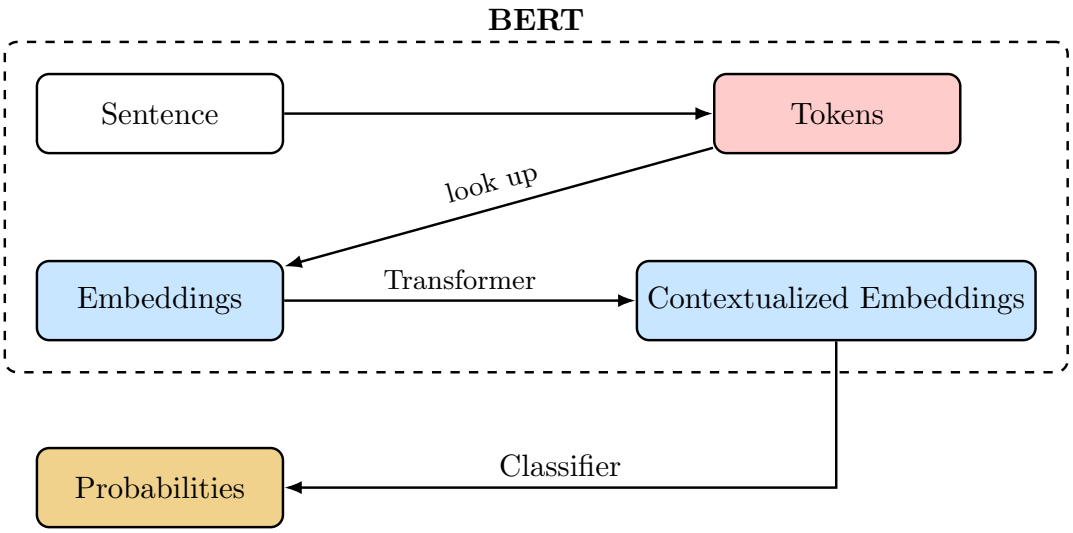

Figure 2.3: The `BertForTokenClassification` pipeline.

Chapter 3

Data

This chapter presents the data resources used in this work, along with the preprocessing and augmentation strategies applied to construct the final datasets. Given the central role of data quality and representativeness in learning-based approaches, particular attention is paid to the selection, cleaning, and expansion of available corpora.

The chapter begins with a description of the original learner corpus, Falko-MERLIN [19], including its structure, annotation scheme, and statistical properties. This is followed by a detailed account of the preprocessing steps undertaken to adapt the dataset to the specific task of real-word error detection.

Due to the limited size of manually annotated learner data, the chapter further introduces methods for synthetic data generation. These methods are designed to increase both the quantity and diversity of training examples while preserving realistic error patterns. Two complementary approaches are presented: LLM-based sentence rewriting with preserved errors, and tuple-based corruption derived from error–correction pairs.

Together, these steps result in a set of datasets that support robust training and evaluation of models for real-word error detection.

3.1 Original Data Set

In this thesis, the base dataset is the Falko–MERLIN dataset, used via the German subset of the MultigED 2023 shared task dataset [19], which aggregates learner corpora from both Falko [20] and MERLIN [21].

Falko [20]: The Falko corpus was developed at Humboldt University Berlin between 2004 and 2008 as part of research on German as a foreign language. It is a freely available, richly annotated learner corpus consisting primarily of essays and

summaries written by advanced learners under controlled conditions. The corpus was designed to support the systematic analysis of learner language, with a particular focus on error patterns and differences between learner and native speaker texts. To this end, Falko employs a multi-layer annotation scheme, including detailed error annotations and "target hypotheses," i.e., corrected versions of learner texts. The corpus comprises several subcorpora and a large number of texts, enabling both qualitative and quantitative studies in learner corpus linguistics.

MERLIN [21]: MERLIN is an error-annotated learner corpus of written texts in German, Italian, and Czech. The data originates from standardized language examinations and are aligned with the Common European Framework of Reference for Languages (CEFR). The platform provides access to the texts together with their proficiency levels, and includes documentation on corpus structure and annotation. MERLIN is intended for applications in both language teaching and research.

Falko-MERLIN [19]: Falko-MERLIN combines the Falko and MERLIN corpora into a unified resource for the study of German learner language. The integration aims to increase data size and diversity while improving comparability across datasets. As part of this process, the original annotation schemes—particularly the fine-grained annotations in Falko—were simplified to a binary token-level classification (correct vs. incorrect). This reduction supports consistent cross-corpus analysis and facilitates the application of computational methods.

In the following, an example is provided in which each token is labeled as correct (c) or incorrect (i):

Example:

Diese	c
drei	c
Leidenschaften	c
hat	i
die	c
Menschheit	c
zu	c
der	c
hohen	c
Zivilisation	c
gebracht	c
.	c

As a whole, Falko-MERLIN contains 24,079 sentences. This dataset is already split into train/validation/test splits where the test split is not annotated. Therefore, there are 21,742 usable annotated sentences. These sentences consist of 344,552 tokens, where 52,283 contain errors, giving an error rate of 0.152.

Since this thesis focuses exclusively on real-word error detection, a preprocessing step was required to clean the Falko-MERLIN dataset. Specifically, sentences containing non-word errors were removed.

To identify such errors, a case-sensitive token-level lookup was performed against a German word list [22]. Each token in the dataset was compared directly to entries in this list. This particular list was selected because, among the resources identified during a brief survey, it provided the most extensive coverage of German vocabulary, thereby increasing the likelihood of matching valid tokens.

Formally, an error was classified as a *real-word error* if the erroneous token exists in the word list. Conversely, tokens not present in the list were treated as non-real-word errors. For simplicity, all sentences containing at least one non-word error were removed from the dataset.

This cleaning process resulted in a substantial reduction in dataset size, as shown in Table 3.1.

Dataset Version	**Sentences**	**Tokens**	**Errors**	**Error Rate**
Original Falko-MERLIN	24,079	381,134	57,897	0.152
Cleaned Dataset	12,476	173,131	16,304	0.094

Table 3.1: Dataset statistics before and after cleaning.

The cleaned dataset was randomly split into training, test and validation datasets with an 80/10/10 split respectively.

The test set contains 1,248 sentences, comprising 17,417 tokens, of which 1,660 are labeled as erroneous. This corresponds to an error rate of 0.0953.

3.2 Synthetic Data Generation

High-quality annotated data is essential for training and evaluating models in error detection and correction tasks. However, manually curated datasets are often limited in size and costly to produce. To address these challenges, this work employs synthetic data generation techniques to create large-scale datasets with controlled error patterns.

The approach leverages a LLM API in two complementary ways. First, it is used to generate paraphrased variants of erroneous sentences while preserving the under-

lying errors. Second, it is applied to produce corrected versions of these sentences, enabling the extraction of error–correction pairs. These pairs are subsequently used to inject realistic errors into clean text data, creating a further dataset.

The synthetic data generation process leverages a LLM accessed via the OpenAI API [23]. Specifically, the model `gpt-5.1` [24] was used with a temperature setting of 0.7 to balance diversity and consistency in the generated outputs.

The API was invoked using a standard chat completion interface, where prompts were provided as user messages. Additionally, the `service_tier` parameter was set to `flex`. The exact prompts used for generation are provided in Appendix A.

To ensure the usefulness of the generated data, the error distribution of the synthetic dataset was carefully controlled to mirror the distribution observed in real-world data. This was achieved by calibrating the frequency of injected errors based on statistics derived from the original annotated corpus. By aligning the proportion of erroneous tokens and sentences with those found in authentic data, the synthetic dataset maintains realistic noise characteristics, thereby improving its suitability for both training and evaluation purposes.

3.2.1 LLM-Based Rewriting with Preserved Errors

The objective of this step was to generate synthetic variations of sentences while preserving the original errors. To achieve this, an LLM API was used to rewrite sentences containing grammatical or lexical mistakes. The key requirement was that the rewritten sentences should maintain the original errors, while altering the surrounding context and phrasing.

An example thereof is shown on Table 3.2.

Example of LLM-based rewriting with preserved errors
Original sentence: Feminismus ist einen stärken Einfluss heute in Kultur.
Rewritten sentence: Feminismus ist keinen stärken Einfluss mehr in Kultur.
Labels: (c, i, c, i, c, c, c, i, c) → unchanged

Table 3.2: Example of sentence rewriting while preserving error annotations. Modified tokens are highlighted.

Each original sentence was rewritten three times using the API, resulting in multiple diverse variants per input. Following a post-processing and cleaning stage to remove unsuitable outputs, the resulting synthetic dataset comprised approximately 20,000 sentences.

3.2.2 Tuple-Based Corruption

The goal of this step was to systematically extract error–correction pairs and use them to generate additional corrupted text. First, the LLM API was used to produce corrected versions of the original erroneous sentences. By comparing each corrected sentence with its corresponding faulty input, pairs of the form (incorrect word, corrected word) were identified.

An example thereof is shown on Table 3.3.

Example of tuple-based corruption
Original erroneous sentence: Für diesen Frauen, Feminismus existiert nicht.
Extracted correction tuple: (diesen → diese)
Clean Wikipedia sentence: Selbst diese Filme waren zumeist nur wenig verbreitet und liefen fast ausschließlich auf so genannten Filmkunstbühnen.
Corrupted sentence: Selbst diesen Filme waren zumeist nur wenig verbreitet und liefen fast ausschließlich auf so genannten Filmkunstbühnen.

Table 3.3: Example of tuple-based error injection. A correction pair extracted from learner data is applied to a clean sentence to introduce a realistic error.

Each original faulty sentence was corrected exactly once. After applying filtering and cleanup procedures, this process yielded a set of error–correction tuples. These tuples were subsequently used to inject realistic errors into a Wikipedia text corpus, enabling the creation of a larger synthetic dataset. In total, this approach resulted in 50,000 artificially corrupted sentences.

Relation to Prior Work The use of error–correction pairs for synthetic data generation is consistent with established approaches in NLP, particularly in the

context of grammatical error correction and data augmentation. Prior work has shown that leveraging such pairs to inject realistic noise into clean text can improve the robustness and scalability of training datasets [12]. Our method follows this general paradigm by extracting (incorrect, corrected) tuples from LLM-generated corrections and reusing them to construct a larger corpus of artificially corrupted sentences.

Chapter 4

Model Architecture

This chapter introduces the model architectures investigated in this thesis. We begin with a baseline model based on a standard token classification architecture using BERT [5], which serves as a reference point for all subsequent experiments. Building on this foundation, we then propose and analyze custom classification head architectures that extend the baseline by incorporating representations from multiple intermediate layers of the transformer.

The goal of these modifications is to explore whether combining information from different levels of abstraction within BERT can improve token-level prediction performance. In particular, we investigate two main strategies for combining layer-wise representations: concatenation and pooling-based methods. Both approaches are designed to retain complementary linguistic information captured at different depths of the model.

For clarity, we focus here on the architectural design and intuition behind these approaches. A brief overview of the underlying machine learning concepts (e.g., linear layers, dropout, activation functions, softmax, Feed Forward Networks (FFN), LayerNorm, and binary cross entropy loss) is provided in Appendix B.

4.1 Baseline Model

The baseline model used in this thesis is `BertForTokenClassification` from the Hugging Face transformers library [25].

The model uses the final contextualized token embeddings produced by BERT. These embeddings have dimensionality 768 and are passed through a linear classification layer that maps them to a 2-dimensional logit vector. Applying the softmax function yields class probabilities for each token. This architecture is shown on a high abstraction level in Figure 4.1.

For training, there is also a dropout layer before the linear layer (which does

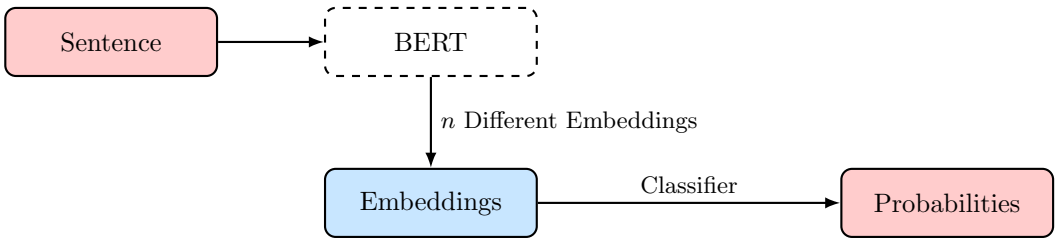

Figure 4.1: High Level Custom Head Architecture at Inference.

not have an impact at inference). The concrete implementation of the model is as follows:

```
self.bert = BertModel(config, add_pooling_layer=False)
self.dropout = nn.Dropout(classifier_dropout)
self.classifier = nn.Linear(config.hidden_size, config.num_labels)
```

4.2 Custom Classification Head Architecture

The baseline model relies solely on the final layer embeddings. However, intermediate layers in BERT capture different levels of linguistic information, with lower layers encoding syntactic features and higher layers more semantic information [26]. The custom head architecture leverages this by incorporating embeddings from multiple layers.

On a higher level, this means, the model takes multiple layer deep embeddings from BERT and then uses a classifier on those. However, how to combine those embeddings depends on the head.

Concatenation: One choice for the head is to combine the different embeddings through concatenating them into a single vector, which is then used with linear-layers and ReLUs to get the logits.

For example, with embeddings 0 and 12 the pipeline looks as follows:

$$\underbrace{X_0 = \begin{pmatrix} x_0 \\ \vdots \\ x_{767} \end{pmatrix}, X_{12} = \begin{pmatrix} y_0 \\ \vdots \\ y_{767} \end{pmatrix}}_{\text{Embeddings}} \xrightarrow{\text{concatenate}} [X_0|X_{12}] = \begin{pmatrix} x_0 \\ \vdots \\ x_{767} \\ y_0 \\ \vdots \\ y_{767} \end{pmatrix} \xrightarrow{\text{classifier}} \text{Logits}$$

However, the amount of embeddings as well as which ones is variable and can be decided upon initializing a model.

Here is the final implementation of the classifier where `n` is the amount of embeddings used and `d` the dimension of an embedding (in this case 768). The classifier first reduces the concatenated representation back to the original embedding dimension, applies a non-linear transformation, and finally maps to the output space.

```
self.classifier = nn.Sequential(
    nn.Linear(d*n, d),
    nn.ReLU(),
    nn.Dropout(dropout),
    nn.Linear(d, num_labels)
)
```

Summarizing, the concatenation head gets multiple embeddings, concatenates those and applies the classifier for the logits. Through the softmax function these logits are then transformed into the prediction probabilities.

Pooling: Another way to combine different embeddings is through pooling. In general, pooling refers to a merging process of multiple data points. In this case, it is concerned with combining n embeddings (vectors of dimension 768) to a single vector of dimension 768, with the goal of preserving the most extreme features.

There are many different kinds of pooling. The following is a list of the pooling methods used in this work.

- **Mean pooling** sums the vectors up and then divides by the number of vectors.
$$v_1, \dots, v_n \mapsto v = \frac{1}{n} \sum_{i=1}^{n} v_i$$

- **Max pooling** takes the entry wise max over all n vectors.
$$v_1 = \begin{pmatrix} v_{1,1} \\ \vdots \\ x_{1,768} \end{pmatrix}, \dots, v_n = \begin{pmatrix} v_{n,1} \\ \vdots \\ x_{n,768} \end{pmatrix} \mapsto v = \begin{pmatrix} \max\limits_{i \in \{0,\dots,n\}} v_{i,1} \\ \vdots \\ \max\limits_{i \in \{0,\dots,n\}} v_{i,768} \end{pmatrix}$$

- **Attention Pooling** uses a learned activation function ($score : \mathbb{R}^{768} \to \mathbb{R}^{+}$) to calculate how important a vector is and then calculates a weighted sum.
$$v_1, ..., v_n \mapsto v = \sum_{i=1}^{n} score(v_i) \cdot v_i$$

For all pooling methods it is also possible (but not necessary) to normalize the vectors before pooling them.

This gives the following pipeline for the custom classification head using pooling:

$$\text{BERT} \rightarrow n \text{ Embeddings} \xrightarrow{\text{Pooling}} \text{Combined Embeddings} \xrightarrow{\text{classifier}} \text{logits}$$

The following code snippet shows the design of the classifier for the pooling classification heads. The classifier applies a linear layer to the pooled vector, which then yields the logits. This is the same design as the base-model, except as an input it gets the pooled vector.

```
self.classifier = nn.Sequential(
    nn.Dropout(dropout),
    nn.Linear(d, num_labels)
)
```

Summarizing, the pooling head gets multiple embeddings, which are then combined using one of the pooling-functions[1]. The classifier is then applied to this new vector, which yields the logits. Through softmax, those are transformed into probabilities.

[1] This pooling function is fixed for any given model.

Chapter 5

Experimental Evaluation

This chapter describes the experimental framework used to evaluate the proposed models for GED. It outlines the dataset preparation and augmentation strategies, the evaluation metrics used for performance comparison, the model variants under investigation, and the training configuration.

The goal of this setup is to ensure a fair and systematic comparison between different architectural choices and data augmentation techniques, while isolating their individual contributions to overall model performance.

5.1 Data Splits

The original dataset was cleaned up yielding a clean version of 12,476 sentences. This clean dataset was split into training, test and validation sets, with an 80/10/10 split respectively.

With sentence rewriting, an additional 20,000 sentences were created. These were added in batches of size 5,000.

Through the the tuple based corruption of the Wikipedia corpus, an additional 50,000 sentences were generated. These were also added in batches of size 5,000.

Due to computational and resource constraints, the two augmentation strategies were not combined. Instead, they were evaluated separately to isolate their individual impact on model performance.

All data splits are displayed in Table 5.1.

5.2 Evaluation Metrics

To enable a systematic comparison between models, appropriate evaluation metrics for binary classification are required. In this work, three standard metrics are

Table 5.1: Dataset composition and augmentation strategy.

Dataset	**Size**	**Description**
Cleaned Dataset	12,476	Dataset after preprocessing
Training Set	9,981	80% of cleaned dataset
Validation Set	1,248	10% of cleaned dataset
Test Set	1,247	10% of cleaned dataset
Synthetic Data		
Sentence Rewriting	5k – 20k	LLM-based rewriting while preserving errors
Error Pairs	5k – 50k	Tuple-based corruption from Wikipedia corpus

employed: precision, recall, and the F_1 score.

Precision measures the proportion of correctly predicted positive instances among all predicted positives:

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}.$$

This metric is particularly important in scenarios where false positives are costly.

Recall measures the proportion of correctly identified positive instances among all actual positives:

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}.$$

Recall is especially relevant when failing to detect positive instances is undesirable.

F_1 Score is the harmonic mean of precision and recall:

$$F_1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}.$$

It provides a balanced measure when dealing with class imbalance.

In general, precision and recall exhibit a trade-off. A model that predicts all instances as positive achieves maximal recall but low precision. Therefore, a primary evaluation focus must be selected.

In the context of GED, recall is considered more important. Missing an actual error (false negative) is more detrimental than incorrectly flagging a correct sentence (false positive), as the latter can be more easily verified and dismissed by a user. Consequently, recall is treated as the primary evaluation metric in this work.

5.3 Model Variants

As introduced in Section 4, multiple model architectures were evaluated. The selected variants differ primarily in how they aggregate information from transformer layers.

Baseline Model: This model consists of a pre-trained *BERT-base-german-cased* encoder with a linear classification head. It serves as the primary benchmark for assessing the impact of architectural modifications and data augmentation.

Concatenation Head: This variant concatenates embeddings from multiple layers, preserving layer-specific information. It is primarily used to analyze which layers contribute most to error detection.

Mean Pooling: This variant calculates the element-wise average across the selected embeddings. This approach aims to provide a balanced representation of the token's features across different depths of the network.

Max Pooling: This strategy selects the maximum value across layers for each dimension, capturing the most prominent features. It is often effective in identifying specific local anomalies like grammatical errors [27].

Attention Pooling: This variant employs a learnable attention mechanism to weigh layer contributions dynamically, allowing the model to adaptively focus on the most informative representations.

5.4 Training Configuration

The training process was structured into two main phases. The first phase focused on establishing a performance floor using the baseline model across various data augmentations, while the second phase optimized the custom head architectures using the most promising data configuration.

Baseline Training The baseline model was trained in all data splits to evaluate the effectiveness of synthetic data generation strategies. This included the initial cleaned dataset and subsequent versions added in increments of 5,000 sentences for both the LLM-based rewriting and the tuple-based corruption methods.

Custom Model Optimization For the custom head variants (Concatenation, Mean, Max, and Attention pooling), the training set was fixed to the original data combined with 15,000 rewritten sentences. This decision was based on prior experiments, which suggest that this is the optimal amount of additional synthetic data.

The configuration of these models followed a specific experimental flow:

Layer Selection: Using the Concatenation head, the model always included the final embedding paired with exactly one other embedding from the transformer. This "one-to-one" probing strategy enables the identification of layers that contribute most significantly to recall performance.

Iterative Pooling: Based on the ranking from the probing phase, the pooling heads were then trained using an increasing number of layers – starting from the most informative layer and adding more embeddings, in order of their proven utility. This was done up to five layers.

Hyperparameter Tuning: To ensure that each architecture reaches its full potential, a hyperparameter search was conducted over the following space:

- Learning Rate: 10^{-5} to 5×10^{-4} (logarithmic scale)
- Batch Size: {8, 16, 32}
- Epochs: {2,3,4,5}
- Weight Decay: 0.0 to 0.3

As displayed in Table 5.2, for the *baseline model*, the hyperparameter search returned a learning rate of 10^{-5}, a training batch size of 16, 5 epochs, and a weight decay of 0.01. Following the search, the optimal parameters for the *custom head* models were identified as a learning rate of 2×10^{-5}, a batch size of 8, 5 epochs, and a weight decay of 0.01.

Hyperparameter	**Baseline Model**	**Custom Head Model**
Learning Rate	10^{-5}	2×10^{-5}
Batch Size	16	8
Epochs	5	5
Weight Decay	0.01	0.01

Table 5.2: Optimal hyperparameters for baseline and custom head models.

In both the baseline and custom model phases, the model checkpoint achieving the highest performance on the validation set was selected for final evaluation on the held-out test set.

Chapter 6

Results

This chapter presents the empirical results of the experiments conducted to evaluate the effectiveness of the proposed approaches. The analysis focuses on three main aspects: (i) The impact of synthetic data augmentation, (ii) the contribution of different BERT layers to model performance, and (iii) the effectiveness of various layer pooling strategies.

Performance is assessed using precision, recall, and F1 score, allowing for a comprehensive evaluation of both the model's accuracy and its ability to retrieve relevant instances. All results are reported on the same evaluation setup to ensure comparability across experiments.

The goal of this chapter is not only to report performance differences, but also to identify trends and trade-offs that provide insights into how different modeling choices affect the behavior of the system.

All results are provided in Appendix C. This chapter presents only the most relevant entries for clarity.

6.1 Baseline Performance

To establish a point of comparison for the proposed methods, a baseline model was evaluated using only the original training dataset without any synthetic data augmentation or complex layer pooling. This baseline configuration only utilizes the final embedding of BERT for the error detection.

The baseline results serve as the benchmark for all subsequent experiments. As shown in Table 6.1, the model achieves a precision of 0.7811, indicating a relatively high degree of exactness in its predictions. However, the recall is significantly lower at 0.4729, suggesting that the model, when trained on the limited original dataset, struggles to identify nearly half of the relevant instances. This results in an initial F1 score of 0.5891.

Table 6.1: Results of the Baseline Model on the Original Dataset.

Configuration	Precision	Recall	F1 Score
Baseline	0.7811	0.4729	0.5891

6.2 Does Synthetic Data Help?

This section evaluates the impact of augmenting the training data with synthetic samples. Two augmentation strategies are considered: pair-based synthetic data and rewritten sentence data. Figures 6.1 and 6.2 illustrate the results of augmenting the training set using the two strategies with varying amounts of additional sentences.

Overall, the results show that synthetic data can improve model performance; however, the extent of this improvement depends strongly on the generation strategy.

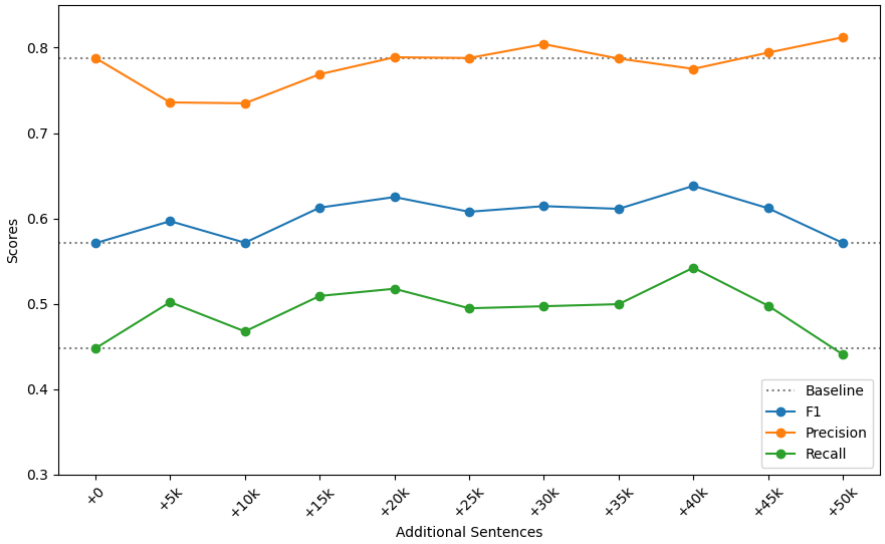

Figure 6.1: Metrics on n additional sentences, generated with pair generation.

Pair-based synthetic data The results indicate that pair-based synthetic data consistently improves recall and overall model balance, while only marginally affecting precision.

As shown in Table 6.2, precision increases by up to 3.98% with 50,000 additional samples, whereas recall improves by up to 14.66% with 40,000 additional samples. The F1 score reaches its maximum improvement of 8.32% at the same point.

Interestingly, the configuration with 40,000 additional samples yields the highest recall and F1 score, despite a slight decrease in precision of 0.75% compared to the baseline. This suggests that the additional data primarily helps the model identify

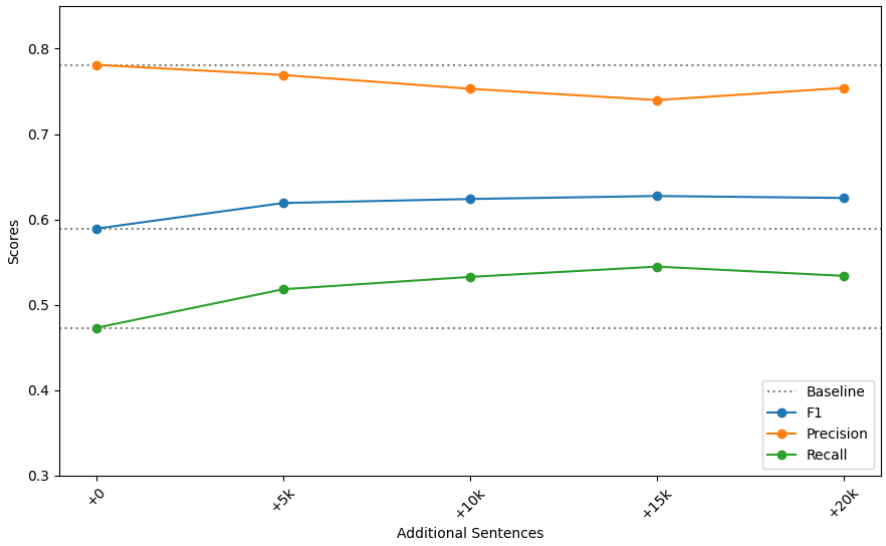

Figure 6.2: Metrics on n additional sentences, generated through rewriting erroneous sentences while preserving the error.

more relevant instances, even at the cost of a small increase in false positives.

The complete results are displayed in Figure 6.1 or in the appendix on Table C.1.

Table 6.2: Partial metrics with Additional Pair Based Generated Training Data.

Additional Data	Precision	Recall	F1 Score
+0	0.7811	0.4729	0.5891
+40,000	0.7752	**0.5422**	**0.6381**
+50,000	**0.8122**	0.4404	0.5711

Rewritten sentence synthetic data In contrast, augmenting the dataset with rewritten sentence synthetic data yields more mixed results.

As shown in Table 6.3, precision decreases when synthetic data is added, while recall increases by up to 15.16% with 15,000 additional samples. The F1 score improves by 6.48% at this point.

However, this improvement comes at a notable cost: precision decreases by 5.29% compared to the baseline. This indicates a clear trade-off between recall and precision. While the model becomes more sensitive to relevant instances, it also produces more false positives.

The complete results are shown in the appendix on Table C.2 or Figure 6.2.

Comparison of augmentation strategies When comparing the two augmentation strategies, pair-based augmentation appears to be more effective overall. It provides a better balance between precision and recall, which translates into

Table 6.3: Partial metrics with Additional Rewritten Sentences Training Data.

Additional Data	Precision	Recall	F1-Score
+0	**0.7811**	0.4729	0.5891
+15,000	0.7398	**0.5446**	**0.6273**

higher gains in F1 score. In contrast, rewritten sentence augmentation introduces a stronger precision–recall trade-off, limiting its overall effectiveness.

However, when focusing specifically on recall, the rewritten sentences achieve the best performance, albeit only marginally. A possible explanation for this behavior is that the rewritten sentences remain more similar to the original data distribution and, consequently, to the training data. This similarity may lead to improved performance on the test set, while not necessarily translating to better generalization on real-world data.

Discussion Several limitations should be noted. First, the observed differences in performance are relatively small in some cases, and no statistical significance testing was conducted. Second, the results are specific to the dataset and generation methods used, which may limit their generalizability to other tasks or domains.

Despite these limitations, the consistency of recall improvements across both strategies suggests a certain degree of reliability.

Summary In summary, synthetic data augmentation can improve model performance, but its effectiveness depends strongly on the chosen strategy. Pair-based augmentation provides the most balanced improvements, particularly by increasing recall without substantially degrading precision. Rewritten sentence augmentation, on the other hand, introduces a stronger trade-off between recall and precision.

These findings highlight the importance of designing and using augmentation methods, as there seem to be differences in data (even when generated through similar means – i.e. same LLM) as well as a "sweet spot" for the right amount of synthetic data used.

6.3 Which BERT Layers Are Most Informative?

To analyze which layers of BERT provide the most informative representations, experiments were conducted using the *custom head, concatenation* model. The training data consisted of original sentences augmented with 15,000 rewritten sentences. In all configurations, exactly two layers were concatenated: the final layer (layer 12) and one additional layer.

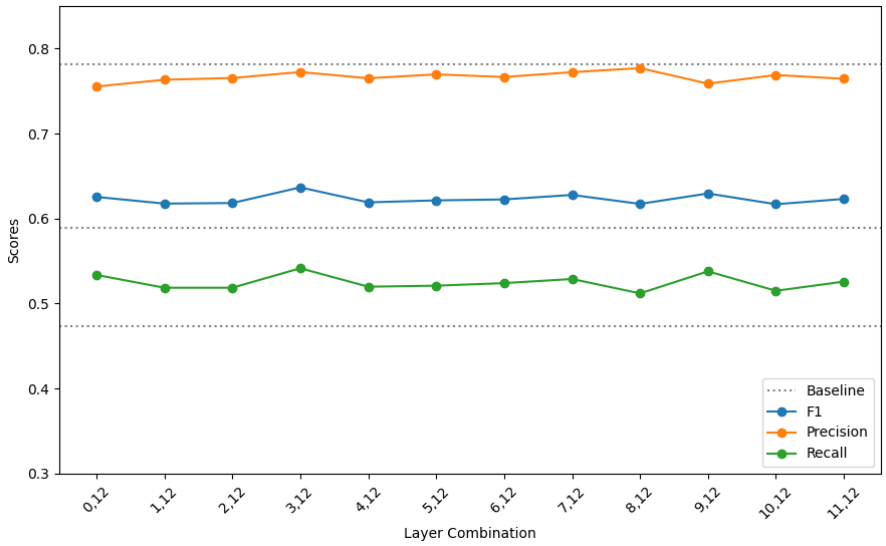

Figure 6.3: Scores Given the Different Embedding-Combinations.

Overall performance trends As shown in Table 6.3, combining the final layer with selected intermediate layers improves performance, particularly in terms of recall and F1 score, while precision remains relatively stable across configurations.

The best-performing configuration is obtained by combining layers 3 and 12, achieving a precision of 0.7723, a recall of 0.5416, and an F1 score of 0.6367. Compared to the baseline model, this corresponds to an improvement in recall and F1 score, while maintaining nearly the same level of precision.

Other layer combinations yield slightly lower but comparable results. For instance, combining layers 9 and 12 achieves a recall of 0.5380 and an F1 score of 0.6295, while layers 7 and 12 result in a recall of 0.5289 and an F1 score of 0.6278.

Overall the scores hardly vary over all layer combinations, indicating, that the impact of the additional layer was marginal.

Layers	Precision	Recall	F1-Score
0,12	0.7553	0.5337	0.6255
3,12	0.7723	**0.5416**	**0.6367**
7,12	0.7722	0.5289	0.6278
9,12	0.7587	0.5380	0.6295

Table 6.4: Partial Metrics of 2 Concatenated Layers.

Interpretation and relation to prior work The observed pattern does not align with prior findings by Tenney et al. [18], who reported that middle and higher layers (5–12) encode more semantic information than lower layers (0–4).

However, this discrepancy may be due to limitations in the measurement setup

rather than a fundamental difference in behavior. In particular, layer 12 was consistently included in the combinations, which may have caused it to dominate the results and overshadow the contribution of other layers. As a result, the apparent impact of the remaining layers may be underestimated, making their individual effects appear negligible.

Most impactful layers with respect to recall Focusing specifically on recall as the primary metric, the most impactful layers when combined with the final layer, are:

- Layer 3 (+14.53% relative recall improvement)
- Layer 9 (+13.77% relative recall improvement)
- Layer 0 (+12.85% relative recall improvement)
- Layer 7 (+11.84% relative recall improvement)

Strengths and limitations of the approach A key strength of this experimental setup is its controlled design, where only the choice of layers is varied while keeping the model architecture and training data constant. This allows for a clear attribution of performance differences to the representations extracted from different layers.

However, the experiment also has several limitations. First, only pairwise combinations of layers were considered, which may overlook more complex interactions between multiple layers. Second, the experiments were conducted on a single dataset with a specific augmentation strategy, limiting the generalizability of the findings. Finally, no statistical significance tests were performed, so small differences between configurations should be interpreted with caution.

Summarizing Overall, the results demonstrate that combining the final BERT layer with carefully selected intermediate layers leads to more informative representations.

The full set of results are shown in Table C.3 or Figure 6.3.

6.4 Does Layer Pooling Improve Performance?

To evaluate whether layer pooling improves performance, several pooling strategies (mean, max, and attention-based pooling) were applied over different layer combinations. The experiments were conducted using the same training setup as in the previous section, with original sentences and 15,000 synthetic rewritten sentences.

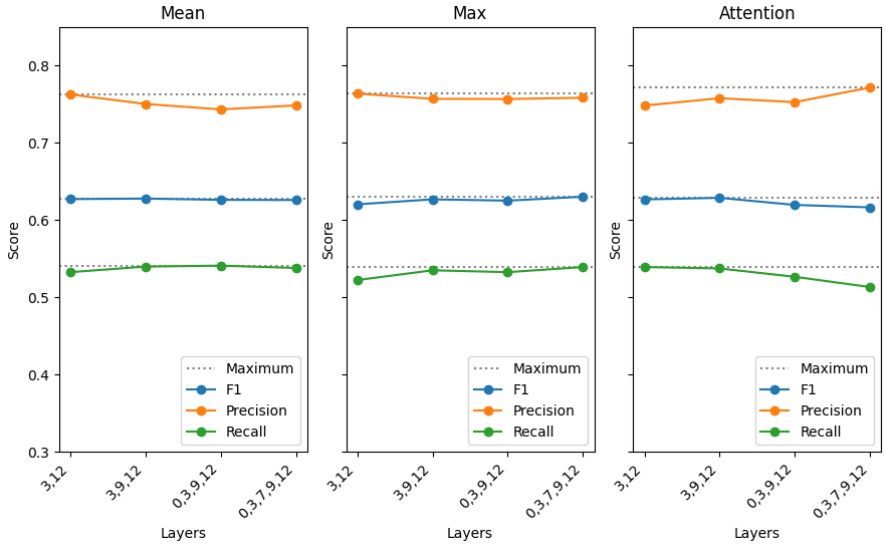

Figure 6.4: Pooling Scores.

The layer combinations were selected based on the results of the prior experiment, ranked according to recall, and ordered from highest to lowest performance, considering up to five layers.

Empirical findings Overall, the results indicate that layer pooling provides only marginal improvements over simpler two-layer concatenation. The best-performing configuration (mean pooling over layers 0, 3, 9, 12) achieves a recall score of 0.5410, which represents only a very small gain compared to other configurations. Across all experiments, differences in recall scores remain below 0.03, suggesting that the impact of pooling strategy is limited.

Mean pooling yields the most stable and consistent results across configurations, while attention-based pooling exhibits higher variability, particularly in recall. These findings suggest that performance is influenced more strongly by the choice of layers than by the pooling mechanism itself.

The full set of results are shown in Figure 6.4 or Table 6.5.

Theoretical interpretation From a theoretical perspective, transformer layers are known to capture different types of linguistic information, with lower layers encoding more syntactic features and higher layers capturing more semantic and task-specific representations [18]. Pooling across multiple layers may therefore dilute these specialized representations rather than effectively combining them.

The stability of mean pooling can be explained by its ability to preserve overall representational structure without introducing additional parameters. In contrast, attention-based pooling requires learning layer weights, which may be difficult to

Table 6.5: Metrics by Pooling Type and Layer Configuration.

Pooling Type	Layers	Precision	Recall	F1 Score
Mean	3, 12	**0.7627**	0.5325	0.6272
	3, 9, 12	0.7504	0.5398	**0.6279**
	0, 3, 9, 12	0.7434	**0.5410**	0.6262
	0, 3, 7, 9, 12	0.7485	0.5380	0.6260
Max	3, 12	**0.7639**	0.5223	0.6204
	3, 9, 12	0.7570	0.5349	0.6269
	0, 3, 9, 12	0.7568	0.5325	0.6252
	0, 3, 7, 9, 12	0.7585	**0.5392**	**0.6303**
Attention	3, 12	0.7483	**0.5392**	0.6268
	3, 9, 12	0.7579	0.5373	**0.6288**
	0, 3, 9, 12	0.7528	0.5265	0.6196
	0, 3, 7, 9, 12	**0.7717**	0.5133	0.6165

optimize given the limited available data, leading to increased variability. This supports the view that layer representations are structured and non-uniform, and cannot be trivially aggregated without potential loss of information.

Relating to exisiting research Previous work on contextualized embeddings has shown that combining information across layers can improve performance, particularly when using learned weighting schemes such as scalar mixing [28]. However, such approaches often rely on sufficient training data to effectively learn optimal combinations. The present results suggest that, in a moderately sized dataset setting, the benefits of layer aggregation are limited.

Strengths and limitations A key strength of this experiment is the controlled setup, in which all configurations were evaluated under identical conditions. This allows for a direct comparison of pooling strategies. Additionally, multiple pooling methods were systematically explored, providing a comprehensive overview.

However, several limitations must be acknowledged. First, the observed performance differences are very small, and no statistical significance testing was conducted, making it difficult to determine whether these differences are meaningful. Second, the dataset size may be insufficient for effectively training attention-based pooling mechanisms. Third, only a limited set of layer combinations was considered, which may not include the optimal configuration.

Quality of results Regarding quality criteria, the results demonstrate good reliability, as performance trends are consistent across pooling strategies. Internal

validity is high due to the controlled setup. However, external validity is limited, as the findings may not generalize to other datasets or tasks. Furthermore, statistical conclusion validity is constrained by the absence of significance testing, which limits the strength of the conclusions.

Conclusion In summary, the findings indicate that while layer pooling is theoretically appealing, its practical benefits in this setting are minimal. The results emphasize the importance of selecting informative layers over increasing architectural complexity through advanced pooling mechanisms.

Chapter 7

Conclusions and Future Work

7.1 Conclusion

This thesis investigates whether synthetic data improves real-word error detection, which BERT layers provide the most informative representations, and whether layer pooling enhances performance.

To address these challenges, both data-centric and model-centric approaches were explored. A cleaned version of the Falko-MERLIN dataset was constructed, focusing exclusively on real-word errors. In addition, two synthetic data generation strategies were introduced: (i) LLM-based sentence rewriting while preserving errors, and (ii) tuple-based corruption using extracted error–correction pairs. On the modeling side, a baseline BERT-based token classification model was compared against several custom head architectures that incorporate representations from multiple transformer layers using concatenation and pooling techniques.

The experimental results indicate that synthetic data augmentation is the primary driver of performance improvements. In particular, tuple-based corruption provides the most balanced gains; improving recall and overall model performance without substantially degrading precision. Rewritten sentence augmentation, while achieving the highest recall, introduces a stronger trade-off by reducing precision.

In contrast, architectural modifications to the model have only a marginal impact on performance. While combining representations from multiple BERT layers leads to slight improvements, these gains are relatively small compared to those achieved through data augmentation. Similarly, different pooling strategies show limited influence, suggesting that increasing architectural complexity does not necessarily translate into better performance in this setting.

From a critical perspective, several limitations must be acknowledged. First, the observed performance improvements, while consistent, are moderate in magnitude. Second, no statistical significance testing was conducted, which limits the strength

of the conclusions. Third, the effectiveness of synthetic data depends heavily on its quality and diversity, which were not evaluated independently in this work. Finally, the experiments were conducted on a single dataset, which may limit the generalizability of the findings.

Overall, the results indicate that improvements in real-word error detection are driven primarily by data rather than model architecture. This suggests that, for this task, focusing on high-quality and diverse training data is more beneficial than increasing model complexity. The findings reinforce the importance of data-centric approaches in modern NLP and provide practical insights for the design of GED systems.

7.2 Future Work

While this thesis demonstrates that data-centric approaches substantially improve real-word error detection, the overall performance is still not sufficient for robust real-world deployment. Future work should therefore focus on improving both data quality and model capacity, with a strong emphasis on data-related aspects.

Data-Centric Improvements (Primary Focus) The most important direction for future research lies in improving the data. The experiments clearly indicate that performance gains are driven primarily by data rather than architectural changes.

First, incorporating additional and more diverse datasets is a natural next step. The current work relies on a relatively limited set of learner data, which restricts coverage of error types and linguistic variation. Integrating other learner corpora, as well as domain-specific or naturally occurring text data, could improve both robustness and generalization.

Second, extending the approach to other languages would provide valuable insights into the generality of the methods. Real-word error detection is not language-specific, but error distributions and linguistic phenomena differ significantly across languages. Evaluating the proposed pipeline on multiple languages would help assess its scalability and adaptability.

Third, future work should consider moving beyond detection towards error correction. While this thesis focuses solely on identifying errors, combining detection with correction would enable end-to-end systems. In particular, jointly modeling detection and correction, or using correction signals to improve detection, could lead to more informative supervision and better representations.

Finally, the quality of synthetic data should be further investigated. This includes improving generation strategies and filtering noisy samples. Understanding

which types of synthetic data are most beneficial remains an open research question.

Model-Centric Improvements Although architectural changes had only a limited impact in this work, several extensions remain worth exploring.

One direction is the use of more complex classification heads. The current heads are relatively simple, and more expressive architectures (e.g., deeper feed-forward networks or structured prediction layers) may better exploit the available representations.

Another straightforward extension is scaling the base model. Replacing the current 12-layer BERT model with a larger 24-layer variant may improve performance by providing richer contextual representations.

Additionally, exploring alternative pretrained models could be beneficial. More recent transformer architectures or models trained on larger and more diverse corpora may yield stronger baseline representations than the one used in this thesis.

Application and Integration Beyond isolated model improvements, an important direction is the integration of real-word error detection into practical applications.

In particular, combining detection with writing assistance or language learning tools offers significant potential. For example, integrating the model into intelligent tutoring systems or proofreading software could provide real-time feedback to users. In such settings, high recall remains crucial, but precision and interpretability also become important factors.

Summary In summary, while both data- and model-centric improvements are possible, the findings of this thesis strongly suggest that future progress will depend primarily on better data. Expanding datasets, improving synthetic data generation, and extending to new languages and tasks are likely to yield the most substantial gains.

Appendix A

Prompt Examples

This appendix documents the prompts used for synthetic data generation via a large language model. The prompts follow a consistent structure consisting of three components: (i) a task description, (ii) formatting instructions, and (iii) an input sentence containing explicitly marked errors.

This design aims to ensure consistency and controllability. By explicitly specifying both the task and the expected output format, the prompts guide the language model to produce structured and easily processable data for subsequent steps.

A.1 Prompt for Error-Correction Pair Extraction

The purpose of this prompt is to generate corrected forms of erroneous tokens. The model is instructed to return only the corrected words corresponding to the marked errors in the input sentence.

Prompt Example:

Verbessere die markierten Wörter im Satz, sodass die Bedeutung ähnlich bleibt.
Gib nur die korrekten Wörter zurück; ein Wort pro Zeile. Wird das Wort einfach gelöscht, schreibe ε.

Oft was am schwierigsten und am gefährlichsten ist , *sei* am nötigsten für die Entwicklung eines Landes .

The output of this prompt is used to construct error–correction pairs by aligning the corrected tokens with their original erroneous counterparts.

A.2 Prompt for Error-Preserving Sentence Rewriting

This prompt is designed to generate paraphrased variants of sentences while preserving the original errors. The model is encouraged to modify multiple parts of the sentence while keeping the marked erroneous tokens unchanged.

Prompt Example:

Verändere den Satz so, dass die markierten Fehler falsch bleiben, aber der Satz eine andere Bedeutung hat (z. B. durch Ersetzen von Verben oder Anpassung der Zeitform). Verändere mehrere Stellen im Satz. Markiere die falschen Wörter so wie zuvor.
Schreibe genau einen Satz pro Zeile und erzeuge drei neue Sätze. Die neuen Sätze müssen die gleiche Anzahl an Wörtern enthalten.

Für so was aber nicht nur , *ich* *wäre* gern nach deiner Rückkehr *dich* zu Hause besuchen .

This prompt enables the generation of diverse sentence variants while maintaining consistent error annotations, which is essential for downstream training tasks.

Appendix B

Machine Learning Basics

Linear Layer: A linear layer applies an affine transformation to an input vector:

$$z \mapsto Wz + b,$$

where W denotes the weight matrix and b the bias vector.

Dropout Layer: A dropout layer is a regularization technique in neural networks that randomly deactivates neurons during training to reduce overfitting [29]. During inference, dropout is disabled and the full network is used.

ReLU: Since compositions of affine transformations remain affine[1], non-linear activation functions are required to enable the model to learn non-linear relationships. One way is through the use of the rectified linear unit (ReLU) function – a non-linear function. The ReLU function is defined as

$$f(z)_i := \max\{0, z_i\},$$

so turning all negative entries of a vector to zero and leaving the positive unchanged.

GELU: Another non-linear activation function with similar properties is the GELU function, defined as

$$f(z)_i = z_i \Phi(z_i)$$

where $\Phi(x) = P(X \leq x)$ is the cumulative distribution function of a standard normal distribution.

[1] A, B Matrices $\quad a, b, v$ Vectors $\implies A(Bv + b) + a = \underbrace{(AB)}_{\text{Matrix}} v + \underbrace{(Ab + a)}_{\text{Vector}}$

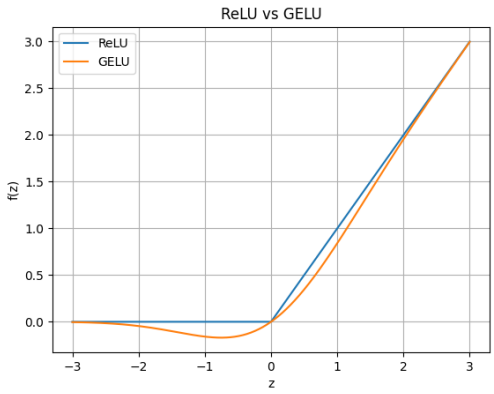

Figure B.1: The ReLU and GELU functions plotted on the range $[-3, 3]$.

Softmax: Since from the linear-layer applied to the embeddings yields logits, which is a vector over the reals, it has to be transformed into a vector representing probabilities. One way to do this transformation is through the use of the softmax function, which is defined as

$$\text{softmax}(z)_i := \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}.$$

This produces a probability distribution over classes, as all entries are non-negative and sum to one.

Feed Forward Network (FFN): This is a neural network where information only flows in one direction. For example with input x:

$$x \mapsto \text{GELU}(W_1 x + b_1) \cdot W_2 + b_2$$

This is actually the exact FFN that is used in BERT.

Layer Normalization: Layer normalization (LayerNorm) is a technique used to stabilize and accelerate the training of deep neural networks by normalizing the inputs across the features for each individual training example [30]. Unlike Batch Normalization, LayerNorm performs the same computation during both training and inference and does not depend on the batch size.

Batch Normalization computes normalized activations as:

$$\hat{x} = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}, \quad y_i = \gamma \hat{x}_i + \beta$$

where μ_B and σ_B^2 denote the mean and variance computed over the mini-batch x,

and γ, β are learnable scaling and shifting parameters.

Binary Cross Entropy: Every model is designed to minimize a loss function. One common loss function is the binary cross entropy, defined as

$$Loss := \mathcal{L} := -\sum_{y \in \text{words}} (y \log(p_y) + (1-y) \log(1-p_y))$$

where y is the label (in this work that is 1 for error, 0 for correct) and p_y the probability of y given from the model.

Appendix C

Result Tables

Here is an extensive list of all the results collected from all experiments.

Table C.1: Metrics with Additional Pair Based Generated Training Data.

Additional Data	Precision	Recall	F1 Score
+0	0.7811	0.4729	0.5891
+5,000	0.7359	0.5018	0.5967
+10,000	0.7348	0.4675	0.5714
+15,000	0.7689	0.5090	0.6125
+20,000	0.7888	0.5175	0.6250
+25,000	0.7879	0.4946	0.6077
+30,000	0.8041	0.4970	0.6143
+35,000	0.7873	0.4994	0.6111
+40,000	0.7752	**0.5422**	**0.6381**
+45,000	0.7942	0.4976	0.6119
+50,000	**0.8122**	0.4404	0.5711

Go to section 6.2.

Table C.2: Metrics with Additional Rewritten Sentences Training Data.

Additional Data	Precision	Recall	F1-Score
+0	**0.7811**	0.4729	0.5891
+5,000	0.7692	0.5181	0.6192
+10,000	0.7530	0.5325	0.6239
+15,000	0.7398	**0.5446**	**0.6273**
+20,000	0.7540	0.5337	0.6250

Go to section 6.2.

Table C.3: Metrics on Concatenating the Layers.

Layers	Precision	Recall	F1-Score
0,12	0.7553	0.5337	0.6255
1,12	0.7633	0.5187	0.6176
2,12	0.7653	0.5187	0.6183
3,12	0.7723	**0.5416**	**0.6367**
4,12	0.7651	0.5199	0.6191
5,12	0.7696	0.5211	0.6214
6,12	0.7665	0.5241	0.6225
7,12	0.7722	0.5289	0.6278
8,12	**0.7770**	0.5120	0.6173
9,12	0.7587	0.5380	0.6295
10,12	0.7689	0.5151	0.6169
11,12	0.7644	0.5259	0.6231

Go to section 6.3.

Table C.4: Metrics by Pooling Type and Layer Configuration.

Pooling Type	Layers	Precision	Recall	F1 Score
Mean	3, 12	**0.7627**	0.5325	0.6272
	3, 9, 12	0.7504	0.5398	**0.6279**
	0, 3, 9, 12	0.7434	**0.5410**	0.6262
	0, 3, 7, 9, 12	0.7485	0.5380	0.6260
Max	3, 12	**0.7639**	0.5223	0.6204
	3, 9, 12	0.7570	0.5349	0.6269
	0, 3, 9, 12	0.7568	0.5325	0.6252
	0, 3, 7, 9, 12	0.7585	**0.5392**	**0.6303**
Attention	3, 12	0.7483	**0.5392**	0.6268
	3, 9, 12	0.7579	0.5373	**0.6288**
	0, 3, 9, 12	0.7528	0.5265	0.6196
	0, 3, 7, 9, 12	**0.7717**	0.5133	0.6165

Go to section 6.4.

Bibliography

[1] Alan M. Turing. Computing machinery and intelligence. *Mind*, 59(236):433–460, 1950.

[2] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.

[3] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Y. Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1631–1642, 2013.

[4] Erik F. Tjong Kim Sang and Fien De Meulder. Introduction to the conll-2003 shared task: Language-independent named entity recognition, 2003.

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics*, pages 4171–4186, 2019.

[6] Fred J. Damerau. A technique for computer detection and correction of spelling errors. *Commun. ACM*, 7(3):171–176, March 1964.

[7] Mark D. Kernighan, Kenneth W. Church, and William A. Gale. A spelling correction program based on a noisy channel model. In *COLING 1990 Volume 2: Papers presented to the 13th International Conference on Computational Linguistics*, 1990.

[8] Daniel Dahlmeier, Hwee Tou Ng, and Siew Mei Wu. Building a large annotated corpus of learner English: The NUS corpus of learner English. In Joel Tetreault, Jill Burstein, and Claudia Leacock, editors, *Proceedings of the Eighth Workshop on Innovative Use of NLP for Building Educational Applications*, pages 22–31, Atlanta, Georgia, June 2013. Association for Computational Linguistics.

[9] Rico Sennrich, Barry Haddow, and Alexandra Birch. Improving neural machine translation models with monolingual data. In *ACL*, 2016.

[10] Jason Wei and Kai Zou. Eda: Easy data augmentation techniques for boosting performance on text classification tasks. In *EMNLP*, 2019.

[11] Qizhe Xie et al. Unsupervised data augmentation for consistency training. *NeurIPS*, 2020.

[12] Lin Long, Rui Wang, Ruixuan Xiao, Junbo Zhao, Xiao Ding, Gang Chen, and Haobo Wang. On llms-driven synthetic data generation, curation, and evaluation: A survey, 2024.

[13] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives, 2014.

[14] Tomas Mikolov et al. Efficient estimation of word representations in vector space. In *ICLR*, 2013.

[15] Jeffrey Pennington, Richard Socher, and Christopher Manning. Glove: Global vectors for word representation. *EMNLP*, 2014.

[16] Branden Chan, Timo Möller, Malte Pietsch, and Tanay Soni. German bert. https://huggingface.co/google-bert/bert-base-german-cased, 2019. Developed by deepset.ai.

[17] Yinhan Liu et al. Roberta: A robustly optimized bert pretraining approach. *arXiv:1907.11692*, 2019.

[18] Ian Tenney et al. Bert rediscovers the classical nlp pipeline. *ACL*, 2019.

[19] Språkbanken Text. Multiged 2023 shared task dataset, 2023. German subset based on Falko-MERLIN dataset.

[20] Humboldt-Universität zu Berlin. Falko – lernerkorpus deutsch als fremdsprache, n.d.

[21] MERLIN Project. Merlin platform, n.d.

[22] Marvin Wendt. German wordlist. https://gist.github.com/MarvinJWendt/2f4f4154b8ae218600eb091a5706b5f4, 2017. GitHub Gist, accessed: 2026-04-18.

[23] OpenAI. Openai api documentation. https://platform.openai.com/docs, 2025. Accessed: 2026-04-24.

[24] Aaditya Singh, Adam Fry, Adam Perelman, Adam Tart, Adi Ganesh, Ahmed El-Kishky, Aidan McLaughlin, Aiden Low, AJ Ostrow, Akhila Ananthram, et al. Openai gpt-5 system card, 2025.

[25] Hugging Face Inc. Transformers documentation: Bertfortokenclassification, n.d.

[26] Yonatan Belinkov and James Glass. What does bert learn about the structure of language? *arXiv preprint arXiv:1906.04341*, 2019.

[27] Yoon Kim. Convolutional neural networks for sentence classification, 2014.

[28] Dongsuk Oh, Yejin Kim, Hodong Lee, H. Howie Huang, and Heuiseok Lim. Don't judge a language model by its last layer: Contrastive learning with layer-wise attention pooling, 2022.

[29] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

[30] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, 2016.

Erklärung

gemäss Art. 30 RSL Phil.-nat.18

Name/Vorname:	Kellenberger, Elias Raphael
Matrikelnummer:	21-120-795
Studiengang:	Computer Science

Bachelor ☑ Master ☐ Dissertation ☐

Titel der Arbeit:	Detection and Generation of Real-Word Errors in German Using Language Models
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.

Bätterkinden, 28.04.2026

Ort/Datum

Unterschrift E. Kellenberger