

Institute of Software Technology

University of Stuttgart  
Universitätsstraße 38  
D-70569 Stuttgart

Bachelorarbeit

# **Deep Learning in Streamlining the Conversion of Natural Language Requirements into Template-Based Architecture**

Max Buchholz

**Course of Study:** Software Engineering

**Examiner:** Prof. Dr. Stefan Wagner

**Supervisor:** Mohammad Kasra Habib, M.Sc.

**Commenced:** Juni 15, 2023

**Completed:** December 15, 2023



## **Kurzfassung**

In dieser Arbeit wird evaluiert, wie die kürzlichen Entwicklungen im Bereich des Machine Learning, insbesondere die Entwicklung von Large Language Models dazu genutzt werden kann, bei der Konvertierung von natürlichsprachlichen Anforderungen, in eine vorlagenbasierte Struktur unterstützen können. Es werden verschiedene Ansätze implementiert, getestet und evaluiert, sowie ein existierender Datensatz um Anforderungen in umstrukturierter Form ergänzt. Die untersuchten Ansätze erweisen sich als potenzielle Ansätze für zukünftige Arbeiten und sind in der Lage die Aufgabe im Ansatz zu bewältigen. Daher kann eine auf maschinellem Lernen basierende Umwandlung von Anforderungsstatements den Prozess der Umwandlung vorhandener Statements in eine normierte Struktur verbessern.

## **Abstract**

This thesis evaluates how recent development in Machine Learning, especially Large Language Models could be used to help with the conversion of natural language based requirement statements into a template-based structure. To evaluate different approaches, they have been implemented, tested and evaluated, on an existing dataset, complemented by converted requirement statements. It seems that the tested approaches are feasible to be used and further investigated, as they were already able to provide a decent performance. Therefore, machine learning based conversion of requirement statements can improve the process of converting existing statements into a normed structure.



# Contents

<b>1</b>	<b>Introduction</b>	<b>13</b>
1.1	Motivation . . . . .	13
1.2	Problem Statement . . . . .	13
1.3	Structure . . . . .	14
<b>2</b>	<b>Background and Fundamentals</b>	<b>15</b>
2.1	Different Types of Requirements . . . . .	15
2.2	Motivation for template-based requirement statements . . . . .	15
2.3	Template based structures for requirement statements . . . . .	17
2.4	Measuring the quality of requirement statements . . . . .	19
2.5	Large Language Models . . . . .	20
<b>3</b>	<b>Related Work</b>	<b>23</b>
<b>4</b>	<b>Study Design</b>	<b>25</b>
4.1	Research Questions . . . . .	25
4.2	Methodology . . . . .	25
<b>5</b>	<b>Results</b>	<b>29</b>
5.1	Zero-Shot Learning . . . . .	29
5.2	Fine-tuning with converted requirements . . . . .	31
5.3	Fine-tuning with knowledge sentences . . . . .	32
<b>6</b>	<b>Evaluation and Discussion</b>	<b>35</b>
6.1	Zero-Shot Learning . . . . .	35
6.2	Fine-tuning with converted requirements . . . . .	35
6.3	Fine-tuning with knowledge sentences . . . . .	36
6.4	Comparison of the approaches . . . . .	36
6.5	Threads to validity . . . . .	37
<b>7</b>	<b>Conclusion and Outlook</b>	<b>39</b>
<b>8</b>	<b>Acknowledgement</b>	<b>41</b>
	<b>Bibliography</b>	<b>43</b>
<b>A</b>	<b>Implementation of the Jupyter Notebook</b>	<b>45</b>



## List of Figures

2.1	The core of a requirement and its legal obligation as proposed in [Poh10] . . . . .	18
2.2	Principle of a complete requirement template without conditions as proposed in [Poh10] . . . . .	18
2.3	The complete requirements template with conditions as proposed in [Poh10] . . . . .	18
5.1	bilingual evaluation understudy (BLEU) scores of the Zero-Shot approach . . . . .	30
5.2	BLEU scores of the Fine-Tune approach with converted requirements . . . . .	31
5.3	BLEU scores of the Fine-Tune approach with knowledge sentences . . . . .	32





## List of Tables

5.1	Zero-Shot results . . . . .	31
5.2	Fine-Tune with converted statements . . . . .	32
5.3	Fine-Tune with describing sentences results . . . . .	33
6.1	Final results . . . . .	36



# Acronyms

**BLEU** bilingual evaluation understudy. 7

**CMMI** Capability Maturity Model Integration. 15

**DL** Deep Learning. 20

**LLaMA** Large Language Model Meta AI. 23

**LLM** Large Language Model. 20

**NLP** Natural Language Processing. 14

**RAG** Retrieval Augmented Generation. 20

**RE** Requirements Engineering. 13

**ROUGE** Recall-Oriented Understudy for Gisting Evaluation. 37

**SPICE** Software Process Improvement and Capability Determination. 15



# 1 Introduction

## 1.1 Motivation

Software applications become more complex, bigger and involve more stakeholders in the software development process. To take into account their different needs, it is necessary to make sure, that the requirements of different stakeholders are discussed, evaluated and especially well-formulated. This is part of the Requirements Engineering (RE) task, which plays a critical role in the software development process. Errors in this phase can have significant cost and effort implications, as it is very hard to fix them if they are detected later in the process. Therefore, there is a high interest in avoiding issues, errors or ambiguities. Effective communication and documentation are crucial for involving diverse stakeholders, which is often achieved through the use of natural language. However, the use of natural language poses potential risks associated with language effects. It allows to differently interpret certain parts of a requirement statement or might not specify all relevant parts of a requirement.

Requirements engineering templates help requirements engineers to keep the according statements in a common form and help to improve on different beneficial properties. Depending on the type of requirement, an according template can be used to assist at formulating requirements in an unambiguous way. However, it is also a lot of work to create the requirement statements in a certain template based structure. Usually requirements are collected and extracted from different artifacts like documentation of existing systems, interviews or surveys with stakeholders. This indicates a high potential cost to create the requirement statements in a given template structure. Tools that can help to improve the accuracy or efficiency of this process, can save costs and time of requirements engineers. By automating parts of the process or proposing automated conversions, a requirements engineer can use these as a base to speed up the task of converting statements from general natural language to template-based structures.

## 1.2 Problem Statement

The use of natural language poses potential risks associated with so-called language effects. To mitigate these risks, templates have been introduced to assist in formulating high-quality requirements efficiently. Nonetheless, the challenge remains in converting unstructured requirement statements expressed in natural language into template-based structures. This bachelor thesis proposes leveraging Deep Learning techniques to automate this conversion process, offering potential benefits in terms of efficiency and quality based on templates, like the one proposed by Klaus Pohl et al. [Poh10], where different parts of a requirement statement have been specified.

Furthermore, this research will involve the development and training of a Deep Learning-based model to convert natural language-based requirement statements into the predefined template format. Therefore, the capabilities of Large Language Models in transforming requirement statements will be explored. Additionally, the potential of training the model with knowledge about RE provided in natural language will be examined. Finally, the model is going to be trained using a dataset of requirement statements expressed in natural language, along with corresponding template-based representations. Evaluation metrics will be evaluated to assess the performance of the model in terms of accuracy, efficiency, and quality of the converted requirement statements.

### **1.3 Structure**

In Chapter 2, we take a look at different topics which are used as a base for this thesis. Furthermore, Chapter 3 evaluates existing work of Natural Language Processing (NLP) and RE with the focus on Requirements statements. Chapter 4 elaborates on the study design of this thesis including the research questions, methodologies and challenges that came up. Afterward, Chapter 5 presents the results of the used approach which is then discussed and evaluated in Chapter 6. To conclude this paper Chapter 7 will summarize the most important points and gives an outlook on possible future work.

## 2 Background and Fundamentals

### 2.1 Different Types of Requirements

Requirement statements can have different types, depending on what they describe [Poh10]. In general one can differentiate three different types of requirement statements: The first type are functional requirements, which describe a behavior of the system or a provided feature. Functional requirements can further be divided into data requirements, behavioral requirements and functional requirements. These categories are created based on the perspective on the system: While in the data perspective the focus is for example on the structure of input and output, the functional perspective observes the functionality of a system. The behavioral perspective describes how the software is embedded into the system context. A further type are quality requirements, which define certain qualities of the software, that are not covered by functional requirements. This often highly influences the architecture of the system. Typical topics of quality requirements are scalability, portability or performance. Quality requirements are also named non-functional requirements. Furthermore, there is a third category of requirements, which are called constraints. Constraints allow expressing limitations to the implementation to the system, that are neither of functional nature nor concern the quality of the feature. These can be external factors like business goals. Constraints cannot be implemented directly, but rather one can comply with constraints when implementing a system.

In practice, additional categorization can be used, for example those proposed by Capability Maturity Model Integration (CMMI) or Software Process Improvement and Capability Determination (SPICE). Another option which is often used in practice is categorization based on additional attributes that are linked to a certain requirement statement. One example would be the degree of legal obligation, to further classify requirements based on additional criteria.

### 2.2 Motivation for template-based requirement statements

If one managed to elicit requirements, these requirements are often documented using natural language. This has a huge advantage: In RE, plenty of different stakeholders are involved. Therefore, natural language allows all stakeholders to participate as it can be read and understood by different stakeholders [Poh10]. Additionally, natural language is flexible enough, to describe different circumstances, no matter how rare or special they are. However, the use of natural language can have several side effects.

One of the most obvious issues with natural language, is the subjectivity of it. Requirement statements formulated in natural language, can be subjectively interpreted, depending on the background of a stakeholder. This can lead to potential ambiguities in such statements, which can result in misunderstandings. When these misconceptions are not detected and resolved during RE, they might appear later in the software development process, leading to potentially costly fixes.

These ambiguities can have different characteristics. One type of effect, that can appear when different stakeholders read and understand a requirement, is Nominalization. In this case, a longer, bigger or more complex process is condensed into a single noun. Therefore, a lot of information to describe the full process is lost, and every reader constructs their own interpretation of the process. Two solutions are available to avoid the effect of Nominalization: Either the requirements engineer should create a definition of the used term, where all necessary details are described. This way individual requirements can then reference the process by the term. Alternatively, the term should be replaced by a more detailed description of the process or event that is referred to. As a criterion one can check, that the details of mentioned process is clear to all stakeholders. Otherwise, it is necessary to further mitigate ambiguities.

Another possible effect, which might appear in requirement statements frequently, are nouns without reference index. As with the aforementioned nominalized processes, it should always be completely specified. In linguistics this is called a missing index of reference. Frequent examples are *data* or *user*. Here the question arises, which user or data is meant? This ambiguity can be resolved by changing the reference.

One further issue is the use of universal quantifiers, which declare amount and frequency. They state something about a group of objects. When universal quantifiers are used, it might happen, that certain objects in this group do not adhere to the stated property or behavior. This can lead to stakeholders grouping special cases and exceptions into one group, by using universal identifiers. Other side effects are misunderstandings in implementation, as the exceptions or edge cases are not stated in the requirement statements, but rather mentioned as part of a bigger group. To avoid this language effect, the requirement can be skimmed for trigger words like *never* or *always*. If those words are found, a check is needed, whether the requirement truly should be meant for all/none of the objects in a specified group.

An additional natural language effect is an incompletely specified condition, where the case that a condition is not fulfilled, is not specified. This way handling these cases is not further defined, and the information is lost in the process. To find cases of incompletely specified conditions, one can look for trigger words like *if ... then*, *in case* or *depending on*. To fix the issue, one needs to check, that the different cases where the condition is not met, are specified in this or another requirement too.

The last prominent issue are incompletely specified process verbs. This is the case, if verbs require several nouns to be completely specified. An example are verbs like *send*, as it implicates that there is a sender, receiver and medium of the send message or object. If any of the mentioned nouns are not specified, this leaves space for interpretation, which again can lead to misunderstanding resulting in costs when the issue needs to be fixed later in the software development process. Although there is no structured way to find the cases of this issue, one can use active voice to avoid the issue in advance. This way the sentence requires a requirements engineer to specify who is sending what to whom.



## 2.3 Template based structures for requirement statements

A requirement template allows the user to formulate requirement statements in a specified reproducible and comparable way. This approach reduces the risk of before discussed language effects. Additionally, a template can help speeding up the process of formulating high quality requirement statements and can improve the overall quality of requirement statements [Poh10]. Templates can be complemented by glossaries, which specify frequently used words at a central place to allow the usage of these terms in the requirements, without further specifying them each time. This also reduces the redundancy and minimizes conflicting specifications of individual words.

To formulate the requirement, several steps can be followed: First, one needs to specify the legal obligation of the requirement. To reflect this in the requirements contained verbs shall, should, will or may can be used. This allows to decide, whether the requirement is anything from legally obligatory up to desirable. As a next step, one determines what the core of a requirement is, meaning the functionality which is specified, usually called the process. This activity should only be specified by using verbs that depict the behavior of a software. It is important to define the core as clearly as possible, because otherwise the overall requirement quality suffers. The third step for functional requirements is to identify, whether the activity gets executed autonomously, interactively or by interfacing with another system. This leads to the differentiation of autonomous system activities, user interactions or interface requirements. Based on this categorization the requirement should be structured in one of the following templates.

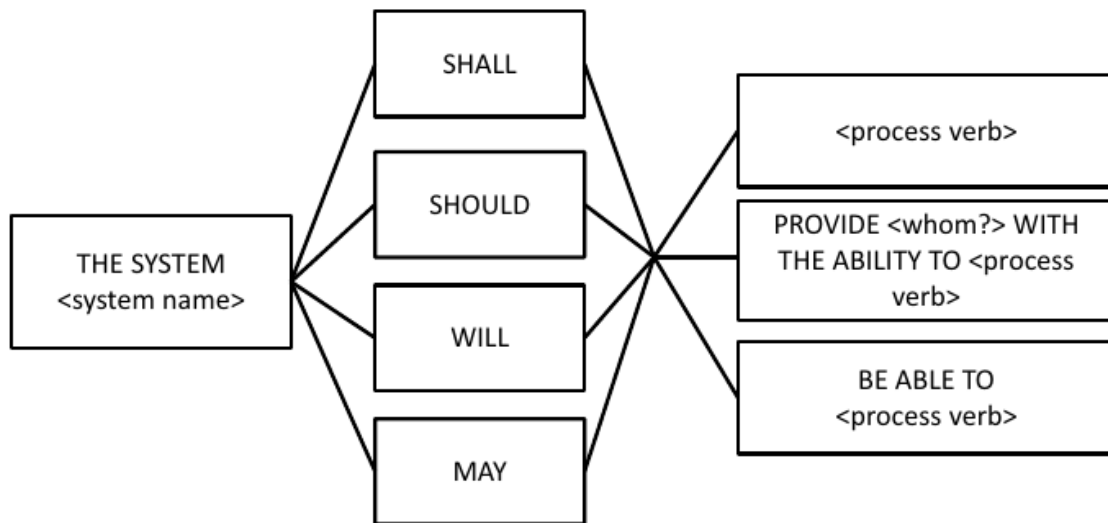
Type 1 of the templates is used for autonomous system activities, where no user interaction is present. Therefore, the template contains an identifier of the system, SHALL, SHOULD, WILL or MAY, depending on its legal obligation and the process verb, describing what the system is meant to perform [Bra97].

Type 2 of the templates is used for activities which require user interaction, providing the user with a functionality. Therefore, the system identifier gets combined with a legal obligation indicator, provide, the identifier of a user, *with the ability to* and a process verb. The identifier of a user specifies whom the functionality should be provided to (e.g. a system administrator versus a user), while the process verb again describes the activity.

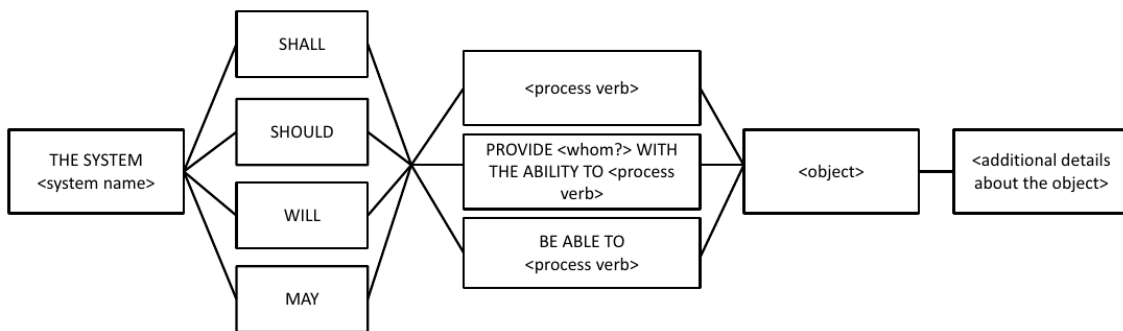
Type 3 describes an interface requirement, which means a functionality of the software triggered by an interface to another system. For that reason, no user interaction is needed, and the template consists of the system identifier, the legal obligation indicator, *be able to* and the process verb, describing the activity. If one combines all three types to one visualization, the result can be 2.1.

The next step to further complete the statement, is to insert additional objects. as some processes need them to be fully specified. For example the aforementioned process *send* does not only need the specification of a sender, but also the receiver. As a result, the object and possible further information about it get amended to the template types, if necessary. After adding an object to the overall template, the resulting structure can be seen in 2.2.

The final step of the template construction is to check for possible conditions. Sometimes, certain activities should only be performed at a specific time or under certain circumstances. Therefore, one can add a condition before the constructed template, when and under what conditions the

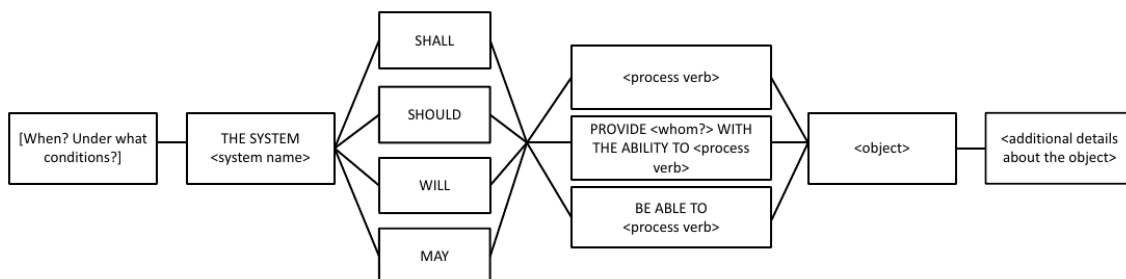


**Figure 2.1:** The core of a requirement and its legal obligation as proposed in [Poh10]



**Figure 2.2:** Principle of a complete requirement template without conditions as proposed in [Poh10]

requirement is valid. As mentioned before, it is important that in another requirement it is also specified, what happens, when the condition is not met, to make sure, that all possible circumstances are documented properly. 2.3 visualizes the resulting structure.



**Figure 2.3:** The complete requirements template with conditions as proposed in [Poh10]

This template based structure for requirement statements was also added to the ISO/IEC/IEEE norm 29148:2018 [18]. As part of the chapter 5.2.4 Requirements constructs, it is described how requirements should be built. In the following quality criteria are described, to which a requirement should align. A well-formed requirement shall be measurable, bounded by constraints, can be verified, define the performance under a certain circumstance and shall be met by the system. The norm does make use of the same template structure, but names the part of the statement differently. The condition is also labeled *Condition*, whereas the system identifier is described as the subject of the statement. The process verb is described as *Action*, while the object is also called *Object*. The additional information about the object, is described as *Constraint of Action*, which is more specific to what is described by the aforementioned template.

To have a better possibility of comparison to other existing projects, aligning to the norm, ISO 29148 norm identifiers are used in this thesis.

## 2.4 Measuring the quality of requirement statements

To automatically evaluate the quality of requirement conversions, one would need to evaluate different characteristics that show, if a requirement statement is of good quality. While some of them are easy to measure (e.g. the absence of certain formulations) others cannot be measured easily (e.g. correct labeling of the certain parts of the requirement statements). Even the easy to measure ones, are not suited as training goals, as a deep learning model would just avoid the usage of certain terms to fulfill the metric. However, measuring the quality of the statement conversion in one form of natural language to another form is a problem, not specific to our use case, but rather often used in machine translation. Therefore, we used a wide-spread metric of machine-translation to evaluate the quality of our results.

A widely used metric is the BLEU score. It measures whether a translation matches the predefined human translated statement or not. The score does not take into account semantics of the translated sentences, but rather is based on the existence of overlapping n-grams between the two sentences [LO04; PRWZ02].

There are multiple known limitations of the BLEU score: It does not take into account semantics of the sentence, which may lead to results that are fitting the human translation, but not their meaning. Due to how the score is calculated, shorter translations get a higher score, compared to longer sentences. A penalty can be introduced as a countermeasure. BLEU scores also cannot be compared between different datasets, nor among various languages. While the latter is not an issue, as we stay in English for our work, the former issue makes it hard to compare our approach between different datasets of requirement statements. Another issue is the high dependence of BLEU score in regard to its parameters, further complication the comparison across multiple appliances. As the BLEU score cannot measure all aspects of the translated sentence, we complement the automated by manual evaluations. Note that they cannot be used for fine-tuning or training of models.

### 2.5 Large Language Models

Large Language Model (LLM)s are Deep Learning (DL) models trained on a big corpus of general natural language texts. They have become widespread in recent years, after the transformers architecture [VSP+17] enabled their creation. Many different areas of NLP are affected by the impacts: These new types of models achieve impressive performance in NLP tasks, although they are not trained specifically on that task. They are pretrained on different domain-agnostic tasks like next token prediction or mask infilling to learn from the corpus, certain meaning and semantics from the natural language. The Transformer architecture usually consists of an encoder, that encodes the input as vector and a decoder, that allows to convert the output vector to be converted back into natural language. However, different variations (including decoder-only and encoder-only) have been created and showed promising results in different areas.

As such researchers started to evaluate the capabilities of LLM in a wider area of appliances reaching from translation tasks, over programming tasks, to rephrasing existing texts. Hereby different approaches are available, to specifically adapt the model for various tasks. When one refers to these tasks, they are usually called downstream tasks, while the tasks used for training the foundation model are called pretraining tasks. Different approaches have been developed to make use of Large Language Models for downstream tasks. The following are presented in order of increasing quality and complexity.

The first approach to use a Large Language Model is to use it as is and formulate the task in natural language. Therefore, the LLM makes use of their trained knowledge about natural language in general, which allows it to perform tasks, it has not been trained on. This approach is also called Zero-Shot-Learning. With zero examples the model infers on its own, what the user might want and therefore generates the most likely output.

The second approach is called Prompt Engineering. By crafting the prompts in a specific way, improvements compared to Zero-Shot-Learning are enabled. As there is no training needed, one can have quick iterations for development and may not even need coding. Different methods for prompt engineering exist. One of them is Few-shot learning: Compared to Zero-Shot-Learning the model gets a few examples presented, often with further information complemented. Another method is Chain-of-Thought, which is similar to explaining a human some given task step-by-step, which can enable the LLM to solve a given task already properly. Another method is ReAct, where the prompt engineer creates thoughts and actions of the LLM and observes how to further guide it to the task. Prompt Engineering can enable huge improvements compared to Zero-Shot-Learning, without the need of big labeled data to train a specialized model.

Another approach is Retrieval Augmented Generation (RAG), where an external knowledge base is used to let the LLM answer prompts properly. This does also not require any training, however the model needs to be connected with a database, the model can query. This is helpful for checking the facts to avoid output containing wrong information.

The last approach is fine-tuning. It provides the best performance among the presented approaches, but requires training of the model. Additionally, one needs a dataset available for training the model. Fine-tuning can update the already trained weights and therefore tailor a LLM to some specific task. One challenge here is that the model can be trained too much, resulting in a decreasing performance. In such a case it might lose its general capabilities as an excessive amount of weights is trained again.

Experiments have been conducted with the zero-shot and two fine-tuning approaches. RAG was omitted, as managing the generator and retriever components was too complex, compared to fine-tuning a model, and access to a database fitting the use case was not available.



## 3 Related Work

One fundamental base of this work is another bachelor thesis by Tobias Senger [Sen22]. A dataset based on open and closed-source software collecting functional and non-functional requirements was created. Furthermore, the possibility to train a Deep Learning model based on the dataset and using it to classify the given statements into functional and non-functional requirements has been shown. The model outperformed manual classification of statements and state-of-the-art existing models.

In Ouyang et al [OWJ+22] language model has been fine-tuned with human feedback, to make sure it outputs respectful and true statements. This can already improve the output. This indicates that prompt engineering can already leverage the certain capabilities of large language models, without fine-tuning on huge data corpus. They also showed, that smaller models can sometimes perform better in a certain task and that a higher number of parameters can be a worse choice.

Tom B Brown et al. [BMR+20] presented in their paper, how Language Models can be effectively used for few-shot learning, to avoid the need to fine-tune them on a downstream task. They indicate that the performance of few-shot-learning increases with a bigger sized language model. Therefore, they introduced the 175 billion parameters sized GPT-3 model and evaluated in on a number of downstream tasks. The showed promising results in a wide range of NLP tasks and discussed potential implications for future appliances of few-shot learning.

When choosing to work with a Large Language Model, there are plenty of options available [ZZL+23]. One of the most prominent model is Large Language Model Meta AI (LLaMA) [TMS+23] with its current second version. It offers state-of-the-art performance and has therefore become the base for many models in existence. As the model is openly available, it can be used to fine-tune for downstream tasks and tailor it to own needs. An additional advantage is, that one is not locked with a specific vendor and can receive help by the wider research and development community.





## 4 Study Design

1. Conduct literature review on converting requirement statements to formal structures
2. Prepare labeled dataset with converted requirements into a predefined template format
3. Evaluate Zero-Shot Learning approach
4. Train DL model for requirement statement conversion

### 4.1 Research Questions

RQ1: What are the limitations and challenges of using Deep Learning for automatic conversion of unstructured requirement statements into template-based structures, and how can these be addressed?

RQ2: How does the performance of Deep Learning models compare to other existing approaches or tools for converting natural language requirement statements into template-based structures?

RQ3: What are the existing methods and techniques available for converting requirement statements expressed in natural language into a structured template format?

RQ4: How effectively can the proposed tool convert natural language-based requirement statements of diverse forms into the predefined template proposed by Klaus Pohl?

### 4.2 Methodology

This section illustrates the methods we used to explore different approaches for converting requirement statements and to answer our research questions.

#### 4.2.1 Creation of a labeled dataset of converted requirements statements.

To test certain approaches on the conversion of requirement statements, it is necessary, to have a dataset with labeled data. Labeling in this case means to convert given requirement statements into a template-based-structure. As base the sward dataset has been used, which is a collection of requirements from different projects. It contains detailed information about each requirement statement including the project name, if available a subproject name, the type of requirement, the statement in different languages (English, German and Italian). Additionally, it contains the category, whether it is a functional or non-functional requirement, subcategories or subclasses for non-functional requirements, based on the source project and the original language code and a

---

**Listing 4.1** Example of the manual conversion of statements

---

```
// Given statement
The user shall be prompted to provide the low level document path.
// Converted version
[Missing][Condition][Missing][Subject]shall prompt [Action] the user [Object] to provide
the low level document path [Constraint of Action].
```

---

unique identifier called REQID. This dataset was complemented by the classification based on two different models, to classify whether this is a functional or non-functional requirement and in which subclass it should go. This task was done by a semi-supervised approach.

As the template proposed by ISO 29148:2018 is just meant for functional requirement statements, this type of requirement has been considered for creating the dataset. Doccano was used to add another column to the dataset, which includes the converted requirement statement based on the ISO-template. For each statement a converted template including labels for the different sections were added. When creating the dataset, we faced different challenges, to be resolved. One challenge was, how to deal with requirement statements, that contained several sentences per statement in the original dataset. Therefore, we used a model to split each requirement statement into single sentences, how it should be the case. Another issue was that sometimes, it was not even clear, whether a requirement statement even is a requirement statement. This could happen due to mistakes in the creation of the original dataset or by issues when splitting up the requirements to individual sentences. Such cases were ignored and no statements where this was the case have been used for this, to keep the scope of the work feasible. Another issue was to decide which parts should be completed in the labeling process. We completed as little as possible information (such as missing actors) to make sure the dataset is about the conversion, not the augmentation of missing information of requirement statements. From practical point of view, it makes more sense, to know which parts are missing, then letting a Language Model without any domain knowledge complete the missing parts.

### 4.2.2 Implementation

The experiments are implemented in a Jupyter Notebook, which allows combining markdown based and code cells for interactive computing. As the dataset is not yet available for public, it requires the dataset provided as an input. Then it loads the dataset and prepares training sets, based on the converted requirement statements. Out of the 128 converted requirements we get a training dataset of 55 samples, a validation set of 28 and a test dataset of 27 samples. Therefore, training slices of 8, 16, 32 and 55 samples are created. To compare the approach properly, a baseline is implemented, that does not convert the requirements at all, but just returns the statement it receives as input as output. This will be used to control the set-up of the study as well as helping to find any implementation issues. The first implemented approach is the zero-shot approach. Therefore, the model is loaded and asked to perform the conversion without any further training or given examples. To describe the task to the model, a prompt has been created, which gives instructions to the model. Afterward, the model is trained on all available requirement statements to check

whether fine-tuning gives an improvement in performance. At the end the fine-tune approach is implemented by training the model on sentences which contains information about requirement statements in natural language format.



# 5 Results

## 5.1 Zero-Shot Learning

After getting prompted to convert a given sentence into a requirement statement, the model performed a transformation. The prompt can be seen in Listing 5.1. As we worked with the chat-version of the model, it generated a chat like response to generate a converted requirement. This resulted in initially pretty low BLEU scores, as the answer contained a lot of additional phrasing surrounding the converted requirement. Sentences like *Sure! Here's the transformation of the sentence you provided:* lead to a lower BLEU score, as less overlapping n-grams are found. As a countermeasure the results have been post-processed, by cutting unneeded additional parts of the response away.

When converting the sentences, the model is not able to reliably convert the requirement statements into a usable form. However, it is able to sometimes rephrase the sentence from passive to active, which is a desired behavior in regard to applying a template based structure. Different issues arose, as it converted the requirements. One issue was that huge parts of the sentence were put into a specific part (e.g. the Condition), leading to the final result of a much longer sentence, then the original one. But in the same example the model was able to identify the subject (the user) and an action (shall provide). If no condition was part of the sentence, the model was not always able to leave out the condition accordingly, but rather filled it with seemingly random parts of the whole sentence.

Another effect that happened, was that the model created new labels (e.g. [Take Medication]) instead of using the provided ones as desired. Therefore, additionally it used one or two words to summarize the content of a requirement part and created a new label out of it. However, in one case almost all words have been transformed to labels. Another label that was sometimes created, was the [dot] label, which was often appended at the end of a part or at the end of the whole requirement. Sometimes the term dot was also appended without the square brackets, indicating a label. Another observable phenomena was, that the labels were sometimes generated in lowercase. This was the case, when the label was placed in front of and after the part. In this case, the label after the part is in lowercase, while the label generated at the beginning was in uppercase.

---

**Listing 5.1** The prompt that was given the model as instruction for the Zero-Shot Learning

---

Please transform the following sentence into a requirement statement by segmenting it into the components [Condition], [Subject], [Action], [Object], and [Constraint of Action], with the respective labels provided after each part. For example, after the Condition segment, include [Condition]. Make sure the requirement ends with a dot. The sentence to be transformed is: "

---

## 5 Results

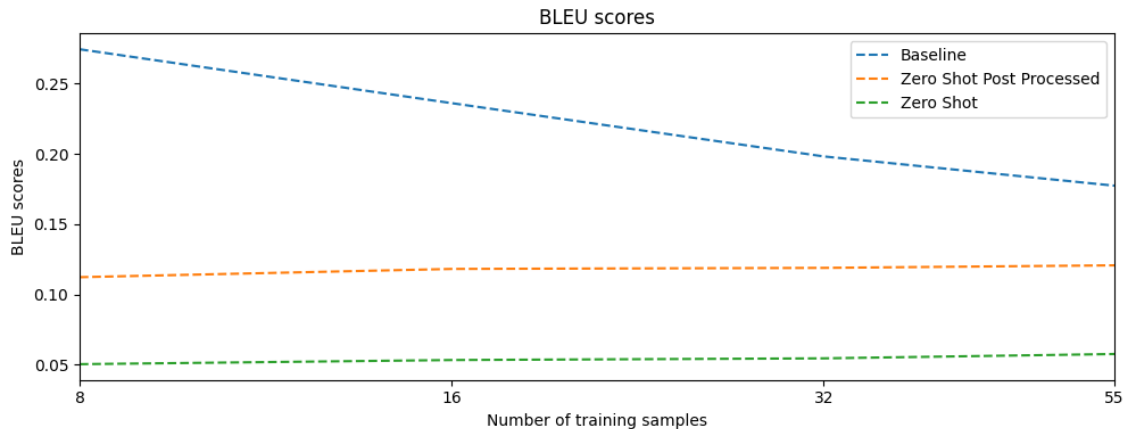
---

### Listing 5.2 Example result, where the model came up with own labels

---

[Condition] The solution shall be capable of helping users to take their medication. [Medication][Subject] Users shall be able to take their medication using the solution. [User][Action] Users shall be able to take their medication. [Take Medication][Object] The solution shall provide a means for users to take their medication. [Solution][Constraint of Action] The solution shall ensure that users take their medication as prescribed. [Prescribed Medication]

---



**Figure 5.1:** BLEU scores of the Zero-Shot approach

Another often observed issue was, that the model did not produce outputs in the desired format. It did often include the label for *Condition*, *Subject*, *Action*, *Object* and *Constraint of Action*, but failed to put the label behind the part as requested in the prompt. In addition, this should reduce the BLEU score, as it reduces the overlapping n-grams. However sometimes it further left out parts of the template.

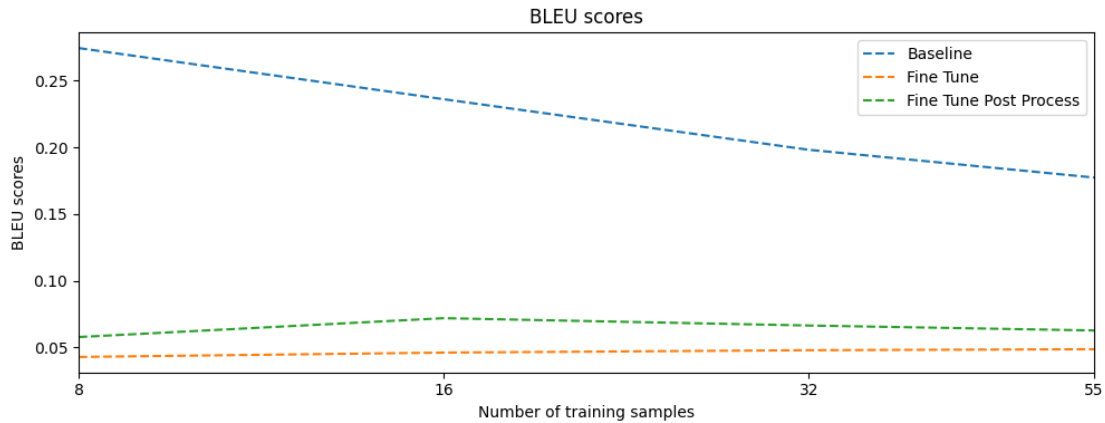
In some cases it likewise happened that the model filled in missing constraints with the condition, that it already provided in the *Condition* part. However, in other cases it kept the constraint empty, while filling the most parts correctly. Furthermore, sometimes the model was not able to grasp, whether a noun was the subject or the object of a requirement. It was also not able to make use of trigger words. In one example the sentence started with an *if*, which for a human could indicate a condition. However, the model removed the trigger word and labeled the resulting rest of the sentence as a condition, which it would not be considered by a human anymore.

Another repeating pattern was the creation of three dots at the end of some parts of the requirement: Parts of the sentence were labeled and complemented by three dots at the end of the sentence. It was also not able to make sure that every label only appears once, as in some cases the label appeared several times. For example instead of restructuring the sentence, the model labeled an object at the end as well as in the middle of the sentence. The labeling as an object was correct, however it should not appear several times per requirement.

Regarding the BLEU score, the Zero-Shot Learning approach performed worse, compared to the implemented baseline. By applying the post-processing of the output, the BLEU score has been roughly doubled, but is still less than the score of the naive baseline.

Approach	8	16	32	55
Baseline	0.27454663271	0.236174118	0.1982366365	0.1773996606
Zero-Shot	0.05028514874	0.053235761	0.0544333986	0.0575102602
Zero-Shot post-processed	0.11223096322	0.118104029	0.1188649085	0.1206674582

**Table 5.1:** BLEU score with the specified number of training samples



**Figure 5.2:** BLEU scores of the Fine-Tune approach with converted requirements

## 5.2 Fine-tuning with converted requirements

In general the results with the fine-tuned approach were not able to output just a converted requirement statement. Rather the model generated a long response including several phrases regarding the original requirement, but often not contained a statement, that could be considered a converted version of the requirement statement.

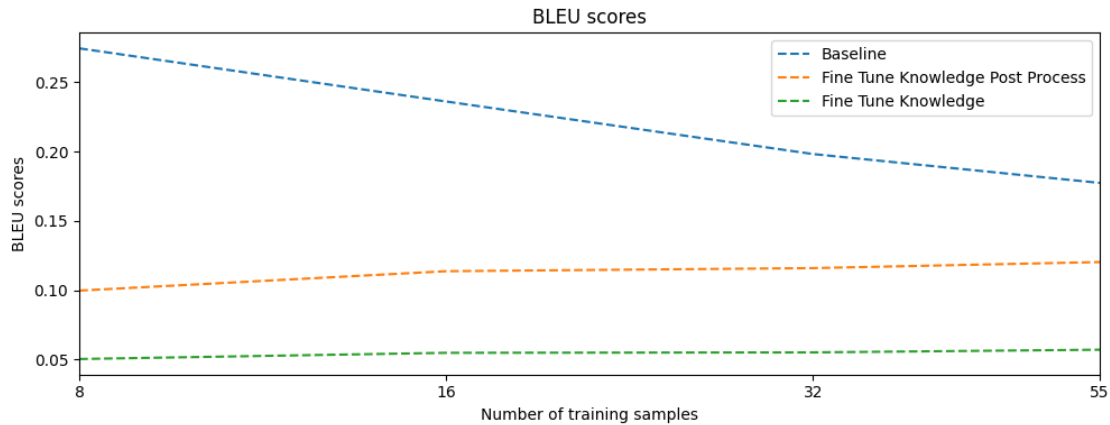
However, the model generated paragraphs describing somewhat the original requirement statement. It included references to existing norms (e.g. ISO/IEC 25010:2011 or ISO/IEC 13849-1:2015) or existing techniques that are used in Software Engineering. The model also missed the labels completely, that were desired to check for the template structure of the converted requirement.

An additional issue is the creation of additional facts. If the prompted requirement statement included only a prompt for the user to enter the name, the model generated several sentences describing additional constraints or conditions, that are not mentioned in the requirement statement. The added information was not per se useless or wrong, as it often completed the information that might have been missing in the context of the original statement. Although it was not tasked to generate additional context, it did increase the size of the output leading to poor BLEU scores. An example for additional structures that the model made up in the output are lists of constraints that need to be considered.

Regarding the BLEU score, the Fine-Tuning Learning approach performed worse, compared to the implemented baseline and compared to the Zero-Shot approach. By applying the post-processing of the output, the BLEU score has been slightly improved, however it is still significantly less compared to the score of the naive baseline.

Approach	8	16	32	55
Baseline	0.27454663271	0.236174118	0.1982366365	0.1773996606
Fine-Tune	0.04280143706	0.046075103	0.0478674156	0.0486135922
Fine-Tune post-processed	0.05774818184	0.071899075	0.0663980883	0.0626860623

**Table 5.2:** BLEU score with the specified number of training samples



**Figure 5.3:** BLEU scores of the Fine-Tune approach with knowledge sentences

### 5.3 Fine-tuning with knowledge sentences

The results of the third approach are in general similar to the ones in the Zero-Shot approach. The fine-tuning did not to further improve the resulting statements or was able to grasp the task better, compared to the Zero-Shot learning. Two of the models issues are, that it comes up with its own labels and lowercase versions of the provided templates. Also, the model sometimes generated the same label multiple times or left the label away, although trigger words could help it identify, that a certain part is needed. Sometimes the whole sentence was repeated in several sections in different similar re-phrasings, while other labels were correctly placed. The issue regarding the three dots happened as well. Furthermore, the model appended the term dot at the end of a statement, but it did not create the dot-label, surrounded by square brackets. Once the model added also an emoticon at the end of the statement.

An issue that did not happen with Zero-Shot-Learning but with this approach, was the addition of information, that was not initially provided by the original requirement. The model added the additional information in parentheses, when the provided information could be misinterpreted or misunderstood.

Regarding the BLEU score, the Fine-Tuning approach with knowledge sentences instead of converted requirements performed similar to the Zero-Shot Learning. It performed worse, compared to the implemented baseline. By applying the post-processing of the output, the BLEU score has been significantly improved, however it is still considerably less compared to the score of the naive baseline.



---

<b>Approach</b>	<b>8</b>	<b>16</b>	<b>32</b>	<b>55</b>
Baseline	0.27454663271	0.236174118	0.1982366365	0.1773996606
Fine-Tune	0.09965364839	0.113653780	0.1158917268	0.1202504119
Fine-Tune post-processed	0.05026113358	0.054726930	0.0550478200	0.0569796184

**Table 5.3:** BLEU score with the specified number of training samples



## 6 Evaluation and Discussion

### 6.1 Zero-Shot Learning

Overall the Zero-Shot approach showed promising results. The model without further retraining was capable to recognize some parts of the requirement and sometimes convert it to a template structure. However, there are several issues, which prevent the usage of the model as is. Instead of using the chat model, an experiment with the classical version can allow avoiding additional sentences in the before and after the converted statement. However, using the chat version might also be a promising path forward, if the final appliance is interactive, so a requirements engineer is able to iteratively improve the converted requirement. This could also help with utilizing the BLEU Score for further automated evaluations.

Some of the mentioned issues, like the repetition of whole sentences or filling parts that are not available in the specific requirement could be circumvented by more explicitly phrasing the task in the prompt, that if a part is missing, the model should label it with a Missing label. This might be already able to avoid the most general issues. As the model showed, it is able to e.g. rephrase existing passive sentences into active, therefore one could this way leverage already existing capabilities better to assist manual conversion. Additionally, the prompt can be complemented by explaining certain basic requirements for the converted requirement statement, to avoid these issues in the result. Problems were this applies are the three dots, duplication of labels, lowercase of provided labels and the creation of new labels. If these issues can be prevented with further prompt engineering, the Zero-Shot approach might already enable usable results which could aid a requirements engineer. Additionally, it might make sense to check for easily controllable properties. If one of the properties is not met, the model can be prompted to generate another candidate. If one provides the failed check and the information what was wrong, this could potentially lead to a performance improvement.

### 6.2 Fine-tuning with converted requirements

The fine-tuning with converted requirement statements did not excel great results. Most likely the created dataset of 128 converted requirements was not big enough to get proper results. A bigger dataset could show, whether the size can lead to a better performance. Another way would be to use the engineered prompt of the Zero-Shot approach to leverage its strengths and afterward fine tune it to further increase the performance. Mentioning the required ISO-norm and template structure in more detail, might have helped to guide the model to a better prompt as well. As the issue of generating additional facts did not arise for the zero-shot learning approach, the improved prompt might also resolve this issue for the fine-tuning approach.

**Listing 6.1** Example fine-tuning result, where the model came up with a lot of extra information

Here is a possible ISO-normed requirement statement for the requirement "The user shall be prompted to provide the new project name": ISO 27001:2013 - 14.1.3 Project Management - 14.1.3.2 Project Name The organization shall ensure that the user is prompted to provide the new project name in a clear and unambiguous manner, using a suitable input mechanism, before the project is created. The input mechanism shall be designed to ensure that the user provides a valid project name that meets the organization's naming conventions and is unique within the system. The organization shall ensure that the project name is verified against the naming conventions and is unique within the system before the project is created. The organization shall ensure that the user is informed of any errors or inconsistencies in the project name, and provides the user with the opportunity to correct them before the project is created. The organization shall ensure that the project name is recorded and stored in a secure and accessible manner, in accordance with the organization's security policies and procedures.

Approach	BLEU score at 55 samples
Baseline	0.1773996606287743
Zero-Shot	0.12066745829026043
Fine-tuned with statements	0.06268606232637361
Fine-tuned with knowledge	0.1202504119457006

**Table 6.1:** Final results of the different approaches

### 6.3 Fine-tuning with knowledge sentences

The results of the fine-tuning approach based on knowledge sentences did not show any improvements compared to the Zero-Shot approach therefore the same evaluations from Zero-Shot learning apply, indicating that additional fine-tuning with this model does not improve the output further. The better results compared to the other fine-tuning approach probably dependent on the usage of a common prompt template of this and the Zero-Shot approach. An interesting further experiment would be to see how the combination of this and their approaches would allow for a better performance, however on its own this approach seems to be the least promising, as a training is required and no real benefit was created from the creation of the dataset and the trained model.

### 6.4 Comparison of the approaches

The BLEU score was lower than the naive baseline for all the three approaches, as seen in Table 6.1. This raises the question if the BLEU score is a fitting metric to measure the quality of the converted requirement statements. While it has issues, it is still useful as a metric: The BLEU score reflected the performance of the three models in comparison to each other. Therefore, it is a usable metric to indicate the performance of different approaches. Additionally, the BLEU score did reflect the finding of the manual evaluation, that the currently presented approaches are not yet good enough for real appliances. The naive baseline of just using the statement as is needs to be outperformed, before the approaches can be considered somewhat useful for appliance. However, it is questionable

if the BLEU score can be considered a useful metric, if the approaches superseded the baseline, as it might not be able to reflect fine-grained issues in converted requirement statements. Therefore, further metrics need to be assessed for evaluations. Possible candidates might be variations of the BLEU score, that address certain shortcoming of it [SK23]. Another option is to replace or complement the BLEU based evaluation with completely different metrics like Recall-Oriented Understudy for Gisting Evaluation (ROUGE).

### **6.5 Threads to validity**

One threat is the manual creation of the dataset and the manual evaluation of the converted requirement statements. As there are no automated methods to convert requirements yet, the manual evaluation was necessary and performed with create care. Additionally, some checks were implemented to make sure the converted requirements follow a common format and fulfill certain criteria (e.g. all labels included). To further minimize the threat in future work, additional checks might be implemented and a review of the converted requirements should be added if possible.

Another threat is the usage of a specific model. As pointed out usage of the LLaMA 2 model is based on a wide usage and report of success when applying LLaMA 2 in different areas, the usage of various models and especially distinct model architectures might reveal the influence of a chosen model on the capabilities for this specific task. Therefore, a possible future work is to compare the approaches based on different used models.



## 7 Conclusion and Outlook

The experiments showed promising results in applying LLM to the conversion of requirement statements. By evaluating on the results, it appeared, that the model as-is having different issues, and it is not trivial to fine-tune a model for this task. However, different possible approaches like further prompt engineering and combination of various methods could be able to further increase the performance and would allow a real appliance of the approach to assist requirements engineers in the future.





## 8 Acknowledgement

The author acknowledges support by the High Performance and Cloud Computing Group at the Zentrum für Datenverarbeitung of the University of Tübingen, the state of Baden-Württemberg through bwHPC and the German Research Foundation (DFG) through grant no INST 37/935-1 FUGG. Furthermore, the author would like to thank his supervisor for the good supervision and all proofreaders for their helpful feedback.



## Bibliography

- [18] “ISO/IEC/IEEE International Standard - Systems and software engineering – Life cycle processes – Requirements engineering”. In: *ISO/IEC/IEEE 29148:2018(E)* (2018), pp. 1–104. doi: [10.1109/IEEESTD.2018.8559686](https://doi.org/10.1109/IEEESTD.2018.8559686) (cit. on p. 19).
- [BMR+20] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. “Language models are few-shot learners”. In: *arXiv preprint arXiv:2005.14165* (2020) (cit. on p. 23).
- [Bra97] S. O. Bradner. *Key words for use in RFCs to Indicate Requirement Levels*. RFC 2119. Mar. 1997. doi: [10.17487/RFC2119](https://doi.org/10.17487/RFC2119). URL: <https://www.rfc-editor.org/info/rfc2119> (cit. on p. 17).
- [LO04] C.-Y. Lin, F. J. Och. “ORANGE: a Method for Evaluating Automatic Evaluation Metrics for Machine Translation”. In: *COLING 2004: Proceedings of the 20th International Conference on Computational Linguistics*. Geneva, Switzerland: COLING, Aug. 2004, pp. 501–507. URL: <https://www.aclweb.org/anthology/C04-1072> (cit. on p. 19).
- [OWJ+22] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, et al. “Training language models to follow instructions with human feedback”. In: *Advances in Neural Information Processing Systems 35* (2022), pp. 27730–27744 (cit. on p. 23).
- [Poh10] K. Pohl. *Requirements engineering: fundamentals, principles, and techniques*. Springer Publishing Company, Incorporated, 2010 (cit. on pp. 13, 15, 17, 18).
- [PRWZ02] K. Papineni, S. Roukos, T. Ward, W.-j. Zhu. “BLEU: a Method for Automatic Evaluation of Machine Translation”. In: 2002, pp. 311–318 (cit. on p. 19).
- [Sen22] T. Senger. “A Unified Open and Closed - Source Software Requirements Dataset”. In: (2022) (cit. on p. 23).
- [SK23] A. M. Shweta Chauhan Philemon Daniel, A. Kumar. “AdaBLEU: A Modified BLEU Score for Morphologically Rich Languages”. In: *IETE Journal of Research* 69.8 (2023), pp. 5112–5123. doi: [10.1080/03772063.2021.1962745](https://doi.org/10.1080/03772063.2021.1962745). eprint: <https://doi.org/10.1080/03772063.2021.1962745>. URL: <https://doi.org/10.1080/03772063.2021.1962745> (cit. on p. 37).

- [TMS+23] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. C. Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. Kloumann, A. Korenev, P. S. Koura, M.-A. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. E. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, T. Scialom. *Llama 2: Open Foundation and Fine-Tuned Chat Models*. 2023. arXiv: 2307.09288 [cs.CL] (cit. on p. 23).
- [VSP+17] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, I. Polosukhin. “Attention is all you need”. In: *Advances in neural information processing systems* 30 (2017) (cit. on p. 20).
- [ZZL+23] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong, et al. “A survey of large language models”. In: *arXiv preprint arXiv:2303.18223* (2023) (cit. on p. 23).

# **A Implementation of the Jupyter Notebook**

# Main

December 13, 2023

## 1 Dealing with few to no labels

### 1.1 Installing necessary libraries

```
[ ]: ! source venv/bin/activate
! pip -q install transformers datasets pandas scikit-multilearn nltk nlpaug
↳sacre Moses "accelerate>=0.20.1"
```

### 1.2 Preparing the Data

Note: the Notebook assumes the dataset to be available as dataset.csv file in the current directory and the labeled dataset part available as admin.csv .

```
[2]: import pandas

dataset_file = "dataset.csv"
labeled_dataset_file = "admin.csv"
dataframe_requirements = pandas.read_csv(dataset_file, low_memory=False)
dataframe_labeled = pandas.read_csv(labeled_dataset_file, low_memory=False)

# Explicitly convert the data type of "REQID_expanded" column in
↳dataframe_labeled to match dataframe_requirements
dataframe_labeled["REQID_expanded"] = dataframe_labeled["REQID_expanded"].
↳astype(dataframe_requirements["REQID_expanded"].dtype)

dataframe_requirements = pandas.merge(dataframe_requirements,
↳dataframe_labeled[["REQID_expanded", "label"]], on="REQID_expanded",
↳how="left")
dataframe_requirements.rename(columns={"label_x": "label"}, inplace=True)
dataframe_requirements["label"] = dataframe_requirements["label"].fillna("None")
requirement_statements = dataframe_requirements["text"]
print(f"DataFram shape: {dataframe_requirements.shape}")
```

DataFram shape: (69787, 26)

```
[3]: columns = ["REQID_expanded", "Project Name", "text", "label"]
dataframe_requirements.loc[70, columns].to_frame()
```

---

```
[3]:
      REQID_expanded      70
      Project Name      (RETRO).NET
      text      A project created via the LOAD RTM function ma...
      label      None
```

```
[4]: dataframe_requirements["Project Name"].value_counts().to_frame().T
```

```
[4]: Project Name  IlmSeven  PURE  COEST  AB  N  L  \
      count      21510  13564  4270  3263  3055  3021

      Project Name  A Dataset of Crowdsourced Smarthome Requirements with Creativity
      Ratings  \
      count      2882

      Project Name  A Requirements data sets (user stories)  V ... WHO  \
      count      1869      1673  1509 ... 114

      Project Name  OPENCOSS  Q  NIST  Gaia-X Federation Services  MobStr  \
      count      108  97  80      72  68

      Project Name  Design requirements of the app for bridging communication between
      a Deaf patient and healthcare personnel__Version 2  \
      count      63

      Project Name  Promise Modis  \
      count      62

      Project Name  Eye Tracking Experiments Data Set - Linking Use Cases and
      Associated Requirements: On the Impact of Linking Variants on Reading Behavior
      \
      count      20

      Project Name  Moving-block System
      count      14
```

```
[1 rows x 45 columns]
```

```
[5]: dataframe_requirements["split"] = "unlabeled"
      mask = dataframe_requirements["label"] != "None"
      dataframe_requirements.loc[mask, "split"] = "labeled"
      dataframe_requirements["split"].value_counts().to_frame()
```

```
[5]:
      count
      split
      unlabeled  69659
      labeled    128
```

```
[6]: len_before = len(dataframe_requirements)
dataframe_requirements = dataframe_requirements.drop_duplicates(subset="text")
print(f"Removed {(len_before-len(dataframe_requirements))/len_before:.2%}␣
↳duplicates.")
```

Removed 8.86% duplicates.

### 1.3 Creating Training Sets

```
[7]: import numpy
from sklearn.preprocessing import MultiLabelBinarizer

mlb = MultiLabelBinarizer()
all_projects = dataframe_requirements["Project Name"].unique()
mlb.fit([all_projects])
```

```
[7]: MultiLabelBinarizer()
```

```
[8]: from skmultilearn.model_selection import iterative_train_test_split

def balanced_split(dataframe, test_size=0.5):
    ind = numpy.expand_dims(numpy.arange(len(dataframe)), axis=1)
    project_names = mlb.transform(dataframe["Project Name"])
    ind_train, _, ind_test, _ = iterative_train_test_split(ind, project_names,␣
↳test_size)
    return dataframe.iloc[ind_train[:, 0]], dataframe.iloc[ind_test[:,0]]
```

```
[9]: from sklearn.model_selection import train_test_split

dataframe_clean = dataframe_requirements[["text", "Project Name", "split",␣
↳"label"]].reset_index(drop=True).copy()
dataframe_unlabeled = dataframe_clean.loc[dataframe_clean["split"] ==␣
↳"unlabeled", ["text", "Project Name", "label"]]
dataframe_labeled = dataframe_clean.loc[dataframe_clean["split"] == "labeled",␣
↳["text", "Project Name", "label"]]

numpy.random.seed(0)
dataframe_train, dataframe_temp = balanced_split(dataframe_labeled, test_size=0.␣
↳5)
dataframe_valid, dataframe_test = balanced_split(dataframe_temp, test_size=0.5)
```

```
/opt/bwhpc/common/jupyter/tensorflow/2023-10-10/lib/python3.9/site-
packages/sklearn/preprocessing/_label.py:900: UserWarning: unknown class(es) ['
', '(', ')', '-', '.', ':', 'B', 'C', 'D', 'E', 'F', 'G', 'I', 'M', 'a', 'b',
'c', 'd', 'e', 'f', 'g', 'h', 'i', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's',
't', 'u', 'v', 'x', 'y'] will be ignored
warnings.warn(
```



```
[10]: from datasets import Dataset, DatasetDict

dataset = DatasetDict({
    "train": Dataset.from_pandas(dataframe_train.reset_index(drop=True)),
    "valid": Dataset.from_pandas(dataframe_valid.reset_index(drop=True)),
    "test": Dataset.from_pandas(dataframe_test.reset_index(drop=True)),
    "unsup": Dataset.from_pandas(dataframe_unlabeled.reset_index(drop=True))})
```

## 1.4 Creating Training Slices

```
[11]: numpy.random.seed(0)
all_indices = numpy.expand_dims(list(range(len(dataset["train"]))), axis=1)
indices_pool = all_indices
project_names = mlb.transform(dataset["train"]["Project Name"])
# train set is of size 64 further increasing possible after bigger train set is
↳available
train_samples = [8, 16, 32]
train_slices = []
last_k = 0
for i, k in enumerate(train_samples):
    # Split off samples necessary to fill the gap of the next split size
    indices_pool, project_names, new_slice, _ =
↳iterative_train_test_split(indices_pool, project_names, (k-last_k)/
↳len(project_names))
    last_k = k
    if i==0:
        train_slices.append(new_slice)
    else:
        train_slices.append(numpy.concatenate((train_slices[-1], new_slice)))

# Add full dataset as last slice
train_slices.append(all_indices)
train_samples.append(len(dataset["train"]))
train_slices = [numpy.squeeze(train_slice) for train_slice in train_slices]
```

```
/opt/bwhpc/common/jupyter/tensorflow/2023-10-10/lib/python3.9/site-
packages/sklearn/preprocessing/_label.py:900: UserWarning: unknown class(es) ['
', '(', ')', '-', '.', ':', 'B', 'C', 'D', 'E', 'F', 'G', 'I', 'M', 'a', 'b',
'c', 'd', 'e', 'f', 'g', 'h', 'i', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's',
't', 'u', 'v', 'x', 'y'] will be ignored
warnings.warn(
```

```
[12]: print("Target split sizes:")
print(train_samples)
print("Actual split sizes:")
print([len(x) for x in train_slices])
```

Target split sizes:

[8, 16, 32, 55]

Actual split sizes:

[8, 16, 32, 55]

## 1.5 Implementing a Baseline

```
[13]: import nltk
from nltk.translate.bleu_score import sentence_bleu
from collections import defaultdict

nltk.download('punkt')
bleu_scores = defaultdict(list)

def calculate_bleu_scores_baseline(train_slices, dataset):
    return_bleu_scores = []

    for train_slice in train_slices:
        # Get training slice and test data
        dataset_train_sample = dataset["train"].select(train_slice)
        current_bleu_scores = []
        for sample in dataset_train_sample:
            # Tokenize the sentences (split into words)
            reference = nltk.word_tokenize(sample["label"])
            hypothesis = nltk.word_tokenize(sample["text"])

            # Calculate BLEU score
            bleu_score = sentence_bleu([reference], hypothesis)
            current_bleu_scores.append(bleu_score)

        bleu_score = sum(current_bleu_scores) / len(current_bleu_scores)
        # Store metrics
        return_bleu_scores.append(bleu_score)
    return return_bleu_scores

bleu_scores["Baseline"] = calculate_bleu_scores_baseline(train_slices, dataset)
```

```
[nltk_data] Downloading package punkt to
[nltk_data]   /home/st/st_us-051500/st_st172013/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
/home/st/st_us-051500/st_st172013/.local/lib/python3.9/site-
packages/nltk/translate/bleu_score.py:552: UserWarning:
The hypothesis contains 0 counts of 3-gram overlaps.
Therefore the BLEU score evaluates to 0, independently of
how many N-gram overlaps of lower order it contains.
Consider using lower n-gram order or use SmoothingFunction()
  warnings.warn(_msg)
/home/st/st_us-051500/st_st172013/.local/lib/python3.9/site-
packages/nltk/translate/bleu_score.py:552: UserWarning:
```

---

The hypothesis contains 0 counts of 4-gram overlaps.  
Therefore the BLEU score evaluates to 0, independently of  
how many N-gram overlaps of lower order it contains.  
Consider using lower n-gram order or use SmoothingFunction()  
warnings.warn(\_msg)

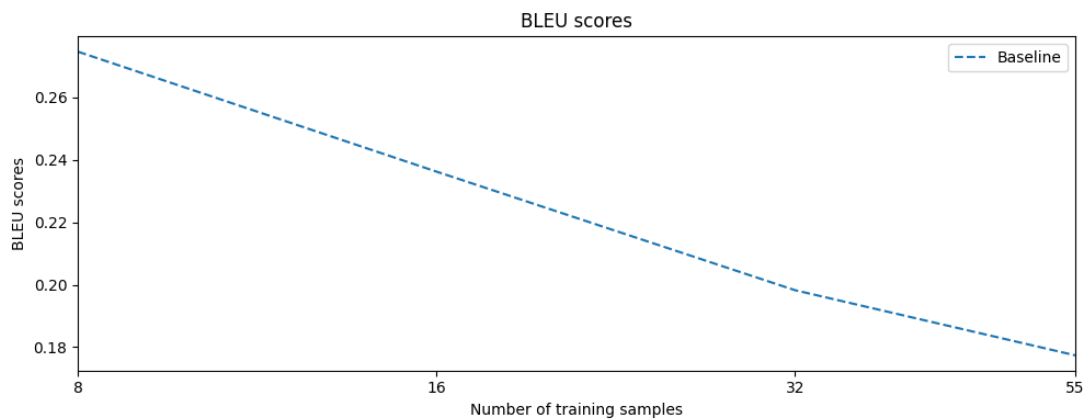
```
[14]: import matplotlib.pyplot as pyplot

def plot_metrics(bleu_scores, sample_sizes):
    fig, ax0 = pyplot.subplots(1, 1, figsize=(10, 4))

    for run in bleu_scores.keys():
        ax0.plot(sample_sizes, bleu_scores[run], label=run, linestyle="dashed")

    ax0.set_title("BLEU scores")
    ax0.set_ylabel("BLEU scores")
    ax0.legend(loc="upper right")
    for ax in [ax0]:
        ax.set_xlabel("Number of training samples")
        ax.set_xscale("log")
        ax.set_xticks(sample_sizes)
        ax.set_xticklabels(sample_sizes)
        ax.minorticks_off()
    pyplot.tight_layout()
    pyplot.margins(x=0, tight=True)
    pyplot.show()
```

```
[15]: plot_metrics(bleu_scores, train_samples)
```



## 1.6 Zero-Shot approach (Working with No Labeled Data)

```
[16]: from huggingface_hub import notebook_login
```

```
notebook_login()
```

```
VBox(children=(HTML(value='<center> <img\nsrc=https://huggingface.co/front/\nassets/huggingface_logo-noborder.sv...
```

```
[ ]: from transformers import pipeline
```

```
pipeline = pipeline("text-generation", model="meta-llama/Llama-2-7b-chat-hf")
```

```
[17]: prompt = "Please transform the following sentence into a requirement statement,\nby segmenting it into the components [Condition], [Subject], [Action],\n[Object], and \n[Constraint or Action], with the respective labels provided after each part.\nFor example, after the Condition segment, include [Condition]. Make sure the\nrequirement \nends with a dot. The sentence to be transformed is: "\nstatement = "When signal x is received, the system shall set the signal x_\nreceived bit within 2 seconds."
```

```
[18]: def post_process_zero_shot(raw_result):  
    result = raw_result[0]["generated_text"]  
    result = result[len(prompt) + statement + 18:]  
    result = result[result.index("["):]  
    result = result.splitlines()[:-1]  
    result = list(filter(None, result))  
    new_result = []  
    for string in result:  
        if string.startswith("[Constraint or Action]"):   
            new_result.append(string[len("[Constraint or Action]"):])  
            new_result.append(string[:len("[Constraint or Action]")])  
        elif " " in string:  
            new_result.append(string[string.index(" "):])  
            new_result.append(string[:string.index(" ")])  
        else:  
            new_result.append(string)  
    result = new_result  
    new_result = []  
    for string in result:  
        if string.startswith(" "):  
            string = string.lstrip() + " "  
        new_result.append(string)  
    result = "".join(new_result)  
    if not result.endswith("."):   
        result += "."
```

```
    return result

    #print(post_process_zero_shot(output))
```

```
[19]: import pandas

def zero_shot_pipeline(example):
    statement = example["text"]
    print(statement)
    output = pipeline(f"<s>[INST] {prompt + statement} [/INST]")
    example["result"] = output[0]["generated_text"]
    return example

def zero_shot_pipeline_with_cache(example):
    # Check if the result is already in cache
    cache_file = "result_cache.csv"

    cache = pandas.read_csv(cache_file)
    if example["text"] in cache["text"].values:
        return cache[cache['text'] == example['text']]

    # If not in the cache, calculate the result
    result = zero_shot_pipeline(example)

    # Update the cache DataFrame
    cache = pandas.concat([cache, pandas.DataFrame({'text': [example['text']],
    ↪ 'result': [result]})], ignore_index=True)

    # Save the updated cache to the CSV file
    cache.to_csv(cache_file, index=False)
    return result

dataset_zero_shot = dataset["test"].map(zero_shot_pipeline_with_cache,
    ↪ num_proc=1)
```

Map: 0% | | 0/27 [00:00<?, ? examples/s]

```
[20]: from nltk.translate.bleu_score import sentence_bleu

def calculate_bleu_scores_zero_shot(train_slices, dataset):
    return_bleu_scores = []
    for train_slice in train_slices:
        threshold = len(dataset["text"])
        for value in train_slice:
            # Get training slice and test data
            dataset_train_sample = dataset.select(train_slice)
```

```

    current_bleu_scores = []
    for sample in dataset_train_sample:
        # Tokenize the sentences (split into words)
        reference = nltk.word_tokenize(sample["text"])
        hypothesis = nltk.word_tokenize(sample["result"])

        # Calculate BLEU score
        bleu_score = sentence_bleu([reference], hypothesis)
        current_bleu_scores.append(bleu_score)

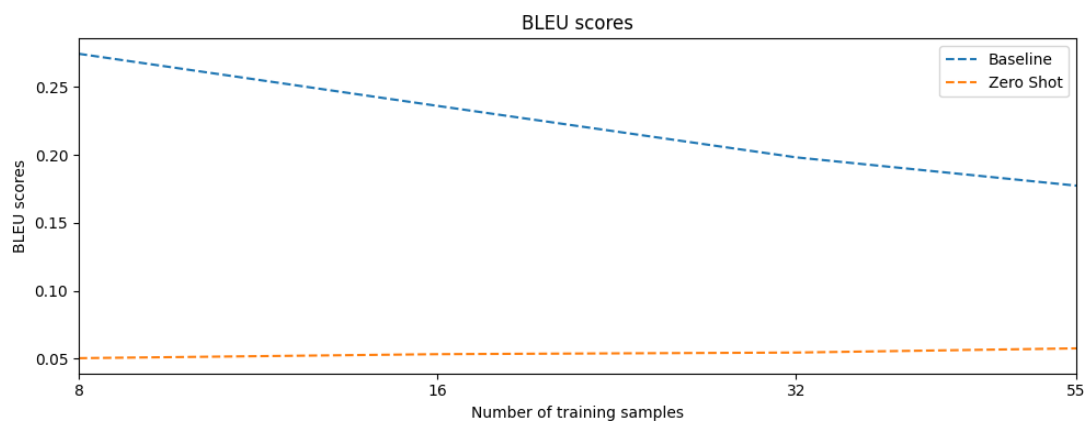
    bleu_score = sum(current_bleu_scores) / len(current_bleu_scores)
    # Store metrics
    return_bleu_scores.append(bleu_score)
return return_bleu_scores

zero_shot_train_slices = []
for train_slice in train_slices:
    new_train_slice = []
    for element in train_slice:
        if element < len(dataset_zero_shot["text"]):
            new_train_slice.append(element)
    zero_shot_train_slices.append(new_train_slice)

bleu_scores["Zero Shot"] = calculate_bleu_scores_zero_shot(zero_shot_train_slices, dataset_zero_shot)

```

```
[21]: plot_metrics(bleu_scores, train_samples)
```



---

## 2 Fine Tuning a requirement conversion in LLaMa 2

In this notebook, we will teach LLaMa 2 an how to convert requirement statements.

### 2.1 Setup the Model

The following section performs all the setup of the model. This includes

- Installing any dependencies
- Setting any configuration
- Downloading the Base Model

#### 2.1.1 Install dependencies

In order to get started we need to install the appropriate dependencies

```
[ ]: ! source venv/bin/activate

# install dependencies

# we use the latest version of transformers, peft, and accelerate
!pip install -q accelerate peft transformers

# install bitsandbytes for quantization
!pip install -q bitsandbytes

# install trl for the SFT library
!pip install -q trl

# we need sentencepiece for the llama2 slow tokenizer
!pip install sentencepiece

# we need einops, used by falcon-7b, llama-2 etc
# einops (einsteinops) is used to simplify tensorops by making them readable
!pip install -q -U einops

# we need to install datasets for our training dataset
!pip install -q datasets
```

#### 2.1.2 Settings

The following configures our settings for finetuning our model

```
[23]: # The model that you want to train from the Hugging Face hub
model_name = "meta-llama/Llama-2-7b-chat-hf"

# The instruction dataset to use
dataset_name = "Maxnet/re-knowledge"
```

## A Implementation of the Jupyter Notebook

---

```
# Fine-tuned model name
new_model = "llama-2-7b-req-know"

# Output directory where the model predictions and checkpoints will be stored
output_dir = "./results"
```

```
[24]: #####
# QLoRA parameters
#####

lora_r = 32 # Defines the size or dimensionality of the vectors used within the
↳ attention computation. A higher value can capture more intricate patterns
↳ but might be computationally expensive and cause overfitting.
lora_alpha = 16 # Helps in managing the magnitude of values.
lora_dropout = 0.2 # 20% of the neurons in LoRA layers would be turned off at
↳ each training step.

#####
# bitsandbytes parameters
#####

use_4bit = True # Reducing to 4-bit precision instead of 32.
bnb_4bit_compute_dtype = "float16" # Less precise than float32 but allows
↳ computations to be faster.
bnb_4bit_quant_type = "nf4" # A specific 4-bit quantization technique
use_nested_quant = False # Activate nested quantization for 4-bit base models
↳ (double quantization)

#####
# TrainingArguments parameters
#####

num_train_epochs = 20 # number of full passes through the training data
# Enable fp16/bf16 training (set bf16 to True with an A100) used to speed up
↳ training and reduce memory usage
fp16 = False
bf16 = False
# Batch size per GPU for training and evaluation
per_device_train_batch_size = 4 # 4 seems reasonable
per_device_eval_batch_size = 4
# Number of update steps to accumulate the gradients for
gradient_accumulation_steps = 2 # 2 is fine, as we're a small batch
# Enable gradient checkpointing
gradient_checkpointing = True
# Maximum gradient normal (gradient clipping)
max_grad_norm = 0.3 # standard setting
# Initial learning rate (AdamW optimizer)
```



---

```

learning_rate = 2e-4 # standard learning rate
# Weight decay to apply to all layers except bias/LayerNorm weights
weight_decay = 0.001 # standard weight decay
# Optimizer to use
optim = "paged_adamw_32bit" # default optimizer
# Learning rate schedule
lr_scheduler_type = "cosine"
# Number of training steps (overrides num_train_epochs)
max_steps = -1 # needs to be -1, otherwise overrides epochs
# Ratio of steps for a linear warmup (from 0 to learning rate)
warmup_ratio = 0.03 # standard warmup ratio
# Group sequences into batches with same length
group_by_length = True # speeds up the training
# Save checkpoint every X updates steps
save_steps = 0 # we're not gonna save
# Log every X updates steps
logging_steps = 10 # same value as used by Meta

#####
# SFT parameters
#####

# Maximum sequence length to use
max_seq_length = None # no max sequence length
# Pack multiple short examples in the same input sequence to increase efficiency
packing = False # don't need packing
# Load the entire model on the GPU 0
device_map = {"": 0}

```

### 2.1.3 Download the base model

The following will download the base model, in this case the llama-2-7b-chat-hf model.

```

[25]: import torch
from transformers import (AutoTokenizer, AutoModelForCausalLM,
    BitsAndBytesConfig, pipeline, logging)

# Load tokenizer and model with QLoRA configuration
compute_dtype = getattr(torch, bnb_4bit_compute_dtype)

# load the quantized settings, we're doing 4 bit quantization
bnb_config = BitsAndBytesConfig(
    load_in_4bit=use_4bit,
    bnb_4bit_quant_type=bnb_4bit_quant_type,
    bnb_4bit_compute_dtype=compute_dtype,
    bnb_4bit_use_double_quant=use_nested_quant,

```

```

)

# Check GPU compatibility with bfloat16
if compute_dtype == torch.float16 and use_4bit:
    major, _ = torch.cuda.get_device_capability()
    if major >= 8:
        print("=" * 80)
        print("Your GPU supports bfloat16: accelerate training with bf16=True")
        print("=" * 80)

# Load base model
model = AutoModelForCausalLM.from_pretrained(model_name,
    ↪quantization_config=bnb_config,
                                           device_map=device_map)

# Load base model
model = AutoModelForCausalLM.from_pretrained(
    model_name,
    quantization_config=bnb_config,
    device_map=device_map
)

# don't use the cache
model.config.use_cache = False
model.config.pretraining_tp = 1

# Load the LLaMa tokenizer from the model (llama2)
tokenizer = AutoTokenizer.from_pretrained(model_name, trust_remote_code=True,
    ↪use_fast=False)
tokenizer.pad_token = tokenizer.eos_token
tokenizer.padding_side = "right" # Fix weird overflow issue with fp16 training

```

```

2023-12-13 09:04:21.085788: I tensorflow/core/util/port.cc:111] oneDNN custom
operations are on. You may see slightly different numerical results due to
floating-point round-off errors from different computation orders. To turn them
off, set the environment variable `TF_ENABLE_ONEDNN_OPTS=0`.
2023-12-13 09:04:21.995180: E
tensorflow/compiler/xla/stream_executor/cuda/cuda_dnn.cc:9342] Unable to
register cuDNN factory: Attempting to register factory for plugin cuDNN when one
has already been registered
2023-12-13 09:04:21.995219: E
tensorflow/compiler/xla/stream_executor/cuda/cuda_fft.cc:609] Unable to register
cuFFT factory: Attempting to register factory for plugin cuFFT when one has
already been registered
2023-12-13 09:04:21.995252: E
tensorflow/compiler/xla/stream_executor/cuda/cuda_blas.cc:1518] Unable to
register cuBLAS factory: Attempting to register factory for plugin cuBLAS when
one has already been registered

```

---

```
2023-12-13 09:04:22.353218: I tensorflow/core/platform/cpu_feature_guard.cc:182]
This TensorFlow binary is optimized to use available CPU instructions in
performance-critical operations.
```

```
To enable the following instructions: AVX2 AVX512F AVX512_VNNI FMA, in other
operations, rebuild TensorFlow with the appropriate compiler flags.
```

```
2023-12-13 09:04:41.404475: W
```

```
tensorflow/compiler/tf2tensorrt/utils/py_utils.cc:38] TF-TRT Warning: Could not
find TensorRT
```

```
Loading checkpoint shards: 0%|          | 0/2 [00:00<?, ?it/s]
```

```
Loading checkpoint shards: 0%|          | 0/2 [00:00<?, ?it/s]
```

## 2.2 Run the Model

The following tests the capabilities of the language model prior to fine tuning.

```
[27]: # Ignore warnings
logging.set_verbosity(logging.CRITICAL)

prompt = "Please transform the following sentence into a requirement statement,
↳by segmenting it into the components [Condition], [Subject], [Action],
↳[Object], and \
[Constraint or Action], with the respective labels provided after each part.
↳For example, after the Condition segment, include [Condition]. Make sure the
↳requirement \
ends with a dot. The sentence to be transformed is: "
statement = "When signal x is received, the system shall set the signal x
↳received bit within 2 seconds."

pipe = pipeline(task="text-generation", model=model, tokenizer=tokenizer,
↳max_length=200)
result = pipe(f"<s>[INST] {prompt + statement} [/INST]")
print(result[0]['generated_text'])
```

```
<s>[INST] Please transform the following sentence into a requirement statement
by segmenting it into the components [Condition], [Subject], [Action], [Object],
and [Constraint or Action], with the respective labels provided after each part.
For example, after the Condition segment, include [Condition]. Make sure the
requirement ends with a dot. The sentence to be transformed is: When signal x is
received, the system shall set the signal x received bit within 2 seconds.
[/INST] Sure! Here's the transformation of the sentence into a requirement
statement:
```

```
[Condition] When signal x is received
[Subject] the system
[Action] set the signal x received bit
[Object] within 2 seconds
[Constraint or Action].
```

So the requirement statement is: When signal x is received, the system shall set the signal x received bit within 2 seconds.

### 2.3 Train the Model

The following section is about taking your dataset and then finetuning the model

#### 2.3.1 Load Dataset

The following code will load your dataset, ready to be fine tuned by the model

```
[19]: from datasets import load_dataset

# Load dataset (you can process it here)
dataset_fine_tune = load_dataset(dataset_name, split="train")
```

#### 2.3.2 Fine Tune the Model

The following section will take your dataset, and fine tune the model with it.

```
[51]: import torch
from transformers import (
    AutoModelForCausalLM,
    AutoTokenizer,
    BitsAndBytesConfig,
    HfArgumentParser,
    TrainingArguments,
    pipeline,
    logging,
)
from peft import LoraConfig
from trl import SFTTrainer
import os

# Load LoRA configuration
peft_config = LoraConfig(
    lora_alpha=lora_alpha,
    lora_dropout=lora_dropout,
    r=lora_r,
    bias="none",
    task_type="CAUSAL_LM",
)

# Set training parameters
training_arguments = TrainingArguments(
    output_dir=output_dir,
    num_train_epochs=num_train_epochs, # uses the number of epochs earlier
    per_device_train_batch_size=per_device_train_batch_size,
```

---

```

    gradient_accumulation_steps=gradient_accumulation_steps,
    optim=optim,
    save_steps=save_steps,
    logging_steps=logging_steps,
    learning_rate=learning_rate,
    weight_decay=weight_decay,
    fp16=fp16,
    bf16=bf16,
    max_grad_norm=max_grad_norm,
    max_steps=max_steps,
    warmup_ratio=warmup_ratio,
    group_by_length=group_by_length,
    lr_scheduler_type=lr_scheduler_type,
    report_to="tensorboard"
)

# Set supervised fine-tuning parameters
trainer = SFTTrainer(
    model=model,
    train_dataset=dataset_fine_tune,
    peft_config=peft_config,           # use our lora peft config
    dataset_text_field="text",
    max_seq_length=max_seq_length,
    tokenizer=tokenizer,              # use the llama tokenizer
    args=training_arguments,         # use the training arguments
    packing=packing,
)

if not os.path.exists(new_model):
    # Train model
    trainer.train()

    # Save trained model
    trainer.model.save_pretrained(new_model)

```

Detected kernel version 4.18.0, which is below the recommended minimum of 5.5.0; this can cause the process to hang. It is recommended to upgrade the kernel to the minimum version or higher.

```

{'loss': 5.1977, 'learning_rate': 0.00019744105246469263, 'epoch': 1.82}
{'loss': 3.6472, 'learning_rate': 0.00018522168236559695, 'epoch': 3.64}
{'loss': 2.7714, 'learning_rate': 0.000164140821963114, 'epoch': 5.45}
{'loss': 2.2181, 'learning_rate': 0.00013639049369634876, 'epoch': 7.27}
{'loss': 1.7516, 'learning_rate': 0.00010485622221144484, 'epoch': 9.09}
{'loss': 1.3566, 'learning_rate': 7.281699277636572e-05, 'epoch': 10.91}
{'loss': 1.132, 'learning_rate': 4.360429701490934e-05, 'epoch': 12.73}
{'loss': 0.9712, 'learning_rate': 2.025571894372794e-05, 'epoch': 14.55}
{'loss': 0.8, 'learning_rate': 5.199082004372957e-06, 'epoch': 16.36}

```

```
{'loss': 0.8602, 'learning_rate': 0.0, 'epoch': 18.18}
{'train_runtime': 133.2264, 'train_samples_per_second': 6.605,
'train_steps_per_second': 0.751, 'train_loss': 2.0705929374694825, 'epoch':
18.18}
```

```
[ ]: from peft import PeftModel

# Reload model in FP16 and merge it with LoRA weights
base_model = AutoModelForCausalLM.from_pretrained(
    model_name,
    low_cpu_mem_usage=True,
    return_dict=True,
    torch_dtype=torch.float16,
    device_map=device_map,
)
model = PeftModel.from_pretrained(base_model, new_model)
model = model.merge_and_unload()

# Reload tokenizer to save it
tokenizer = AutoTokenizer.from_pretrained(model_name, trust_remote_code=True)
tokenizer.pad_token = tokenizer.eos_token
tokenizer.padding_side = "right"
```

```
[31]: model.push_to_hub(new_model, use_temp_dir=False)
tokenizer.push_to_hub(new_model, use_temp_dir=False)
```

```
[31]: CommitInfo(commit_url='https://huggingface.co/Maxnet/llama-2-7b-
req/commit/78353e02475ef8281f89e5b2252e7b57e1e0b515', commit_message='Upload
tokenizer', commit_description='',
oid='78353e02475ef8281f89e5b2252e7b57e1e0b515', pr_url=None, pr_revision=None,
pr_num=None)
```

## 2.4 Evaluate the model and compare to zero shot

```
[20]: from transformers import pipeline

pipeline = pipeline("text-generation", model="llama-2-7b-req")
```

```
2023-12-07 16:46:06.862010: I tensorflow/core/util/port.cc:111] oneDNN custom
operations are on. You may see slightly different numerical results due to
floating-point round-off errors from different computation orders. To turn them
off, set the environment variable `TF_ENABLE_ONEDNN_OPTS=0`.
2023-12-07 16:46:07.899743: E
tensorflow/compiler/xla/stream_executor/cuda/cuda_dnn.cc:9342] Unable to
register cuDNN factory: Attempting to register factory for plugin cuDNN when one
has already been registered
2023-12-07 16:46:07.899782: E
tensorflow/compiler/xla/stream_executor/cuda/cuda_fft.cc:609] Unable to register
```

```

cuFFT factory: Attempting to register factory for plugin cuFFT when one has
already been registered
2023-12-07 16:46:07.899815: E
tensorflow/compiler/xla/stream_executor/cuda/cuda_blas.cc:1518] Unable to
register cuBLAS factory: Attempting to register factory for plugin cuBLAS when
one has already been registered
2023-12-07 16:46:08.192299: I tensorflow/core/platform/cpu_feature_guard.cc:182]
This TensorFlow binary is optimized to use available CPU instructions in
performance-critical operations.
To enable the following instructions: AVX2 AVX512F AVX512_VNNI FMA, in other
operations, rebuild TensorFlow with the appropriate compiler flags.
2023-12-07 16:46:32.471731: W
tensorflow/compiler/tf2tensorrt/utils/py_utils.cc:38] TF-TRT Warning: Could not
find TensorRT

Loading checkpoint shards: 0%|          | 0/2 [00:00<?, ?it/s]

```

```
[26]: prompt = "Convert the following requirement to a ISO-normed requirement,
↳statement: "
```

```
[27]: import pandas

def fine_tune_pipeline(example):
    statement = example["text"]
    output = pipeline(f"<s>[INST] {prompt + statement} [/INST]")
    example["result"] = output[0]["generated_text"]
    return example

def fine_tune_pipeline_with_cache(example):
    # Check if the result is already in cache
    cache_file = "result_cache_fine_tuned_post_processed.csv"

    cache = pandas.read_csv(cache_file)
    if example["text"] in cache["text"].values:
        return cache[cache['text'] == example['text']]

    # If not in the cache, calculate the result
    result = fine_tune_pipeline(example)

    # Update the cache DataFrame
    cache = pandas.concat([cache, pandas.DataFrame({'text': [example['text']],
↳'result': [result]})], ignore_index=True)

    # Save the updated cache to the CSV file
    cache.to_csv(cache_file, index=False)
    return result

```

```
dataset_fine_tuned = dataset["test"].map(fine_tune_pipeline_with_cache,
↳ num_proc=1)
```

Map: 0%| | 0/27 [00:00<?, ? examples/s]

```
[28]: from nltk.translate.bleu_score import sentence_bleu

def calculate_bleu_scores_fine_tune(train_slices, dataset):
    return_bleu_scores = []
    for train_slice in train_slices:
        threshold = len(dataset["text"])
        for value in train_slice:
            # Get training slice and test data
            dataset_train_sample = dataset.select(train_slice)
            current_bleu_scores = []
            for sample in dataset_train_sample:
                # Tokenize the sentences (split into words)
                reference = nltk.word_tokenize(sample["text"])
                hypothesis = nltk.word_tokenize(sample["result"])

                # Calculate BLEU score
                bleu_score = sentence_bleu([reference], hypothesis)
                current_bleu_scores.append(bleu_score)

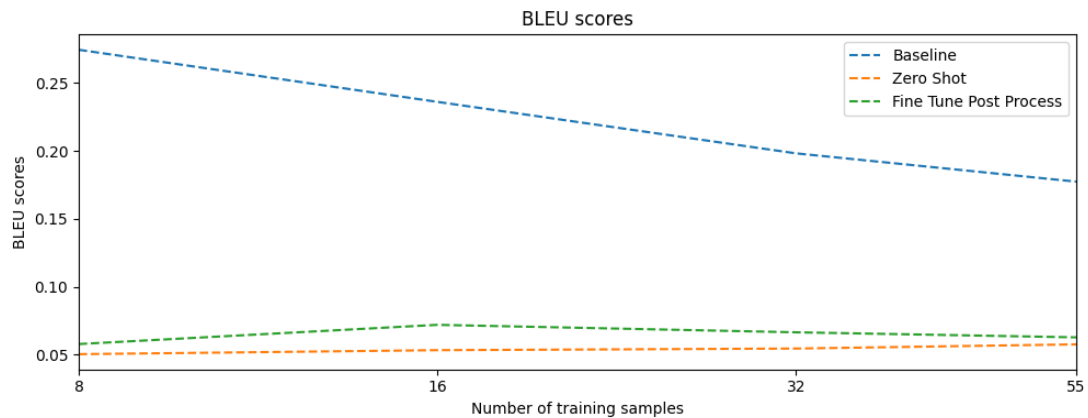
            bleu_score = sum(current_bleu_scores) / len(current_bleu_scores)
            # Store metrics
            return_bleu_scores.append(bleu_score)
    return return_bleu_scores

fine_tune_train_slices = []
for train_slice in train_slices:
    new_train_slice = []
    for element in train_slice:
        if element < len(dataset_fine_tuned["text"]):
            new_train_slice.append(element)
    fine_tune_train_slices.append(new_train_slice)

bleu_scores["Fine Tune Post Process"] =
↳ calculate_bleu_scores_fine_tune(fine_tune_train_slices, dataset_fine_tuned)
```

```
[29]: plot_metrics(bleu_scores, train_samples)
```





### 3 Fine Tuning a requirement conversion in LLaMa 2 with knowledge

In this notebook, we will teach LLaMa 2 an how to convert requirement statements by telling him about RE

#### 3.1 Setup the Model

The following section performs all the setup of the model. This includes

- Installing any dependencies
- Setting any configuration
- Downloading the Base Model

##### 3.1.1 Install dependencies

In order to get started we need to install the appropriate dependencies

```
[ ]: ! source venv/bin/activate

# install dependencies

# we use the latest version of transformers, peft, and accelerate
!pip install -q accelerate peft transformers

# install bitsandbytes for quantization
!pip install -q bitsandbytes

# install trl for the SFT library
!pip install -q trl

# we need sentencepiece for the llama2 slow tokenizer
```

```
!pip install sentencepiece

# we need einops, used by falcon-7b, llama-2 etc
# einops (einsteinops) is used to simplify tensorops by making them readable
!pip install -q -U einops

# we need to install datasets for our training dataset
!pip install -q datasets
```

### 3.1.2 Settings

The following configures our settings for finetuning our model

```
[31]: # The model that you want to train from the Hugging Face hub
model_name = "meta-llama/Llama-2-7b-chat-hf"

# The instruction dataset to use
dataset_name = "Maxnet/requirements-conversion"

# Fine-tuned model name
new_model = "llama-2-7b-req-know"

# Output directory where the model predictions and checkpoints will be stored
output_dir = "./know-results"

[32]: #####
# QLoRA parameters
#####

lora_r = 32 # Defines the size or dimensionality of the vectors used within the
↳ attention computation. A higher value can capture more intricate patterns
↳ but might be computationally expensive and cause overfitting.
lora_alpha = 16 # Helps in managing the magnitude of values.
lora_dropout = 0.2 # 20% of the neurons in LoRA layers would be turned off at
↳ each training step.

#####
# bitsandbytes parameters
#####

use_4bit = True # Reducing to 4-bit precision instead of 32.
bnb_4bit_compute_dtype = "float16" # Less precise than float32 but allows
↳ computations to be faster.
bnb_4bit_quant_type = "nf4" # A specific 4-bit quantization technique
use_nested_quant = False # Activate nested quantization for 4-bit base models
↳ (double quantization)
```

```

#####
# TrainingArguments parameters
#####

num_train_epochs = 20 # number of full passes through the training data
# Enable fp16/bf16 training (set bf16 to True with an A100) used to speed up
↳ training and reduce memory usage
fp16 = False
bf16 = False
# Batch size per GPU for training and evaluation
per_device_train_batch_size = 4 # 4 seems reasonable
per_device_eval_batch_size = 4
# Number of update steps to accumulate the gradients for
gradient_accumulation_steps = 2 # 2 is fine, as we're a small batch
# Enable gradient checkpointing
gradient_checkpointing = True
# Maximum gradient normal (gradient clipping)
max_grad_norm = 0.3 # standard setting
# Initial learning rate (AdamW optimizer)
learning_rate = 2e-4 # standard learning rate
# Weight decay to apply to all layers except bias/LayerNorm weights
weight_decay = 0.001 # standard weight decay
# Optimizer to use
optim = "paged_adamw_32bit" # default optimizer
# Learning rate schedule
lr_scheduler_type = "cosine"
# Number of training steps (overrides num_train_epochs)
max_steps = -1 # needs to be -1, otherwise overrides epochs
# Ratio of steps for a linear warmup (from 0 to learning rate)
warmup_ratio = 0.03 # standard warmup ratio
# Group sequences into batches with same length
group_by_length = True # speeds up the training
# Save checkpoint every X updates steps
save_steps = 0 # we're not gonna save
# Log every X updates steps
logging_steps = 10 # same value as used by Meta

#####
# SFT parameters
#####

# Maximum sequence length to use
max_seq_length = None # no max sequence length
# Pack multiple short examples in the same input sequence to increase efficiency
packing = False # don't need packing
# Load the entire model on the GPU 0

```

```
device_map = {"": 0}
```

### 3.1.3 Download the base model

The following will download the base model, in this case the llama-2-7b-chat-hf model.

```
[ ]: import torch
from transformers import (AutoTokenizer, AutoModelForCausalLM,
    BitsAndBytesConfig, pipeline, logging)

# Load tokenizer and model with QLoRA configuration
compute_dtype = getattr(torch, bnb_4bit_compute_dtype)

# load the quantized settings, we're doing 4 bit quantization
bnb_config = BitsAndBytesConfig(
    load_in_4bit=use_4bit,
    bnb_4bit_quant_type=bnb_4bit_quant_type,
    bnb_4bit_compute_dtype=compute_dtype,
    bnb_4bit_use_double_quant=use_nested_quant,
)

# Check GPU compatibility with bfloat16
if compute_dtype == torch.float16 and use_4bit:
    major, _ = torch.cuda.get_device_capability()
    if major >= 8:
        print("=" * 80)
        print("Your GPU supports bfloat16: accelerate training with bf16=True")
        print("=" * 80)

# Load base model
model = AutoModelForCausalLM.from_pretrained(model_name,
    quantization_config=bnb_config,
                                           device_map=device_map)

# Load base model
model = AutoModelForCausalLM.from_pretrained(
    model_name,
    quantization_config=bnb_config,
    device_map=device_map
)

# don't use the cache
model.config.use_cache = False
model.config.pretraining_tp = 1

# Load the LLaMa tokenizer from the model (llama2)
tokenizer = AutoTokenizer.from_pretrained(model_name, trust_remote_code=True,
    use_fast=False)
```

---

```
tokenizer.pad_token = tokenizer.eos_token
tokenizer.padding_side = "right" # Fix weird overflow issue with fp16 training
```

## 3.2 Run the Model

The following tests the capabilities of the language model prior to fine tuning.

```
[ ]: # Ignore warnings
logging.set_verbosity(logging.CRITICAL)

prompt = "Please transform the following sentence into a requirement statement
↳by segmenting it into the components [Condition], [Subject], [Action],
↳[Object], and \
[Constraint or Action], with the respective labels provided after each part.
↳For example, after the Condition segment, include [Condition]. Make sure the
↳requirement \
ends with a dot. The sentence to be transformed is: "
statement = "When signal x is received, the system shall set the signal x
↳received bit within 2 seconds."

pipe = pipeline(task="text-generation", model=model, tokenizer=tokenizer,
↳max_length=200)
result = pipe(f"<s>[INST] {prompt + statement} [/INST]")
print(result[0]['generated_text'])
```

## 3.3 Train the Model

The following section is about taking your dataset and then finetuning the model

### 3.3.1 Load Dataset

The following code will load your dataset, ready to be fine tuned by the model

```
[21]: from datasets import load_dataset

# Load dataset (you can process it here)
dataset_fine_tune = load_dataset(dataset_name, split="train")
```

Repo card metadata block was not found. Setting CardData to empty.

### 3.3.2 Fine Tune the Model

The following section will take your dataset, and fine tune the model with it.

```
[ ]: import torch
from transformers import (
    AutoModelForCausalLM,
    AutoTokenizer,
    BitsAndBytesConfig,
```

```
HfArgumentParser,
TrainingArguments,
pipeline,
logging,
)
from peft import LoraConfig
from trl import SFTTrainer
import os

# Load LoRA configuration
peft_config = LoraConfig(
    lora_alpha=lora_alpha,
    lora_dropout=lora_dropout,
    r=lora_r,
    bias="none",
    task_type="CAUSAL_LM",
)

# Set training parameters
training_arguments = TrainingArguments(
    output_dir=output_dir,
    num_train_epochs=num_train_epochs,      # uses the number of epochs earlier
    per_device_train_batch_size=per_device_train_batch_size,
    gradient_accumulation_steps=gradient_accumulation_steps,
    optim=optim,
    save_steps=save_steps,
    logging_steps=logging_steps,
    learning_rate=learning_rate,
    weight_decay=weight_decay,
    fp16=fp16,
    bf16=bf16,
    max_grad_norm=max_grad_norm,
    max_steps=max_steps,
    warmup_ratio=warmup_ratio,
    group_by_length=group_by_length,
    lr_scheduler_type=lr_scheduler_type,
    report_to="tensorboard"
)

# Set supervised fine-tuning parameters
trainer = SFTTrainer(
    model=model,
    train_dataset=dataset_fine_tune,
    peft_config=peft_config,                # use our lora peft config
    dataset_text_field="text",
    max_seq_length=max_seq_length,
    tokenizer=tokenizer,                    # use the llama tokenizer
```

---

```

    args=training_arguments,                # use the training arguments
    packing=packing,
)

if not os.path.exists(new_model):
    # Train model
    trainer.train()

    # Save trained model
    trainer.model.save_pretrained(new_model)

```

```

[ ]: from peft import PeftModel

# Reload model in FP16 and merge it with LoRA weights
base_model = AutoModelForCausalLM.from_pretrained(
    model_name,
    low_cpu_mem_usage=True,
    return_dict=True,
    torch_dtype=torch.float16,
    device_map=device_map,
)
model = PeftModel.from_pretrained(base_model, new_model)
model = model.merge_and_unload()

# Reload tokenizer to save it
tokenizer = AutoTokenizer.from_pretrained(model_name, trust_remote_code=True)
tokenizer.pad_token = tokenizer.eos_token
tokenizer.padding_side = "right"

```

```

[23]: model.push_to_hub(new_model, use_temp_dir=False)
       tokenizer.push_to_hub(new_model, use_temp_dir=False)

```

```
tokenizer.model:  0%|          | 0.00/500k [00:00<?, ?B/s]
```

```

[23]: CommitInfo(commit_url='https://huggingface.co/Maxnet/llama-2-7b-req-
know/commit/7b342de74ceda67cba487506d5b60339a8dd41a6', commit_message='Upload
tokenizer', commit_description='',
oid='7b342de74ceda67cba487506d5b60339a8dd41a6', pr_url=None, pr_revision=None,
pr_num=None)

```

### 3.4 Evaluate the model and compare to zero shot

```

[22]: from transformers import pipeline

pipeline = pipeline("text-generation", model="llama-2-7b-req-know")

```

```
2023-12-07 18:59:07.787677: I tensorflow/core/util/port.cc:111] oneDNN custom
operations are on. You may see slightly different numerical results due to
```

```
floating-point round-off errors from different computation orders. To turn them
off, set the environment variable `TF_ENABLE_ONEDNN_OPTS=0`.
2023-12-07 18:59:08.497055: E
tensorflow/compiler/xla/stream_executor/cuda/cuda_dnn.cc:9342] Unable to
register cuDNN factory: Attempting to register factory for plugin cuDNN when one
has already been registered
2023-12-07 18:59:08.497096: E
tensorflow/compiler/xla/stream_executor/cuda/cuda_fft.cc:609] Unable to register
cuFFT factory: Attempting to register factory for plugin cuFFT when one has
already been registered
2023-12-07 18:59:08.497127: E
tensorflow/compiler/xla/stream_executor/cuda/cuda_blas.cc:1518] Unable to
register cuBLAS factory: Attempting to register factory for plugin cuBLAS when
one has already been registered
2023-12-07 18:59:08.793200: I tensorflow/core/platform/cpu_feature_guard.cc:182]
This TensorFlow binary is optimized to use available CPU instructions in
performance-critical operations.
To enable the following instructions: AVX2 AVX512F AVX512_VNNI FMA, in other
operations, rebuild TensorFlow with the appropriate compiler flags.
2023-12-07 18:59:28.007761: W
tensorflow/compiler/tf2tensorrt/utils/py_utils.cc:38] TF-TRT Warning: Could not
find TensorRT

Loading checkpoint shards:  0%|          | 0/2 [00:00<?, ?it/s]
```

```
[34]: prompt = "Please transform the following sentence into a requirement statement,
↳by segmenting it into the components [Condition], [Subject], [Action],
↳[Object], and \
[Constraint or Action], with the respective labels provided after each part.
↳For example, after the Condition segment, include [Condition]. Make sure the
↳requirement \
ends with a dot. The sentence to be transformed is: "
```

```
[35]: import pandas

def fine_tune_pipeline(example):
    statement = example["text"]
    output = pipeline(f"<s>[INST] {prompt + statement} [/INST]")
    example["result"] = output[0]["generated_text"]
    return example

def fine_tune_pipeline_with_cache(example):
    # Check if the result is already in cache
    cache_file = "result_cache_fine_tuned_know.csv"

    cache = pandas.read_csv(cache_file)
    if example["text"] in cache["text"].values:
        return cache[cache['text'] == example['text']]
```



```

# If not in the cache, calculate the result
result = fine_tune_pipeline(example)

# Update the cache DataFrame
cache = pandas.concat([cache, pandas.DataFrame({'text': [example['text']],
↪ 'result': [result]})], ignore_index=True)

# Save the updated cache to the CSV file
cache.to_csv(cache_file, index=False)
return result

dataset_fine_tuned = dataset["test"].map(fine_tune_pipeline_with_cache,
↪ num_proc=1)

```

Map: 0% | 0/27 [00:00<?, ? examples/s]

```

[36]: from nltk.translate.bleu_score import sentence_bleu

def calculate_bleu_scores_fine_tune(train_slices, dataset):
    return_bleu_scores = []
    for train_slice in train_slices:
        threshold = len(dataset["text"])
        for value in train_slice:
            # Get training slice and test data
            dataset_train_sample = dataset.select(train_slice)
            current_bleu_scores = []
            for sample in dataset_train_sample:
                # Tokenize the sentences (split into words)
                reference = nltk.word_tokenize(sample["text"])
                hypothesis = nltk.word_tokenize(sample["result"])

                # Calculate BLEU score
                bleu_score = sentence_bleu([reference], hypothesis)
                current_bleu_scores.append(bleu_score)

            bleu_score = sum(current_bleu_scores) / len(current_bleu_scores)
            # Store metrics
            return_bleu_scores.append(bleu_score)
    return return_bleu_scores

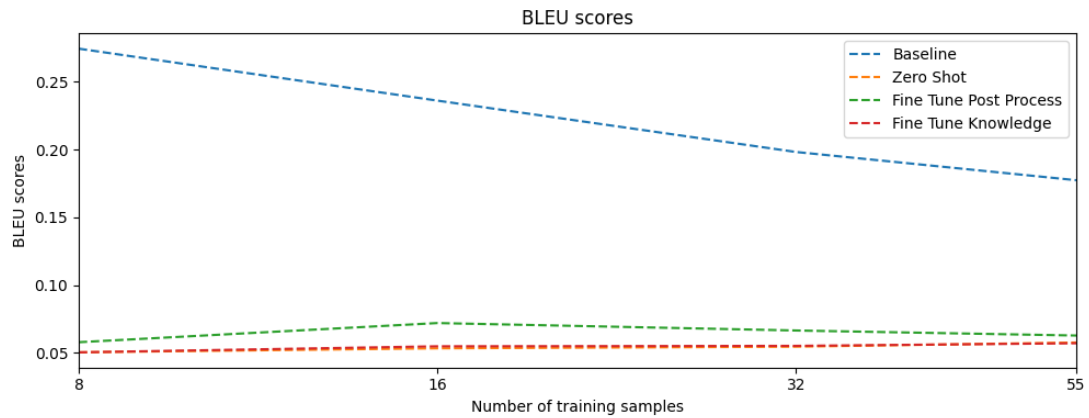
fine_tune_train_slices = []
for train_slice in train_slices:
    new_train_slice = []
    for element in train_slice:

```

```
if element < len(dataset_fine_tuned["text"]):  
    new_train_slice.append(element)  
fine_tune_train_slices.append(new_train_slice)
```

```
bleu_scores["Fine Tune Knowledge"] =   
↳ calculate_bleu_scores_fine_tune(fine_tune_train_slices, dataset_fine_tuned)
```

```
[37]: plot_metrics(bleu_scores, train_samples)
```



### **Declaration**

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature