<Corinna Grabner> <BSc>

<Training of the Language Model> <PikabERT>

MASTERARBEIT

zur Erlangung des akademischen Grades
Diplom-Ingenieurin

STUDIUM
Informatics

Alpen-Adria-Universität Klagenfurt
Faculty of Technical Science

BEGUTACHTER
<Assoc. Prof. Dipl-Ing. Dr. Konstantin Schekotihin>

Institut für Angewandte Informatik

Version <0.01> Klagenfurt am Wörthersee, January 4, 2023

<Vor der Abgabe
mit Version der AAU abgleichen oder besser
Version der AAU einbetten!>

Zitat, falls gewünscht
von wem?

Affidavit

I hereby declare in lieu of an oath that

- the submitted academic paper is entirely my own work and that no auxiliary materials have been used other than those indicated;
- I have fully disclosed all assistance received from third parties during the process of writing the paper, including any significant advice from supervisors;
- any contents taken from the works of third parties or my own works that have been included either literally or in spirit have been appropriately marked and the respective source of the information has been clearly identified with precise bibliographical references (e.g. in footnotes);
- to date, I have not submitted this paper to an examining authority either in Austria or abroad and that
- the digital version of the paper submitted for the purpose of plagiarism assessment is fully consistent with the printed version.

I am aware that a declaration contrary to the facts will have legal consequences.

<Corinna Grabner, BSc> e.h. Klagenfurt, **<01.08.2021>**

<Please check against the AAU version of this page or embed the AAU version!>

)

Acknowledgements

Unser Dank gilt Raphael Wigoutschnigg, da diese Vorlage für wissenschaftliche Arbeiten der Gruppe Systemsicherheit (syssec) auf den LaTeX-Files seiner Diplomarbeit aufbaut.

Abstract

Infineon's Failure Analysis department examines the produced chips for errors in production and also deals with customer complaints. There are over 2,000 types of chips, which differ only minimally in layout. For the analysis, microscopes, lasers and X-Ray equipment is used. The department makes a very important contribution to the quality assurance of products at Infineon. If a product is not working properly, the resulting error code can be determined and the now called target product is sent to the failure analysis (FA) laboratory. Detection and localization of faults in semiconductors is a knowledge-intensive and tedious task. To increase the chances of success, an electrical engineer should be able to get all available information about the samples of similar past jobs. Various support systems used in Failure Analysis (FA), like databases, wikis, or file shares, often have this information stored as documents describing previous analysis reports of similar samples, best practices, specifications, customer reports, etc. However, accessing knowledge contained in these documents can be problematic, since in most cases, such support systems only provide rudimentary search functionality, like keyword matching. As a result, to find relevant information about jobs similar to the considered one, an engineer must query multiple systems, manually evaluate returned reports looking for similar characteristics, and asserting the value of each document for the current problem.
Modern Natural Language Processing methods (NLP) already showed their efficiency in various applications, including automatic translators, Recommender Systems or chatbots. Among these applications, text classification is one of the most promising to solve the FA search problem by automatically associating labels with a report denoting physical or electrical faults described in it, applied methods and tools, etc. The engineers can then use these labels to perform various tasks, like identifying similar jobs or getting statistics on possible faults, tools, or methods.
This is why one of the first applications of Artificial Intelligence (AI) tools at the FA laboratory of Infineon consisted on a classifier of the FA reports, using word2vec embeddings and clustering models, which satisfactory results.
With the apparition of more sophisticated tools in NLP during the last few years, the development of Transformer models until the publication of BERT, the results obtained by these models surpass anything utilized in previous times in almost every field where Transformers have been tested.

Zusammenfassung

Die Fehleranalyseabteilung von Infineon untersucht die produzierten Chips auf Produktionsfehler und kümmert sich auch um Kundenreklamationen. Es existieren über 2.000 Chiptypen, die sich nur minimal im Layout unterscheiden. Für die Analyse werden Mikroskope, Laser und Röntgengeräte verwendet. Die Abteilung leistet einen sehr wichtigen Beitrag zur Qualitätssicherung der Produkte bei Infineon. Funktioniert ein Produkt nicht richtig, kann der resultierende Fehlercode ermittelt und an das FA-Labor gesendet werden. Das Erkennen und Lokalisieren von Fehlern in Halbleitern ist eine wissensintensive und langwierige Aufgabe. Um die Erfolgschancen zu erhöhen, sollte ein Elektroingenieur in der Lage sein, alle verfügbaren Informationen über ähnliche vergangene Jobs zu erhalten. Verschiedene Unterstützungssysteme, die in der Fehleranalyse (FA) verwendet werden, wie Datenbanken, Wikis oder Dateifreigaben, haben diese Informationen oft als Dokumente gespeichert, die frühere Analyseberichte ähnlicher Proben, Best Practices, Spezifikationen, Kundenberichte usw. beschreiben. Jedoch kann das Zugreifen auf das enthaltene Wissen in diesen Dokumenten problematisch sein, da solche Unterstützungssysteme in den meisten Fällen nur rudimentäre Suchfunktionen, wie z. B. Keyword-Matching, bereitstellen. Um relevante Informationen zu Jobs zu finden, die dem betrachteten ähnlich sind, muss ein Ingenieur daher mehrere Systeme abfragen, zurückgegebene Berichte manuell nach ähnlichen Merkmalen auswerten und den Wert jedes Dokuments für das aktuelle Problem bestätigen.
Moderne Methoden der Verarbeitung natürlicher Sprache (NLP) haben ihre Leistungsfähigkeit bereits in verschiedenen Anwendungen unter Beweis gestellt, darunter automatische Übersetzer, Recommender-Systeme oder Chatbots. Unter diesen Anwendungen ist die Textklassifizierung eine der vielversprechendsten, um das FA-Suchproblem zu lösen, indem Etiketten automatisch mit einem Bericht verknüpft werden, der darin beschriebene physikalische oder elektrische Fehler, angewandte Methoden und Werkzeuge usw. bezeichnet. Die Ingenieure können diese Etiketten dann zur Durchführung verwenden verschiedene Aufgaben, wie das Identifizieren ähnlicher Jobs oder das Abrufen von Statistiken über mögliche Fehler, Werkzeuge oder Methoden. Aus diesem Grund bestand eine der ersten Anwendungen von Tools für künstliche Intelligenz (KI) im FA-Labor von Infineon in einem Klassifikator der FA-Berichte unter Verwendung von word2vec-Einbettungen und Clustering-Modellen, die zufriedenstellende Ergebnisse lieferten.
Mit der Entstehung immer ausgefeilterer NLP-Tools in den letzten Jahren, der Entwicklung von Transformer-Modellen bis zur Veröffentlichung von BERT, übertreffen die Ergebnisse dieser Modelle alles, was in früheren Zeiten in fast allen Bereichen, in denen Transformers getestet wurden, verwendet wurde.

Wichtige Hinweise zur vorliegenden LaTeX-Vorlage

- Diese Vorlage kann ohne Änderungen mittels PDFLaTeX von MikTeX compiliert werden. Daher müssen auch die Abbildungen als PDF vorliegen.
- Muss zur Compilierung aber LaTeX genutzt werden, da psfrag (oder Ähnliches verwendet wird),dann darf das Package hyperref nicht verwendet werden → Package hyperref (inkludiert am Beginn dieser Datei) auskommentieren und \newcommand{\href}[2]{#2} definieren. Abbildungen müssen dann als eps vorliegen.
- Damit das Literaturverzeichnis erzeugt wird, muss "bibtex thesis" aufgerufen werden.
- Damit das Abkürzungsverzeichnis erzeugt wird, muss "nomenclature.bat" aufgerufen werden.
- Einige Dokumente die den Umgang mit LaTeX und diversen Packages beschreiben, finden Sie im Verzeichnis "README".
- Textstellen (wie Datumsangeben oder Nafkmen) die anzupassen sind, wurden teilweise mit Platzhaltern der Form <**Bechreibung**> versehen! Nicht markierte Stellen der Titelseite, die gegebenenfalls anzupassen sind, umfassen
  - Bezeichnung des Studiums und
  - Name der betreuenden Assistentin bzw. des betreuenden Assistenten.
- Bitte Sicherstellen, dass die Titelseite und die Eidesstattliche Erklärung der aktuellen Fassung der AAU entsprechen. Die zugehörigen Seiten können auch auf der Upload-Seite der AAU erzeugt und in dieses Dokument eingebunden werden.
- Vor dem Ausdruck für die gebundene Masterarbeit ist die boolesche Variable "FINAL" (siehe Zeile 49 in dieser Datei) auf "true" zu setzen! Nicht ersetzte Platzhalter – das sind Kommandos der Form \ph{...} – werden dann beim Aufruf von PDFTex zu einer Fehlermeldung führen.
- Die aktuelle Version dieses Dokuments finden Sie auf der Website des Instituts für Angewandte Informatik `https://www.aau.at/en/ainf/teaching/templates`.

Table of Contents

1 Introduction

Infineon technologies Austria is one of the leading semiconductor companies worldwide. With around 4,820 employees, the company makes an important contribution to shaping the digital and networked future. Through the development of microelectronics and their constant improvement, Infineon enables efficient energy management, intelligent mobility and secure, seamless communication.

1.1 Motivation

Detection and localization of faults in semiconductors is a knowledge-intensive and tedious task. To increase the chances of success, an electrical engineer should be able to get all available information about the samples of similar past jobs. Various support systems used in Failure Analysis (FA), like databases, wikis, or file shares, often have this information stored as documents describing previous analysis reports of similar samples, best practices, specifications, customer reports, etc. However, accessing knowledge contained in these documents can be problematic, since in most cases, such support systems only provide rudimentary search functionality, like keyword matching. As a result, to find relevant information about jobs similar to the considered one, an engineer must query multiple systems, manually evaluate returned reports looking for similar characteristics, and asserting the value of each document for the current problem. Furthermore, the report corpus includes a quantity of over one million documents, which means a long search time if you are looking for a specific topic. These procedure is depicted in Figure 1.1

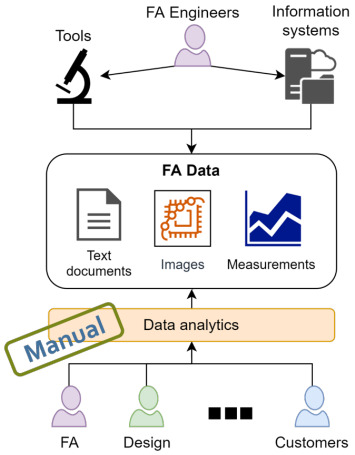

Fig. 1.1: FA Analysis Process

In order to accelerate this process of troubleshooting and to make it easier to find similarities between past and new jobs, the integration of artificial intelligence in the failure analysis process is suitable. Integrating AI into Infineons failure analysis could look like in Figure 1.2 where AI infrastructures are used to load and process ressources and help not only FA employees, but all Infineon employees and customers to better understand the existing data through appropriate processing and to be able to use it in a targeted manner.

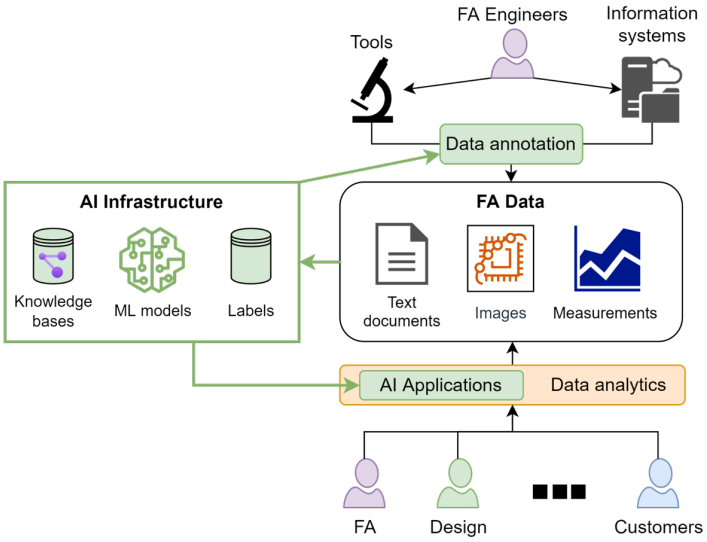

Fig. 1.2: FA Analysis Process with AI Integration

Modern Natural Language Processing methods (NLP) already showed their efficiency in various applications, including automatic translators, Recommender Systems or chatbots. Among these applications, text classification is one of the most promising to solve the FA search problem by automatically associating labels with a report denoting physical or electrical faults described in it, applied methods and tools, etc. The engineers can then use these labels to perform various tasks, like identifying similar jobs or getting statistics on possible faults, tools, or methods.

This is why one of the first applications of Artificial Intelligence (AI) tools at the FA laboratory of Infineon consisted on a classifier of the FA reports. To support the classifier and improve performance as best as possible, a language model is trained on in-domain text data as shown in Figure 1.3. The model should support engineers during the analysis and report writing process.

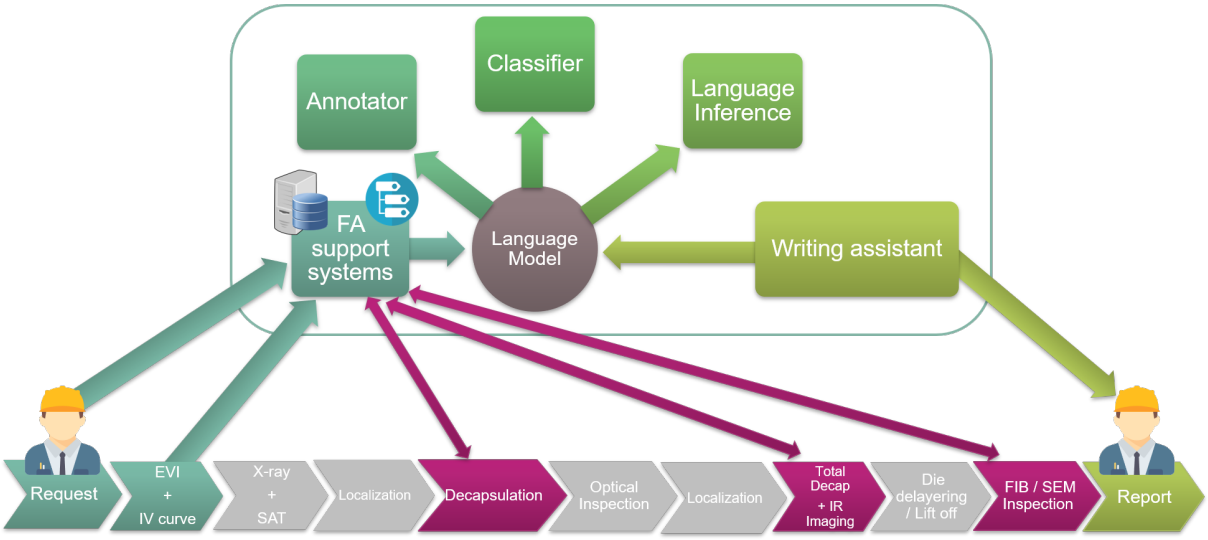

Fig. 1.3: FA-BERT Integration into FA Analysis Prozess

After receiving a in-domain trained language model it can be used for several downstream tasks like classification of electrical and physical failures, improving the classification results by segmenting FA texts according to their topic, Named Entity Recognition (NER) were the model will be integrated in the production environment for automated annotation of FA reports or Natural Language Inference (NLI) tasks for question answering and information retreival.

1.2 Problem Description

The goal of this project is to develop a FA report classifier with a BERT model. In order to do so, the task has been divided into two phases:
First, we have developed a Language Model based on the state-of-the-art model BERT. Therefore we had to consider our specific domain and select the most appropriate model for our domain. Since many successors of BERT have been developed but none of them aimed at the electrical domain, we had to select a model as close as possible to our field of study in order to achieve a satisfying performance later.
Second, we focused on defining the structure of the classifier, which consisted on the BERT network and additional classification layers. To test the results, we have defined a series of classification problems based on the FA reports.
Given the nature of this project the two phases were developed in parallel so the joint performance hasn't been tested yet.

1.3 Research Questions

This thesis will address the following research questions:

- What kind of BERT models already exist and which one to choose as an entrypoint for training?
- What data should be used for further training and how to collect it?
- How to train and evaluate the resulting model?

The following chapters in this thesis will cover the research questions described above. Before that, as a basis for the content that follows, some terms and topics related to natural language processing will be discussed.

1.4 Contributions

RQ1: What kind of BERT models already exist and which one to choose as an entrypoint for training?
In order to fulfill the goal of developing an innovative language model trained on the semiconductor domain, it was necessary to find an existing language model which has already been trained on english text similar to electrical engineering or semiconductor topics. With this language model to use as entry point for our training, we assumed it would be easier to achieve our desired goals. After extensive search and research, we found possible candidates for the model entry point. One of the largest BERT models is Med-BERT, which has been trained on medical and biomedical documents. Unfortunately, this model is not suitable as a starting point for our training, since words in medical texts are very similar to those in semiconductor texts but have a different semantic meaning. Because of this, the model can become "confused" when trying to understand the text. Other promising models are SciBERT and S2ORC-SciBERT. Both models are trained using scientific, english texts, with the training corpus consisting of chemistry, physics, but also mainly medical topics. In contrast to SciBERT, S2ORC-SciBERT has a slightly larger vocabulary and was also trained on a larger data set, which also includes topics such as computer science and engineering.
After some evaluation processes on error analysis reports, S2ORC-SciBERT could be chosen as the entry point. The performance of S2ORC-SciBERT is slightly better than that of SciBERT.

RQ2: What data should be used for further training and how to collect it?
To further train the language model entrypint on a domain specific dataset to achieve the desired results, the specific dataset had to be collected and transformed into a shape which allows to train a BERT based language model. To collect the domain specific texts, we used *GoogleScholar*, *SemanticScholar* and *IEEE* to download as many papers as possible. Also we used parts of the S2ORC dataset and filtered for texts with the topics chemistry, physics, computer science, mathematical science and engineering. After collecting the papers, the raw text had to be extracted. Therefore we were allowed to use a text extractor tool developed by Infineon collegues in Bangalore and did not had to develop it on our own. The raw text had been saved in csv files and could be loaded to create a okenized dataset ready for training. In total the training data frame included 1 783 789 rows which is about 10GB of data.

RQ3: How to train and evaluate the resulting model?
As a training method we decided to choose the most common way of how to pretrain a BERT-based model: using Masked Language Modeling. This technique had already been used for training the original BERT model and will now also been used in an adapted way for our training experiments. In order to achieve a satisfying result, we wanted our model to understand the FA reports as good as possible. With the whole-word-masking technique, which is an adaption of the traditional masked language modeling algorithm, we achieved a lower loss during training. The evaluation of the final model on the FA report job summaries show, that our trained model has a better understanding of the texts than the initial model.

2 Literature Review

In this section we describe some techniques used in NLP, trying to give an overview of the current state of art and what is necessary to train a language model.

2.1 Natural Language Processing

Communication and the sharing of knowledge is an essential part of human society. History has shown that the best way to transmit knowledge is in writing. It is not difficult for a person to learn to read and to understand the context in texts. For computers, this task is much more difficult. Natural Language Processing (NLP) is a subfield of Artificial Intelligence (AI) which deals with teaching computers to understand and interpret human language and has emerged in 1940. The initial need of NLP was the translation from one language into another which was used during the second world war. Nowadays it is widely used in health care, spam detection, sentiment and cognitive analysis. NLP also provides computers with the ability to read text, hear speech, and communicate with humans. Every person had contact with a NLP device at least once in their life. The best example everyone might know are personal assistants like Siri or Alexa. These are applications which work with NLP and have learned to communicate with humans and nearly seem humanely.
Generally speaking, NLP breaks down language into shorter, more basic pieces, called tokens (words, periods, etc.) which get saved as vectorial representations. The technique of mapping words to real vectors is called word embedding. To understand the relationships of the tokens, the model gets trained by predicting some hidden part of the text using some other part of their surrounding text. One way to calculate the representations of the vectors is the method *Word2Vec*. Until BERT has emerged, Word2Vec was the most common technique to create word embeddings. It is a two-layer neural network which processes text and turns it into a numerical form that can be understand by deep neural networks. Given enough data, Word2vec can make highly accurate guesses about a word's meaning based on past appearances. Those guesses can be used to establish a word's association with other words. The diference between Word2Vec and BERT will be covered in Section 2.3.3.

2.2 Text Representation

Text representation is a fundamental problem of NLP and necessary for preparing raw text as input for a language model. It aims to create a numerical representation of the unstructured text and make them mathematically computable. It is also called text vectorization or feature extraction.
Texts cannot be processed by ML models directly. Tokenization is a method that transforms

sequences of characters into a sequence of integers. As an example we can take the sentence:

"This is a cat."

A tokenization transforms this sentence into [‘This’, ‘is’, ‘a’, cat’]. Punctuations will be removed. This step is important to help the model understanding the meaning of a text. The model can

- Count the number of words in the text
- Count the frequency of the word, that is, the number of times a particular word is present

In modern LMs, tokenizers are trained together with the model to identify the best possible transformations, which can happen on both word and sub-word levels. The set of obtained tokens determines the vocabulary of an LM. If a word appears in the vocabulary of LM, it can be represented as a vector in the target space. Otherwise, a word is split into parts until each of them can be mapped to a set of tokens from the vocabulary. In the worst case, a word is split into individual letters. Such splitting may significantly reduce the quality of word embeddings in the vector space, thus reducing the quality of features extracted for the classification layers of a network. Therefore, it is essential either to select an LM whose vocabulary provides good coverage of the main terms used in an application domain or to extend the vocabulary of an LM with these terms and fine-tune it on domain-specific texts.

The next chapters cover the most common techniques of text representation.

2.2.1 One-Hot Encoding

One-hot encoding is a type of representation were words in a document get converted in a V-dimension vector. When combining all, the result is a single document in the shape of a 2-D array as shown in the example below. In this example we assume that the document contains only the sentence **This is a cat.**:

Word	**Vector Representation**
This	[1, 0, 0, 0]
is	[0, 1, 0, 0]
a	[0, 0, 1, 0]
cat	[0, 0, 0, 1]

Tab. 2.1: One Hot Encoding Example

Although it is very simple, one-hot encoding has the disadvantage of sparsity. Every single sentence creates a vector of n*m size where n is the length of sentence m is a number of unique words in a document and 80 percent of values in a vector is zero. Also each document is of a different length which creates vectors of different sizes. Later in chapter 2.3.3 it will be described that vectors of different sizes cannot be feed into the model. As the last disadvantage one-hot encoding does not capture semantics. The actual meaning of a sentence should be observed in numbers which is not possible whith this technique as the example shows.

2.2.2 Term Frequency - Inverse Document Frequency

Term Frequency - Inverse Document Frequency (TF-IDF) is a statistical measure to determine how important a word is within a document. It determines not only how often a word appears in a single document, but also how often it appears in the entire document corpus. Very common words like "this" or "and" get a lower score, even though they occur often, because they are not important for the context of a document. This will be determined by comparing the occurrence of these words among the documents. This strategy is often used in NLP for text analysis. For example, in a failure analysis report, a word like failure would get a high score. By using other important words, the report could be automatically assigned to a certain failure type and thus classified.
TF-IDF for a word in a document is calculated by multiplying the term frequency and the inverse document frequency.

$$tfidf(t, d, D) = tf(t, d) * idf(t, D) \tag{2.1}$$

Where

$$tf(t, D) = log(1 + freq(t, d)) \tag{2.2}$$
$$idf(t, D) = log(N/count(dED, tEd)) \tag{2.3}$$

The term frequency *tf* is either a simple count of instances a word appears in a document or it can be adjusted by taking the length of the document to account. The calculation can be seen in (2.3) where t is the term and D the document to analyse. The inverse document frequency *idf* is determined accross the document corpus. The closer this metric is to zero, the more common a word is. The calculations can be seen in (2.4) where N is the number of documents.

2.2.3 Bag of Words

Bag of Words is one of the most used text vectorizing techniques. It is a method for simplifying the representation of documents and the importance of the words they contain which is used in NLP and Information Retreivel (IR). Here the sentences in the documents are divided into their individual words and counted. The resulting object contains a list of the contained words and the number of their occurrences. It is called a “bag” of words, because any information about the order or structure of words in the document is discarded. The model is only concerned with whether known words occur in the document, not where in the document. We will describe the concept of bag-of-words with the following example: The below snippet contains the first few lines of the book "A Tale of Two Cities" by Charles Dickens.

It was the best of times,
it was the worst of times,
it was the age of wisdom,
it was the age of foolishness,

Tab. 2.2: Example document for bag of words technique

In this example, each line will be treated as an individual document. The model vocabulary is containing the following 10 words:

- “it”
- “was”
- “the”
- “best”
- “of”
- “times”
- “worst”
- “age”
- “wisdom”
- “foolishness”

Using this vocabulary, vector representations can be created for any document. For this purpose, a suitable scoring method must be chosen. The scoring method can be binary scoring or frequencies. Table 2.3 is showing an example of the resulting vector representations for a document which contains all of the lines in our example.

Scoring Method	**Resulting Vector Representation**
Binary	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
Frequencies	[4, 4, 4, 1, 4, 2, 1, 2, 1, 1]

Tab. 2.3: Example of scoring methods for a document including the entire corpus

As a second example Table 2.4 is showing the resulting vectors for a document which would include only the first two lines of our corpus in Table 2.2:

Scoring Method	**Resulting Vector Representation**
Binary	[1, 1, 1, 1, 1, 1, 1, 0, 0, 0]
Frequencies	[2, 2, 2, 1, 2, 2, 1, 0, 0, 0]

Tab. 2.4: Example of scoring methods for a document including the first two lines

In comparison to one-hot encoding, the resulting vectors are of fixed size and can be feed into machine learning models. There is still the disadvantage of sparsity because the vectors can get very long.

2.2.4 Word2Vec Architecture

Word2vec is not a singular algorithm, it is a family of model architectures and optimizations that can be used to learn word embeddings from large datasets. The strength of Word2Vec lies in assigning similar vectors to words with similar meanings. With other methods like one-hot

encoding, all the words in a vocabulary are independent of each other. With Word2Vec the embedding can be obtained using two methods (both involving Neural Networks): Skip Gram and Common Bag Of Words (CBOW).

Common Bag of Words

This method takes the context of each word as the input and tries to predict the word corresponding to the context. Considering the example *Have a great day*, let the input to the Neural Network be the word, *great*. The goal is to predict a target word *day* using the single context input word *great*. In the process of predicting the target word, the network learns the vector representation of the target word. Figure 2.1 shows the architecture of a simple CBOW model with one hidden layer where *Wvn* is the weight matrix that maps the input x to the hidden layer which is a V*N dimensional matrix. *W'nv* is the weight matrix that maps the hidden layer outputs to the final output layer which is a N*V dimensional matrix. The hidden layer neurons just copy the weighted sum of inputs to the next layer.

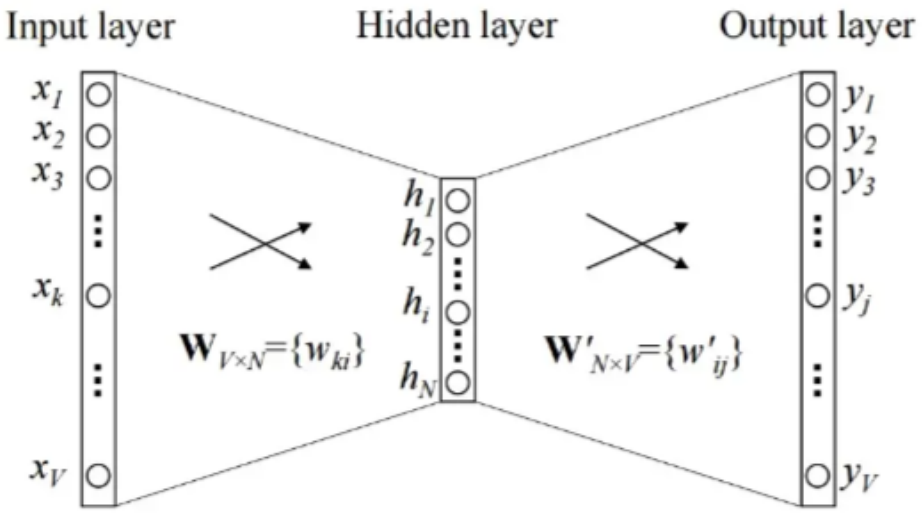

Fig. 2.1: Simple CBOW Network with one hidden layer

The model in Figure 2.1 uses a single context word to predict the target word. The same calculations and predictions can be done by using multiple context words as an input and feed them into the hidden layer.

Skip Gram

The Skip Gram model is an alternative to the CBOW model and works the other way around. It uses the target word to predict the context and produces the representation of the tarked word during the process. An example of the model is shown in Figure 2.2.

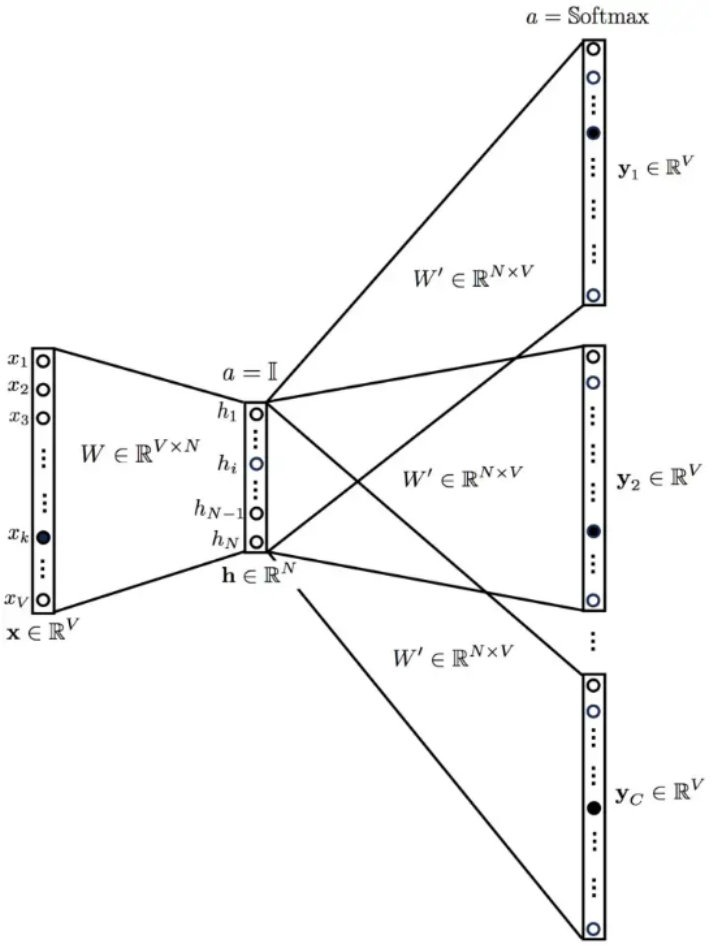

Fig. 2.2: Simple CBOW Network with one hidden layer

When putting the target word into the network, the model outputs C probability distributions which means that for each context position, we get C probability distributions of V probabilities, one for each word.
In both cases, the models are using back propagation to learn.

2.3 Masked Language Modelling

Masked Language Modeling (MLM) is a learning technique which got introduced with the language model BERT in 2018. For this training method some tokens of the input sentence will get masked as in the example *Today is a [MASK] day.* Before discussing the method in deep, we want to introduce the BERT model and the transformer architecture.

2.3.1 Transformer

The transformer in the field of NLP is a new architecture which is able to solve sequence-to-sequence tasks while handling dependencies in the text. Transformers basically consist of an encoder and a decoder. The encoder reads the input text and the decoder produces a

prediction. Transformers work in small increments. In each step, an attention mechanism is applied to understand the relationships between the words in a sentence, regardless of their positions. Both encoder and decoder sections of transformer are a stack of six identical layers of multi-head attention and feed forward sublayers. The architecture is depicted in Figure 2.3.

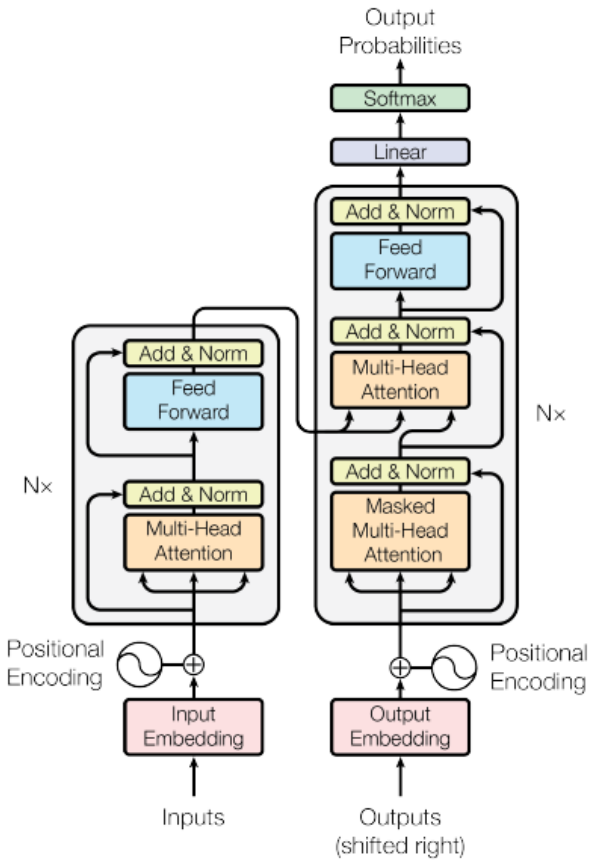

Fig. 2.3: Transformer Architecture

The encoder maps an input sequence of symbol representations *x1, ... , xn* to a sequence of representations *z = (z1, . . . , zn)*. Given *z*, the decoder then generates an output sequence *(y1, . . . , ym)* of symbols one element at a time. Each layer has an output of dimension 512.

2.3.2 Evolution of Language Models

It has always been difficult to teach a computer to really understand natural language. While they are capable of processing and storing large amounts of data, they lack language context. This problem continued until NLP became popular in the Artificial Intelligence field. NLP enables computers to read, analyze, interpret and gather information from long texts and single written words. In contrast to before without NLP, meaning and sense of texts can now be determined. In the beginning, a separate model was developed and used for each of these tasks. Since the development of BERT in 2018, this has changed. Engineers are now able to use a single model to handle the most common NLP tasks such as Classification, Sentiment Analysis or Question Answering. The timeline depicted in Figure 2.4 shows the evolution of NLP models from *Bag OF Words* until *BERT*.

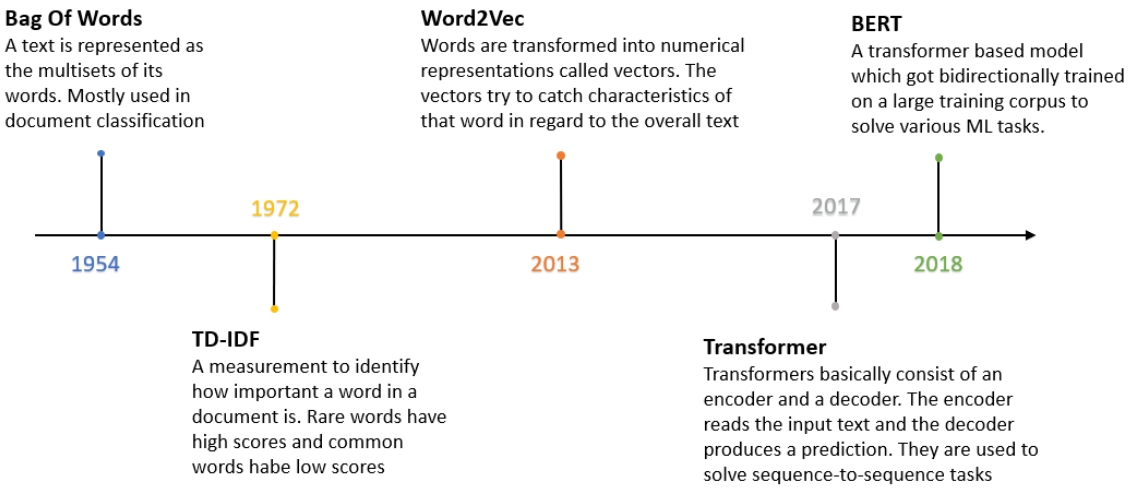

Fig. 2.4: Timeline of Model evolution in NLP.

The following sections will briefly describe all of the mentioned models to cover all the important topics which will be discussed in this paper and to prepare for the technical details of BERT.

2.3.3 BERT

BERT stands for Bidirectional Encoder Representations from Transformers. It makes use of a transformer and an attention mechanism that learns contextual relations between words or sub-words in a text. A Transformer usually includes two separate mechanisms, an encoder that reads the text input and a decoder that produces a prediction for the task, which got already introduced in chapter 2.3.1. For producing a language model, only the encoder mechanism is necessary. The BERT model was developed by Google researchers in 2018 and is already pre-trained using a combination of masked language modeling objective and next sentence prediction on a large corpus. The original BERT model comes in two sizes: BERT-base (trained on BooksCorpus: 800 million words) and BERT-large (trained on English Wikipedia: 2,500 million words). In contrast to previous language models which looked at a text sequence either from left to right or combined left-to-right and right-to-left training, BERT is using the bidirectional approach. The paper's results show that a language model which is bidirectionally trained can have a deeper sense of language context and flow than single-direction language models. To do this, the authors introduced a new technique which is called Masked Language Modelling. This objective randomly masks 15% tokens from an input sequence during training. The model has to predict the original token based on the context. Out of these 15%, the algorithm exchanges 80% of the tokens with the [MASK] token, 10% with any random word and the other 10% remain the same word. The second pre-training technique introduced in BERT is Next Sentence Prediction that takes in a sentence-pair input, replaces the second input sentence with a random sentence for 50% of the training steps, and trains on the sentence pairs to learn sentence relationships. The input to BERT's encoder is a sequence of tokens previously converted into vectors. Later, these tokens can be processed in the neural network. In order to be able to do this, a few steps must first be carried out:

1. CLS and SEP tokens at the beginning and end of each sentence

2. Segment embeddings for each token to distinguish between sentences.
3. Position embeddings for each token to identify the position in a sentence.

These three steps are depicted in Figure 2.5.

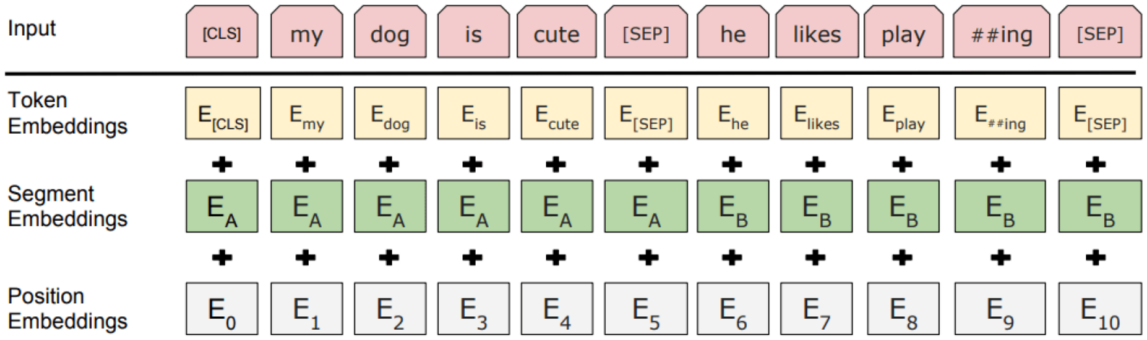

Fig. 2.5: Embeddings of Text Input with BERT.

BERT is optimized to either get a single text sentence or a pair of text sentences as input. In the case of two sentences, each token in the first sentence receives embedding A, and each token in the second sentence receives embedding B as visible in Figure 2.5. As each layer in the transformer architecture produces an ouptut of dimension 512, the supported sequence length of BERT is also 512 tokens.

The training of the BERT language model is divided into pre-training and fine-tuning. The pre-training process starts with next sentence prediction tasks followed by MLM tasks. For the NSP, 50% of the time the sentence B is replaced with a random sentence. After the pre-training, the fine-tuning process follows which includes an added classification layer. In particular, the fine-tuning process is aimed at certain NLP downstream tasks. During fine-tuning only batch size, learning rate and the number of training epochs are adapted. All other hyperparamethers stay the same as during pre-training. The two training processes are depicted in Figure 2.6[?].

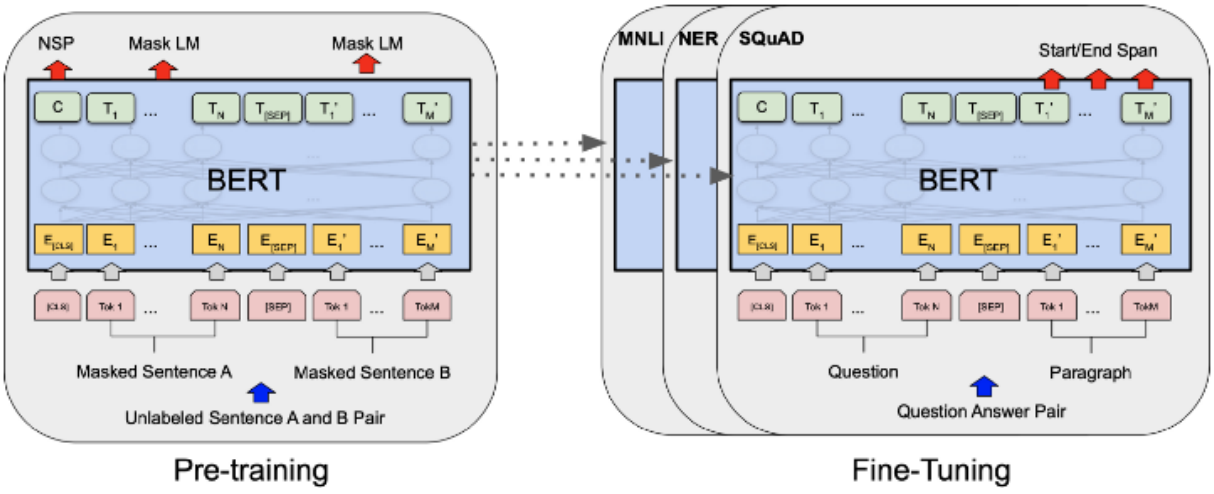

Fig. 2.6: Pre-Training and Fine-Tuning Process of BERT

Difference between BERT and Word2Vec

Word2Vec is contex-independent, so it only has a numeric vector to represent a word. If a word has several meanings, these are combined into a vector.

BERT, on the other hand, is context-dependent and thus allows multiple numeric vectors as a representation for a word, depending on the context in which the word occurs.

An example of the difference is the word bank, which can appear in a financial context as well as in a beach or park context. Word2Vec will always generate the same vector for this word and can therefore lead to an inaccurate representation. BERT can distinguish the two different semantic meanings and thus also generate two different vectors.

1. The next difference is that Word2Vec doesn't care about the position of words in a sentence. BERT, on the other hand, uses the position (index) of a word as input for calculating the vector.
2. Word2Vec only needs one word as input and delivers a vector as output. BERT, on the other hand, needs an entire sentence as input because it needs the context of the sentence to calculate the vector.
3. Word2Vec can have problems if a word is not stored in the vocabulary and no vector can be generated. BERT can also create a vector for subwords that are not stored in the vocabulary and is therefore not limited by the vocabulary.

3 Selection of Language Model

This chapter will cover the first two research questions about existing BERT models and how well they fit to the semiconductor domain. In this way we were able to find a suitable entrypoint for training our FA-BERT model.

3.1 Overview of BERT based Models

Since the development of BERT in 2018, many models that are based on BERT but fine-tuned for a specific task appeared. The following sections cover the most common known BERT models.

3.1.1 RoBERTa

RoBERTa is the name of an optimized BERT pretraining approach developed by Facebook and Washington University. It is entirely based on BERT and modifies some training methods and parameters. The name stands for Robustly Optimized BERT Pre-training Approach. For the pretraining, some methods and parameters of BERT were changed and adjusted. These changes and adjustments are essentially the following:

- Remove Next Sentence Prediction (NSP) from pretraining
- Replacement of static masking with dynamic masking
- Increasing the batch size of the sequences
- Increase in training iterations
- Use of significantly more training data

To increase the training data the developers used further text data with a total size of 160 gigabytes are used. This data includes the Common Crawl News dataset, which has 63 million articles and is 76 GB in size, the Open Web Text corpus, which is 38 GB in size, and stories from Common Crawl, which are 31 GB in size.
The optimization of the BERT pretraining with RoBERTa shows significant improvements compared to the original BERT training in various NLP benchmarks. In NLP benchmarks such as GLUE (General Language Understanding Evaluation), SQuAD (Stanford Question Answering Dataset) or RACE (ReAding Comprehension from Examinations), RoBERTa achieved top scores that had not been achieved before. For example, RoBERTa achieved a top position with an accuracy of 88.5 in the GLUE benchmark.

https://www.bigdata-insider.de/was-ist-roberta-a-1116137/

3.1.2 Med-BERT

https://arxiv.org/abs/2005.12833 Med-BERT is a BERT based model which got trained on the EHR (electronical health records) dataset. The EHR is containing data from 28.490.650 patients. Similarly to RoBERTa they also exlcuded Next Sentence Prediction from the training task ans used MLM only. As already described in Section 2.3.3, BERT based models usually have the additional tokens SEP and CLS in their input layer. The CLS token summarizes the information of the input int two sentences, but as the EH records are usually much longer than two sentences, the CLS token was not helpful. Consequently, the developers removed the CLS and SEP tokens from the input layer.
Figure 3.1 depicts the Med-BERT structure. As an input the model gets three types of embeddings, code, serialization and visit embeddigs. Code embeddings are representing the diagnosis code were serialization embeddings denote the relative order of each code for each visit. Visit embeddings are neccessary to distinguish between different visits.

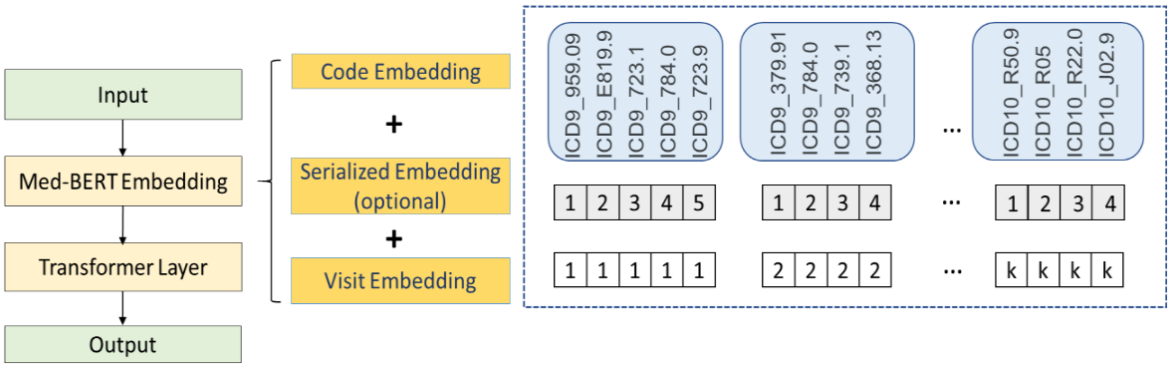

Fig. 3.1: Med-BERT structure

3.1.3 SciBERT

The development of BERT was a milestone in Natural language processing. Since the vocabulary of BERT is restricted, the application of the model in some domains does not lead to a satisfactory performance. To achieve this goal in different scientific fields, the model of BERT got adapted to SciBERT. The SciBERT model follows the same architecture as BERT but is instead pre-trained on scientific text. The corpus consists of 1,14M papers derived from *Google Scholar*, 18% papers from the computer science domain and 82% from the broad biomedical domain. With that the developers want to investigate the performance on downstream tasks when having and in-domain vocabulary. The authors used the full text of the papers instead of only the abstracts. The average paper length is 154 sentences which results in 2.769 tokens. The final corpus size has 3.17B tokens, similar to the 3.3B tokens on which BERT was trained [**?**].
As pretraining BERT for long sentences can be slow, the developers set a maximum sentence length of 128 tokens, and train the model until the training loss stops decreasing. After that, they continue training the model allowing sentence lengths up to 512 tokens. To obtain the best possible performing model, they tried to train different versions of SciBERT:

- Training the models cased and uncased

- Fine-tune the BERT- base model with original BaseVocab
- Training a BERT model from scratch with the constructed SciVocab.

The SciVocab was previously created out of the collected data corpus using the *SentencePiece* library. Similary to the BERT baseVocab, it is a *WordPiece* vocab. https://arxiv.org/abs/1903.10676 Although the training of the model with scientific texts sounds promising, the amount of 82% biomedical domain could be a drawback for our task.

3.1.4 S2ORC-SciBERT

The S2ORC-SciBERT is a SciBERT model which was further trained on the S2ORC dataset which will be presented in Section 4.1.2. It got published simultaneously with the dataset itself from the same developers. Compared to SciBERT it has a wider corpus including 20 different domains like Computer Science, Mathematical Science and Engineering which could be beneficial for the FA domain. However it was also trained on Medical and Biology data which again could be a drawback for the model as with SciBERT and Med-BERT. https://paperswithcode.com/paper/gorc-a-large-contextual-citation-graph-of

3.2 Choosing the Model Entrypoint

Since BERT models studied in this paper were trained on different text corpora, their tokenizers might provide different coverage of important terms used in the FA domain, such as the ones stored in FA ontology. To determine the coverage, we first analyze the tokenization of the ontology keywords with the three language models: traditional BERT, SciBERT and S2ORC-SciBERT. With this analysis it was possible to gain a first impression about the vocabulary of the models and how well they fit to the electrical domain.
The models got downloaded from the website Huggingface and can be load into the code as follows:

```
model = BertModel.from_pretrained("/path/to/model/")
tokenizer = BertTokenizer(vocab_file = '../path/to/tokeizer/vocab.txt')
```

To receive the keywords, we executed an existing function provided from our supervisor Christian Burmer. It executes a script with a request to the SparQL Database receiving the ontology keywords in a list. This keywords also occur in the reports as indicators for the content.
After receiving the keywords from the ontology and having a deeper look at them, we found some misspellings like in the word *flasover*, which misses a h to become *flashover*. These typos could confuse the models and lead to some mistokenization. To prevent this, we will first run some spell checking and correction methods on the keywords before further processing it. Therefore we tried different packages like the *Pyspellchecker* or *SymSpellPy* which are based on the Peter Novig's method. As both of the two packages did not work, we skimmed the keywords manually and corrected misspellings.
With the correct spelled keywords we started the experiment of tokenizing them with the three different BERT models. The code for tokenization was the same for every model:

```
# Run tokenizer.
tokenized_word = s2orc_scibert_tokenizer.tokenize(word)
```

```
#differentiate good and badly tokenized words
if len(tokenized_word) == 2:
        good_tokenized_words_s2orc += 1
        good_words.extend(tokenized_word)
elif len(tokenized_word_c) > 2:
        badly_tokenized_words_s2orc += 1
        bad_words.extend(tokenized_word)
```

We created a loop which hands every word over to the tokenizers and also analyzes if the resulting word is included in the models vocabulary or not. A word which is already in the vocabulary will not be divided into subtokens and so consists of only one token. If it is not included in the vocabulary, we determined if the word was good or badly tokenized. A word got considered as badly tokenized if it was split in more than two tokens. Otherwise the tokenization got considered as good. At the end of the execution, we created a dataframe depicting how the individual words got splitted into tokens. An example of the dataframe is depicted in Figure 3.2. Here the keyword "lifted clip" would get considered as badly tokenized word and the keyword "requestor" as good.

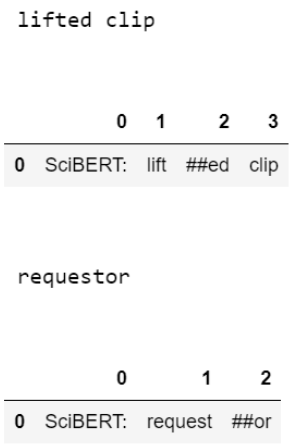

Fig. 3.2: Example of tokenization with SciBERT

Next we analyzed the word stems of the failure analysis reports and again examined the tokenization of the words with the language models. We did not analyze the word stems of the whole reports, but only the word stems of the most frequently used words. To obtain the frequency distribution and the word stems of the reports, we used the NLTK (Natural Language Toolkit) library. It includes the so called *SnowballStemmer*, because according to the documentation, this stemmer works better for English text. Stemmers remove morphological affixes from words, leaving only the word stem. To run the tokenization experiment, we also had to lower case the word stems and transform plural words into singles. The code can be seen in the code below. For the tokenization itself, the same code has been used as already explained above.

```
#create a new stemmer
stemmer = SnowballStemmer("english")

#test the stammer on pluralized words
```

```
most_common_stems = nltk.word_tokenize(str(most_common))
singles = [stemmer.stem(plural) for plural in most_common_stems]
word_stems = set(singles)
print(' '.join(singles))
```

As a result of the tokenization of the keywords it can be seen that SciBERT and S2ORC-SciBERt both perform better on the keywords as the traditional BERT model. Both models contain a few more keywords in their vocabulary than the BERT model. The resulting statistic can be seen in Figure 3.3. We expected this result because the training data of these two models fits better to our electrical domain than the data BERT was trained with which is more widespread.

```
#Keywords:  362
#Words in BERT vocabulary: 166
#Words in SciBERT vocabulary: 169
#Words in S2ORC SciBERT vocabulary: 178
```

Fig. 3.3: Comparison of the performance of the three models on the keywords

For the tokenization of the FA reports and their word stems we assumed to see at least a slight difference between SciBERT and S2ORC-SciBERT but without fine-tuning the models it is not possible to determine a better model which can be seen on the results of the tokenization analysis in Figure 3.4.

```
SciBERT:
In vocabulary:  39
Good tokenized words:  7
Badly tokenized words:  1
S2ORC-SciBERT:
In vocabulary:  39
Good tokenized words:  7
Badly tokenized words:  1
```

Fig. 3.4: Comparison of the performance on the word stems

In order to finally decide between SciBERT and S2ORC-SciBERT, a further analysis was performed in which the keywords were assigned a weighting based on their occurrence frequency. In that way the keywords get weighted according to their importance. Words that appear very often in the reports must be well recognized and tokenized by the model. Therefore, using the frequency and the number of subtokens, a weight is calculated as follows:

$$n = |t_1, t_2, ...t_n| \tag{3.1}$$

$$\rightarrow w = n \cdot sx_m \tag{3.2}$$

were n is the number of tokens a keywords got split into and w is the calculated wheight a keywords gets assigned.
The resulting weights got stored as tuples containing the keyword and its assigned wheight:

```
[('kw1', 0.002654), ('kw2', 0.062375), ('kw3', 0.12209),...]
```

After this weighting we again performed a tokenization and examined the good and badly tokenized keywords, but both models performed again very similar. SciBERT tokenized 129 keywords good, S2ORC-SciBERT 120. Because the FA reports contain out of more than only keywords, we decided to choose the S2ORC-SciBERT as an entrypoint for our further experiments. It has more electrical and physical papers in its training corpus than the original SciBERT and therefore might work better for us.

4 Data Collection

In order to train a language model, a sufficiently large amount of data is needed. Since the model presented in this paper will be used for textual classification, the data must be available in written form. Most of the data used for the report classification come from the FA laboratory. The laboratory has two sources to save information: an ontology which is the formalization of FA knowledge in a computer-friendly format, and the FA reports which comprise information that describes all diagnostic steps, results, and observations for a job analysis. These two resources would have a large enough amount of data to train the model. However, experiments with previous models indicated that these documents are too specific to serve as the sole data source for a model in the semiconductor domain. The content of the documents in the ontology and the FA reports contains too specific vocabulary and too little general vocabulary about electronics and semiconductors. Models which were trained only with this data did not have sufficient performance. For this reason, additional sources were sought to train the model more extensively. This chapter starts by introducing the inddividual datasets and how they got collected before covering the data cleaning process to prepare it as an input for the PikaBERT model.

4.1 Data Sources

The remaining dataset was created out of different data sources which got categorized in four datasets for an easier use in later experiments. The four datasets are: the *FA ontology and reports*, the *S2ORC Dataset*, the *Infineon Dataset* and the *Additional Dataset*. The following chapters introduce every dataset with its content, size and chatateristics.

4.1.1 Failure Analysis Ontology and Reports

The FA laboratory stores most of its knowledge as free texts, thus they might be ambiguous and cannot be processed by the software automatically. To avoid these issues, standard definitions of FA concepts used in the domain are required. Moreover, these definitions must be stored in a way that they can be used by both engineers and software tools alike. One possible solution to this challenge is to formalize the knowledge about the FA domain as an ontology, a knowledge base specifically designed to store terminological definitions. This is the case for the FA laboratory, where an ontology is being developed, currently storing hundreds of failures, tasks, tools, etc. The structure of an ontology includes classes, instances of these classes and properties. Individuals are descriptions of real-world entities, like sample integrated circuits of a job or tools available in a lab. Classes are defining parts of the world by summarizing properties of a collection of individuals.

For training the classification models, we have considered the historical data of the FA laboratory. These reports contain a series of fields, including the job identification number, cus-tomer comment regarding the issues found in devices, an anal-ysis report describing applied methods, found physical defects, their locations, electrical characterization, and other details. Some of this data is structured and can be retrieved from corre-sponding databases. However, the essential parts relevant to the fault identification process are described only in textual form and, therefore, cannot be processed automatically. Consider three samples taken from fault analysis reports of an FA Laboratory presented in Error! Reference source not found.. The examples show selected sentences describing the findings of an engineer and the labels indicating the physical faults and their electrical signatures. Both types of labels are organized in an ontology, which is a hierar-chical structure rep-resenting a taxonomy of faults. This ontology allows for the de-velopment of software tools helping to label reports manually and working with the classification results. Along with the job summary, the reports contain a series of fields including, an identification number of the device, lists of tasks performed on the device, the images obtained by certain tools such as X-Ray,the description of such images attached, etc. A subset of the reports also includesfault labels, both of the electrical signature of the faulty device and of the final physical failure.For the classification phase, these two faults are considered the target output while the job summary and image descriptions will constitute the input.

4.1.2 S2ORC Dataset

The S2ORC Dataset was collected by the Allen Institute of Artificial Intelligence and it has a corpus of 81.1M English-language academic papers from different domains. The texts got extracted from PDFs including abstracts and inline mentions of citations. The selection of subject areas is extensive and ranges from medicine and biology to geography as depicted in Figure 4.1.

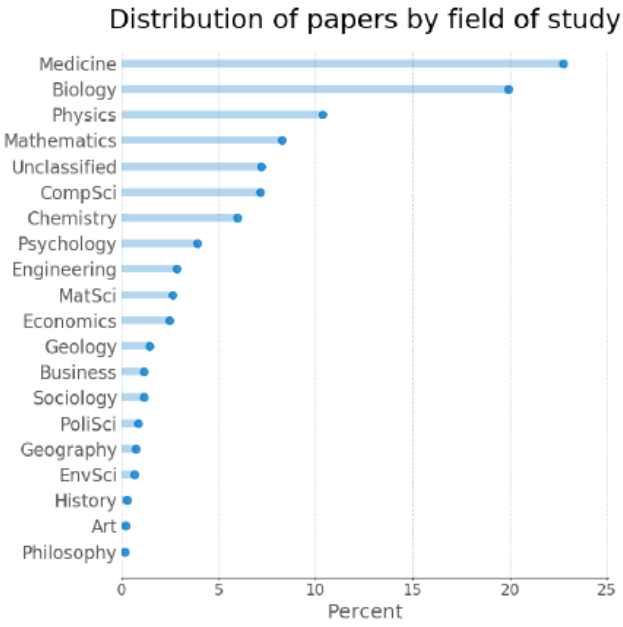

Fig. 4.1: Different Domains in the S2ORC Dataset.

Around 40% of the papers belong to the biomedical field, which is counterproductive for training a model in semiconductor domain. The vocabulary in the medical field partly overlaps with that in electronics, but has different meanings. A good example of this are the words neuron and electron, which are used in medicine in the field of neurology and whose semantics are related to the human brain. In electronics, this means negatively or neutrally charged particles and is related to current and voltage. For this reason, we decided to shrink the S2ORC data set and filter out only those papers that fit together in the subject area with electronics and semiconductor. https://paperswithcode.com/paper/gorc-a-large-contextual-citation-graph-of

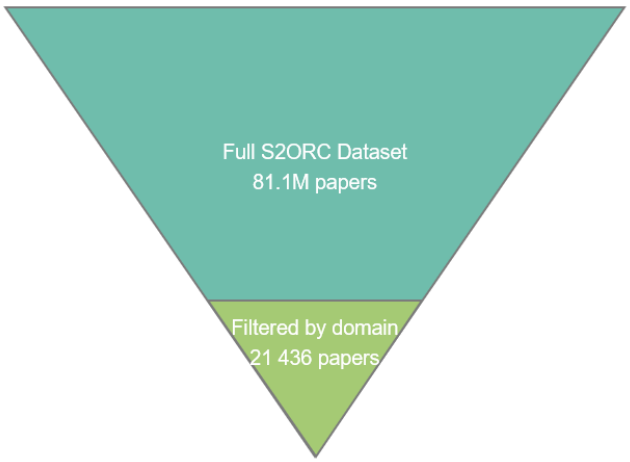

Fig. 4.2: Filtering of Domains in the S2ORC Dataset.

We decided to use the following domains:

- Physics
- Computer Science
- Chemistry
- Mathematical Science
- Engineering

As the selected domains form the minority of the s2orc datset, it shrank together from 81.1 Million papers to a size of about 21.000 papers as visible in Figure 4.2.

4.1.3 Infineon Dataset

In addition, we created an Infineon Dataset comprising rele-vant textual data available on the company's intranet. This dataset comprises 1.687 papers covering research topics and best-practices methods in the semiconductor domain. It was collected by employees and engineers of Infineon . Be-cause these papers got collected from engineers in different countries, we had to filter out non-English language papers. After filtering and extracting the raw text, we used approxi-mately 31.5MB of text data to create a dataset.

4.1.4 Additional Data

To increase the specificity of the training data for the FA domain, we also searched the Web for FA and electrical en-gineering papers. In this work we were using open search engines, like FreeFullPDF and GoogleScholar , as well as specific sources, like IEEE, to collect the data. The text was extracted from the papers and converted into a text represen-tation. The resulting dataset comprises 12.31MB of raw text, where 2.33MB got collected from FreeFullPDF and Google-Scholar and the other 9.98MB from IEEE.

After finishing collecting papers we ended with four independent in domain datasets collected from four different sources. We decided to unite the two smallest datasets when using it for training, what will be covered in the following chapters.

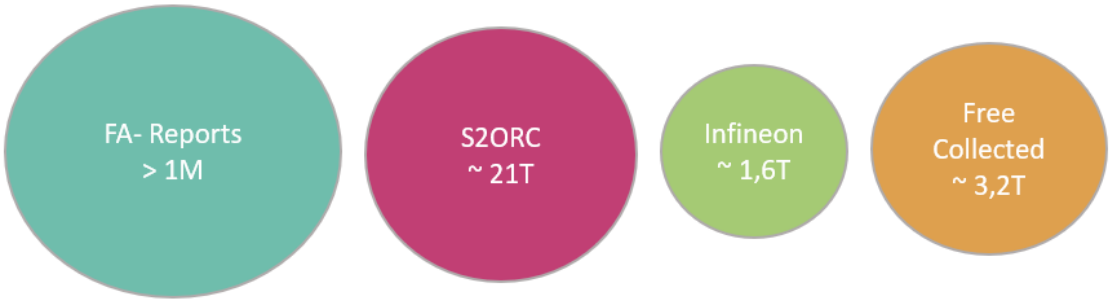

Fig. 4.3: Overview of all collected datasets.

4.2 Data Preprocessing

For later usage during training, we combined the Infineon and the Additional Datasets in one. Moreover, since the biggest part of the S2ORC Dataset was medical and biomedical data, we filtered this dataset by using the FA ontology keywords, retaining only documents that comprise at least one keyword. The resulting filtered dataset contains approximately 542MB of raw text. For later discussions, we introduce the following abbreviations for the datasets:

- **inf**: Infineon Dataset + Additional Dataset;
- **s2**: a part of the documents from S2ORC dataset fil-tered on keywords extracted from the ontology.

After finishing collecting the papers for the dataset, it was necessary to extract the raw text out of the documents. Therefore we got access to an text extractor API developed by Infineon collegues in Bangalore. The raw texts got extracted and UTF8 converted to be saved as excel sheets.
Not all of the extracted text is important or useful for the dataset. To remove unuseful data, a notebook was created were the text got prepared before being cleaned. Authors, metadata about the images, tables and other not contiguous text was remoed. The resulting text got again saved as excel sheets.

4.2.1 Data Cleaning

To use the FA reports for AI purposes, their content must be pre-processed and cleaned to erase all data that can disrupt the training process. One of the biggest issues regarding the FA reports is the wide variety of styles in which they are written, as they come from several laboratories around the globe, some of them dealing with specific purposes, while others deal with a broader range of failures. It is also essential to consider that although all they are written in English, this is not a standard language as engineers are only rarely native speakers.
The data cleaning process was the same for all datasets. A notebook was created were the excel sheets got loaded into dataframes. The methods used for cleaning the text are imported from the Natural Language Toolkit (nltk) package which implements functions for normalization, stemming and lemmatization. The following steps describe the procedure of text cleaning:

1. **Remove Punctuations and Special Characters**: Punctuatins and special characters got removed inbetween and at the end of every sentence by using regular expressions because a language model is not able to interpret those characters.
2. **Remove Numbers**: The input got checked if it is a number or not. Also all words which are not strictly all letters got removed to avoid confusing the model.
3. **Normalization**: After removing undesired words and characters, the entire text got normalized. This means that the resulting text contains only out of lowercased letters.
4. **Remove Stop-Words**: The nltk package contains a collection of english stop-words like "and", "or", "but", "for" etc. Due to their low entropy they are useless for the training and therefore removed from the text.
5. **Lemmatization**: Lemmatization considers the context of a word and converts it to its meaningful base form, which is called Lemma. For instance, lemmatizing the word "Caring" would return "Care". This has been done for the entire dataframe by iterating over the words.
6. **Remove Empty Lines**: Due to the previous cleaning some cells or lines of the dataframe may be empty now and represented as NaN values in the dataframe. These NaN values have to be replaced with spaces, otherwise they will cause errors during further processing.

4.2.2 Dataset Formatting

In comparison to traditional neural networks, BERT-based models are like a black box for the engineer. For training the model we used the so-called “transformer” class from Huggingface. The trainer method of the transformer class needs a model, a tokenizer and the training dataset as input. This input dataset needs to have a specific format and datatype otherwise the function would rise an error. To do so the previous created dataframes got converted into dataframes of two cells, an index and a text cell. An example is shown in Table 4.1.

	text
0	The following parameters are most often considered...
1	Nitrogen fertilization stands out as an important...

Tab. 4.1: Example of transformed dataframe shape

The base class *Dataset* from Huggingface implements a Dataset backed by an Apache Arrow table. This type of object can be created out of the dataframe just created with the following code:

Listing 1: Transformation into Dataset Dictionary

```
train_dataset = Dataset.from_dict(df)
datasets = datasets.DatasetDict({"train:" train_dataset})
```

The dataset object allows to specify a training and a test dataset. Since training the language model is an unsupervised learning method, only the training dataset is needed.

BERT based models can handle a block size of maximum 512 charakters as input. Previous training sessions, as described in Chapter 3, have shown that a block size should be chosen which is as similar as possible to that of the subsequent area of application. According to that we chose a block size of 128. With a map function the full dataset was tokenized into blocks of the defined block size. This process will get covered in the following section. The resulting dataset objects were saved and could be loaded later for training.

5 Experiments

In this section we will cover the process of preparing the collected dataset, described in section 4 and training different types of tokenizers and language models until we found the best performing one.

5.1 Tokenization of the dataset

In order to use the dataset created earlier in the chapter 4 for the BERT based model as training input, it must be transformed into a suitable format. Since we are using the Huggingface trainer, we also need to follow the Huggingface training data format.
In the first step, the entire text from the CSV file is converted into a dataset dictionary. This is possible with one line of code after importing the datasets package.

Listing 2: Creating the Dataset Dictionary

```
from datasets import Dataset
training_data = Dataset.from_dict(df)
datasets = datasets.DatasetDict({"train": train_data})
```

When using this method, training and test data can be stored together in one object and accessed via the datasets["train"] or datasets["test"] index. The resulting datasets["train"] dictionary has the feature "text" to access the data and 1.052.587 rows as visible in the output depicted in Figure 5.1.

```
Dataset({
    features: ['text'],
    num_rows: 1052587
})
```

Fig. 5.1: Format of Dataset Dictionary

The next step is to apply the tokenizer to the entire text. For this we use the map method from the datasets library. First we define a method shown in Listing 3 which calls the tokenizer on the text.

Listing 3: Tokenize Function to call on the text

```
def tokenize_function(examples):
return tokenizer(examples["text"])
```

The map function, shown in Listing 4, can be used to apply the tokenizer function from Listing 3 to the entire dataset. To speed up the process, we pass the textitnumber_poc=4 attribute, which enables multithreading and splits the process into four threads.

Listing 4: Applying the tokenize function on the Text

```
tokenized_datasets = datasets.map(tokenize_function, batched=True, num_proc=4,
        remove_columns=["text"])
```

The "text" collumn from the dictionary has now been replaced by the textitinput_ids which is needed by the modell as an input for training.

```
DatasetDict({
    train: Dataset({
        features: ['input_ids', 'token_type_ids', 'attention_mask'
        num_rows: 1052587
    })
})
```

Fig. 5.2: Text collumn has been replaced by calling tokenizer on the dataset.

After the dataset has been tokenized, the next step is to split it into individual chunks. The size of these chunks depends on the available resources and the maximum context size of the model. The context size can be determined with the method textitmodel_max_length. For BERT based models, like ours, it is 512 tokens. We need to concatenate all of our texts together and then split the result in small chunks of a certain textitchunk_size. To do this, we will use the textitmap method again, with the option textitbatched=True. This option actually lets us change the number of examples in the datasets by returning a different number of examples than we got. This way, we can create our new samples from a batch of examples.
First, we grab the maximum length our model was pretrained with. This might be a big too big to fit in our GPU RAM. In the first try we chose a chunk size of 64, which turned out not to be suitable because longer sentences were truncated, while shorter sentences were populated with [PAD] tokens (id: 0) until they reach the desired length. Using a small chunk size can be detrimental in real-world scenarios, so it is recommended to use a size that corresponds to the use case the model will be applied to. Because of that we chose a chunk size of 128.

Listing 5: Definition of the method group_text

```
chunk_size = 128

def group_texts(examples):
# Concatenate all texts
concatenated_examples = {k: sum(examples[k], []) for k in examples.keys()}
# Compute length of concatenated texts
total_length = len(concatenated_examples[list(examples.keys())[0]])
# We drop the last chunk if it's smaller than chunk_size
total_length = (total_length // chunk_size) * chunk_size
# Split by chunks of max_len
result = {
        k: [t[i : i + chunk_size] for i in range(0, total_length, chunk_size)]
```

```
        for k, t in concatenated_examples.items()
}
# Create a new labels column
result["labels"] = result["input_ids"].copy()
return result
```

5.2 Training of the Tokenizer

Also the tokenizer itself can be improved before using it in the training method together with the model. Training a tokenizer can be done in two ways:

1. Extending the vocabulary manually
2. Pre-training the tokenizer

Tokenizers of BERT-derived models usually have empty positions allowing us to add required tokens directly. Extend-ing the vocabulary of a tokenizer means that it won't get trained explicitely but important words like our ontology keywords can be added to the vocabulary of the tokenizer and therefore recognized as one single token by the model later. Pre-training the tokenizer means that the tokenizer will get trained on a prepared dataset without using the model. Again the engineer has no impact on the behavior of the tokenizer during the training loops. We created four types of trained tokenizers. For later discussions we introduce the following abbreviations:

- **ext**: A tokenizer with the ontology keywords added to its vocabulary manually
- **small**: A tokenizer trained on inf dataset
- **large**: A tokenizer trained on inf+s2 dataset;
- **flt**: A tokenizer trained on the s2 dataset

5.2.1 Extending the tokenizers vocabulary

For the first experiment we tried to enhance the language models performance by extending the tokenizers vocabulary. The tokens which we added to the vocabulary were the ontology keywords. As already discussed in the Section 4.2, some of the keywords included misspellings. We again removed or corrected these keywords to avoid to sabotage our model.
Before extending the tokenizer, it's vocabulary had a size of 31944. To add the keywords to he tokenizers vocabulary, we had to execute the following code:

Listing 6: Extending the Tokenizer Vocabulary

```
for keyword in keywords:
        tokenizer.add_tokens(keyword)
```

The keywords got added to the vocabulary and increased the vocabulary size to 32131. After saving the tokenizer it can be used in the training process like any other pretrained tokenizer.

5.2.2 Pre-Training the tokenizer

Training a tokenizer is very simple with the huggingface methods. To train a tokenizer we first need to create a training dataset again. This dataset should have the same format as already

discribed in chapter 4. For our experiments, we assembled three different datasets listed in 5.2.2 to train three tokenizers. These tokenizers were then to be trained together with the language model in different tests. The different data sets were used for testing purposes to find the best possible model.

1. full dataset containing **inf** and **s2**
2. smaler dataset including only **inf**
3. **s2** alone

To create the training corpora we used the same methods as already described in Section 4.2.2. Chunking the data into a block size of 128 was not neccessary.
After creating the training corpus, the tokenizer can be trained with one single line of code. The textittrain_new_from_iterator method needs the training corpus and the new vocabulary size as an input. We chose the highest possible vocabulary size, which is 52.000.

Listing 7: Training the Tokenizer and resize Vocabulary

```
tokenizer = old_tokenizer.train_new_from_iterator(training_corpus, 52000)
```

After training the tokenizer with the smaller training corpus, it tokenized the ontology keywords into 542 tokens instead of 661, which means an improvement in performance.

5.3 Training Process

After creating the training data as described in section 5.1, it is necessary to prepare some requirements for the training process. It also includes the process of how to chunk the training dataset into a specific block size which is needed by the model as an input. We adapted the code regarding to our needings.
The training itself contains out of the five steps:

1. Defining the model and tokenizer checkpoints
2. Chunking the dataset into blocks
3. Defining the training arguments
4. Defining the Huggingface Trainer
5. Call the train method

Defining the model and tokenizer checkpoints can be done in the same way as shown in the code **??**. The chunking of the dataset is described in section **??**. The training arguments are variables which are needed by the trainer object. Here different values like the learning rate, the name of the model and how to report the logging can be set.

Listing 8: Creating the Training Arguments

```
training_args = TrainingArguments(
"checkpoint_LM_s2_s2orc",
evaluation_strategy = "epoch",
learning_rate=2e-5,
weight_decay=0.01,
report_to="tensorboard,
```

```
logging_strategy="steps"
)
```

Listing 9: Creating the Trainer Object

```
trainer = Trainer(
model=model,
args=training_args,
train_dataset=lm_datasets["train"],
data_collator=data_collator,
)
```

The training can be started by calling the method *train()* from the transformers trainer object.

Listing 10: Starting the training

```
trainer.train()
```

5.3.1 Moving the bias towards FA domain

In the first training experiments it was our goal to move the bias of the pretrained S2ORC-SciBERT further towards the semiconductor and failure analysis domain. Therefore we use different datasets.

For later discussions and easier use we introduce the following abbreviations:

- **inf**: Infineon Dataset (FA-reports, FA-papers, IEEE papers, FreeFullPDF papers)
- **s2**: S2ORC dataset filtered on keywords

For the first few experiments we used the inf dataset together with the different trained and extended tokenizers explained in section 5.2. The inf dataset should be a good complement to the BERT based model which has already been pre-trained with the S2ORC dataset. With the help of the trained tokenizers, this training material should prepare for the classification of FA reports. We trained the models for 10 epochs and a learning rate of 2e-5 as described in Listing 9. In the descriptions of the following experiments we refer to the abbreviations described in section 5.2.2 and section 5.3.1.

Experiment 1:
For the first experiment we trained the S2ORC-SciBERT with the *inf* dataset and used the pre-trained tokenizer *small* for 10 epochs. We started with a loss of about 8. The loss curve falls steadily. A small hill can be seen in the area around 40k steps. This indicates that the model was not training optimally at this point and had to adjust its weights. After 10 epochs the loss reached a value of 3.471.

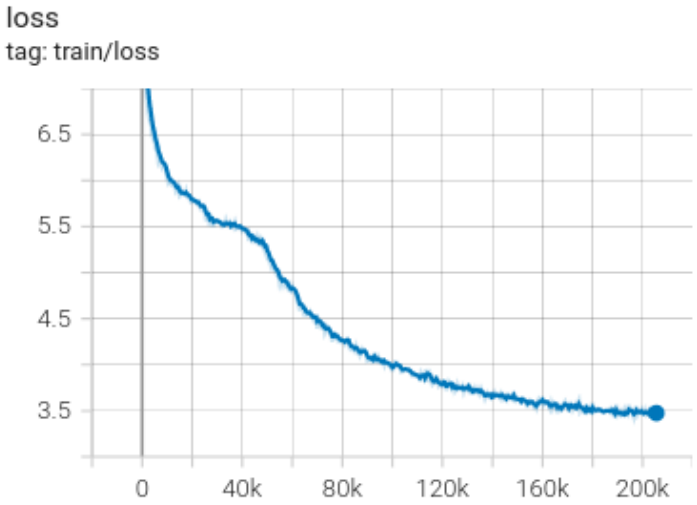

Fig. 5.3: Loss Function during Training with Infineon Dataset and small trained tokenizer.

Experiment 2:
In the second experiment we used the *inf* dataset again together with the pre-trained tokenizer *large* for 10 epochs. We started with a loss of about 8.5. The loss curve falls steeper at the beginning before it flattens out. Although the training seemed promising, int the end the loss reached again a value of 3.471 after 10 epochs.

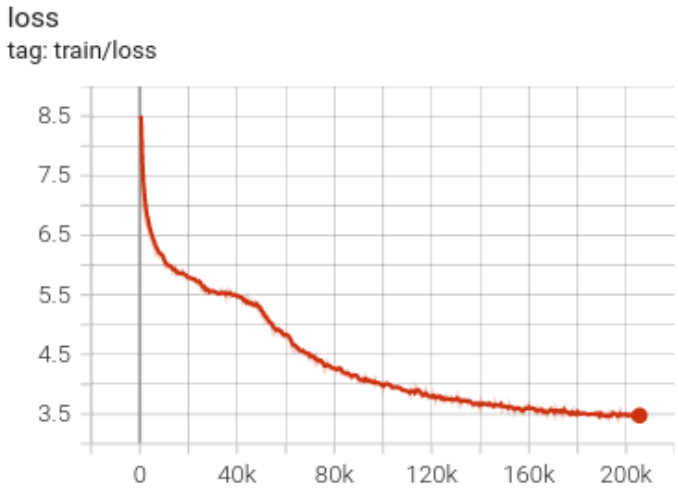

Fig. 5.4: Loss Function during Training with Infineon Dataset and large trained tokenizer.

Experiment 3:
To compare a pre-trained tokenizer with the extended tokenizer approach, we used the *ext* tokenizer together with the inf dataset in experiment three. The loss curve behaves similarly to that of experiment two. In the end it flattens out at a loss value of 3.475, which is slightly higher than in experiment one and two.

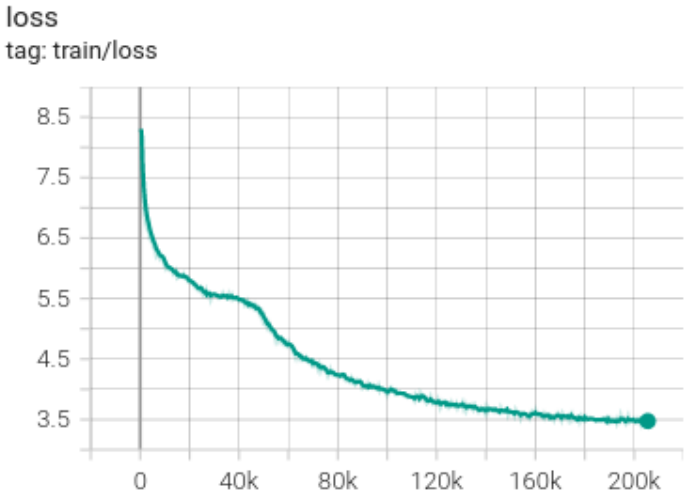

Fig. 5.5: Loss Function during Training with Infineon Dataset and extended tokenizer.

Experiment 4:
For experiment four we went for a bigger dataset, *inf* dataset combined with *s2* dataset, and trained it together with the tokenizer flt which is the one with the largest vocabulary. Surprisingly, the loss curve hardly changes at all and flattens out at a loss value of again 3.471 after 10 epochs.

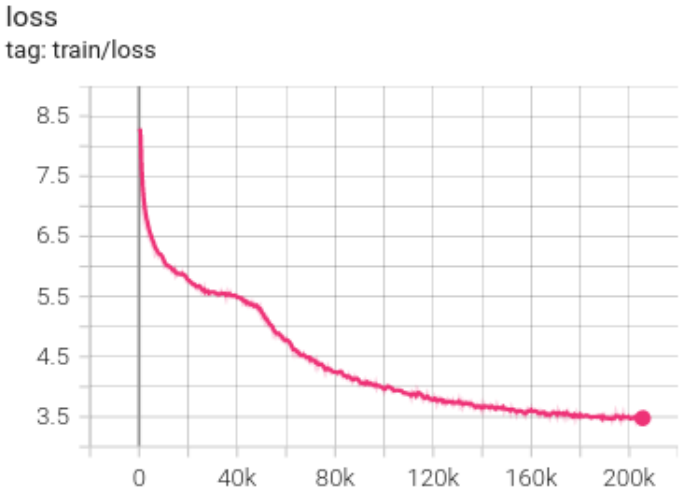

Fig. 5.6: Loss Function during Training with tokenizer trained on s2.

Experiment 5:
As for the last experiments the loss curve hardly changed, we decided to increase the number of training epochs to 15. We sticked with the combination fo the datasets *inf* and *s2* and used the tokenizer *large* as it showed the steepest loss curve at the beginning. The additional 5 epochs showed an improvement compared to the previous training sessions, the loss courve flattens out at a value of 3.088.

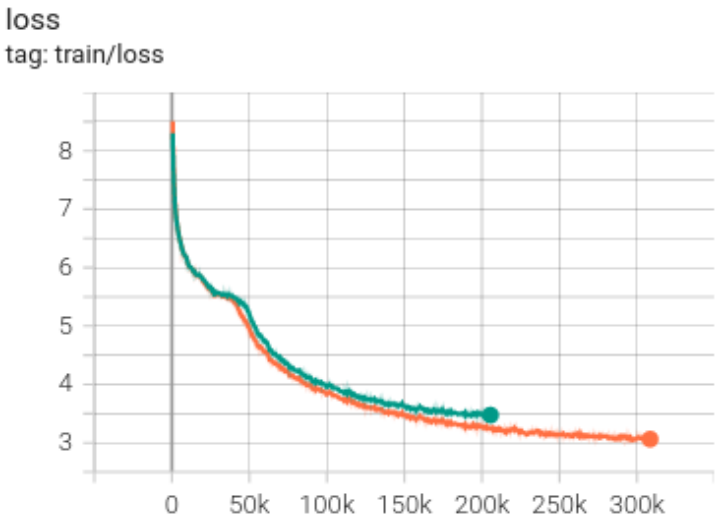

Fig. 5.7: Loss Function during Training with largest dataset and tokenizer trained on large.

The various loss functions hardly differ from each other. On the basis of the last curve, which was trained for 15 epochs, it can be seen that the longer training could certainly lead to a better performance. Nevertheless, the loss function already flattens out too much in this area. For this reason, further measures regarding the improvement of the training were researched.

5.3.2 Trying different types of losses

Another attempt to improve the performance of the model is to implement a different loss function. Cross entropy loss seemed to be particularly suitable. In the case of masked language modeling, this metric compares the token predicted by the model with the ground truth and assigns a value between 0 and 1 to the prediction. 0 stands for a perfect prediction. During training, the values are aggregated to represent the loss function.
The loss function used by the Trainer object from Huggingface Transformers is shown in Listing 11.

Listing 11: Trainer compute loss function

```
def compute_loss(self, model, inputs, return_outputs=False):
        if self.label_smoother is not None and "labels" in inputs:
                labels = inputs.pop("labels")
        else:
        labels = None
        outputs = model(**inputs)
        # Save past state if it exists
        if self.args.past_index >= 0:
                self._past = outputs[self.args.past_index]

        if labels is not None:
                loss = self.label_smoother(outputs, labels)
        else:
        # We don't use .loss here since the model may return tuples
```

```
        # instead of ModelOutput.
        loss = outputs["loss"] if isinstance(outputs, dict) else outputs[0]

        return (loss, outputs) if return_outputs else loss
```

quelle: https://github.com/huggingface/transformers/blob/v4.17.0/src/transformers/trainer.pyL2006

which means that, by default, the model itself is responsible for computing some sort of loss and returning it to outputs. To implement an own loss function, we had to overwrite the Trainer class and the loss function and defining a cross entropy loss function to return it to outputs. The code is shown in listing 12. We use the cross entropy loss function from *torch.nn* here. In most cases the *CrossEntropyLoss()* function gets some weights as inputs, but because in our case pretraining a BERt model is an unsupervised learning task, we have no weights to assign and leave it empty.

Listing 12: Cross Entropy Loss

```
class MyTrainer(Trainer):
        def compute_loss(self, model, inputs, return_outputs = False):
                labels = inputs.pop("labels")
                outputs = model(**inputs)
                logits = outputs.get("logits")

                loss_fct = nn.CrossEntropyLoss()
                loss = loss_fct(logits.view(-1, tokenizer.vocab_size),
                                labels.view(-1))
                return (loss, outputs) if return_outputs else loss
```

5.3.3 Changing the Masking Method

The original masking method which is used by all BERT based models is that every token in an input blocked gets masked by the probability of 15%. hier fehlt noch was

Based on the latest experiments, it can be seen that our models are not sufficiently well trained in the semiconductor area. Another attempt to improve the performance is to adjust the masking strategy during the training. After some research we came across the Whole Word Masking strategy. With this method, not only some tokens are randomly masked, but it is also checked whether the token to be masked is a standalone token or a subtoken. If a subtoken, which is part of a word, should get masked, the algorithm also masks all the surrounding tokens that belong to the word. In addition we increased the masking probability from 15% to 20%. The code is visible in Listing 13. Therefore we created a new data collator class which has the whole word masking method implemented and handed this data collator to the trainer shown in Listing 14.

nicht vergessen tokenizer map funktion geändert damit word ids im dataset

```
DatasetDict({
    train: Dataset({
        features: ['input_ids', 'token_type_ids', 'attention_mask', 'word_ids'],
        num_rows: 1052587
    })
})
```

Fig. 5.8: Loss Function during Training with Infineon Dataset and large trained tokenizer.

Listing 13: Data Collator implementing Whole Word Masking

```
wwm_probability = 0.2

def whole_word_masking_data_collator(features):
        for feature in features:
        word_ids = feature.pop("word_ids")

        # Create a map between words and corresponding token indices
        mapping = collections.defaultdict(list)
        current_word_index = -1
        current_word = None
        for idx, word_id in enumerate(word_ids):
        if word_id is not None:
        if word_id != current_word:
        current_word = word_id
        current_word_index += 1
        mapping[current_word_index].append(idx)

        # Randomly mask words
        mask = np.random.binomial(1, wwm_probability, (len(mapping),))
        input_ids = feature["input_ids"]
        labels = feature["labels"]
        new_labels = [-100] * len(labels)
        for word_id in np.where(mask)[0]:
        word_id = word_id.item()
        for idx in mapping[word_id]:
        new_labels[idx] = labels[idx]
        input_ids[idx] = tokenizer.mask_token_id
        feature["labels"] = new_labels

        return torch_default_data_collator(features)
```

Figure 5.9 shows an example of how the whole word masking method works. The green examples shows how the algorithm is masking all surrounding subtokens which belong to a word were the blue example show some random picked tokens to mask which do not belong to any other word.

```
'>>> ['[CLS]', 'content', 'downloaded', 'i', '##ops', '##ci', '##ence', 'scr', '##oll', 'text', '[SEP]', '[CLS]',
'download', '[MASK]', 'ip', 'address', '137', '222', '24', '##9', '19', 'content', '[MASK]', '26', '02', '2015',
'19', '34', 'note', 'terms', '[MASK]', 'apply', '[SEP]', '[CLS]', 'high', 'performance', 'microm', '##achi', '##n
ed', 'ga', '##n', 'si', 'high', 'electron', 'mobility', 'transistor', 'back', '##side', 'diamond', '##like', 'car
bon', 'titanium', 'heat', 'dissipation', 'layer', 'view', 'table', 'contents', 'issue', 'journal', 'home', '##pa
', '##ge', '[MASK]', 'appl', 'phys', '[MASK]', '8', '011', '##00', '##1', '[MASK]', 'i', '##ops', '##ci', '##ence
', 'i', '##op', 'org', '188', '##2', '07', '##86', '8', '1', '011', '##00', '##1', 'home', '[MASK]', '[MASK]',
'[MASK]', 'contact', 'i', '##ops', '##ci', '##ence', '[SEP]', '[CLS]', '[MASK]', 'performance', 'microm', '##achi
', '##ned', 'ga', '##n', 'si', 'high', 'electron', 'mobility', 'transistor', 'back', '##side', 'diamond', '##like
', 'carbon', 'titanium', 'heat', 'dissipation', '[MASK]', '[MASK]', '[MASK]', '[MASK]', 'chi', '##u', '##1', '*',
'chi']'
```

Fig. 5.9: Example of Whole Word Masking Method.

Listing 14: Trainer

```
trainer = MyTrainer(
model=model.to(device),
args=training_args,
train_dataset=lm_datasets['train'],
data_collator=whole_word_masking_data_collator,
tokenizer=tokenizer
)
```

We trained the model with the new masking method again for 20 epochs and the same learning rate and weight decay as in the other experiments. The results show a final loss of 2.51, which is the lowest loss in all experiments so far. The loss curve is depicted in Figure 5.10. This model could allow a good performance for the FA classifier after further training epochs.

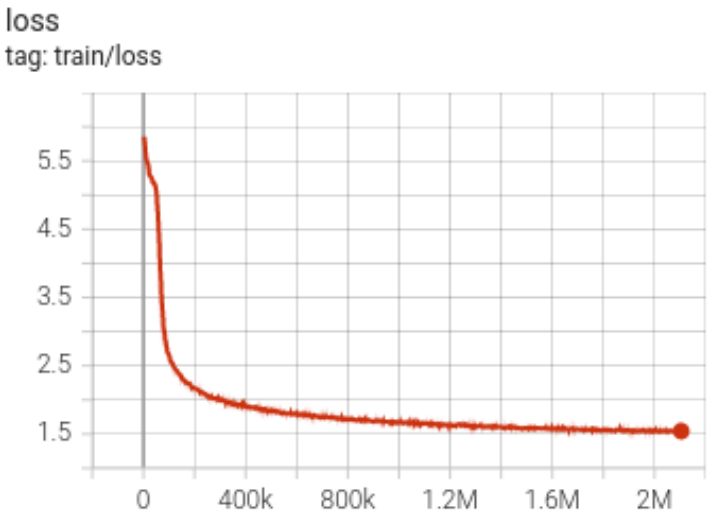

Fig. 5.10: Loss Function during Training with Whole Word Masking.

6 Results

6.1 Performance Measures

6.2 Discussion of Results

Inhalt...

7 Conclusion

From the previous experiments, we can conclude that a BERT-based model was a good choice for the FA model due to the broadly pre-trained training corpus and ease of use. It is also easy to apply and incorporate for the following classification. Since no suitable models exist in the semiconductor domain yet and most of the BERT models were trained with mainly medical and biomedical papers, it was cumbersome to find a suitable model entry point since the medical vocabulary can lead to a degradation of performance on semiconductor texts.

7.1 Future Work

Bibliography

[Ashb00] J. D. M. Ashbourn: Biometrics : Advanced Identify Verification: The Complete Guide. Springer Verlag (2000).

[BeKP91] A. Beutelspacher, A. G. Kersten, A. Pfau: Chipkarten als Sicherheitswerkzeug. Springer Verlag, Berlin (1991).

[BiSh91] E. Biham, A. Shamir: Differential Cryptanalysis of DES-like Cryptosystems. *In: CRYPTO '90: Proceedings of the 10th Annual International Cryptology Conference on Advances in Cryptology*, Springer-Verlag, London, UK (1991), 2–21.

[Breu84] R. Breuer: Computer-Schutz durch Sicherung und Versicherung. Karamanolis-Verlag (1984).

[Chen00] Z. Chen: Java Card Technology for Smart Cards: Architecture and Programmer's Guide (The Java Series). Addison-Wesley (2000).

[DaRi00] J. Daemen, V. Rijmen: The Block Cipher Rijndael. *In: CARDIS '98: Proceedings of the The International Conference on Smart Card Research and Applications*, Springer-Verlag, London, UK (2000), 277–284.

[JoMV01] D. Johnson, A. Menezes, S. A. Vanstone: The Elliptic Curve Digital Signature Algorithm (ECDSA). In: *International Journal on Information Security*, 1, 1 (2001), 36–63.

[MeVO96] A. J. Menezes, S. A. Vanstone, P. C. V. Oorschot: Handbook of Applied Cryptography. CRC Press, Inc., Boca Raton, FL, USA (1996).

[NIST77] NIST: Data Encryption Standard, *Federal information processing standards publication*, Bd. 46. National Institute for Standards and Technology (1977).

[NIST00] NIST: FIPS PUB 186-2 Digital Signature Standard (DSS). National Institute for Standards and Technology, Gaithersburg, MD, USA (2000), `http://www.itl.nist.gov/fipspubs/fip186-2.pdf`.

[RiSA78] R. L. Rivest, A. Shamir, L. M. Adleman: A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. In: *Communications of the ACM*, 21, 2 (1978), 120–126.