Automating Forms creation using AI

Master of Science in Technology Thesis

Master's Degree Programme in Information and Communication Technology

Department of Computing, Faculty of Technology

Software Engineering

Author:

Vuyelwa David Ruwodo

Supervisor:

Tuomas Mäkilä

24.06.2024

Turku

Department of Computing, Faculty of Technology
University of Turku

Master of Science in Technology Thesis

Subject: Software Engineering
Author(s): Vuyelwa David Ruwodo
Title: Automating Forms creation using AI
Supervisor(s): Tuomas Mäkilä
Number of pages: 59 pages
Date: 24.06.2024

Abstract.

This study investigates the potential of artificial intelligence (AI) in automating form creation within software development, focusing on the interpretation of functional specification documents for automated form generation. The research addresses the inefficiencies in current form creation practices, particularly for organizations with dynamic requirements and contract programmers facing resource constraints.

The study employs a multifaceted methodology, including a comprehensive literature review on the historical progression of AI in software development and a technological feasibility study. The research explores the capabilities of Large Language Models (LLMs) in formatting raw functional specifications and generating synthetic form components.

Key findings reveal that while LLMs show promise in handling simple form components and small-scale generation tasks, they struggle with complex relationships and full-form generation. The study evaluates various fine-tuning techniques and their effectiveness across different models, highlighting the importance of high-quality, task-specific training data.

Results indicate that while AI demonstrates potential in certain aspects of form creation, current models fall short of producing production-ready code for complex forms. The research also uncovers unexpected performance variations between model sizes and the effectiveness of language-specific models.

This study contributes to the growing body of knowledge on AI-assisted software development, offering insights into the current capabilities and limitations of AI in form creation. It concludes by suggesting future research directions and practical implications for both researchers and practitioners in the field.

Key words: AI, LLM, Automation, Forms

Acknowledgements

I would like to extend my gratitude to several individuals and organizations who played crucial roles in the completion of this thesis. Foremost, I am deeply indebted to Farrokh Mehryary for his invaluable guidance in the technical implementation of this study. His expertise and support were instrumental in navigating the complex aspects of the research.

I am thankful to my supervisor, Tuomas, whose patience, insight, and unwavering support throughout the project were essential to its success.

Special thanks go to Alice and Tanatswa, whose presence and contributions created a conducive and inspiring working environment, fostering productivity and creativity.

I would also like to express my gratitude to the IT Center for Science (CSC) for providing the necessary computational resources to train the models used in the latter part of my thesis. Their support was crucial in enabling the advanced analytical work required for this research.

In the spirit of transparency and embracing modern research tools, I acknowledge the use of AI-assisted writing tools in the preparation of this thesis. These tools were employed to refine and enhance various sections, contributing to the overall clarity and coherence of the document. This approach reflects the evolving landscape of academic writing and research methodologies.

Contents

1 Introduction

The evolution of software development has been characterized by a continual search for efficiency and effectiveness in various processes. Form creation, an essential aspect of software development, traditionally involves repetitive and manual coding practices. Historically, organizations with static form-creation processes have combated this inefficiency by developing custom automation tools, while organizations and contact programmers with ever-changing form-creation processes grapple with resource constraints that limit their ability to do the same. The advent of artificial intelligence (AI) in coding, particularly with tools like ChatGPT, has opened up new possibilities for automating routine tasks, including form creation. However, the author acknowledges that using AI may not always be the fastest or best approach for organizations with static form creation methods, as they can develop a traditional tool for form creation once and use that continuously. This study sets the stage for exploring how AI can revolutionize this aspect of software development.

1.1 Problem statement

Current practices in form creation can be inefficient, especially for companies with ever-changing form creation methods who face unique challenges. These companies are in a constant state of flux, which makes the development of dedicated form-creation automation tools unfeasible. For contract programmers, the transient nature of their projects means that creating new tools for each project is impractical. The potential of AI in automating form creation, particularly through understanding and interpreting functional specifications, has not been fully explored. This gap presents an opportunity to enhance efficiency, adaptability, and scalability in software development processes.

1.2 Research objectives

The primary objectives of this research are to **explore the potential of artificial intelligence (AI) in automating the process of form creation** and to **evaluate the feasibility and effectiveness of using AI to interpret functional specification documents for automated form generation**. This study aims to investigate how AI technologies can streamline the development of forms by reducing manual effort and increasing accuracy. By assessing the capability of AI to understand and process functional specifications, the research seeks to determine the practicality of deploying AI-driven solutions in real-world scenarios, ultimately enhancing efficiency and productivity in form generation tasks.

1.3 Research Questions

1. How has artificial intelligence evolved over the years in the field of software development?
2. How effective is artificial intelligence, particularly LLMs like ChatGPT, in automating the process of form creation by interpreting and generating code from functional specification documents?

These research questions are designed to cover both the historical evolution and the current practical application of AI in software development. This dual focus ensures that the study provides a comprehensive understanding of how AI has reached its current state and how effectively it can be applied to specific tasks within the field. By addressing both broad and specific aspects, the research can offer valuable insights for both academic inquiry and practical implementation.

Research question 1 aims to provide a historical overview and contextual background for the study. Understanding the evolution of AI in software development is crucial for several reasons:

- **Historical Context:** It allows the researcher to trace the advancements and milestones in AI, highlighting how these have progressively contributed to current capabilities. This historical perspective can help identify the technological trends and pivotal moments that have shaped the field.
- **Technological Progress:** By examining the development of AI technologies over time, the study can identify key innovations and breakthroughs that have shaped modern AI applications. This can include understanding the transition from rule-based systems to machine learning and then to advanced neural networks.
- **Foundation for Current Capabilities:** The historical perspective helps to understand the foundation upon which current AI technologies, including those used for code generation, are built. This can elucidate the factors that have enabled recent advancements, such as

improvements in computational power, algorithmic developments, and the availability of large datasets.

- **Informing Future Directions:** Insights into the evolution of AI can provide valuable lessons and guide future research and development in AI-driven software tools. By understanding the successes and challenges of the past, researchers and practitioners can better navigate future innovations and avoid repeating past mistakes.

Research Question 2 targets the core of the study's practical investigation, focusing on the application and evaluation of modern AI technologies in the form creation process, this inquiry is crucial for several reasons:

- **Practical Application:** Evaluating the effectiveness of AI in automating form creation addresses a concrete and significant task within software development. This has direct implications for improving efficiency and productivity in the field. By automating routine and repetitive tasks, developers can focus on more complex and creative aspects of software development.
- **Evaluating AI Capabilities:** Assessing how well AI can interpret and generate code from functional specifications provides valuable insights into the current state of AI technology. This evaluation can help determine the practical limits of AI, identifying areas where it excels and where it may still fall short.
- **Evaluating Real-World Impact:** Assessing AI's ability to interpret and generate code from functional specifications provides insights into its real-world applicability. This helps to determine whether AI can effectively replace or augment human effort in routine tasks, leading to potential cost and time savings. By understanding the practical impact of these technologies, organizations can make informed decisions about adopting AI tools.
- **Guiding Future Development:** The findings can inform developers and organizations about the practicality of adopting AI tools for form creation. This guidance can help direct investments and development efforts towards the most promising areas of AI technology, ensuring that resources are used effectively to enhance software development processes.

1.3 Methodology

This study will employ a comprehensive methodological approach, commencing with a Literature Review to explore the historical evolution of artificial intelligence (AI) in response to the first research question. The literature review aims to elucidate the current landscape of AI and inform decisions regarding its application in the form creation process. Following this, Quantitative Analysis will be conducted to evaluate the feasibility and efficacy of using AI to interpret functional specification documents for automated form generation, addressing the second research question. The effectiveness of this methodological approach has been demonstrated in prior studies evaluating large language models (LLMs), as evidenced by [1]. This approach aligns with standard practices in the evaluation of LLMs, as observed in [2], [3].

1.4 Contribution

This study holds significant implications for the field of software development. By exploring the integration of AI in form creation, it aims to introduce a method that could potentially save time, reduce costs, and increase adaptability for various stakeholders, particularly companies and contract programmers with non-static form creation methods. The findings could pave the way for more extensive use of AI in software engineering, leading to broader transformations in how software is developed. Furthermore, the study could contribute valuable insights to the ongoing discourse on the role of AI in automating and optimizing routine tasks in software development. For the tech industry at large, the implications of this research include potential shifts in best practices and the standardization of more efficient methodologies.

2 History of AI

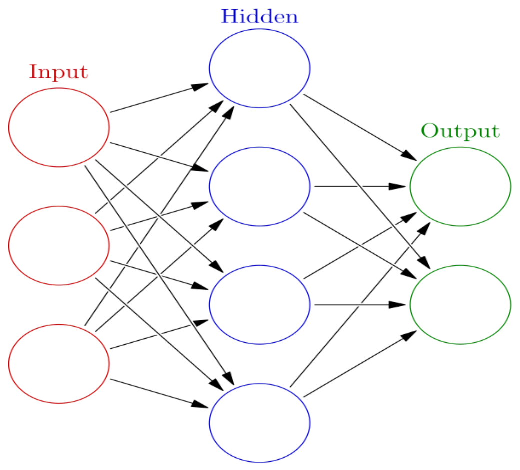
This chapter delves into the rich history of artificial intelligence (AI), tracing its development from early conceptualizations to modern advancements. It covers the initial theoretical foundations laid by classical philosophers, the seminal contributions of pioneers like Alan Turing, the evolution from rule-based systems to neural networks, and the significant milestones that have shaped the field. The chapter also discusses the different types of AI, highlighting their capabilities and the ongoing challenges in achieving more advanced forms of intelligence.

2.1 Early Concepts and Theoretical Foundations (Rule-Based Systems)

“AI is defined as machine intelligence or intelligence demonstrated by machines, in contrast to the natural intelligence displayed by humans” [4]. The seeds of AI can be traced back to classical philosophers and their efforts to understand human thinking as a symbolic system, but its formal inception is often attributed to the mid-20th century. In 1950, Alan Turing, often referred to as the "father of computer science" [5], published a seminal paper titled "Computing Machinery and Intelligence" [6]. In this paper, he proposed the idea of a machine that could simulate human intelligence and introduced the famous Turing Test as a criterion for machine intelligence. This period also saw the development of foundational concepts like the “Logical Theorist" [7], recognized as the first AI program.

Logical Theorist was a rule-based AI system developed to prove mathematical theorems using a set of rules representing logical relationships between different mathematical concepts. Logic Theorist was a landmark achievement in AI, however, it was also limited by its reliance on hand-coded rules which made it brittle and difficult to adapt to new problems. Rule-based systems like Logic Theorist are effective for tasks that can be clearly defined by rules, but they can be inflexible and difficult to maintain as new knowledge and requirements emerge. This is because hand-coding rules is a time-consuming and error-prone process, and it can be challenging to anticipate all possible scenarios that the system may encounter.

2.2 Emergence of Neural Networks



Picture 1: Artificial neural network architecture [8]

While the early focus was on rule-based systems, a transformative shift occurred with the exploration of artificial neural networks (ANN), inspired by the structure of the human brain. This marked a paradigmatic evolution in the field of AI, steering away from deterministic approaches toward more adaptive and dynamic systems. [9]

In 1957, Frank Rosenblatt introduced a groundbreaking concept known as "The Perceptron" [10]. This simple yet revolutionary model simulated an artificial neuron capable of learning to classify patterns. The Perceptron achieved this by adjusting its weights based on the input data it received. This innovation laid the foundation for ANN development, opening doors to more complex architectures that could learn and adapt autonomously.

The foundational concepts, exemplified by the “Logical Theorist" [7] , remained crucial during this period. However, researchers began envisioning systems that could learn and adapt more dynamically, mimicking the plasticity of the human brain. The Perceptron was a pivotal milestone, showcasing the potential for machines to autonomously learn and improve their performance over time.

This transition from rigid rule-based systems to the flexible learning capabilities of ANNs marked a profound shift in AI's trajectory. While rule-based systems like the Logical Theorist were effective for clearly defined tasks, they faced limitations in adaptability. The emergence of

ANNs addressed these limitations, offering a more sophisticated approach to problem-solving by enabling systems to learn and generalize from data.

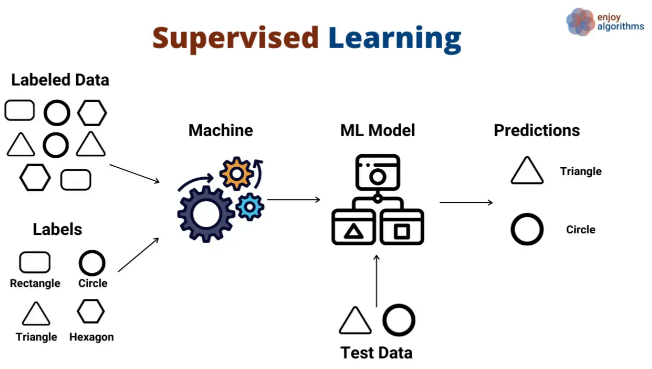
This period of exploration in the late 1950s laid the groundwork for subsequent advancements in ANN architectures, setting the stage for the dynamic and ever-evolving landscape of artificial intelligence.

2.3 The Birth of AI Research and Early Challenges

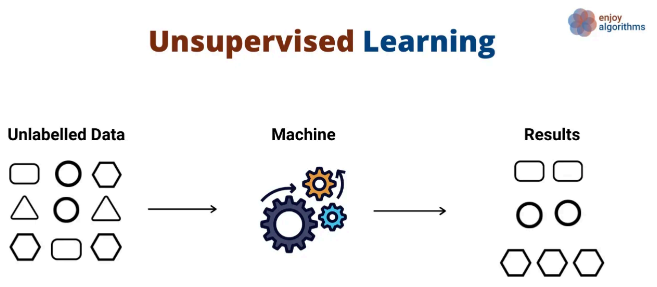
The Dartmouth Conference in 1956, often cited as the birth of AI as a field [11], brought together researchers who believed that machines could be made to simulate aspects of human intelligence. However, the initial optimism of the 1960s faced a reality check in the 1970s and early 1980s, a period known as the "AI winter," characterized by reduced funding and interest in AI research due to unmet expectations [12].

2.4 The Revival and Rise of Machine Learning

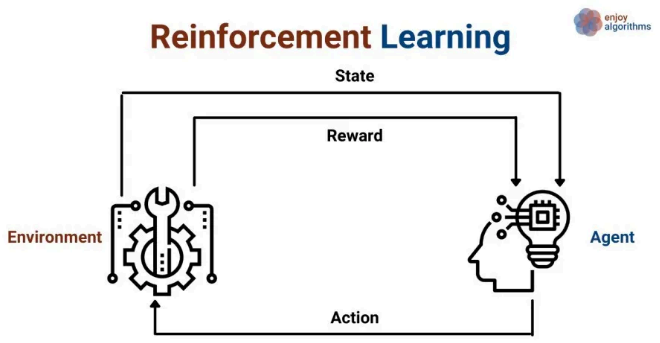
The revival of AI in the late 1980s and early 1990s is attributed to the emergence of machine learning (ML), a subfield of AI that is defined as the “study of algorithms that enable computer systems to learn through experience”. ML shifted focus from rule-based algorithms like Logical Theorist to learning-based algorithms like The Perceptron. ML algorithms build a model based on sample data, known as "training data", to make predictions or decisions without being explicitly programmed to do so.”[4]. Machine learning techniques can be broken down into 3 categories [13]:



Picture 2: Supervised Learning [14]



Picture 3: Unsupervised Learning [14]



Picture 4: Reinforcement Learning [14]

- **Supervised learning:** This type of learning is where the machine is given labeled data, and it learns to map inputs to outputs. For example, a machine learning model could be trained to classify images of cats and dogs by being shown a large number of images that have already been labeled as either cats or dogs.

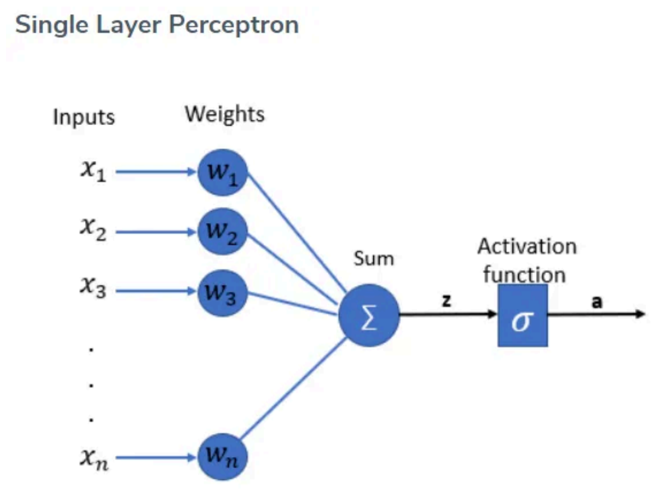
- **Unsupervised learning**: This type of learning is where the machine is given unlabeled data, and it learns to identify patterns in the data. For example, a machine learning model could be trained to cluster documents together based on their content without being given any specific labels for the documents.
- **Reinforcement learning**: This type of learning is where the machine learns by interacting with its environment and receiving rewards or punishments for its actions. For example, a reinforcement learning model could be trained to play a game of chess by being given feedback on its moves.

2.5 Deep Learning

Machine learning gave rise to a new field, “Deep learning” [15]. This subfield utilizes artificial neural networks (ANNs), complex architectures composed of multiple layers of interconnected nodes. To train these models, a process called “backpropagation” [16] is employed. This algorithm adjusts the weights and connections between neurons to minimize the discrepancies between the model's predictions and the actual data.

Over time, various deep learning models have emerged such as:

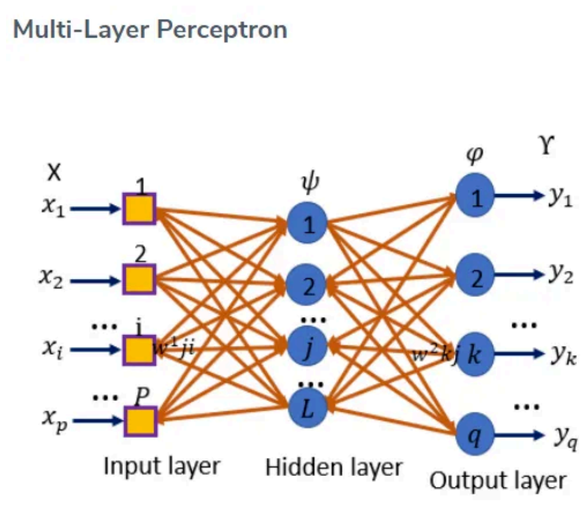
2.5.1. Perceptron (1957)



Picture 5: Single layer Perceptron [17]

The Perceptron was the first artificial neuron, designed by Frank Rosenblatt in 1957 [10]. It was a simple model with a single layer of neurons that used a threshold function to classify binary data.
The Perceptron was able to learn to classify data by adjusting the weights of its connections. However, it was limited in its ability to handle complex data, as it could only learn linear decision boundaries.
The Perceptron's linear decision boundaries made it difficult to classify data that was not linearly separable. This led to limitations in its applicability to real-world problems.

2.5.2. Multilayer Perceptron (MLP) (1962)



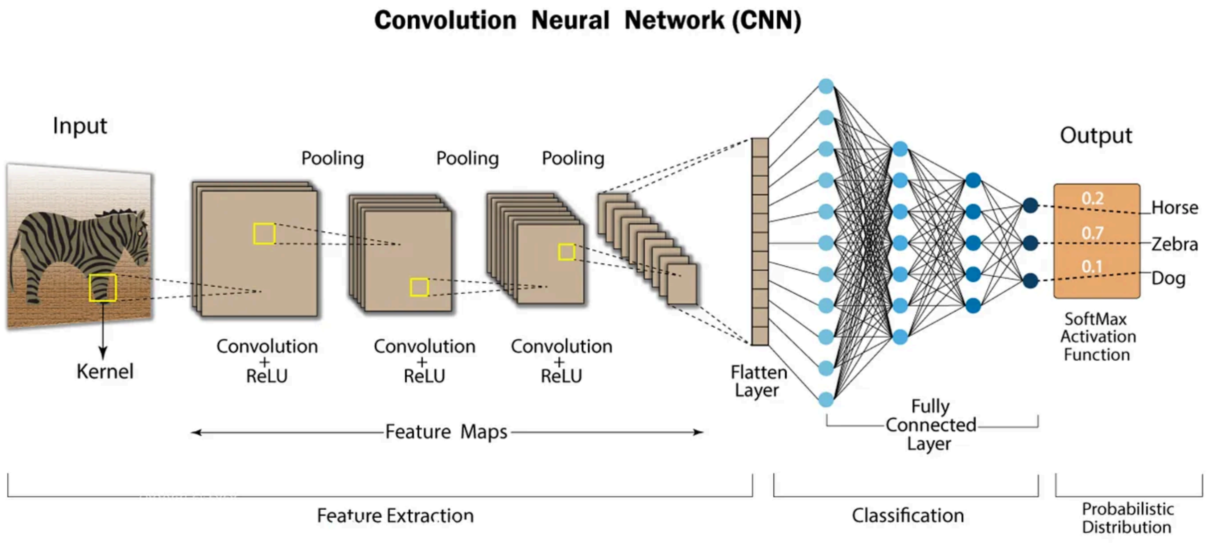
Picture 6: Multilayer Perceptron [17]

The Multilayer Perceptron (MLP) was an extension of the Perceptron that introduced multiple layers of neurons [18]. This allowed MLPs to learn to classify data with more complex decision boundaries. The MLP added more layers of neurons and used activation functions to introduce non-linearity into the model. This allowed MLPs to learn more complex decision boundaries and classify multi-class data. While MLPs were an improvement over Perceptrons, they still struggled with complex data due to the vanishing gradient problem. This problem made it difficult for MLPs to learn long-range dependencies in the data.

2.5.3. Feed-forward Neural Networks (FNNs) (1970s)

Feed-forward Neural Networks (FNNs) emerged as a response to the limitations of Perceptrons and MLPs. FNNs were designed to process data in a single pass, without any feedback loops or recurrent connections. This allowed them to learn more complex patterns in data but still limited their ability to capture temporal relationships. FNNs were characterized by their use of multiple layers of artificial neurons, with each layer processing the output from the previous layer. They were widely used for tasks such as image classification, speech recognition, and language translation. FNNs differed from the Perceptron in their ability to learn more complex patterns in data but were still limited in their ability to handle sequential data. [19]

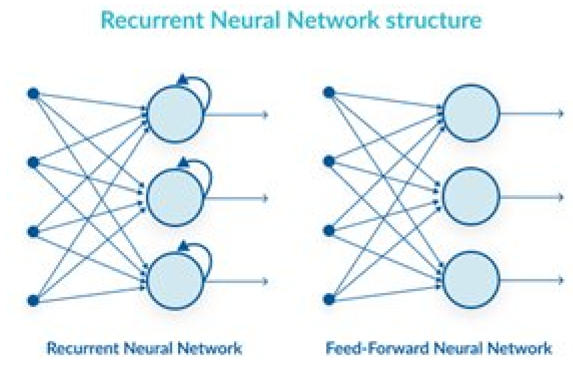
2.5.4. Convolutional Neural Network (CNN) (1980s)



Picture 7: Convolutional Neural Network [20]

The Convolutional Neural Network (CNN) was specifically designed for image recognition tasks. It used a specialized architecture that allowed it to extract features from images efficiently. CNNs used convolutional layers and pooling layers to extract features from images. These layers were designed to capture local patterns and reduce the dimensionality of the data.
While CNNs were very effective for image recognition, they were not well-suited for sequential data, such as natural language. This is because CNNs were not designed to capture temporal relationships in data.[21]

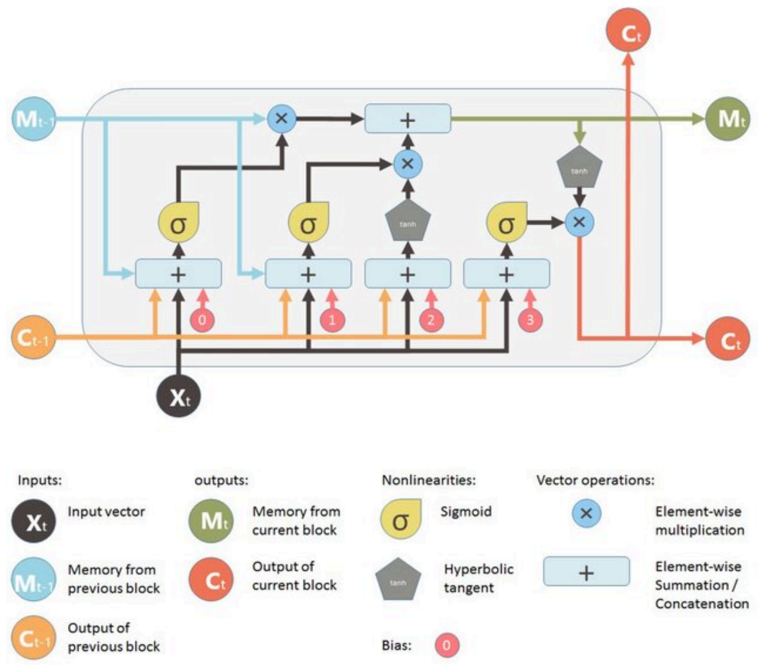
2.5.5. Recurrent Neural Network (RNN) (1980s)


Recurrent Neural Network structure
Recurrent Neural Network
Feed-Forward Neural Network

Picture 8: Recurrent and Feed-Forward Neural Networks [22]

The Recurrent Neural Network (RNN) was designed to process sequential data, such as natural language. It used a feedback loop to allow it to retain information over time. RNNs used a recurrent structure that allowed them to process sequences of data one element at a time. This allowed them to capture temporal relationships in the data [23]. Before RNNs neural networks used a Feed-forward architecture. RNNs were still affected by the vanishing and exploding gradient problems [24], [25]. These problems made it difficult for RNNs to learn long-range dependencies in the data.

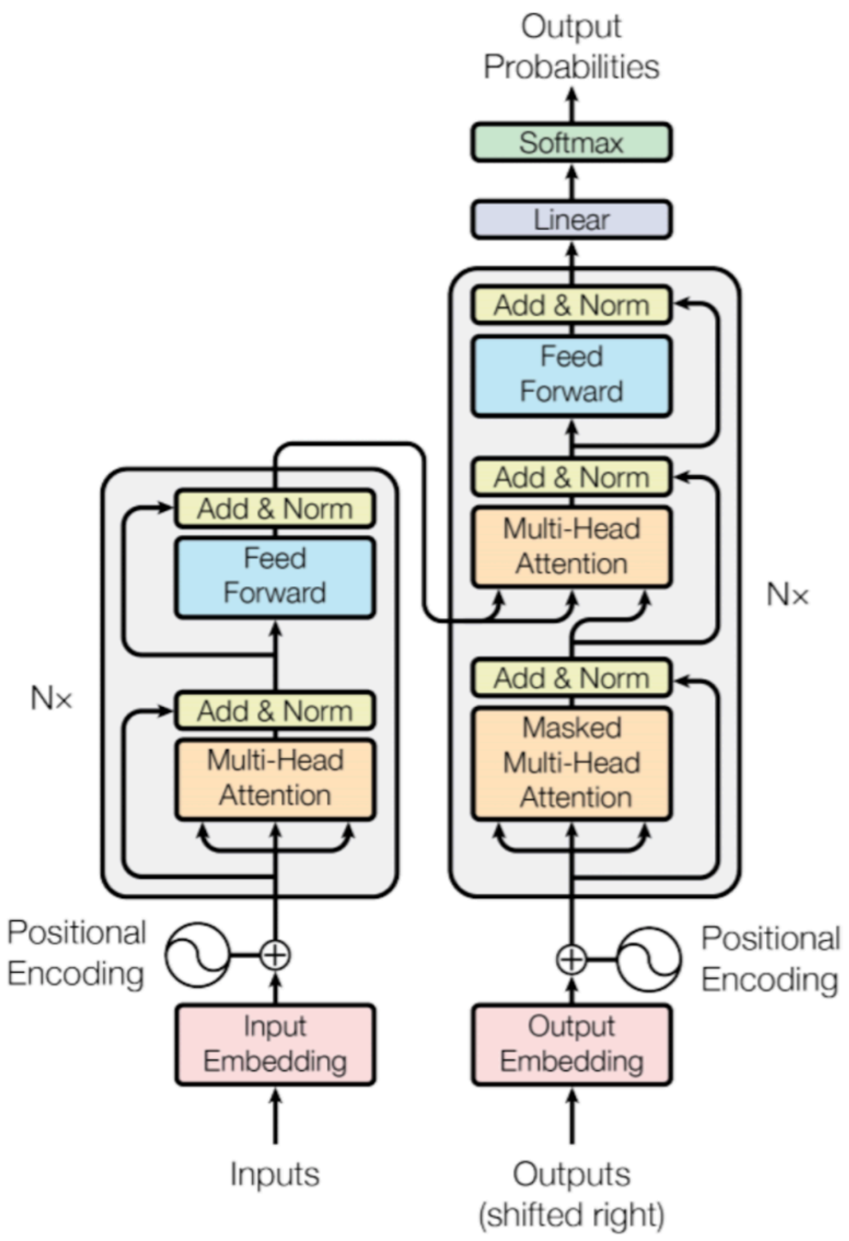
2.5.6. Long Short-Term Memory (LSTM) (1997)


Inputs:
X_t Input vector
M_{t-1} Memory from previous block
C_{t-1} Output of previous block
outputs:
M_t Memory from current block
C_t Output of current block
Nonlinearities:
σ Sigmoid
tanh Hyperbolic tangent
Bias: 0
Vector operations:
× Element-wise multiplication
+ Element-wise Summation / Concatenation

Picture 9: Long Short-Term Memory architecture [26]

The Long Short-Term Memory (LSTM) network was developed to address the vanishing and exploding gradient problems in RNNs. It uses an internal memory cell to store information over time. LSTMs used gating mechanisms to control the flow of information through the network. These gates allowed LSTMs to selectively remember or forget information over time [27]. LSTMs were more computationally expensive than RNNs due to their complex architecture. This made them less suitable for training on large amounts of data.

2.5.7. Transformer (2017)



Picture 10: Transformer architecture [28]

The Transformer was a new architecture for natural language processing that used an attention mechanism to focus on specific parts of the input sequence when generating each output token. The attention mechanism allowed the Transformer to capture long-range dependencies in the input sequence without being affected by the vanishing and exploding gradient problems [28].

The Transformer was more computationally expensive than RNNs and LSTMs due to the complexity of its attention mechanism. This made it less suitable for training on small amounts of data.

2.5.8. Next-Generation Neural Networks (2020s)

The next generation of neural networks, such as Graph Neural Networks (GNNs) [29], Vision Transformers (ViTs) [30], Multimodal Transformers (MMTs) [31], and Bidirectional Encoder Representations from Transformers (BERT) based models [32], have been developed to address the limitations of previous architectures. GNNs are designed to process graph-structured data, while ViTs and MMTs are designed to process visual and multimodal data, respectively. BERT-based models, such as RoBERTa and DistilBERT, have been developed to improve upon the performance of language models on a range of NLP tasks. These architectures use novel techniques, such as graph attention, multimodal fusion, and masked language modeling, to capture complex patterns in data. They have been shown to achieve state-of-the-art performance on a range of tasks, including graph classification, image classification, multimodal language translation, and question answering.

The mainstream adoption of AI for image and text generation further propelled the field into prominence. Models such as the Transformer revolutionized how we generate realistic images and coherent text. This adoption expanded AI's reach into creative domains, enabling applications such as art generation, content creation, and even deepfake technology. With the ability to produce convincing visual and textual content, AI's impact on various industries and everyday life became even more pronounced, marking a significant shift in how we interact with and perceive artificial intelligence.

2.6 Types of AI

In the realm of AI, understanding the different types of AI is crucial for grasping the breadth and potential of the field. **Narrow** or Weak AI, the most commonly implemented form, is designed for specific tasks such as language translation, code generation, or driving autonomous vehicles, and does not possess consciousness or self-awareness. In contrast, **General** or Strong AI, which remains largely theoretical, refers to AI systems capable of generalized human cognitive abilities, implying the ability to reason, solve problems, and understand abstract concepts across diverse domains [6], [33], [34]. The most speculative and advanced concept is **Artificial Superintelligence** (ASI), a form of AI that surpasses human intelligence in all aspects, including creativity, general wisdom, and problem-solving [6]. While Narrow AI has seen significant practical application, the development of General AI and ASI remains a combination of theoretical research and aspirational goals, encompassing various ethical, technical, and existential considerations [35], [36].

3 AI in Software Development

In recent years, the application of artificial intelligence (AI) in software development has gained significant attention and has been the focus of numerous research studies. One area of particular interest is the use of pre-trained models (PTMs) in software engineering (SE) tasks. “Instead of training a model from scratch with large amounts of data, human beings can learn to solve new problems with very few samples” [37]. Machines can do the same using transfer-learning, a 2 phase learning framework, Pre-trained Models (PTMs) in the context of software development are trained through a meticulous process that enables them to understand and manipulate code effectively. This training process is crucial for preparing PTMs to handle a wide array of software engineering tasks, leveraging vast amounts of code data to learn patterns, syntax, semantics, and the intricacies of various programming languages.

3.1. Data Collection and Preprocessing

The first step in training PTMs for code understanding involves collecting a large dataset of source code. This dataset typically includes code from open-source projects, software repositories, and other coding platforms. The collected code spans multiple programming languages and domains, providing a rich diversity that helps in building a robust model.

Once the data is collected, it undergoes preprocessing to standardize and clean it. This may involve [38]:

- Removing redundant or non-informative parts of the code, such as comments and white spaces.
- Normalizing variable names and other identifiers to reduce the variability across different coding styles.
- Pre-tokenization
- Tokenizing the code into smaller units (tokens) that can be processed by the model.

3.2 Normalization

Normalization entails processes designed to standardize textual data [39]. This includes:

- **Removal of unnecessary whitespace**: Ensuring that spacing in the text does not interfere with tokenization.
- **Case conversion**: Typically to lowercase to maintain consistency across variants of the same word (e.g., "Hello" vs. "hello").
- **Accent removal**: Stripping characters of diacritical marks to reduce the vocabulary size and simplify text processing.
- **Unicode normalization:** Applying strategies like NFC (Normalization Form Canonical Composition) or NFKC (Normalization Form Compatibility Composition) to ensure consistent representation of characters in texts.

3.3 Pre-tokenization

Following normalization, pre-tokenization involves the segmentation of text into manageable entities, typically words, before they undergo subword division [40]. This step often utilizes simple strategies like splitting by whitespace and punctuation. For example, the pre-tokenization in a BERT tokenizer focuses on white spaces and punctuation marks for splitting, while GPT-2 includes whitespace as a token (Ġ), preserving the spaces to aid in reconstructing the original text during decoding.

Notably, different tokenizers may follow distinct rules [41]:

- **BERT Tokenizer:** Operates on white spaces and punctuation, standardizing spacing but removing extra spaces.
- **GPT-2 Tokenizer**: Retains spacing information by using a special symbol, and does not remove double spaces.
- **T5 Tokenizer**: Utilizes SentencePiece, which treats spaces as a special character (_), and does not split on punctuation. It adds a space token at the start by default and manages spaces effectively between words.

3.4 The transition from Pre-tokenization to Tokenization

Following the initial pre-tokenization steps where text is cleaned and segmented into preliminary entities like words, the subsequent phase involves applying more refined tokenization techniques. These strategies are crucial in breaking down the text into tokens that are not only manageable but also meaningful for computational models to process effectively. Among the prominent techniques employed are Sentencepiece Tokenization, Byte-Pair Encoding (BPE) Tokenization, WordPiece Tokenization, and Unigram Tokenization, each tailored to optimize different aspects of language modeling and vocabulary management. [42], [43], [44], [45]

3.4.1 SentencePiece

SentencePiece is a versatile tokenization tool designed primarily for use in neural machine translation (NMT) systems, which often grapple with the challenge of open vocabulary problems. Unlike traditional tokenization methods, SentencePiece offers a comprehensive solution by providing a pre-tokenization-free framework that treats text as a raw sequence of Unicode characters piece [43]. This novel approach allows for seamless application across different languages, especially those without clear word delimiters like Chinese and Japanese.[43]

SentencePiece provides [46]:

- **Language-Independent Design**: SentencePiece processes text as a sequence of raw Unicode characters. This makes it universally applicable to any language, without requiring language-specific rules or pre-tokenization (e.g., Moses tokenizer for segmented languages).
- **Integration of Established Subword Algorithms**: SentencePiece supports the implementation of both the byte-pair encoding (BPE) and the unigram language model as core algorithms for token generation. These methods allow the tokenizer to optimize vocabulary size effectively while handling linguistic variations within text.
- **Subword Regularization** Techniques: SentencePiece enhances the robustness and generalization capabilities of models through subword regularization and BPE-dropout. These techniques involve on-the-fly adjustment of subword segmentations during

training, adding a stochastic element to token generation, which prevents models from overfitting on particular tokenization patterns.

- **Efficiency and Performance**: The tool is designed to be both fast and lightweight. It can process approximately 50k sentences per second with a memory footprint of only around 6MB, making it ideal for large-scale text processing in resource-limited environments.
- **Fixed Vocabulary Size**: SentencePiece is unique in its approach to fixing the vocabulary size prior to model training. This contrasts with other subword tokenization methods that typically adjust their vocabulary based on the number of merge operations performed. SentencePiece defines a target vocabulary size (e.g., 8k, 16k, 32k tokens), enabling precise control over the complexity and capacity of the NMT model.
- **Consistent Tokenization and Detokenization**: Using the same model file guarantees consistency in the tokenization and detokenization processes. This feature ensures that the original text can be accurately reconstructed from its tokenized form, fostering reliable evaluation and deployment of trained models.

SentencePiece addresses a common issue in text tokenization where whitespace information often gets lost. By converting whitespace into a visible character ("_", U+2581), ensures that spaces are treated as explicit tokens and preserves the original text structure. This approach allows for reversible tokenization, where tokenized sequences can be perfectly converted back to the original text.

Example:

Before: "Hello World."

After Tokenization: "[Hello] [_Wor] [ld] [.]"

After Detokenization: "Hello World." (reconstructed accurately)

Directly from raw text, SentencePiece can generate sequences of vocabulary IDs, streamlining the processing pipeline for NMT models by eliminating intermediate steps typically required by other tokenization approaches. The design and operation of SentencePiece make it particularly advantageous for neural machine translation applications, where consistent and flexible handling of diverse linguistic inputs is crucial. By detaching the tokenization process from the constraints of predefined word boundaries and language-specific idiosyncrasies, SentencePiece empowers developers to implement more effective and inclusive NMT solutions. Its ability to handle text as purely Unicode sequences and to maintain reversible transformations stands out as a major advancement over traditional tokenization methods, paving the way for more adaptable and powerful language technology tools.

3.4.2 Byte-Pair Encoding (BPE) Tokenization

Byte-pair encoding originated as a data compression technique and has been adeptly repurposed for NLP tasks. BPE iteratively merges the most frequent pairs of bytes or characters until a set vocabulary size is reached. This method effectively handles unknown words by decomposing them into recognizable subwords, thus enhancing the model's ability to generalize across varied textual inputs. It is particularly beneficial when addressing large and complex vocabularies within texts. [47]

The training process of BPE involves an initial examination of the text corpus, which, following normalization and pre-tokenization, issues a basic vocabulary of unique symbols or characters. For instance, let's consider the corpus contains words like "hug", "pug", "pun", "bun", and "hugs". The foundational vocabulary derived would consist of single characters: ["b", "g", "h", "n", "p", "s", "u"]. [48]

If a token or character that was absent in the training corpus appears during tokenization, it is regarded as an unknown token ([UNK]). This often occurs with unique or less frequent characters, such as emojis, which can cause challenges when analyzing certain texts.

An innovative aspect of BPE used in the GPT-2 and RoBERTa tokenizers is byte-level BPE. Instead of interpreting words as sequences of Unicode characters, byte-level BPE considers them sequences of bytes. This approach ensures every conceivable character is accounted for in the base vocabulary, preventing them from being transformed into unknown tokens. The base vocabulary in this method typically consists of 256 characters, corresponding to the byte values.

Post-base vocabulary setup, BPE learns new tokens by merging the most frequent pairs of characters or existing tokens. Initially, frequent bi-character tokens are formed and as the process iterates, longer subwords emerge from these pairs. The algorithm focuses on identifying the most frequent adjacent tokens ("pairs") within words, merging them to form a new token, which is then included in the vocabulary.

Example Walkthrough:

Using the corpus example ["hug", "pug", "pun", "bun", "hugs"] with identified frequencies:

Initial tokenization by characters:

("h" "u" "g", 10), ("p" "u" "g", 5), ("p" "u" "n", 12), ("b" "u" "n", 4), ("h" "u" "g" "s", 5)

Let's consider pair analysis:

The pair ("u", "g") appears most frequently (20 times in our corpus). Thus, the first merge rule learned is ("u", "g") -> "ug".

This merging process continues, identifying and merging the next frequent pair. For instance, the next could be ("u", "n"), which after merging would yield:

Vocabulary and corpus update: Vocabulary: ["b", "g", "h", "n", "p", "s", "u", "ug", "un"]

The iterative process is continued, progressively merging pairs until the desired vocabulary size is achieved or all frequent pairs have been merged.

The actual tokenization of new inputs based on the established merge rules:

1. **Normalization**: Text is cleaned up, removing unwanted characters and inconsistencies.
2. **Pre-tokenization**: Text is split into words or other significant components.
3. **Character Splitting:** Words are further broken down into individual characters as per the initial vocabulary.
4. **Merge Rule Application:** Sequentially apply the learned merge rules to the characters, grouping them into the learned tokens.

Example Tokenization

Using the word "bug" with the trained merge rules:

- Initial split into characters: ["b", "u", "g"]
- Applying a merge rule: ("u", "g") -> "ug"
- Final tokenization: ["b", "ug"]

This systematic tokenization process, grounded in basic characters and explicit merge rules, allows BPE to be highly efficient and broadly applicable, especially in dealing with diverse and extensive vocabularies. Byte-pair encoding thus serves as a robust method for addressing various challenges in NLP, from handling unseen words to efficiently managing large text corpora.

3.4.3 WordPiece Tokenization

Building on the foundational ideas of tokenization, WordPiece further optimizes the process by focusing not just on frequency but also on the likelihood of language models. Starting from basic characters, WordPiece merges them into tokens based on their contribution to the overall probability of the text structure, allowing for a nuanced understanding of word composition and usage contexts. This technique is integral to the functioning of models like Google's BERT, optimizing both the coverage of the vocabulary and the depth of semantic recognition.

Though Google has not open-sourced the exact implementation details of WordPiece, what's understood about the algorithm is largely derived from the literature. The process can be delineated as follows:

Initialization: Begins with a small vocabulary that includes special tokens used by the model and a set of basic characters from the initial alphabet. For BERT, WordPiece utilizes a prefix (like "##" for subsequent chunks of a word after the first character) to distinguish subsequent characters within words. For example, the word "word" would be segmented as:

"w ##o ##r ##d"

Learning Merge Rules: Similar to BPE, WordPiece then learns to merge these characters and subwords. However, unlike BPE’s frequency-based approach, WordPiece employs a scoring system for pairs of tokens using the formula:

Score= freq_of_pair/(freq_of_first_element×freq_of_second_element)

This scoring technique means that merges are favored not only based on raw frequency but also adjusted against the prevalence of the individual tokens being merged. This criterion aims to balance the coverage and cohesion of the vocabulary.

Example Scenario [49]:

Consider the tokens following initial segmentation and their frequencies:

("hug", 10), ("pug", 5), ("pun", 12), ("bun", 4), ("hugs", 5)

Splitting these into characters with manipulations for internal parts of words marked by prefixes:

("h" "##u" "##g", 10), ("p" "##u" "##g", 5), ("p" "##u" "##n", 12), ("b" "##u" "##n", 4), ("h" "##u" "##g" "##s", 5)

The initial vocabulary would be: ["b", "h", "p", "##g", "##n", "##s", "##u"]. If for example, the merge "##g" and "##s" has the highest score (not because of their individual frequency but due to the lower occurrence of these pairs making their relative score higher), it becomes the first merge to form "##gs".

High-frequency tokens ("##u", "##g"):

- Frequency of pair ("##u", "##g") = 20
- Frequency of "##u" = 31 (total appearances in pairs)
- Frequency of "##g" = 20

The score calculation would be: Score = 20/(31*20) = 20/620 = 1/31

All pairs containing "##u" have this score due to its high frequency in various tokens.

Lower-frequency tokens ("##g", "##s"):

Frequency of pair ("##g", "##s") = 5

Frequency of "##g" = 20

Frequency of "##s" = 5

The score for this pair would be: Score = 5/(20*5) = 5/100 = 1/20

Since this score is higher than the score for pairs involving "##u," the merge between "##g" and "##s" is prioritized. The effective scoring formula reduces the bias towards merging extremely frequent sub-tokens, which might otherwise dominate the vocabulary, leading to less efficient token utilization.

As merges continue, tokens like "hu", "hug", etc., may eventually be created based on the scoring criterion, incorporating sequences that best describe common linguistic patterns while promoting an efficient vocabulary.

The learning process involves repetitively merging according to the best score until the desired vocabulary size is reached.

Once trained, WordPiece tokenizes new text inputs by:

- Normalizing and pre-tokenizing the text.
- Splitting into the smallest units based on the initial vocabulary.
- Recursively applying merge rules learned during training to form the tokens detected as most meaningful for linguistic structures.

The example, "thug", would thus be tokenized potentially as ["t", "hug"], applying learned sequences and considering what pieces are recognized under the trained vocabulary and rules.

WordPiece tokenization offers a sophisticated method of parsing text into subwords, where the decision-making process critically balances the frequency of occurrence with the distributive properties of sub-units. This method supports models in effectively dealing with diverse and complex input texts, making intricate predictions about word structures and meanings, thereby empowering deeper semantic understanding in tasks like translation, text summarization, and contextual prediction in NLP applications.

3.4.4 Unigram Tokenization

Unigram Tokenization adopts a probabilistic approach to build its vocabulary. It starts with an extensive set of potential subwords and uses a statistical method to gradually eliminate the least probable tokens. This results in a compact and highly efficient vocabulary set that reflects the true usage patterns of the language being modeled. The Unigram model is especially effective for languages with complex morphologies, as it naturally integrates morphological variants into its token selection process. [50]

Unigram's training strategy begins with a significantly large vocabulary compiled from common substrings in pre-tokenized words or using methods such as a high-threshold BPE. This initial vocabulary is then streamlined through an iterative process of elimination based on token utility for corpus representation:

- **Loss Computation:** Starting with a comprehensive vocabulary, the Unigram model calculates the corpus encoding loss for the existing set of tokens. This loss quantifies how effectively the vocabulary covers the corpus.
- **Evaluating Tokens for Removal**: For each token in the vocabulary, Unigram assesses the potential increase in corpus loss if that token were removed. Tokens that minimally increase the loss—indicating they are less crucial—are flagged for removal.
- **Batch Token Removal:** Rather than removing tokens one at a time, Unigram removes them in batches. It typically eliminates the lowest (p%) contributors to corpus coverage, where (p) is a tunable hyperparameter. This process continues until the vocabulary shrinks to the desired size.
- **Preservation of Base Characters:** Essential characters are retained throughout to ensure complete tokenizability of any corpus text.

Example of Initial Vocabulary and Token Ethication Process with a sample corpus used in earlier segments such as [51]:

("hug",10),("pug",5),("pun",12),("bun",4),("hugs",5)

Unigram might begin with all strict substrings:

Initial Vocabulary=["h","u","g","hu","ug","p","pu","n","un","b","bu","s","hug","gs","ugs"]

Each substring's frequency is measured, and as part of the probabilistic model, the probability of each substring is computed based on its frequency against the total frequency of all tokens.

In Unigram, the likelihood of a sequence of tokens is calculated as the product of individual probabilities, assuming token independence. This simplifies language modeling since each token's probability is treated discretely from others. The resulting model prefers fewer, larger tokens when tokenizing text due to probabilistic weighting, aligning with intuitive text segmentation.

For instance, tokenization options for "pug" would be evaluated as follows:

$"pug" \rightarrow ["p", "u", "g"]\ Probability\ =\ (5/210\ *\ 36/210\ *\ 20/210)\ =\ 0.000389$

$"pug" \rightarrow ["pu", "g"]\ Probability\ =\ (5/210\ *\ 20/210)\ =\ 0.0022676$

The option with a higher probability ("pu", "g") would be chosen due to its higher composite likelihood.

Unigram's iterative removal of tokens involves recalculating loss each time a set of tokens is proposed for deletion. This involves:

- Tokenizing the corpus using the current vocabulary.
- Calculating the negative log-likelihood of the corpus under the current tokenization.
- Proposing token removal that results in the smallest increase in the total corpus loss.

In practice, adjusting the vocabulary using loss calculations ensures that Unigram remains sensitive to the nuances of language data it's trained on. This adaptive feature makes it suitable for natural language tasks requiring an understanding of complex, variable morphological structures often seen in highly inflective languages.

3.5 Mapping and Vectorization in NLP

After the text is tokenized, each token is mapped to a unique integer ID, which is essential since machine learning models require numerical input rather than raw text. This mapping is constructed during the tokenizer's training phase, where a vocabulary index is created. Within this index, every distinct token is associated with a specific integer ID. For example, in English, the tokens "the," "dog," and "runs" might be assigned IDs like 1, 1234, and 567, respectively. As new texts are processed, each token is replaced by its corresponding integer ID according to the vocabulary index. For tokens not found in the vocabulary, known as out-of-vocabulary (OOV) tokens, they are either omitted, substituted with a unique identifier for unknown tokens (typically marked as <UNK>), or decomposed into smaller known sub-tokens.[52]

Once tokens are converted to sequences of these IDs, they must be transformed into a numerical format that can capture both the linguistic characteristics and the semantic relationships of the words. This transformation, known as vectorization, occurs in one of two ways depending on the complexity of the natural language processing (NLP) model involved. In more sophisticated models, token IDs are used to fetch vectors from a pre-trained embedding layer, which contains densely packed floating-point numbers that represent tokens in a multidimensional space. These vectors are structured so that tokens with similar meanings are positioned closely in the vector space, allowing the model to effectively deduce context and semantics. Techniques like Word2Vec, GloVe, or embedding layers found in transformer models such as BERT or GPT are typically utilized to generate these semantically rich vectors. In simpler models, these token IDs may be converted into one-hot encoded vectors. However, while one-hot encoding is straightforward, it does not capture semantic relationships as it treats every token as equidistant from every other token.

As this section follows pre-tokenization, it's important to note how the foundational groundwork of initial text segmentation and normalization significantly impacts the efficiency and effectiveness of these advanced tokenization methods. Each tokenization strategy thereby builds on the cleaned and segmented text provided by pre-tokenization to further refine the interpretability and usefulness of the text for various NLP applications. These tokenization methods not only facilitate a deeper understanding of language nuances but also enhance the overall performance of machine-learning models in processing and generating human-like text.

3.6 Model Pre-training

After preprocessing, the next phase is the pre-training of the model. This stage is crucial and involves teaching the model the basic structure and syntax of programming languages through unsupervised learning techniques. Common pre-training tasks include:

- **Masked Language Modeling** (MLM) [53]: Random tokens in the code are masked, and the model is trained to predict the masked tokens based on their context. This task helps the model learn the syntax and other contextual relationships within the code.
- **Next Token Prediction** [54]**:** The model predicts the next token in a sequence, helping it understand the flow and common patterns in coding practices.

These tasks enable the model to develop an initial understanding of how code is structured and functions, preparing it for more specialized tasks.

3.7 Fine-tuning for Specific Tasks

With a foundational knowledge base established during pre-training, the PTM is then fine-tuned for specific software engineering tasks. This stage involves training the model on a smaller, task-specific dataset under supervised learning conditions, where the model learns from examples with known outcomes. The fine-tuning process adjusts the model's weights specifically for tasks such as code completion, bug fixing, or code summarization, enhancing its performance on these tasks. [55]

3.7.1 Parameter-Efficient Fine-Tuning (PEFT)

Parameter-Efficient Fine-Tuning (PEFT) is an approach used to adapt a pretrained model to a new task while modifying only a small portion of the model's parameters. This technique is particularly vital in scenarios where deploying large-scale models is computationally expensive or where the environmental footprint of training such models needs to be minimized. PEFT allows for retaining the general capabilities of a large pretrained model while customizing it efficiently for specific tasks. [56]

For example, when considering the bigscience/T0_3B model, which has 3 billion parameters, full fine-tuning demands a substantial 47.14GB of GPU memory and 2.96GB of CPU memory. In sharp contrast, fine-tuning the same model using PEFT with the LoRA (Low-Rank Adaptation) method requires only 14.4GB of GPU memory while maintaining the same CPU memory usage. This represents a remarkable 70% reduction in GPU resource requirements, showcasing the significant efficiency gains offered by the PEFT-LoRA approach in optimizing the utilization of computational resources.[57]

Despite this considerable reduction in resource usage, PEFT strategies like LoRA allow the model to retain the massive prelearning from its original training. This is crucial because it means there's no significant trade-off in performance capabilities; the model still performs effectively across tasks by leveraging its base knowledge and making specific, efficient adjustments. However “PEFT techniques are slower to converge than full tuning in low/medium-resource scenarios” meaning that “for lower-resource datasets, if we prioritize training speed and less on hardware constraints, full tuning is a better option”[58]. Several strategies have been developed to implement PEFT, each focusing on altering a small subset of the model’s parameters while keeping the majority frozen (unchanged). Here are some of the fundamental techniques.

3.7.1.1 Adapter Layers

Concept:

Adapter Layers are small, trainable modules inserted between the pre-existing layers of a pre-trained model. Each adapter typically consists of a down-projection (reducing dimensionality), a non-linear activation function, and an up-projection (restoring dimensionality).

Implementation:

Adapters are added to each layer of a Transformer model, for example, without modifying the original weights of the model. Only the adapter weights are updated during training. This method allows for task-specific learning without substantial changes to the overall network architecture.[59]

Strengths:

- Modularity: Easy to add or remove from existing models, allowing for flexibility in model configuration.
- Next Token Prediction**:** Requires training only a small fraction of the total model parameters, significantly reducing computational overhead.
- Next Token Prediction: Does not disturb the original pre-trained parameters, maintaining the generalized capabilities learned during pretraining.

Weaknesses:

- Performance: While usually effective, the performance might not reach the same peak as full model training, especially for tasks that are highly divergent from the pretraining data.

3.7.1.2 Low-Rank Adaptation (LoRA)

Concept:

Low-rank adaptation (LoRA) involves modifying the weights of the pre-trained model using updates that are constrained to have a low-rank structure. This method approximates the original high dimensionality space using fewer parameters, capturing the most critical information needed for the new task.

Implementation:

This technique typically involves adding a low-rank matrix to existing weights or fully replacing original weights with low-rank approximations. During training, only these low-rank matrices are updated, while the bulk of the original weights remain unchanged.[60]

Strengths:

- Efficiency in Large Models: Can be very effective in reducing the parameter count in very large models.

- Fine-grained Control: Offers more direct control over the model’s learned representations, potentially leading to better performance on tasks closely related to the pretraining domain.

Weaknesses:

- Complexity: Implementation and tuning can be more complex than simple adapters. Determining the appropriate rank and integration method can require extensive experimentation.
- Risk of Overfitting: With insufficient regularization, low-rank updates might overfit to the fine-tuning data due to their capacity to adjust core aspects of the model more comprehensively.

3.7.1.3 Prompt Tuning

Concept:

Prompt Tuning is a technique associated primarily with fine-tuning large language models, especially those based on the Transformer architecture. It involves appending trainable tokens, or "prompts", to the input sequence to guide the model toward desired outputs without altering the underlying model weights. These prompts effectively act as tunable parameters that adjust the context in which the model processes inputs, exploiting the model's existing knowledge to generate task-specific responses.

Implementation:

Soft prompts, which are vectors of trainable parameters, are added to the input of the model. These prompts are designed to interact with the model's embedding layer and influence the activations throughout all subsequent layers. During fine-tuning, only the prompt parameters are adjusted while the main model parameters remain fixed. This confines the learning process to the tunable prompts.

Strengths:

- Parameter Efficiency: Extremely efficient since it only requires tuning a small set of parameters relative to the full model's parameter count.
- Versatility: Prompts can be designed to elicit specific behaviors from the model, enabling customization for a wide range of tasks using the same pretrained model.
- Speed: Training is typically faster than full model fine-tuning since fewer parameters are updated.

Weaknesses:

- Task Alignment: May not be as effective for tasks that are significantly different from those the model was originally trained on, since the effectiveness of prompts relies heavily on leveraging pre-existing model knowledge.
- Design Sensitivity: The design and initialization of prompts can be critical, and poor prompt design might lead to suboptimal performance.
- Scalability of Prompt Length: Extending the prompt length increases parameter count and can lead to more complex interactions that are harder to optimize.

3.7.1.4 Summary

Adapter Layers are well-suited for scenarios where the priority is to maintain the integrity of the pretrained model and achieve good performance across a variety of tasks with minimal intervention. They are particularly useful in multitasking settings where you might want to switch between different adapters efficiently.

Low-Rank Adaptation might be preferable when working with extremely large models where even modest proportions of parameter updates are impractical or where a more nuanced adjustment of the model parameters is required, possibly for tasks very similar to the original training data.

Prompt Tuning is particularly useful when dealing with large language models like GPT-3 or BERT, where traditional fine-tuning might be computationally prohibitive. It leverages the deep contextual understanding these models have to adapt to new tasks with minimal adjustments.

This approach is ideal for tasks where the underlying model already possesses a significant amount of relevant knowledge, and slight nudges provided by carefully designed prompts can result in appropriately tailored responses.

3.7.2 What Happens Inside the Model During Fine-Tuning?

In the process of fine-tuning, the model's parameters (i.e., the weights and biases in various layers) are strategically adjusted to minimize the loss function, which quantifies the difference between the model's predictions and the actual desired outcomes for the new task :

3.7.2.1 Parameter Adjustment:

During the fine-tuning phase, the model's parameters are meticulously adjusted to minimize a loss function that's carefully chosen to mirror the objectives of the new task. This fine-tuned loss function could vary from cross-entropy for code generation tasks to mean squared error for regression tasks, depending on what the task requires. The primary method employed to update the model parameters is gradient descent or its variants like stochastic gradient descent or Adam. This method calculates the gradient of the loss function with respect to each model parameter, offering a clear direction on how modifications to these parameters can decrease or increase the loss. As part of the fine-tuning process, weight adjustments are made incrementally in the direction that most effectively reduces the loss. This typically involves minor, progressive changes to the parameters, aligned opposite to the gradient and scaled by a specific learning rate. Through this iterative approach, the model's parameters are gently shifted towards values that minimize loss. Consequently, the model's outputs become more closely aligned with the desired outcomes of the new task, thereby substantially enhancing the model's performance in achieving specific objectives.

3.7.2.2. Backpropagation:

Fine-tuning utilizes backpropagation, the standard method for training neural networks. In this process, the error between the model’s predictions and the true outputs (loss) is calculated and then used to update the model weights. This error is propagated backward through the network (hence "backpropagation"), from the last layer to the first, adjusting the weights to decrease the error.

3.7.2.3. Learning Rate Adjustment:

Typically, a lower learning rate is used in fine-tuning than was used in the initial training of the model. This prevents the weights from changing too drastically, preserving the useful features the model has already learned from the large dataset while still allowing it to adapt to nuances of the new, smaller dataset.

3.7.2.4. Feature Adaptation:

The first layers of neural networks (especially in contexts like vision with Convolutional Neural Networks, or language with Transformers) learn general features that are widely applicable across many tasks (e.g., edges in images, syntax in text). During fine-tuning, more emphasis is often placed on adjusting the deeper layers because these layers are responsible for learning task-specific features.

3.7.2.5. Avoiding Overfitting

Fine-tuning must be handled carefully to avoid overfitting, especially when the new dataset is much smaller than the original training set. Overfitting occurs when a model learns the details and noise in the training data to the extent that it negatively impacts the performance of the model on new data. Techniques such as dropout, early stopping, or introducing regularization are commonly used to mitigate this risk. [61]

3.7.3 Uses of Pretrained Models

In their paper, "Deep Learning Meets Software Engineering: A Survey on Pre-Trained Models of Source Code", Changan Niu, Chuanyi Li, Bin Luo, and Vincent Ng provide a comprehensive survey of the use of PTMs in SE tasks. [62]

The authors identify and categorize 18 SE tasks to which PTMs have been applied. These tasks include:

1. **Code completion:** PTMs can be used to predict the next token or symbol in a code snippet based on the context, thereby assisting developers in completing their code more efficiently.
2. **Code search**: PTMs can be used to search for relevant code snippets within a large codebase, saving developers time and effort.
3. **Code refactoring**: PTMs can be used to automatically refactor code to improve its readability, maintainability, and performance.
4. **Code repair:** PTMs can be used to automatically repair code bugs and errors, reducing the time and effort required for manual debugging.
5. **Code rewriting**: PTMs can be used to rewrite code to improve its quality, readability, and maintainability.
6. **Code summarization**: PTMs can be used to summarize long code snippets, providing developers with a quick overview of the code's functionality.
7. **Code generation**: PTMs can be used to generate new code based on a set of inputs and constraints, such as generating test cases or creating new functions.
8. **Code comprehension**: PTMs can be used to analyze and understand the meaning and functionality of code, allowing developers to quickly grasp the essence of complex codebases.
9. **Code retrieval**: PTMs can be used to retrieve relevant code snippets from a large codebase, based on a given query or context.
10. **Code clustering**: PTMs can be used to cluster similar code snippets together, allowing developers to identify common patterns and best practices.
11. **Code classification**: PTMs can be used to classify code into different categories, such as functional or non-functional code, or code that follows best practices or not.
12. **Code sentiment analysis**: PTMs can be used to analyze the sentiment of code, such as identifying code that is likely to be buggy or difficult to maintain.

13. **Code feature extraction**: PTMs can be used to extract relevant features from code, such as identifying the programming language, frameworks, or libraries used.
14. **Code similarity analysis:** PTMs can be used to compare and analyze the similarity between different code snippets, allowing developers to identify duplicate code or code that can be refactored.
15. **Code evolution analysis:** PTMs can be used to analyze the evolution of code over time, allowing developers to identify changes in code quality, maintenance, and functionality.
16. **Code review**: PTMs can be used to assist in code review, such as identifying code that is difficult to understand, has potential bugs, or does not follow best practices.
17. **Code testing**: PTMs can be used to assist in code testing, such as identifying potential test cases, generating test data, or predicting test results.
18. **Code debugging**: PTMs can be used to assist in code debugging, such as identifying potential bugs, predicting bug locations, or suggesting fixes.

For this paper, we will mostly focus on Code generation.

3.8 Evaluation Metrics for LLMs in Code Generation

When evaluating Large Language Models (LLMs) in code generation, several metrics are traditionally used. Each metric has its strengths and limitations, and understanding these can help in selecting the right tools for comprehensive evaluation.

3.8.1 BLEU (Bilingual Evaluation Understudy)

- Description: Originally designed for machine translation, BLEU measures how many words and phrases in the generated output match a reference output, adjusted for proportion and order.
- Limitations: BLEU is primarily effective in contexts where the exact wording is important, such as translation. However, it may not be suitable for evaluating code generation where syntactic variety and functional correctness can be more important than exact text matches. BLEU also struggles with evaluating the quality of individual texts or

outputs that deviate syntactically from the reference but are still correct or even optimal in execution.[63]

3.8.2 character n-gram F-score (chrF)

ChrF is a metric used to evaluate the quality of machine translation. It calculates a score called the F-score, which is a balanced measure of precision and recall. Precision is the number of correct translations produced by the system divided by the total number of translations it produces. Recall is the number of correct translations produced by the system divided by the total number of correct translations in the reference set. chrF differs from other metrics like BLEU in that it focuses on character n-grams instead of words. This means it evaluates the similarity between translations based on sequences of characters rather than entire words. This characteristic makes chrF less sensitive to the specific order of words in the translation, which can be helpful in cases where word order might vary between languages or where the translation is grammatically correct but the word order differs from the reference. [64]

While chrF can capture some syntactic nuances better than BLEU, it exhibits negative correlations with syntactic complexity [63]. This suggests that as the complexity of the code increases, chrF may fail to adequately capture similarities between the generated code and the reference, particularly in syntactically rich languages.

3.8.3 Recall-Oriented Understudy for Gisting Evaluation (ROUGE)

ROUGE is used primarily for evaluating automatic summarization and machine translation. It measures the overlap of n-grams, word sequences, and word pairs between the generated text and a set of reference texts. [65]

“A good ROUGE score does not imply good summary quality when readability and grammatical accuracy are also taken into account”[66]. This indicates potential limitations in its applicability to evaluating code, where logical structure and execution correctness are crucial.

3.8.4 Metric for Evaluation of Translation with Explicit Ordering (METEOR)

METEOR considers exact word matches, synonyms, and stemmed versions, providing a more nuanced assessment than BLEU. [67]

While METEOR is designed to align more closely with human judgment by considering synonyms and paraphrases, it might not fully capture the functional accuracy needed in code generation.

3.8.5 Bidirectional Encoder Representations from Transformers Score (BERTScore)

BERTScore leverages contextual embeddings from models like BERT to compare the semantic similarity of words in the generated and reference texts. It is known for its robustness in capturing deeper semantic and syntactic nuances. [68]

“BERTSCORE correlates highly with human evaluations” [68], suggesting its potential suitability for tasks that require understanding complex syntactic structures or semantic content.

Given its sensitivity to syntactic and semantic richness, BERTScore could be particularly useful in evaluating code generation where understanding the underlying logic and functionality is more important than surface textual similarity.

4 Case Study on Form Generation

4.1 Research methodology

Quantitative analysis is a crucial methodological approach used in the field of artificial intelligence (AI) to assess various aspects of AI models, including large language models (LLMs) like ChatGPT. This approach employs numerical data and statistical techniques to quantify relationships, measure performance metrics, and draw conclusions based on empirical evidence. Here's a broader overview of how quantitative analysis is generally used in evaluating LLMs in AI research:

- **Performance Evaluation**: Quantitative analysis is instrumental in evaluating the performance of LLMs. Metrics such as accuracy, precision, recall, F1 score, perplexity, and computational efficiency are quantitatively assessed to benchmark the capabilities of LLMs in tasks like language generation, text completion, sentiment analysis, and more.
- **Analysis of Training Data**: Quantitative analysis plays a vital role in analyzing the training data used to fine-tune LLMs. Statistical techniques are utilized to preprocess data, identify patterns, detect biases, and ensure the quality and representativeness of the datasets, which are crucial for the robustness and generalizability of AI models.
- **Comparison and Benchmarking**: Quantitative analysis facilitates comparative studies and benchmarking of LLMs against existing models or baselines. Through rigorous experimentation and statistical testing, researchers can objectively assess whether new models outperform previous ones, establishing advancements in AI capabilities.

4.2 Background

From the initial idea to the final stage of form creation, the process generally commences when stakeholders provide a set of requirements to a Project Manager. The Project Manager then distills these requirements into a functional specifications document which is handed over to the software development team. This team is tasked with turning these specifications into the required forms. For smaller teams or contract programmers, these specifications often come directly from the stakeholders themselves.

To actualize these forms, the development team typically makes use of in-house form creation tools which are designed to extract relevant information from the functional specification documents and convert it into code. When such tools are not available, and if the demand justifies it due to a large number of forms, the team might opt to develop a tool specifically for translating these specifications into executable code. Conversely, if the quantity of forms is minimal, the most practical approach might be for developers to manually code the specifications.
For those developers or companies consistently working with the same types of forms, it becomes logical to invest in crafting a custom tool that facilitates this conversion process from specifications to code, ensuring predictable and reliable outcomes each time. However, for those working on varied form creation projects—potentially using different frameworks or programming languages—the creation of a distinct tool for each project may not be viable, thus defaulting to manual methods as a more practical solution.

In scenarios where form creation demands fluctuate or span across diverse methodologies, AI presents a particularly beneficial alternative. The identification of this challenge arose from observing the inefficiencies and limitations of traditional methods, especially in dynamic environments where manual coding or bespoke tool development may not be feasible or cost-effective for every project.

The decision to explore AI as a solution stemmed from recognizing AI's potential to automate and optimize the form creation process. AI, particularly in the form of large language models like ChatGPT, offers capabilities in natural language understanding and code generation that can interpret functional specifications and autonomously generate code. This approach promises pragmatic implications such as:

- **Adaptability**: AI can adapt to varying forms and specifications, reducing the need for bespoke tools for each project and enhancing flexibility in software development workflows.
- **Scalability**: By automating form creation tasks, AI enables scalability, allowing developers to handle larger volumes of form creation requests efficiently.

- **Efficiency**: AI-driven automation minimizes manual effort and speeds up the development cycle, potentially reducing time-to-market for software products.
- **Consistency**: AI models can ensure consistent outputs based on learned patterns and specifications, improving reliability across different projects and teams.

Implementing AI in form creation aims to streamline development processes, optimize resource allocation, and ultimately enhance productivity in software engineering. This pragmatic approach targets not only technical feasibility but also addresses practical challenges faced by teams managing diverse and fluctuating form creation requirements.

4.3 Technical Implementation

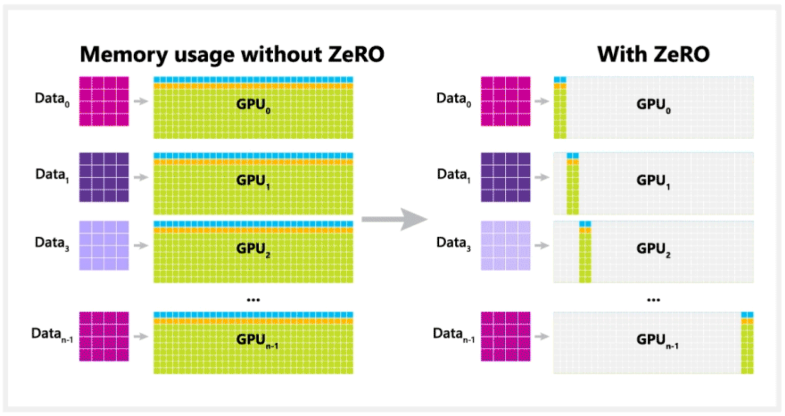
The Mahti compute nodes provided by the CSC – IT Center for Science [69] were used for the training of the models. Specifically, two nodes equipped with four 40GB Nvidia A100 GPUs each were employed. Although Mahti can support a maximum of six nodes, the training was restricted to two nodes. This limitation arose because attempts to use more than two nodes resulted in communication errors between the nodes, causing the multi-node training to fail. Consequently, the largest models that could be trained were limited to 13B parameters.

This study employs the Hugging Face library in conjunction with DeepSpeed to fine-tune large language models (LLMs). Huggingface provides a library of LLM development tools including a place to store the models and datasets [70]. “DeepSpeed is a deep learning optimization library”[71] developed to optimize the LLM training process. Deepspeed has 3 stages of memory optimization. Developed by Microsoft, DeepSpeed addresses the memory and computational challenges associated with scaling LLMs by introducing advanced memory optimization techniques. It features three key stages of memory optimization:

1. **Optimizer State Partitioning (Pos)**: This stage achieves a 4x memory reduction by partitioning the optimizer state, while maintaining the same communication volume as traditional data parallelism.

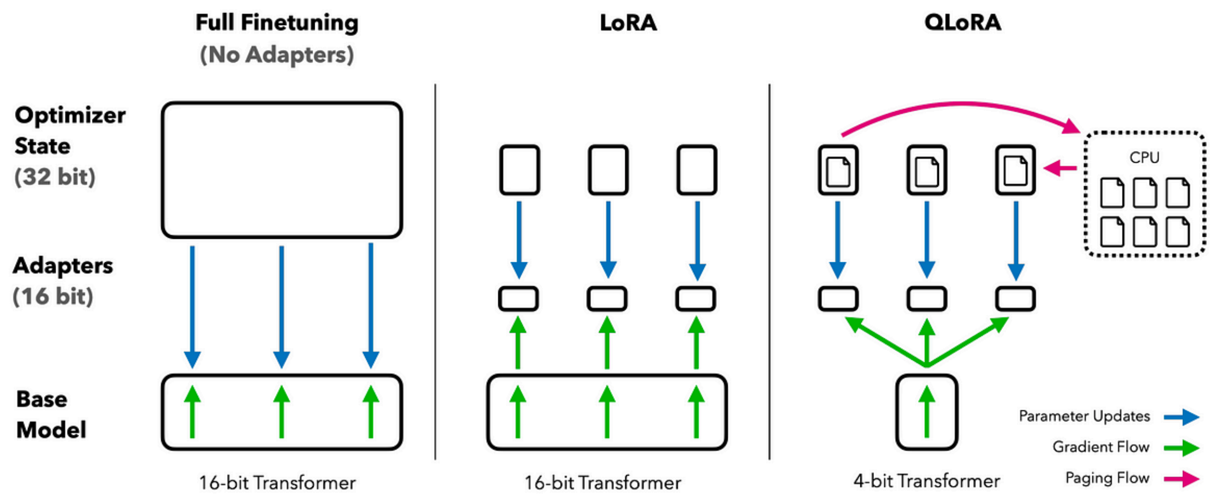
2. **Gradient Partitioning (Pos+g)**: Building on the previous stage, this step introduces gradient partitioning, resulting in an 8x memory reduction with the same communication volume as data parallelism.
3. **Parameter Partitioning (Pos+g+p)**: The final stage incorporates parameter partitioning, offering a linear memory reduction proportional to the degree of data parallelism (Nd). For instance, partitioning across 64 GPUs (Nd = 64) yields a 64x memory reduction, with only a modest 50% increase in communication volume.

DeepSpeed's Stage 3, which partitions models across multiple GPUs, was utilized to optimize model training, making it feasible to handle the LLMs. See Appendix 1 for deepspeed configuration details.



Picture 11: Deepspeed memory usage [72]

PEFT, specifically QLoRA was used to reduce graphics processing unit (GPU) memory requirements as “LoRA performs on-par or better than fine-tuning in model quality”[60]. As demonstrated by [73], "it is possible to finetune a quantized 4-bit model without any performance degradation." This quantization technique significantly reduces memory requirements by representing model parameters with 4 bits instead of the conventional 32 bits, dramatically increasing the efficiency of our fine-tuning process without compromising model performance. See Appendix 2 for configuration details.



Picture 12: Full-Finetuning, LoRa and QLoRA [73]

Five Causal language models from the Hugging Face Hub were selected for fine-tuning on a dataset containing custom form code. Causal language models are normally used for code generation. The models utilized in this process are:

- m-a-p/OpenCodeInterpreter-DS-6.7B
- Artigenz/Artigenz-Coder-DS-6.7B
- deepseek-ai/deepseek-coder-6.7b-instruct
- Finnish-NLP/llama-7b-finnish-instruct-v0.2
- codellama/CodeLlama-7b-Instruct-hf
- codellama/CodeLlama-13b-Instruct-hf

Five of these models were chosen for their code generation capabilities, evaluated using the EvalPlus, an evaluation method described in [74]. The remaining model from Finnish-NLP was selected due to its training on a dataset containing Finnish, the native language of the dataset used in this study.

This study utilized a total of eight datasets to fine-tune the models, capitalizing on both Finnish and English languages through various transformations and synthetic data augmentations:

1. **Base Dataset in Finnish**: Comprised of 52 items.
2. **English Translation of Base Dataset**: The original Finnish dataset was translated into English using the DeepL Translation API, noted for its competitive performance against Google Translate.[75], [76]
3. **Synthetic Finnish Components:** Using generative AI, 200 synthetic form components were created and added to the original Finnish dataset.
4. **Synthetic English Components**: Similarly, 200 synthetic form components were generated in English and added to the translated English dataset.
5. **Extended Finnish-to-English-to-Finnish Dataset**: The Finnish base dataset was back-translated [77] to English and then back to Finnish using the DeepL API to enhance language model training.
6. **Extended English-to-Finnish-to-English Dataset**: Similarly the English base dataset underwent Back-Translation to Finnish and then back again to English to produce synthetic data that's contextually similar.
7. **Consolidated Finnish Dataset:** This dataset combined the original Finnish dataset, the synthetic Finnish components, and the Finnish dataset obtained from back-translation.
8. **Consolidated English Dataset**: This encompassed the English-translated base dataset, synthetic English components, and the English dataset generated from back-translating.

Each of the datasets was formatted using the chat format

```
{"messages": [{"role": "system", "content": "You are now a form code
generation tool. Generate form code for the following."}, {"role": "user",
"content": "{ 'title': '', 'components': [ { 'type': '', 'values': [''],
'label': ''}, { 'type': '', 'value': '', 'label': '', 'condition': { '':
''}}] }"}, {"role": "assistant", "content": ""}]}
```

Code Snippet 1. initial dataset in chat format

```
{"messages": [{"role": "system", "content": "You are now a form code
component generator, I will give you components and I want you to generate
the code for them."}, {"role": "user", "content": "{"component": {"type":
```

```
"", "values": [""], "label": ""}, "code": ""}"}, {"role": "assistant",
"content": ""}]}
```
Code Snippet 2. Component dataset in chat format

The examples in the dataset were tokenized using each of the models' native sentencepiece tokenizer. Each model was fine-tuned using these datasets with the intent of discovering which dataset type performed the best. Post-fine-tuning, each model underwent testing on a separate dataset to evaluate their performance. The evaluation metrics used were, ROUGE, METEOR, chrF, and BERTScore [78], [79], [80]. Appendix 4 contains the evaluation code.

5 Results

The results from Tables 1 to 9 reveal several key insights into the performance of various language models for automated form generation. Base models, as shown in Table 1, demonstrated varied performance, with CodeLlama-13b-Instruct-hf leading in most metrics. Fine-tuning on different datasets, as illustrated in Tables 2-8, generally improved model performance, although the extent of improvement varied. Notably, Dataset 3 (Finnish Components) and Dataset 8 (Combined English) yielded particularly strong results across models.

In terms of model-specific observations, CodeLlama models consistently performed well across different datasets and configurations. The llama-7b-finnish-instruct-v0.2 model showed strong performance, especially on Finnish datasets. In contrast, OpenCodeInterpreter-DS-6.7B and Artigenz-Coder-DS-6.7B generally underperformed compared to other models.

The degree of improvement for finetuning varied across datasets, with some models showing substantial gains while others experienced more modest improvements. This variability highlights the importance of careful dataset selection in the fine-tuning process.

Interestingly, the results challenged some common assumptions about model size. Larger models didn't always outperform smaller ones, suggesting that model size alone doesn't guarantee better performance. This finding emphasizes the need to consider factors beyond just model size when selecting the most appropriate model for a specific task.

The choice of dataset for fine-tuning emerged as a crucial factor influencing model performance. Some datasets consistently yielded better results across models, indicating that the quality and relevance of the training data play a significant role in model performance.

Performance variations across different evaluation metrics (ROUGE, METEOR, CHRF, BERTScore) were observed, highlighting the importance of using multiple metrics for a comprehensive evaluation of model performance. Each metric captures different aspects of the generated output, providing a more holistic view of model capabilities.

The final ranking presented in Table 9 shows a mix of model sizes and fine-tuning approaches among the top performers. This diversity suggests that multiple strategies can lead to competitive results in automated form generation tasks. It underscores the complexity of model selection and fine-tuning, emphasizing the need for careful consideration of model architecture, dataset selection, and fine-tuning strategies to achieve optimal performance.

These findings collectively underscore the nuanced nature of applying language models to automated form generation. They highlight the importance of a multifaceted approach that considers various factors including model architecture, dataset quality, language specificity, and evaluation metrics to achieve the best possible results in this domain.

The scoring mechanism assigns 1 point to the model with the highest evaluation score in each column. A model can accumulate a maximum of 4 points. Subsequently, the top-performing models from each dataset are consolidated into a table for further analysis. Within this aggregated table, scores are recalculated to identify the best-performing model.
The findings are presented as follows:

*The top scores for each evaluation metric (obtained by running eval using the test dataset as both prediction and actual): **Rouge**: 0.425 **Meteor**: 0.268 **Chrf**: 100.000 **BERTScore**: 1.000

Table 1. Base models

Base models un-finetuned.

Model	**Rouge**	**meteor**	**chrf**	**BERTScore**	**Score**
OpenCodeInterpreter-DS-6.7B	0.079	0.079	15.832	0.467	0
Artigenz-Coder-DS-6.7B	0.089	0.125	19.712	0.441	0
deepseek-coder-6.7b-instruct	0.067	0.100	18.497	0.469	0
llama-7b-finnish-instruct-v0.2	0.020	0.059	5.418	0.349	0
CodeLlama-7b-Instruct-hf	0.074	0.104	16.035	0.554	1
CodeLlama-13b-Instruct-hf	**0.121**	**0.141**	**24.730**	**0.550**	**3**

Table 2. Dataset 1

Finetuned on base Finnish Dataset 1.

Model	**Rouge**	**meteor**	**chrf**	**BERTScore**	**Score**
OpenCodeInterpreter-DS-6.7B	0.055	0.063	17.051	0.412	0
Artigenz-Coder-DS-6.7B	0.091	0.110	19.073	0.455	0
deepseek-coder-6.7b-instruct	0.073	0.094	21.840	0.501	0

Model	Rouge	meteor	chrf	BERTScore	Score
llama-7b-finnish-instruct-v0.2	0.017	0.043	5.431	0.273	0
CodeLlama-7b-Instruct-hf	**0.115**	**0.138**	**23.186**	**0.538**	**3**
CodeLlama-13b-Instruct-hf	0.085	0.113	20.086	0.558	1

Table 3. Dataset 2

Finetuned on base English Dataset 2.

Model	Rouge	meteor	chrf	BERTScore	Score
OpenCodeInterpreter-DS-6.7B	**0.037**	**0.098**	**10.904**	**0.387**	**4**
Artigenz-Coder-DS-6.7B	0.012	0.046	5.916	0.376	0
deepseek-coder-6.7b-instruct	0.015	0.046	4.647	0.491	0
llama-7b-finnish-instruct-v0.2	0.017	0.043	5.431	0.273	0
CodeLlama-7b-Instruct-hf	0.016	0.053	4.330	0.316	0
CodeLlama-13b-Instruct	0.018	0.042	7.052	0.255	0

Table 4. Dataset 3

Finetuned on Finnish Components Dataset 3.

Model	Rouge	meteor	chrf	BERTScore	Score
OpenCodeInterpreter-DS-6.7B	0.060	0.044	6.224	0.343	0
Artigenz-Coder-DS-6.7B	0.044	0.155	8.247	0.329	0
deepseek-coder-6.7b-instruct	0.082	0.055	7.940	0.350	0
llama-7b-finnish-instruct-v0.2	0.084	0.209	19.016	0.656	1
CodeLlama-7b-Instruct-hf	**0.098**	**0.214**	**19.847**	**0.594**	**3**
CodeLlama-13b-Instruct-hf	0.040	0.084	5.934	0.300	0

Table 5. Dataset 4

Finetuned on English Components Dataset 4.

Model	**Rouge**	**meteor**	**chrf**	**BERTScore**	**Score**
OpenCodeInterpreter-DS-6.7B	0.061	0.080	9.810	0.273	1
Artigenz-Coder-DS-6.7B	0.058	0.054	4.738	0.315	0
deepseek-coder-6.7b-instruct	0.074	0.072	7.929	0.318	0
llama-7b-finnish-instruct-v0.2	0.015	0.068	9.023	0.303	0
CodeLlama-7b-Instruct-hf	0.055	0.110	11.081	0.465	1
CodeLlama-13b-Instruct-hf	**0.045**	**0.121**	**9.537**	**0.543**	**2**

Table 6. Dataset 5

Finetuned on Finnish Back-Translated Dataset 5.

Model	**Rouge**	**meteor**	**chrf**	**BERTScore**	**Score**
OpenCodeInterpreter-DS-6.7B	0.019	0.051	3.507	0.278	0
Artigenz-Coder-DS-6.7B	0.019	0.062	8.767	0.192	0
deepseek-coder-6.7b-instruct	0.018	0.065	3.078	0.302	0
llama-7b-finnish-instruct-v0.2	**0.082**	**0.140**	**17.111**	**0.624**	**4**
CodeLlama-7b-Instruct-hf	0.054	0.089	7.970	0.421	0
CodeLlama-13b-Instruct-hf	0.037	0.078	3.703	0.389	0

Table 7. Dataset 6

Finetuned on English Back-Translated Dataset 5.

Model	**Rouge**	**meteor**	**chrf**	**BERTScore**	**Score**
OpenCodeInterpreter-DS-6.7B	0.012	0.029	2.210	0.307	0
Artigenz-Coder-DS-6.7B	0.023	0.045	2.579	0.261	0
deepseek-coder-6.7b-instruct	0.009	0.047	1.327	0.264	0
llama-7b-finnish-instruct-v0.2	**0.080**	**0.139**	**19.016**	**0.623**	**4**
CodeLlama-7b-Instruct-hf	0.033	0.072	8.337	0.392	0
CodeLlama-13b-Instruct-hf	0.029	0.061	5.131	0.288	0

Table 8. Dataset 7

Finetuned on combined Finnish Dataset 7.

Model	**Rouge**	**meteor**	**chrf**	**BERTScore**	**Score**
OpenCodeInterpreter-DS-6.7B	0.050	0.066	6.180	0.278	0
Artigenz-Coder-DS-6.7B	0.061	0.141	9.582	0.287	0
deepseek-coder-6.7b-instruct	0.085	0.074	8.738	0.300	0
llama-7b-finnish-instruct-v0.2	**0.079**	**0.118**	**17.282**	**0.562**	**2**
CodeLlama-7b-Instruct-hf	**0.096**	**0.173**	**13.538**	**0.490**	**2**
CodeLlama-13b-Instruct-hf	0.045	0.131	6.370	0.291	0

Table 9. Dataset 8

Finetuned on combined English Dataset 8.

Model	Rouge	meteor	chrf	BERTScore	Score
OpenCodeInterpreter-DS-6.7B	0.045	0.046	4.732	0.299	0
Artigenz-Coder-DS-6.7B	0.056	0.152	10.861	0.283	0
deepseek-coder-6.7b-instruct	0.039	0.044	4.182	0.288	0
llama-7b-finnish-instruct-v0.2	**0.116**	**0.131**	**23.342**	**0.620**	**2**
CodeLlama-7b-Instruct-hf	0.041	0.078	5.140	0.479	0
CodeLlama-13b-Instruct-hf	**0.144**	**0.172**	**14.816**	**0.525**	**2**

Table 9. Top Models

Top models after Scoring.

Model	Dataset	Rouge	meteor	chrf	BERTScore
CodeLlama-13b-Instruct-hf	0	0.121	0.141	24.730	0.550
CodeLlama-7b-Instruct-hf	1	0.115	0.138	23.186	0.538
OpenCodeInterpreter-DS-6.7B	2	0.037	0.098	10.904	0.387
CodeLlama-7b-Instruct-hf	3	0.098	0.214	19.847	0.594
CodeLlama-13b-Instruct-hf	4	0.045	0.121	9.537	0.543
llama-7b-finnish-instruct-v0.2	5	0.082	0.140	17.111	0.624
llama-7b-finnish-instruct-v0.2	6	0.080	0.139	19.016	0.623
llama-7b-finnish-instruct-v0.2	7	0.079	0.118	17.282	0.562
CodeLlama-7b-Instruct-hf	7	0.096	0.173	13.538	0.490
llama-7b-finnish-instruct-v0.2	8	0.116	0.131	23.342	0.620
CodeLlama-13b-Instruct-hf	8	0.144	0.172	14.816	0.525

To rank the models each metric was normalized.

The formula for each metric: $Normalized\ Score\ =\ (Value\ -\ Min)\ /\ (Max\ -\ Min)$

- Rouge: Min = 0.037, Max = 0.144
- Meteor: Min = 0.098, Max = 0.214
- CHRF: Min = 9.537, Max = 24.730
- BERTScore: Min = 0.387, Max = 0.624

Table 10. Top Models ranked

Top models after normalization

Model	**Dataset**	**Normalized Score**	**Rank**
CodeLlama-7b-Instruct-hf	**3**	**0.7806**	**1**
llama-7b-finnish-instruct-v0.2	8	0.7288	2
CodeLlama-13b-Instruct-hf	0	0.7109	3
CodeLlama-13b-Instruct-hf	8	0.6418	4
CodeLlama-7b-Instruct-hf	1	0.6521	5
llama-7b-finnish-instruct-v0.2	6	0.5940	6
llama-7b-finnish-instruct-v0.2	5	0.5705	7
CodeLlama-7b-Instruct-hf	7	0.4741	8
llama-7b-finnish-instruct-v0.2	7	0.4532	9
CodeLlama-13b-Instruct-hf	4	0.2328	10
OpenCodeInterpreter-DS-6.7B	2	0.0225	11

The results presented in Table 9 reveal several noteworthy findings. As initially observed, the Llama models demonstrated superior performance overall, with the CodeLlama-7b-Instruct-hf model fine-tuned on dataset 3 (Synthetic Finnish Components) achieving the highest normalized score of 0.7806 and ranking first among all models tested.

Interestingly, the top three positions are occupied by different model variants, highlighting the importance of both model architecture and dataset selection in achieving optimal performance.

The llama-7b-finnish-instruct-v0.2 model fine-tuned on dataset 8 secured the second position with a score of 0.7288, while the CodeLlama-13b-Instruct-hf model without fine-tuning (dataset 0) ranked third with a score of 0.7109.

It's worth noting that the performance gap between the top-ranked models is relatively small, suggesting that multiple approaches can yield competitive results. The CodeLlama models, in particular, show strong performance across different configurations, occupying four of the top five positions.

The Finnish-specific model (llama-7b-finnish-instruct-v0.2) performed consistently well across different datasets, appearing multiple times in the top half of the rankings. This underscores the value of language-specific models when working with non-English datasets.

Surprisingly, the larger CodeLlama-13b-Instruct-hf model did not consistently outperform its 7b counterpart, indicating that model size alone does not guarantee superior performance in this task.

At the lower end of the rankings, we find that the OpenCodeInterpreter-DS-6.7B model significantly underperformed compared to the Llama variants, suggesting that it may not be well-suited for this particular task or may require different fine-tuning approaches.

When qualitatively comparing the generated output to the expected results, a clear distinction emerged between lower-scoring models and higher-scoring ones. The higher-scoring models demonstrated a better grasp of the dataset structure, as reflected by their evaluation scores. Despite this, the generated output still fell short of the desired standard, indicating the need for further training with larger models known for handling complex patterns more effectively.

To address these shortcomings, models with substantially larger parameters—up to 70 billion compared to the smallest fine-tuned models with 6.7 billion parameters—should be considered.

The rationale for expecting better performance from these larger models includes several key factors:

- **Representation Capacity**: Larger models with more parameters can capture more intricate patterns and relationships within the data. This enhanced capacity allows for a deeper understanding of complex language structures, subtleties, and nuances, resulting in more accurate and contextually appropriate responses.
- **Knowledge Encoding**: Higher parameter models can store a more extensive amount of information from the training data. This expanded "memory" enables the model to retain and retrieve a broader range of facts, linguistic rules, and contextual cues, enhancing its ability to generate relevant and informed outputs.
- **Generalization Ability**: Models with more parameters tend to generalize better across different contexts and tasks. Exposure to a wider variety of examples and patterns during training helps these models handle new and unseen inputs more effectively.
- **Complex Task Handling**: Higher parameter models are better equipped to tackle complex and diverse tasks requiring deep understanding and multi-step reasoning. They can manage tasks involving long-range dependencies, detailed explanations, and intricate problem-solving, which smaller models might struggle with.
- **Fine-tuning Precision**: Larger models offer more fine-grained control over the learned representations. During fine-tuning for specific tasks, these models can adjust their vast number of parameters to achieve higher precision and tailor their outputs more closely to the task requirements.
- **Error Reduction**: With more parameters, LLMs can distribute learning across many parameters, reducing the risk of overfitting to specific data points or patterns. This distribution helps mitigate errors and biases, leading to more robust and reliable performance.

To verify these hypotheses, it is crucial to resolve the communication errors encountered during multi-node training in the CSC environment. Fixing these issues will allow for the effective training of larger models, providing a more definitive assessment of their potential performance improvements.

6 Conclusion

This study has provided insights into the potential and current limitations of using AI, specifically Large Language Models (LLMs), for automating form creation from functional specification documents. By addressing the research questions, "How has artificial intelligence evolved over the years in the field of software development?" and "How effective is artificial intelligence, particularly LLMs like ChatGPT, in automating the process of form creation by interpreting and generating code from functional specification documents?", the multi-stage research process revealed both promising aspects and significant challenges in this application of AI technology.

The initial phase of the study involved a comprehensive Literature Review to explore the historical evolution of AI. This review highlighted significant advancements, from rule-based systems to machine learning, and more recently to advanced neural networks like LLMs. Understanding this progression provided a contextual backdrop for evaluating the current capabilities of AI, particularly in automating software development tasks such as form creation.

In response to the second research question, the study employed Quantitative Analysis to assess the feasibility and efficacy of using AI for interpreting functional specification documents to enable automated form generation. The analysis demonstrated the following findings:

- **Formatting Raw Functional Specifications**: Initially, AI was employed to format raw functional specifications into a structured format. While the models showed proficiency with simple form components, they often struggled with complex relationships, such as conditional field displays with associated help text. This highlights the challenges AI faces in interpreting and reproducing intricate document structures and relationships.
- **Generating Synthetic Form Components**: The generation of synthetic form components yielded promising results using prompt tuning, particularly for small components. However, generating entire forms with custom components using prompt tuning alone proved challenging. This limitation underscores the need for more advanced techniques to handle larger, more complex form structures.

- **Fine-Tuning Models**: Fine-tuning was explored to enhance the models' ability to capture and reproduce custom syntax. Improvements were noted, especially with the CodeLlama-7b-Instruct-hf model fine-tuned on the Synthetic Finnish Components dataset. However, the inconsistent performance across different models raises questions about the cost-effectiveness of fine-tuning in this context.

These findings offer insights for both researchers and practitioners in the field of AI-assisted software development:

- **For Researchers**: This study underscores the need for further investigation into optimizing LLMs for specific, complex tasks like form generation. Future research should focus on developing more sophisticated techniques that can better handle the intricacies of complex form structures and relationships.
- **For Practitioners**: The results suggest that while AI tools can be valuable aids in the form creation process, particularly for simpler components, they are not yet ready to fully automate the creation of complex, custom forms. A hybrid approach leveraging both AI capabilities and human expertise may be the most effective strategy in the near term.

The success of language-specific models, as seen with the Finnish model's performance, indicates a potential direction for developing more localized and specialized AI tools for software development.

In conclusion, this study has illuminated both the potential and limitations of current AI technologies in the domain of form generation. While the current state of LLMs shows promise in certain aspects, it does not yet meet the standards required for fully automated, production-ready form code generation, especially for complex forms. Future advancements may come from a combination of more sophisticated models, refined fine-tuning and prompt-tuning techniques, and a greater focus on task-specific and language-specific training data. As AI technology continues to evolve, it holds the potential to significantly enhance and streamline the software development process, particularly in areas like form creation, even if it may not fully automate these tasks in the immediate future.

References

[1] Y. Chang *et al.*, “A Survey on Evaluation of Large Language Models,” Dec. 28, 2023, *arXiv*: arXiv:2307.03109. doi: 10.48550/arXiv.2307.03109.

[2] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, “Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation,” Oct. 30, 2023, *arXiv*: arXiv:2305.01210. doi: 10.48550/arXiv.2305.01210.

[3] J. Bulian *et al.*, “Assessing Large Language Models on Climate Information,” May 28, 2024, *arXiv*: arXiv:2310.02932. doi: 10.48550/arXiv.2310.02932.

[4] B. Delipetrev, C. Tsinaraki, and U. Kostic, “Historical Evolution of Artificial Intelligence,” JRC Publications Repository. Accessed: Jan. 21, 2024. [Online]. Available: https://publications.jrc.ec.europa.eu/repository/handle/JRC120469

[5] E. G. Daylight, “Towards a Historical Notion of ‘Turing—the Father of Computer Science,’” *History and Philosophy of Logic*, vol. 36, no. 3, pp. 205–228, Jul. 2015, doi: 10.1080/01445340.2015.1082050.

[6] A. M. Turing, “Computing Machinery and Intelligence,” in *Parsing the Turing Test: Philosophical and Methodological Issues in the Quest for the Thinking Computer*, R. Epstein, G. Roberts, and G. Beber, Eds., Dordrecht: Springer Netherlands, 2009, pp. 23–65. doi: 10.1007/978-1-4020-6710-5_3.

[7] A. Newell, J. C. Shaw, and H. A. Simon, “Elements of a theory of human problem solving,” *Psychological Review*, vol. 65, no. 3, pp. 151–166, 1958, doi: 10.1037/h0048495.

[8] “Wikiwand - File:Colored_neural_network.svg,” Wikiwand. Accessed: Jul. 09, 2024. [Online]. Available: https://www.wikiwand.com/en/File:Colored_neural_network.svg

[9] “Artificial Neural Networks Technology – CSIAC.” Accessed: Jun. 27, 2024. [Online]. Available: https://csiac.org/state-of-the-art-reports/artificial-neural-networks-technology/

[10] F. Rosenblatt, “The perceptron: A probabilistic model for information storage and organization in the brain,” *Psychological Review*, vol. 65, no. 6, pp. 386–408, 1958, doi: 10.1037/h0042519.

[11] J. McCarthy, M. L. Minsky, N. Rochester, and C. E. Shannon, “A Proposal for the Dartmouth Summer Research Project on Artificial Intelligence, August 31, 1955,” *AI Magazine*, vol. 27, no. 4, Art. no. 4, Dec. 2006, doi: 10.1609/aimag.v27i4.1904.

[12] D. Crevier, *AI: The Tumultuous History of the Search for Artificial Intelligence*. 1993, p. 386.

[13] T. M. Mitchell, *Machine Learning*, 1st ed. USA: McGraw-Hill, Inc., 1997.

[14] “EnjoyAlgorithms.” Accessed: Jul. 09, 2024. [Online]. Available: https://www.enjoyalgorithms.com/

[15] Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning,” *Nature*, vol. 521, no. 7553, Art. no. 7553, May 2015, doi: 10.1038/nature14539.

[16] T. P. Lillicrap, A. Santoro, L. Marris, C. J. Akerman, and G. Hinton, “Backpropagation and the brain,” *Nat Rev Neurosci*, vol. 21, no. 6, pp. 335–346, Jun. 2020, doi: 10.1038/s41583-020-0277-3.

[17] A. Christopher, “What is Perceptron: A BeginnersTutorial For Perceptron,” Medium. Accessed: Jul. 09, 2024. [Online]. Available: https://antoblog.medium.com/what-is-perceptron-a-beginnerstutorial-for-perceptron-632539884146

[18] M.-C. Popescu, V. E. Balas, L. Perescu-Popescu, and N. Mastorakis, “Multilayer Perceptron and Neural Networks,” vol. 8, no. 7, 2009.

[19] G. Bebis and M. Georgiopoulos, “Feed-forward neural networks,” *IEEE Potentials*, vol. 13, no. 4, pp. 27–31, Oct. 1994, doi: 10.1109/45.329294.

[20] N. Shahriar, “What is Convolutional Neural Network — CNN (Deep Learning),” Medium.

Accessed: Jul. 10, 2024. [Online]. Available: https://nafizshahriar.medium.com/what-is-convolutional-neural-network-cnn-deep-learning-b3921bdd82d5
[21] K. O'Shea and R. Nash, "An Introduction to Convolutional Neural Networks," Dec. 02, 2015, *arXiv*: arXiv:1511.08458. doi: 10.48550/arXiv.1511.08458.
[22] R. Badkas, "MODELLING OF CARDIAC ELECTROPHYSIOLOGY," 2021.
[23] A. C. Tsoi, "Recurrent neural network architectures: An overview," in *Adaptive Processing of Sequences and Data Structures: International Summer School on Neural Networks "E.R. Caianiello" Vietri sul Mare, Salerno, Italy September 6–13, 1997 Tutorial Lectures*, C. L. Giles and M. Gori, Eds., Berlin, Heidelberg: Springer, 1998, pp. 1–26. doi: 10.1007/BFb0053993.
[24] S. Hochreiter, "The Vanishing Gradient Problem During Learning Recurrent Neural Nets and Problem Solutions," *Int. J. Unc. Fuzz. Knowl. Based Syst.*, vol. 06, no. 02, pp. 107–116, Apr. 1998, doi: 10.1142/S0218488598000094.
[25] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training recurrent neural networks," in *Proceedings of the 30th International Conference on Machine Learning*, PMLR, May 2013, pp. 1310–1318. Accessed: Jun. 27, 2024. [Online]. Available: https://proceedings.mlr.press/v28/pascanu13.html
[26] D. Xu, M. Bai, T. Long, and J. Gao, "LSTM-assisted evolutionary self-expressive subspace clustering," *International Journal of Machine Learning and Cybernetics*, vol. 12, Oct. 2021, doi: 10.1007/s13042-021-01363-z.
[27] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997, doi: 10.1162/neco.1997.9.8.1735.
[28] A. Vaswani *et al.*, "Attention is All you Need," in *Advances in Neural Information Processing Systems*, Curran Associates, Inc., 2017. Accessed: Jun. 27, 2024. [Online]. Available: https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html
[29] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A Comprehensive Survey on Graph Neural Networks," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 32, no. 1, pp. 4–24, Jan. 2021, doi: 10.1109/TNNLS.2020.2978386.
[30] K. Han *et al.*, "A Survey on Vision Transformer," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 45, no. 1, pp. 87–110, Jan. 2023, doi: 10.1109/TPAMI.2022.3152247.
[31] P. Xu, X. Zhu, and D. A. Clifton, "Multimodal Learning With Transformers: A Survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 45, no. 10, pp. 12113–12132, Oct. 2023, doi: 10.1109/TPAMI.2023.3275156.
[32] S. Alaparthi and M. Mishra, "Bidirectional Encoder Representations from Transformers (BERT): A sentiment analysis odyssey," Jul. 02, 2020, *arXiv*: arXiv:2007.01127. doi: 10.48550/arXiv.2007.01127.
[33] J. Flowers, "Strong and Weak AI: Deweyan Considerations," presented at the AAAI Spring Symposium: Towards Conscious AI Systems, 2019. Accessed: Jan. 21, 2024. [Online]. Available: https://www.semanticscholar.org/paper/Strong-and-Weak-AI%3A-Deweyan-Considerations-Flowers/35a332c6c414165c3fa02988949f45231fb0b46c
[34] J. R. Searle, "Minds, brains, and programs," *Behavioral and Brain Sciences*, vol. 3, no. 3, pp. 417–424, 1980, doi: 10.1017/S0140525X00005756.
[35] K. Jebari and J. Lundborg, "Artificial superintelligence and its limits: why AlphaZero cannot become a general agent," *AI & Soc*, vol. 36, no. 3, pp. 807–815, Sep. 2021, doi: 10.1007/s00146-020-01070-3.
[36] K. S. Gill, "Artificial super intelligence: beyond rhetoric," *AI & Soc*, vol. 31, no. 2, pp.

137–143, May 2016, doi: 10.1007/s00146-016-0651-x.
[37] X. Han *et al.*, “Pre-trained models: Past, present and future,” *AI Open*, vol. 2, pp. 225–250, Jan. 2021, doi: 10.1016/j.aiopen.2021.08.002.
[38] M. R. J, K. VM, H. Warrier, and Y. Gupta, “Fine Tuning LLM for Enterprise: Practical Guidelines and Recommendations,” Mar. 23, 2024, *arXiv*: arXiv:2404.10779. doi: 10.48550/arXiv.2404.10779.
[39] D. Singh and B. Singh, “Investigating the impact of data normalization on classification performance,” *Applied Soft Computing*, vol. 97, p. 105524, Dec. 2020, doi: 10.1016/j.asoc.2019.105524.
[40] C. W. Schmidt *et al.*, “Tokenization Is More Than Compression,” Feb. 28, 2024, *arXiv*: arXiv:2402.18376. doi: 10.48550/arXiv.2402.18376.
[41] “Summary of the tokenizers.” Accessed: Jun. 27, 2024. [Online]. Available: https://huggingface.co/docs/transformers/en/tokenizer_summary
[42] A. Kolonin and V. Ramesh, “Unsupervised Tokenization Learning,” Dec. 15, 2022, *arXiv*: arXiv:2205.11443. doi: 10.48550/arXiv.2205.11443.
[43] T. Kudo and J. Richardson, “SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing,” Aug. 19, 2018, *arXiv*: arXiv:1808.06226. doi: 10.48550/arXiv.1808.06226.
[44] K. Bostrom and G. Durrett, “Byte Pair Encoding is Suboptimal for Language Model Pretraining,” Oct. 05, 2020, *arXiv*: arXiv:2004.03720. doi: 10.48550/arXiv.2004.03720.
[45] X. Song, A. Salcianu, Y. Song, D. Dopson, and D. Zhou, “Fast WordPiece Tokenization,” Oct. 05, 2021, *arXiv*: arXiv:2012.15524. doi: 10.48550/arXiv.2012.15524.
[46] *google/sentencepiece*. (Jun. 27, 2024). C++. Google. Accessed: Jun. 27, 2024. [Online]. Available: https://github.com/google/sentencepiece
[47] R. Sennrich, B. Haddow, and A. Birch, “Neural Machine Translation of Rare Words with Subword Units,” in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, K. Erk and N. A. Smith, Eds., Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 1715–1725. doi: 10.18653/v1/P16-1162.
[48] “Byte-Pair Encoding tokenization - Hugging Face NLP Course.” Accessed: May 20, 2024. [Online]. Available: https://huggingface.co/learn/nlp-course/en/chapter6/5
[49] “WordPiece tokenization - Hugging Face NLP Course.” Accessed: May 20, 2024. [Online]. Available: https://huggingface.co/learn/nlp-course/en/chapter6/6
[50] T. Kudo, “Subword Regularization: Improving Neural Network Translation Models with Multiple Subword Candidates,” Apr. 29, 2018, *arXiv*: arXiv:1804.10959. doi: 10.48550/arXiv.1804.10959.
[51] “Unigram tokenization - Hugging Face NLP Course.” Accessed: May 20, 2024. [Online]. Available: https://huggingface.co/learn/nlp-course/en/chapter6/7
[52] “Behind the pipeline - Hugging Face NLP Course.” Accessed: Jun. 27, 2024. [Online]. Available: https://huggingface.co/learn/nlp-course/en/chapter2/2
[53] K. Sinha, R. Jia, D. Hupkes, J. Pineau, A. Williams, and D. Kiela, “Masked Language Modeling and the Distributional Hypothesis: Order Word Matters Pre-training for Little,” Sep. 09, 2021, *arXiv*: arXiv:2104.06644. doi: 10.48550/arXiv.2104.06644.
[54] Y. Li, Y. Huang, M. E. Ildiz, A. S. Rawat, and S. Oymak, “Mechanics of Next Token Prediction with Self-Attention,” in *Proceedings of The 27th International Conference on Artificial Intelligence and Statistics*, PMLR, Apr. 2024, pp. 685–693. Accessed: Jun. 27, 2024. [Online]. Available: https://proceedings.mlr.press/v238/li24f.html
[55] K. W. Church, Z. Chen, and Y. Ma, “Emerging trends: A gentle introduction to fine-tuning,” *Natural Language Engineering*, vol. 27, no. 6, pp. 763–778, Nov. 2021, doi: 10.1017/S1351324921000322.
[56] V. Lialin, V. Deshpande, and A. Rumshisky, “Scaling Down to Scale Up: A Guide to

Parameter-Efficient Fine-Tuning," Mar. 27, 2023, *arXiv*: arXiv:2303.15647. Accessed: May 21, 2024. [Online]. Available: http://arxiv.org/abs/2303.15647

[57] *huggingface/peft*. (May 21, 2024). Python. Hugging Face. Accessed: May 21, 2024. [Online]. Available: https://github.com/huggingface/peft

[58] G. Pu, A. Jain, J. Yin, and R. Kaplan, "Empirical Analysis of the Strengths and Weaknesses of PEFT Techniques for LLMs," arXiv.org. Accessed: May 21, 2024. [Online]. Available: https://arxiv.org/abs/2304.14999v1

[59] R. He *et al.*, "On the Effectiveness of Adapter-based Tuning for Pretrained Language Model Adaptation," Jun. 06, 2021, *arXiv*: arXiv:2106.03164. doi: 10.48550/arXiv.2106.03164.

[60] E. J. Hu *et al.*, "LoRA: Low-Rank Adaptation of Large Language Models," Oct. 16, 2021, *arXiv*: arXiv:2106.09685. doi: 10.48550/arXiv.2106.09685.

[61] F. Xue, Y. Fu, W. Zhou, Z. Zheng, and Y. You, "To Repeat or Not To Repeat: Insights from Scaling LLM under Token-Crisis," *Advances in Neural Information Processing Systems*, vol. 36, pp. 59304–59322, Dec. 2023.

[62] "[2205.11739] Deep Learning Meets Software Engineering: A Survey on Pre-Trained Models of Source Code." Accessed: Jun. 23, 2024. [Online]. Available: https://arxiv.org/abs/2205.11739

[63] Y. Wang, Q. Chen, and D. Ataman, "Delving into Evaluation Metrics for Generation: A Thorough Assessment of How Metrics Generalize to Rephrasing Across Languages," in *Proceedings of the 4th Workshop on Evaluation and Comparison of NLP Systems*, D. Deutsch, R. Dror, S. Eger, Y. Gao, C. Leiter, J. Opitz, and A. Rücklé, Eds., Bali, Indonesia: Association for Computational Linguistics, Nov. 2023, pp. 23–31. doi: 10.18653/v1/2023.eval4nlp-1.3.

[64] M. Popović, "chrF: character n-gram F-score for automatic MT evaluation," in *Proceedings of the Tenth Workshop on Statistical Machine Translation*, O. Bojar, R. Chatterjee, C. Federmann, B. Haddow, C. Hokamp, M. Huck, V. Logacheva, and P. Pecina, Eds., Lisbon, Portugal: Association for Computational Linguistics, Sep. 2015, pp. 392–395. doi: 10.18653/v1/W15-3049.

[65] C.-Y. Lin, "ROUGE: A Package for Automatic Evaluation of Summaries," in *Text Summarization Branches Out*, Barcelona, Spain: Association for Computational Linguistics, Jul. 2004, pp. 74–81. Accessed: Jun. 27, 2024. [Online]. Available: https://aclanthology.org/W04-1013

[66] M. Barbella and G. Tortora, "Rouge Metric Evaluation for Text Summarization Techniques," May 26, 2022, *Rochester, NY*: 4120317. doi: 10.2139/ssrn.4120317.

[67] A. Lavie and M. J. Denkowski, "The Meteor metric for automatic evaluation of machine translation," *Machine Translation*, vol. 23, no. 2, pp. 105–115, Sep. 2009, doi: 10.1007/s10590-009-9059-4.

[68] T. Zhang, V. Kishore, F. Wu, K. Q. Weinberger, and Y. Artzi, "BERTScore: Evaluating Text Generation with BERT," Feb. 24, 2020, *arXiv*: arXiv:1904.09675. doi: 10.48550/arXiv.1904.09675.

[69] "IT Center for Science," CSC. Accessed: Jul. 12, 2024. [Online]. Available: https://csc.fi/en/

[70] "Hugging Face – The AI community building the future." Accessed: Jun. 21, 2024. [Online]. Available: https://huggingface.co/

[71] *microsoft/DeepSpeed*. (Jun. 21, 2024). Python. Microsoft. Accessed: Jun. 21, 2024. [Online]. Available: https://github.com/microsoft/DeepSpeed

[72] A. Hagen, "ZeRO & DeepSpeed: New system optimizations enable training models with over 100 billion parameters," Microsoft Research. Accessed: Jul. 12, 2024. [Online]. Available: https://www.microsoft.com/en-us/research/blog/zero-deepspeed-new-system-optimizations-enable-training-models-with-over-100-billion-parameters/

[73] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, “QLORA: Efficient Finetuning of Quantized LLMs”.

[74] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, “Is Your Code Generated by ChatGPT Really Correct?”.

[75] C. M. Hidalgo-Ternero, “Google Translate vs. DeepL: analysing neural machine translation performance under the challenge of phraseological variation,” *MonTI. Monografías de Traducción e Interpretación*, pp. 154–177, 2020, doi: 10.6035/MonTI.2020.ne6.5.

[76] A. Yulianto and R. Supriatnaningsih, “Google Translate vs. DeepL: A quantitative evaluation of close-language pair translation (French to English),” *AJELP: Asian Journal of English Language and Pedagogy*, vol. 9, no. 2, Art. no. 2, Dec. 2021, doi: 10.37134/ajelp.vol9.2.9.2021.

[77] A. Sugiyama and N. Yoshinaga, “Data augmentation using back-translation for context-aware neural machine translation,” in *Proceedings of the Fourth Workshop on Discourse in Machine Translation (DiscoMT 2019)*, A. Popescu-Belis, S. Loáiciga, C. Hardmeier, and D. Xiong, Eds., Hong Kong, China: Association for Computational Linguistics, Nov. 2019, pp. 35–44. doi: 10.18653/v1/D19-6504.

[78] “ROUGE - a Hugging Face Space by evaluate-metric.” Accessed: Jun. 27, 2024. [Online]. Available: https://huggingface.co/spaces/evaluate-metric/rouge

[79] “METEOR - a Hugging Face Space by evaluate-metric.” Accessed: Jun. 27, 2024. [Online]. Available: https://huggingface.co/spaces/evaluate-metric/meteor

[80] “chrF - a Hugging Face Space by evaluate-metric.” Accessed: Jun. 27, 2024. [Online]. Available: https://huggingface.co/spaces/evaluate-metric/chrf

Appendices

Appendix 1: Deepspeed config

```
{
 "comms_logger": {
   "enabled": true,
   "verbose": true,
   "prof_all": true,
   "debug": true
 },
 "zero_optimization": {
   "stage": 3,
   "offload_param": {
     "device": "cpu",
     "pin_memory": false
   },
   "offload_optimizer": {
     "device": "cpu",
     "pin_memory": false,
     "fast_init": true
   },
   "stage3_gather_16bit_weights_on_model_save": true,
   "allgather_partitions": false,
   "overlap_comm": false,
   "reduce_scatter": false,
   "contiguous_gradients": false,
   "round_robin_gradients": true,
   "memory_efficient_linear": true
 },
 "optimizer": {
   "type": "Adam",
   "params": {
     "lr": "auto",
     "betas": [
       0.9,
       0.999
     ],
     "eps": "auto",
     "weight_decay": "auto",
     "torch_adam": false,
     "adam_w_mode": false
   }
 },
 "scheduler": {
   "type": "WarmupDecayLR",
   "params": {
     "warmup_min_lr": "auto",
     "warmup_max_lr": "auto",
     "warmup_num_steps": "auto",
     "total_num_steps": "auto"
   }
 },
 "fp16": {
   "enabled": "auto",
   "auto_cast": true,
   "loss_scale": 0,
   "initial_scale_power": 16,
   "loss_scale_window": 1000,
   "hysteresis": 2,
   "consecutive_hysteresis": false,
   "min_loss_scale": 1
 },
 "bf16": {
   "enabled": true
 },
 "activation_checkpointing": {
   "partition_activations": true,
   "cpu_checkpointing": true,
   "contiguous_memory_optimization": true,
   "number_checkpoints": 1,
   "synchronize_checkpoint_boundary": false,
   "profile": true
 },
 "train_micro_batch_size_per_gpu": "auto",
 "gradient_accumulation_steps": "auto"
}
```

Appendix 2: Model initialization

```
def model_init(model_name, deepspeed_config, train=True, quantization=True,
sparse_attention=True):
   bnb_config = BitsAndBytesConfig(
       load_in_4bit=True,
       bnb_4bit_quant_type="nf4",
       bnb_4bit_use_double_quant=True,
       bnb_4bit_compute_dtype=torch.bfloat16,
       bnb_4bit_quant_storage=torch.bfloat16
   )

   quantization_config = bnb_config if quantization else None
   low_cpu_mem_usage = True if quantization else False
   torch_dtype = torch.bfloat16 if quantization else torch.float32
   attn_implementation = "flash_attention_2" if sparse_attention else None

   model = AutoModelForCausalLM.from_pretrained(
       model_name,
       use_cache=False, # this is needed for gradient checkpointing
       trust_remote_code=True,
       quantization_config=quantization_config,
       attn_implementation=attn_implementation,
       # low_cpu_mem_usage=low_cpu_mem_usage,
       torch_dtype=torch_dtype
   )

   if train==True:
       model.train()
       config = LoraConfig(
           r=16,
           lora_alpha=16,
           target_modules=find_all_linear_target_modue_names(model),
           # target_modules="all-linear",
           lora_dropout=0.1, #prevent overfitting
           bias="none",
           task_type="CAUSAL_LM",
           modules_to_save=["lm_head", "embed_tokens"]  # This argument serves for
adding new tokens.
       )
       model = prepare_model_for_kbit_training(model, use_gradient_checkpointing=True)
       model = get_peft_model(model, config)
```

```
return model
```

Appendix 3: Trainer code

```
dataset_size = len(dataset)
num_train_epochs = 10
batch_size_per_gpu = 1
gradient_accumulation_steps = 1

max_batch_size =  batch_size_per_gpu * gradient_accumulation_steps   #
https://www.deepspeed.ai/docs/config-json/
steps_per_epoch = math.ceil(dataset_size / max_batch_size)
max_steps = steps_per_epoch * num_train_epochs
warmup_steps = int(0.1* max_steps)

training_arguments = TrainingArguments(
       gradient_checkpointing= True,
       gradient_checkpointing_kwargs={'use_reentrant':True},
       num_train_epochs=10,
       max_steps = max_steps,
       load_best_model_at_end = False,
       save_total_limit = 2,
       push_to_hub=True,
       save_strategy = "steps",
       evaluation_strategy = "steps",
       logging_strategy = "steps",
       logging_steps = 1,
       save_steps = steps_per_epoch,
       eval_steps = steps_per_epoch,
       per_device_train_batch_size = batch_size_per_gpu,
       per_device_eval_batch_size = batch_size_per_gpu,
       gradient_accumulation_steps = gradient_accumulation_steps,
       eval_accumulation_steps=1,
       bf16= True,
       warmup_steps = warmup_steps,
       learning_rate = 0.001,
       adam_epsilon = 0.000001,
       weight_decay=0.01,
       remove_unused_columns=True,
       seed = 3407,
       output_dir = output,
       run_name=output,
       deepspeed=deepspeed_config,
   )
```

```
#early stopping not used due to deepspeed stage3 weights
trainer = Trainer(
    model = model,
    args = training_arguments,
    tokenizer = tokenizer,
    train_dataset = dataset,
    eval_dataset = dataset_eval,
    compute_metrics = partial_compute_metrics,
    preprocess_logits_for_metrics = preprocess_logits_for_metrics,
)
```

Appendix 4: Eval function

```
def evaluate(predictions, reference, translation_model_name, file_name = ""):
   model_name = "sentence-transformers/all-mpnet-base-v2"
   model = SentenceTransformer(model_name)

   rouge_metric =  load('rouge')
   meteor_metric =  load('meteor')
   chrf_metric =  load('chrf')

   length = min(len(predictions), len(reference))
   similarities = []
   for i in range(length):
       actual_response = predictions[i]
       expected_response = reference[i]

       if (actual_response and expected_response):

           generated = model.encode(actual_response)
           expected = model.encode(expected_response)

           similarity = np.dot(generated, expected) / (np.linalg.norm(generated) *
np.linalg.norm(expected))
           similarities.append(similarity)

           rouge_metric.add_batch(predictions=[actual_response],
references=[expected_response])
           meteor_metric.add_batch(predictions=[actual_response],
references=[expected_response])
           chrf_metric.add_batch(predictions=[actual_response],
references=[expected_response])

   similarities = np.mean(similarities)
   rouge_score = rouge_metric.compute()["rougeLsum"]
   meteor_score = meteor_metric.compute()["meteor"]
   chrf_score = chrf_metric.compute()["score"]

   rogue_score_formatted = "{:.3f}".format(rouge_score)
   meteor_score_formatted = "{:.3f}".format(meteor_score)
   chrf_score_formatted = "{:.3f}".format(chrf_score)
   similarities_formatted = "{:.3f}".format(similarities)
```

```
scores = {
    "scores": {
        "rouge": rogue_score_formatted,
        "meteor": meteor_score_formatted,
        "chrf": chrf_score_formatted,
        "BERTScore": similarities_formatted,
        "model": translation_model_name,
        "fileName": file_name
    }
}
return scores
```