Leverage from the EU 2014–2020

FIMA Forum for Intelligent Machines

COUNCIL OF TAMPERE REGION

AI-HUB 2.0 PROJECT REPORT:

Application Of Large Language Models in Software Engineering[1]

Muntaha Binte Alam

Tampere University

August 31st, 2023

[1]This work has been made possible thanks to the financial support from the European Union under the REACT-EU initiative, a key element of the EU's response to the COVID-19 pandemic. By fostering resilience, the EU demonstrates its commitment to supporting member states, regions, and communities in navigating through the crisis, promoting a green, digital, and inclusive recovery, and reinforcing the social fabric of Europe.

Abstract

This technical report provides an exploration of Large Language Models (LLMs) within the context of software development, offering insights into their definition, examples, applications, and integration strategies. It focuses into the essential concepts and terminology pertinent to LLMs, including natural language processing (NLP), tokenization, embedding, attention mechanisms, pre-training, transfer learning, fine-tuning, and the transformer model architecture. The report further discusses popular LLM architectures for software engineering, emphasizing cloud-based solutions and their advantages.

In addressing data preparation for LLM training, the document outlines methods for collecting, cleaning, preprocessing, and annotating both code and natural language data. It presents training techniques such as supervised and unsupervised learning, transfer learning, and fine-tuning LLMs for specific software development applications. Evaluation metrics for assessing LLM performance in a software context are also discussed, covering both intrinsic and extrinsic metrics like accuracy, precision, recall, BLEU, perplexity, METEOR, ROUGE, and CHRF score.

Tools and libraries for LLM development, including deep learning frameworks TensorFlow and PyTorch, as well as pre-trained models for code-related tasks, are reviewed. Practical guidance for fine-tuning open-source pretrained models, including step-by-step guides, tips, challenges, and issues, is provided. The document explores optimizing LLMs for software company use, focusing on model compression, pruning, quantization, distillation, and the maintenance of LLMs within the software development lifecycle. Future developments are also addressed.

Version number:**1.0**
State of publicity:**Public**

Preface

Background. The AI Hub 2.0 project is designed to enhance Tampere's role as a regional center for artificial intelligence, focusing on health, well-being, and the integration of sustainable, energy-efficient systems within industrial applications. This initiative, an expansion of the AI Hub established in 2019, aims to broaden its operational scope and impact regionally. Leveraging Tampere's strong healthcare, service, and academic foundations, the project will facilitate the adoption of AI technologies by local industries, particularly in the development of intelligent, energy-efficient machinery.

Context. This technical report is part of work of WP5, which enhances awareness of new technologies, especially the use of generative AI applications. The goal of this work package is to experiment with, introduce, assess risks, plan roadmaps, and develop related expertise in generative AI technologies like ChatGPT, both in research personnel and within the local business community. The key tehnology to learn is Large Language Model and how it can be applied, ie., trained and fine-tuned. The application domain is software engineering, which is foundational to all domains impacted by AI-HUB 2.0. Based on the results, applications can be planned for training and fine-tuning LLMs in other segments

Contents

Abstract **1**

Preface **1**

1 Introduction **5**

1.1 Defination of Large Language Model: 6

1.1.1 Example Of LLMs: 6

1.2 Applications and Use Cases of LLMs in Software Development 6

1.2.1 How LLMs are applied in Software Development . . . 7

2 Integrating LLMs into Software Development **14**

2.1 Key Concepts and Terminology 14

2.1.1 NLP 14

2.1.2 LLM 15

2.1.3 Tokenization 15

2.1.4 Embedding 16

2.1.5 Attention 16

2.1.6 Pre-Training 16

2.1.7 Transfer Learning 16

2.1.8 Fine- Tuning 16

2.1.9 Transformer model 17

2.2 Popular LLM Architectures for Software Engineering 17

2.2.1 LLMs as a Service: Cloud-based Solutions 20

2.2.2 Advantages and Considerations of Cloud-Based Solutions for Large Language Models 22

3 Data Preparation for LLM Training in Software Companies 24

3.0.1 Data Collection: Leveraging Software Documentation and Repositories . . . 24

3.1 Data Cleaning and Preprocessing: Techniques for Code and Text . . . 25

3.1.1 Handling Code data . . . 25

3.1.2 Managing Natural Language Data . . . 26

3.1.3 Data set Splitting and Annotation for Software Development Tasks . . . 27

3.1.4 Annotation Techniques . . . 28

4 LLM Training Techniques for Software Engineering 30

4.1 Supervised Learning with Software-related Datasets . . . 30

4.2 Unsupervised Learning for Code and Natural Language . . . 31

4.3 Transfer Learning for Software-specific Tasks . . . 32

4.4 Fine-tuning LLMs for Software Development Applications . . . 33

5 LLM Evaluation Metrics in Software Context 35

5.1 Intrinsic Metrics . . . 35

5.2 Extrinsic metrics . . . 36

5.2.1 Accuracy . . . 36

5.2.2 Precision . . . 36

5.2.3 Recall . . . 37

5.2.4 Bilingual Evaluation Understudy (BLEU): . . . 37

5.2.5 Perplexity: . . . 37

5.2.6 METEOR: . . . 37

5.2.7 ROUGE: . . . 38

5.2.8 CHRF Score: . . . 38

6 Tools and Libraries for LLM Development in Software Companies 39

6.1 Frameworks for Deep Learning . . . 39

6.1.1 TensorFlow . . . 40

6.1.2 PyTorch . . . 40

6.2 Pre-trained Models for Code Related tasks 42
6.2.1 Starcoder 42
6.2.2 CodeT5 43

7 Practicalities For Fine tuning Open Source Pretrained Models 45
7.0.1 Step by Step Guide in Fine tuning 45
7.0.2 Tips for Fine tuning 50
7.0.3 Challenges and Issuses 51

8 Optimizing LLMs for Software Company Use 53
8.1 Model Compression and Pruning for Efficient Deployment . . 53
8.2 Quantization and Distillation for Resource-Constrained Environments 54
8.2.1 QLORA and LORA: 55
8.3 Monitoring and Maintaining LLMs in Software Development Lifecycle 55

9 Future Trends and Research Directions 57
9.0.1 Challenges in LLM Development for Software Companies 57
9.1 Future Directions of LLM and Software Engineering 59
9.2 Truthfulness of LLMs 60
9.3 Ethical AI and Responsible LLM Development in Software Industry 60

10 Conclusion 62
10.1 Summary of Key Findings for Software Companies 62
10.2 Recommendations for LLM Practitioners in Software Development 63
10.3 Final Thoughts 64

Chapter 1

Introduction

Large Language models (LLM) have been a buzzword that took the internet by storm in recent months, especially after the arrival of ChatGPT. As an answer to this newly launched language model, other LLMs such as Google's bard have been introduced which proves that LLMs are here to stay for a longer time Openai.

Additionally, language models are expanding opportunities since they can automate operations, reduce costs and time, and improve task accuracy. These are a few of the elements that make software companies interested in implementing LLMs in their operations. Large language models, however, are a recent innovation in computer science. Business executives might not be knowledgeable about such models as a result. We penned this study to educate interested software company executives in large language models:

- Definitions
- Applications and Use Cases of LLMs in Software Development
- Challenges and Ethical Considerations in Software Context

1.1 Defination of Large Language Model:

LLM is a type of machine learning model that produces outputs for various natural language processing (NLP) tasks, such as text generation, question answering, and machine translation. Large language models are often trained on enormous volumes of text data, frequently consisting of billions of words, and are typically built on deep learning neural networks such as the Transformer architecture. Larger models, like Google's BERT model, can produce results for a variety of tasks because they are trained with a large data set from a variety of data sources.[Wodecki, July 2, 2022] [Vaswani, 2017]

1.1.1 Example Of LLMs:

Though **ChatGPT** is the most commonly known LLM that has gained attraction these days, but a lot of other LLMs have been here which are also quite worth the mentions.

To name a few common LLMs, The table below introduces the 7 largest large language models by parameter size :

1.2 Applications and Use Cases of LLMs in Software Development

The initial research in the computer science field has indicated that LLMs can help with software engineering jobs quite well. Code generation models like GitHub's CoPilot is one of the examples of the various tasks LLM can assist in software engineering tasks. In this section, we will concentrate on the implications of LLMs in the software business and their potential for software development.[Ozkaya, May-June 2023] Though it may seem like only software engineering depends on the Large language model for different purposes, the

Model	Developer	Parameter Size
WuDao 2.0	Beijing Academy of Artificial Intelligence	1.75 trillion
MT-NLG	Nvidia and Microsoft	530 billion
Bloom	Hugging Face and BigScience	176 billion
GPT-3	OpenAI	175 billion
LaMDA	Google	137 billion
ESMFold	Meta AI	15 billion
Gato	DeepMind	1.18 billion

Table 1.1: Examples of LLMs
[AIMultiple, 2023]

developments of LLMs also need to interact with software professionals in various aspects. In order to examine the interactive relationship between LLM and software engineering, we will also briefly cover this viewpoint in this section.

1.2.1 How LLMs are applied in Software Development

The process of software development involves quite a lot of steps, LLMs can be implemented in many of them. Here, we would like to focus on how Large language models are implemented in those steps.

Requirement Generation :

Software Engineers must have a thorough understanding of every need before they can create a system or piece of software. But doing so can be pretty laborious as requirements documents are detailed and complex sources of knowledge. If we have a database with thousands of requirements, engineers would have to manually connect or group requirements together by hand

without LLM. It would have been frustrating, quite time-consuming, and still prone to errors. With LLMs, things are faster and there is less room for mistakes. The language models revolutionize the process by giving requirement engineers significant information and technological expertise.LLMs are essential in producing precise and contextually aware customer stories, descriptions of products, and feature recommendations. The development team might give explicit guidelines and criteria to LLMs so they can write narratives that support the project's goals. By considering a variety of options and the views of users, this method enables it easier to design and specify project requirements[Abbas]. LLM can not only help to set up the requirements, but they can also help in prioritizing or validating the requirements. By looking for contradictions, ambiguities, or conflicts in the requirement documents, language models can help in requirement validation.[Hou et al., 2023] They can help make sure that the requirements are precise and practical by offering suggestions or clarifications. Following that, the requirements' completeness is assured, strengthening the software development process[Abbas].

Implementation / App development :

Implementation of app development is the actual process when all the source code for a piece of software is written. This process is time-consuming, cumbersome, and sometimes tiring and challenging which calls for automatic code creation. Code generation is a process in which source code is automatically generated based on functional requirements such as natural language descriptions or pseudo-code algorithms. In recent years, large pre-trained language models such as AlphaCode and the GPT3 series have demonstrated impressive capabilities in code generation. Other open-source code generation models include GPT-Neo[Sid Black and Biderman., 2021], GPT-J [Wang and Komatsuzaki, 2021], CodeParrot[Thomas Wolf and Rush., 2020], PolyCoder[Frank F. Xu and Hellendoorn., 2022], and InCoder[Daniel Fried and Lewis., 2022.].

Software developers can give large language models high-level descriptions of what they want the code to do, and the LLMs will produce the relevant

code. As a result, less manual coding will be needed during the initial stages of software development.[Jiang et al., 2023]

Automatic code generation is not the only step where LLMs can contribute, rather they can be useful for debugging, error handling, restructuring, translation, and domain-specific code generation.
Coding errors can be found and fixed with the help of large language models. They are able to examine code samples, spot potential mistakes, and recommend changes or different strategies. This can speed up debugging and increase code quality [Hou et al., 2023].

Large language models can make code completion suggestions based on the context and patterns found in the source. They can help developers write code more quickly and easily by suggesting names for functions, variables, or even complete snippets of code. This can boost output greatly and cut down on syntax errors.[Ciniselli et al., 2021].

Code snippets can be analyzed by language models, which can then suggest refactoring changes to enhance readability, performance, or quality. They can point out redundant code, recommend improved code structure, or offer more effective strategies. This can aid developers in best practices and code optimization.
In order to produce code specifically suited for a given domain or framework, language models can be fine-tuned.[Wang et al., 2021] For instance, models can be trained on certain frameworks or libraries, such as TensorFlow or Django, allowing them to produce code unique to those tools.

Automated Testing with LLMs

Software testing after development is an essential component of software development that takes a lot of time, resources, and focus. Implementing Large Language model in this phase can actually help businesses to provide a better outcome. The requirements that have been introduced in requirement generation phase, or the codes that have been written ; based upon that ; Large Language model can assist in generating test cases.

With the purposes of generating test cases, the LLMs can generate realistic and diverse data sets by understanding data dependencies and as a result the efficiency and coverage of automated tests can be enhanced . They can analyze test case requirements and produce corresponding test scripts which reduces the manual effort required for scripting [Schäfer et al., 2023]

LLMs can understand code semantics, extract patterns, and interpret natural language descriptions related to software functionalities which involves their natural language processing and code comprehension capabilities. With the use of language models, test results, logs, and reports can be analyzed.[Hou et al., 2023] By examining error messages, log entries, or natural language descriptions, the LLMs can assist in discovering trends, patterns, or typical failure circumstances which help with trouble shooting and issue identification.

Additionally, LLMs can help create test cases from descriptions in normal language, promoting improved cooperation between developers and testers. They also assist in identifying test coverage gaps and make pertinent test case recommendations, ensuring thorough testing and lowering the possibility of issues going undetected. LLMs contribute to the creation of more dependable and high-quality software products by increasing test effectiveness and efficiency. [Hou et al., 2023] [Schäfer et al., 2023] .

Language models be considered as intelligent oracles for automated testing. By comparing the expected output or behavior of a system with the actual results obtained during testing, they can help identify deviations, abnormalities, or unexpected outcomes. This can improve the accuracy and reliability of the automated tests.

Deployment

The deployment stage of software developments starts with environment set up and LLM can assist in setting up the environment by analyzing requirements and generating instructions for installing dependencies or libraries.

This step will ensure that the deployment environment is properly prepared to run the software[Gong et al., 2023].

Language models can aid in automating the deployment process by generating deployment scripts or configuration files. They can analyze deployment requirements, infrastructure specifications, and deployment best practices to generate scripts that automate the deployment of the software in various environments.

By analyzing infrastructure requirements and specifications, large language models can provide guidance or generate code snippets for provisioning infrastructure resources [Wan et al., 2022].

LLMs are capable of supporting CI/CD (Continuous Integration/Continuous Deployment) pipelines by analyzing code repositories, build scripts, and deployment configurations. By automating build processes, identifying dependencies, and generating CI/CD pipeline configurations, they can enable smoother and more efficient software deployment workflows[Hou et al., 2023].

Language models can generate release notes and documentation for software deployments.They are able to create release notes that highlight the new features, bug fixes, and known issues while also summarizing the changes. Additionally, they can help create or update deployment documentation, which will make it simpler for users or administrators to comprehend and implement the deployment procedure[Arakelyan et al., 2023].

They can provide suggestions or generate configurations based on environment-specific requirements which ensures that that the software is correctly configured for deployment in different settings.

By examining logs, monitoring data, or system metrics, large Language models can help validate the implementation. They can offer insights for troubleshooting or performance optimization and assist in locating perfor-

mance bottlenecks or other issues in the installed software.

Maintenance

Large Language models can still be helpful in software development process during the maintenance phase. They can assist in locating the underlying causes of problems and offer insights for troubleshooting and resolving them by examining error logs, user input, or system activity.

In some cases, language models can generate automated bug fixes or code patches based on error reports or issue descriptions.[Chen et al., 2023] They can analyze the code base, understand the context, and suggest potential fixes for common or repetitive bugs. These suggestions can then be reviewed and applied by human developers.

Large Language models can offer suggestions for refactoring the code to enhance its efficiency, quality, and maintainability. They can look for anti-patterns or assess the code base and recommend refactoring or different implementations that follow best practices.

Large language models can help optimize software performance by analyzing code, logs, or system metrics. They can identify potential bottlenecks, suggest performance optimizations, or guide developers in applying performance tuning techniques to enhance the efficiency and responsiveness of the software[Hou et al., 2023].

Language models can assist in identifying potential security risks in the software by analyzing code, configuration files, or security guidelines, they can provide insights and recommendations to enhance the security posture of the system. This can include suggestions for secure coding practices, vulnerability scanning, or access control improvements.[Alqarni and Azim, 2022].

Large Language models analyze system logs and monitoring data to identify patterns, anomalies, or recurring issues. By processing and understanding the log entries, they can provide insights into system behavior, performance degradation, or potential issues that require attention.

Language models can support the maintenance of knowledge bases, FAQs (Frequently asked questions), or documentation [Li et al., 2022].They can create or update important documentation by analyzing new data, updates, or modifications to the system. This guarantees that the documentation is accurate and represents the software's current status. .

Language models are capable of assessing the impact of code changes or system updates. By analyzing the code base and relevant dependencies, they can identify potential areas affected by changes and provide insights into potential risks or conflicts that need to be addressed.

Chapter 2

Integrating LLMs into Software Development

One of the reasons large language models are being popular in different business is that they can make the work easier and smoother and also improve the work quality.Consequently, integrating Large Language Models (LLMs) can greatly improve a variety of activities involving natural language processing (NLP) and code generation in the field of software development.This process of integration involves understanding key concepts and terminology, exploring popular LLM architectures for software engineering, and considering cloud-based solutions for accessing LLM capabilities. It's crucial to get to know certain basic terms and concepts in order to comprehend the integration process.

2.1 Key Concepts and Terminology

2.1.1 NLP

NLP is a branch of artificial intelligence (AI) that aims to make it possible for computers to recognize, interpret, and produce human language.[Ruth Brooks]It includes activities like sentiment analysis, named entity identification, machine translation, and text categorization.

2.1.2 LLM

LLMs are deep learning models that have been trained on vast volumes of text data in order to discover linguistic patterns and structures [Hadi et al., 2023]. They can produce text that is human-like and are useful for a variety of NLP applications, including sentiment analysis, language translation, code generation, and more.

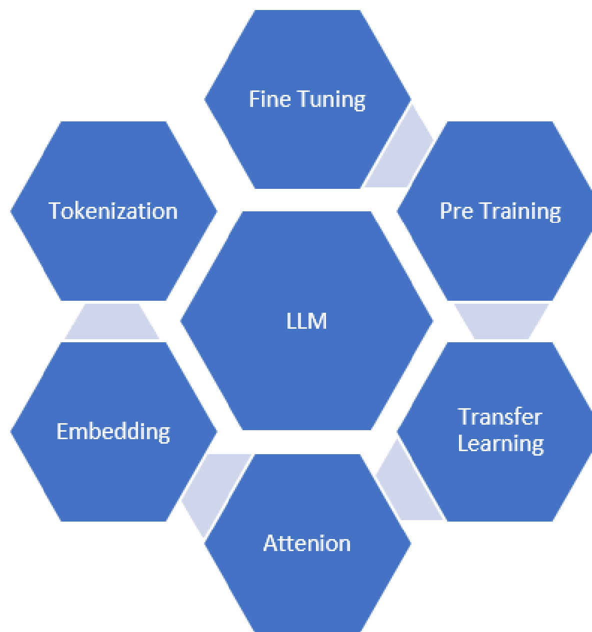

Figure 2.1: LLM Keywords (need to revise and change the caption)

2.1.3 Tokenization

Tokenization is the process of breaking down text into smaller units called tokens.Depending on the chosen tokenization approach, tokens could be words, subwords, or characters. In NLP, tokenization is a basic step that enables models to process text in detail [Roumeliotis and Tselikas, 2023]

2.1.4 Embedding

In a high-dimensional space, the technique of embedding is utilized to represent words or tokens as continuous vectors. Word embeddings record the semantic and syntactic connections between words, allowing models to comprehend the meaning of words in the context of a particular text [Wikipedia, 2023]

2.1.5 Attention

LLM architectures must have attention mechanisms. They enable the model to generate output by focusing on various segments of the input sequence. The model's capacity to recognize long-range dependencies and its comprehension of context are both improved by attention processes.

2.1.6 Pre-Training

An LLM is pre-trained using a sizable corpus of text data to gain general knowledge and linguistic patterns. Pre-training helps the model develop a solid grasp of syntax, semantics, and contextual relationships as it learns to predict missing words or sentences [Hendrycks et al., 2019]

2.1.7 Transfer Learning

Transfer learning is a strategy for using information gained from one job to enhance performance in a related task. LLMs can be fine-tuned on certain downstream tasks, like code generation or natural language interpretation, to adjust their learnt representations to the target domain after being pre-trained on large-scale data-sets [Tormos et al., 2022]

2.1.8 Fine- Tuning

A pre-trained LLM is further trained on a particular task or data-set as part of the fine-tuning phase. The model can fine-tune its parameters to better

match the requirements of the target task or area [Tormos et al., 2022].

LLMs can be fine-tuned to better perform on particular software engineering activities like code generation and code completion by including domain-specific knowledge.

2.1.9 Transformer model

A transformer model is a neural network that follows relationships in sequential input, such as the words in this sentence, to learn context and subsequently meaning.

Transformer models use a growing collection of procedures calledd attention or self-attention to find hidden relationships between even distant data pieces in a series.[nvidia]

2.2 Popular LLM Architectures for Software Engineering

There are three main types of large language models (LLMs) based on the transformer model architecture :

Autoregressive Language Models

Auto-regressive models generate text by predicting the next word in a sequence based on the previous words.They are trained to maximize each word's likelihood given its context within the training data set. The GPT (Generative Pre-trained Transformer) series from OpenAI, of which GPT-4 is the most recent and powerful version, is the most well-known illustration of an autoregressive language model [Recchia, 2021]

Autoencoding Language Models

On the other hand, Auto encoding models discover how to create a fixed-size vector representation (**embeddings**)of input text by reconstructing the original input from a corrupted or hidden version of it. These models are trained to predict words that are either missing or masked in the input text by utilizing the surrounding context.[HuggingFace, b] One of the most well-known models for automatic language encoding was created by Google; which is BERT (Bidirectional Encoder Representations from Transformers). For a range of Natural language processing tasks, including sentiment analysis, named entity identification, and question-answering, it can be fine-tuned The third option is the combination of autoencoding and autoregressive .T5 model is one of the example of such a model [Du et al., 2022] Several LLM architectures have shown promising results in the field of software engineering. Some popular architectures include:

GPT-3 (Generative Pre-trained Transformer 3)

GPT-3 (Generative Pre-trained Transformer 3) is an auto regressive model which can generate output tokens one at a time, conditioned on the previously generated tokens.GPT-3 uses a transformer architecture with multiple layers of self-**attention mechanisms** to model the dependencies between the input and output tokens. GPT- 3 models can be applied to code generation, documentation, and code completion in software development.Developers can now fine-tune GPT-3 on their own data, creating a custom version tailored to their application.[OpenAI] Customizing makes GPT-3 reliable for a wider variety of use cases which can include tasks related to software engineering.

Codex

An LLM, or artificial intelligence model, such as Codex, which has been trained to predict text to follow a given string of input text. Codex is a fine-tuned version of OpenAI's GPT-3 which means that it inherits GPT-3's language capacity and is given further training on a wide range of programming languages [Sam Manning and Eisner4., 3/3/2022].Due to its natural language processing characteristics, it is remarkably capabilities to general-

ize to a variety of coding jobs, including code generation, code completion, code repair, code translation, and code question answering. These features have made it useful for a variety of practical tasks, including providing documentation or unit tests for code snippets, finishing partially written code, writing explanations for code snippets, and correcting errors in code. It can also generate code from natural language descriptions.

Bert

BERT (Bidirectional Encoder Representations from Transformers) is an autoencoding language model that can be fine-tuned for various software engineering tasks such as code classification, code summarization, bug detection, and natural language understanding in software-related documents. BERT's ability to capture contextual information and its pre-training on a large corpus of text make it effective for understanding code and text in software engineering applications.

Bert-SE a model based on Bert has been introduceed for software engineering domain which is destined for textual classification in the field of software engineering.[Eliane Maria, 18/9/2020]

Code-Bert

Code-BERT is bimodal pre-trained model based on Bert architecture for natural language (NL) and programming language (PL) like Python, Java, JavaScript, etc. CodeBERT can capture the semantic correlation between natural language and programming language.It generates general-purpose representations which can extensively serve both NL-PL generation (code documentation generation) and comprehension (natural language code search) tasks. [Zhangyin Feng1, 18/9/2020]

Code2Vec

Code2Vec is a model which learns distributed representations of code snippets. It can be used for tasks such as code similarity detection, code rec-

ommendation, or even code clone detection. By encoding code snippets into vectors,we can compare or match them based on their semantic similarities . The model is more focused on learning code representations rather than generating or predicting code sequences.[URI ALON, 30/10/2018]

Transformer-XL

Transformer-XL is a variant of the transformer model that is designed to handle longer sequences and capture longer-range dependencies. It can be useful for software engineering tasks that involve analyzing and understanding large code bases or lengthy software documentation.

These are just some examples of the models that can be used for software engineering related tasks, but there exists also other models that can be useful in this context.

2.2.1 LLMs as a Service: Cloud-based Solutions

Large Language models are increasingly being offered as a service through cloud-based solutions, providing convenient access to their capabilities and Companies can get the most potential from Large Language Models.Several well-known platforms and services follows as below:

OpenAI API

Developers can make API calls online as the OpenAI API is hosted on OpenAI's cloud infrastructure. The potential of OpenAI's models can thus be utilized without the need to build and maintain own computational infrastructure.[Greg Brockman]The cloud-based solution from OpenAI is built to manage many API requests and can grow to suit any application's needs. This guarantees that program can keep performance while managing heavy

workloads. A simple method for incorporating the models into own programs or services is the OpenAI API. It is simple to interact with the API and process the results because developers can perform API calls using conventional HTTP requests and receive responses in JSON format.

Hugging Face

Hugging Face is a popular platform for NLP models with a wide range of pre-trained Large Language models. These models can be useful for user specific tasks such as text generation,text classification , sentiment analysis etc. Hugging face provides a library and API for easy integration of LLMs into different tasks which can be related to software engineering. They are continuously introducing new models in their website.[HuggingFace, a]

Microsoft Azure ML

The Microsoft Azure ML platform provides a range of cloud-based machine learning capabilities, including LLMs [azu, 2023].It offers tools for training, fine-tuning, and deploying Large Language Models in software development projects.Azure ML offers a versatile and adaptable framework that can be tailored to different software engineering use cases. This process allows developers to leverage machine learning techniques to enhance the processes and results of software development.

Google Cloud AI Platform

Google Cloud AI is a cloud-based platform provided by Google that offers a wide range of tools and services for building, deploying, and managing machine learning models [goo, A]. It provides a scalable and flexible infrastructure to support various software engineering tasks.It supports training models on powerful GPU and TPU accelerators, enabling faster training and experimentation on large data sets. Google Cloud AI Platform supports version control and collaboration for managing models, experiments, and code. It can integrate with Git and provides services for tracking changes, managing code repositories, and enabling collaboration among team members.

2.2.2 Advantages and Considerations of Cloud-Based Solutions for Large Language Models

Undoubtedly, cloud-based LLM services enable developers to take advantage of the power of language models without the need for significant resources from their end. Yet, we cannot help considering the challenges and issues related in this context.

Accessibility: Pre-trained models are easily available through cloud-based LLM services which makes it easier for developers to leverage them into their applications without requiring a considerable amount of processing power. Language-related functionality can be developed and deployed more quickly thanks to this accessibility.

Scalability: Depending on demand, cloud platforms can dynamically scale their processing capacity. This scalability, which enables effective resource usage and supports variable workloads, is especially helpful during the fine-tuning or inference stages of LLMs.

Cost-effectiveness: Software businesses can save money by using cloud-based LLM services rather than purchasing pricey hardware up front. It is more cost-effective because they can pay for the resources they actually use. Better cost control and scalability are made possible by this pay-as-you-go concept. Cloud service providers use strong security precautions to protect users' privacy and data.

Security and Privacy: To guarantee the security of sensitive code and data, cloud providers adopt strong security measures and data protection processes. Developers can benefit from the security infrastructure offered by the cloud provider by utilizing cloud-based LLM services which reduces potential security risks.

However, there are other factors to take into account and difficulties that come with using LLMs as a service, such as:

Cost management: While cloud-based solutions provide flexibility, it's crucial to monitor and optimize costs while fine-tuning or employing LLMs in production to prevent unforeseen costs.

Dependency on External Services: Software firms that rely on cloud-based LLM services are reliant on the accessibility and dependability of the service provider.

Privacy Issues: When dealing with proprietary or sensitive code, privacy concerns may arise when utilizing cloud-based solutions. Analyzing data privacy rules and ensuring compliance with legal and regulatory standards are necessary for this purpose.

Chapter 3

Data Preparation for LLM Training in Software Companies

Effective data preparation is one of the most important task in the training of Large Language Models (LLMs) for software organizations. This section addresses the difficulties associated with LLM training in software development environments by concentrating on the critical phases in data preparation. Software businesses can gather, clean, pre-process, split, annotate, and enhance the data to guarantee the best training outcomes by utilizing software documentation, repositories, and other pertinent sources.

3.0.1 Data Collection: Leveraging Software Documentation and Repositories

Gathering Comprehensive Software Documentation

The process can be initiated by identifying the documentation sources. We need to determine the various sources of software documentation relevant to data collection. These may include user manuals, technical specifications, design documents, API documentation, code comments, issue trackers, and version control repositories. The data collection may also include external

resources such as forums, online communities, and knowledge bases related to the software. These sources can contain valuable information, discussions, and best practices that can add to the documentation collection process.

Extracting Data from Software Repositories

Several techniques exists for extracting code snippets from version control systems like **Git, SVN, or Mercurial**.One approach is to analyze the differences (diff) between consecutive commits in the version control system. These differences provides information about the added, modified, or deleted lines of code. By parsing the differences, developers can extract the relevant code snippets that were changed or added in each commit. Another technique is to extract code snippets at the file level. Developers can iterate through the files in each commit and extract the code snippets based on specific criteria, such as function definitions, class definitions, or code blocks enclosed within specific markers or annotations.

3.1 Data Cleaning and Preprocessing: Techniques for Code and Text

3.1.1 Handling Code data

In the process of cleaning code data for further use, we may face some challenges which can include removing comments, normalizing formatting, handling variable names, and addressing syntactical inconsistencies. By applying suitable techniques both code and text data can be cleaned and prepossessed [Hou et al., 2023] which will result in more consistent, meaningful, and manageable data for further analysis, modeling, or training of large language models.

Removing Comments

When developing software , programmers add comments to keep a track on the task ,but these code comments often contain non-executable text and

should be removed to focus solely on the executable code. This can be achieved by identifying and stripping out comment lines or blocks.

Duplicated instance deletion

It is common and frequent for duplicate instances. When prepossessing these data, duplicated instance deletion techniques can be applied to remove duplicate samples from the dataset which can ensure data integrity [Xu et al., 2022]

Intial Data Segmentation

This step is important as large language model may not be able to long sequence of undtructed data. Therefore, initial data segmentationcan help in splitting data into various categories as required ; For example, to split into sentences or words [Kou et al., 2023] [Hou et al., 2023].

3.1.2 Managing Natural Language Data

When cleaning and prepossessing textual data some techniques such as removing noise, handling special characters, dealing with stop words, and applying stemming or lemmatization should be focused on.[Thanaki, 2017]

Removing Noise

Textual data may contain irrelevant or noisy elements, such as HTML tags, special characters, or URLs. Cleaning techniques like regular expressions or library functions can be used to remove or replace such noise.

Handling Special Characters:

Special characters, such as punctuation marks or non-alphanumeric symbols, may not contribute significantly to the meaning of the text. Removing or replacing them can help streamline the data. Techniques like character filtering or Unicode normalization can be employed.[Thanaki, 2017]

Dealing with Stop Words:

Stop words are common words ; for example; "the," "is," or "and" .They often appear frequently in text but carry little semantic value. Removing stop words can reduce noise and focus on the more informative content. Stop word lists or libraries can be used for this purpose.

Applying Stemming or Lemmatization:

Stemming and lemmatization are techniques to reduce words to their base or root form, capturing their essential meaning. This helps in reducing vocabulary size and dealing with word variations. Stemming algorithms or lemmatization libraries (e.g., NLTK)[nltk] can be utilized.

3.1.3 Data set Splitting and Annotation for Software Development Tasks

This section focuses on exploring the approaches for dividing the data set into training, validation, and testing sets to ensure proper evaluation and model performance estimation.

Random Splitting

In random splitting, the order of the data set's index is used to produce the training, validation, and test sets. Therefore, before dividing, the entire data set should be shuffled to avoid problem with class imbalance. [medium] This approach ensures an unbiased distribution of data across the splits but may not take into account specific characteristics or dependencies within the data set.

Stratified Splitting

In stratified splitting, the data set is divided while maintaining the distribution of specific attributes or labels across the splits. This is particularly

useful when dealing with imbalanced data sets or when certain attributes are crucical for model evaluation. [ludwig]

Time-based Splitting

In this process, the data set splitting is done based on a temporal order, where the training set contains data from earlier time periods, the validation set includes data from a more recent period, and the testing set represents the most recent data. This is common in scenarios where the model needs to generalize to future unseen data.[ludwig][Chalokia]

Cross-validation

In Cross Validation, the data set is partitioned into multiple subsets or folds, with each fold used as a validation set while training the model on the remaining folds. Therefore, cross validation provides more robust evaluation by utilizing all data for training and validation.

3.1.4 Annotation Techniques

Data annotation, also known as data labeling ; as these terms are used interchangeably, is the task of labeling objects for machine learning algorithms in data-sets. This section discusses methods for annotating the data with task-specific labels or tags to facilitate supervised or semi-supervised learning approaches.

Manual Annotation

In manual annotation, human experts annotate the data by assigning labels or tags based on predefined criteria. Although this can be a time-consuming process, it provides high-quality annotations tailored to the specific task.

Automated Annotation

In automated annotation, we leverage existing tools or algorithms to automatically assign labels or tags to the data. This can involve techniques such

as keyword matching, topic modeling, or machine learning-based approaches. Automated annotation can speed up the process but may require additional validation and refinement.

Active Learning

Active learning is the process of combining manual and automated annotation by starting with a small labeled data set and iterative selecting informative samples for human annotation based on the model's uncertainty or confidence. This reduces the annotation effort while achieving effective training data.

Chapter 4

LLM Training Techniques for Software Engineering

For training large language models for software engineering related task , the core knowledge about different techniques for training LLM is important. This idea helps to efficiently choose which approach will be helpful for that specific task with task-specific data set. These technique may include supervised learning, unsupervised learning, transfer learning and fine tuning and this chapter will focus on these techniques.

4.1 Supervised Learning with Software-related Datasets

Supervised learning, also known as supervised machine learning, is defined by its use of labeled data sets to train algorithms that to classify data or predict outcomes accurately. The model modifies the weights when input data is fed into it until the model is adequately fitted; which happens because of the **cross-validation** process. [ibm].

In software engineering tasks, the technique will involve using labeled software-related data sets, where each input example is associated with a corresponding target label. In the context of software engineering, the input examples

can include code snippets, natural language descriptions, or a combination of both [Hou et al., 2023]

To train a language model using supervised learning, the developers would typically provide the input examples along with their corresponding labels to the model during the training process. The model then learns to map the input examples to the appropriate output labels based on the provided training data. This approach can be used for various software engineering tasks, such as code classification, code summarization, and more.

To help the model generalize well to new data, it is crucial to make sure the data collection includes a variety of scenarios and examples.

4.2 Unsupervised Learning for Code and Natural Language

Software engineering can also use unsupervised learning methods to develop language models. Unsupervised learning, in contrast to supervised learning, relies on the discovery of patterns, correlations, and structures within the data itself rather than on labeled data.

Language models can learn representations and embeddings that capture important information from the input data without the use of explicit labels by utilizing unsupervised learning.

One popular unsupervised learning technique is pre-training language models [Ge et al., 2023] using methods like auto encoders, generative adversarial networks (GANs), or self-supervised learning. By predicting missing pieces, fixing corrupted data, or resolving deceptive problems, these strategies enable the model to learn meaningful representations of code and natural language.

4.3 Transfer Learning for Software-specific Tasks

In transfer learning, the knowledge of an already trained machine learning model is applied to a different but related problem. With transfer learning, we generally try to re-use what has been learned in one task to improve generalization and performance in another task. We move the weights that a network has picked up in "task A" to a fresh "task B." The most typical approach is to apply what a model has learnt from a task with a lot of labeled training data to a new task with little to no training data. We begin the learning process using patterns discovered while completing a comparable task, as opposed to starting from scratch. [builtin].

In the context of software engineering, transfer learning can be applied to train language models on a large, general-purpose data set and then fine-tune them on specific software-related tasks.

By using transfer learning, language models can benefit from the general knowledge learned during pre-training, which helps them bootstrap their understanding of software-specific tasks with limited task-specific training data. This approach can lead to better performance, especially when the target task has a small or limited labeled data set [Tormos et al., 2022] For transfer learning in software engineering, a common approach is to use a large-scale language model pre-trained on a vast corpus of code and natural language, such as GitHub repositories or Stack Overflow. This pre-trained model can then be fine-tuned on specific software-related tasks, such as code completion, bug detection, or code summarization, using task-specific labeled data.

Transfer learning can be used for code-related tasks by pre-training transformer models on a generic data set using a self-supervised task, such as filling masked words in sentences. Then, these models are fine-tuned to support specific code-related tasks, such as automatic bug-fixing, injection of code mutants, generation of assert statements, and code summarization.
A single model can be fine-tuned to support multiple tasks, possibly exploit-

ing the benefits of transfer learning. This means that knowledge acquired to solve a specific task can be useful to boost performance on another task. The Text-To-Text Transfer Transformer (T5) model is a pre-trained transformer model that has been used to support code-related tasks. The T5 model achieved better performance compared to state-of-the-art baselines in the four code-related tasks mentioned above.[Mastropaolo et al., 2022]

4.4 Fine-tuning LLMs for Software Development Applications

Fine-tuning in large language models (LLMs) involves re-training pre-trained models on specific data sets which allows the model to adapt to the specific context of the business needs. This process can help to create highly accurate language models, tailored to own specific business use cases.[simform].

In the context of software development applications, fine-tuning large language models involve taking a pre-trained model that has been pre-trained on a diverse data set containing code and natural language, and updating its parameters on a task-specific data set. By fine-tuning on task-specific data, the LLMs can adapt to the specific requirements and nuances of software engineering tasks which will lead to improved performance on the target task.

Large language models (LLMs) pre-trained on vast source code ("Code LLM") have achieved remarkable progress in code intelligence. For instance, with the help of AI generative tools, software developers can now create and maintain their codes easily, and eventually improve their productivity significantly.[Yue Wang]

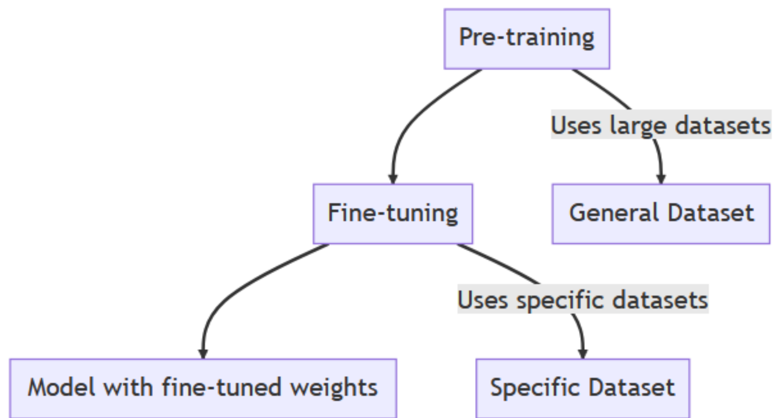

Figure 4.1: LLM fine tuning (need to revise and change the caption)

Chapter 5

LLM Evaluation Metrics in Software Context

In the software context, Large Language Model's evaluation metrics are used to assess the performance of language models specifically for software-related tasks. Evaluating language models in this context is crucial to ensure they are effective, accurate, and reliable.The choice of evaluation metrics depends on the specific software application and use case. Different tasks require different metrics, and a combination of metrics is often used to provide a comprehensive evaluation of a language model's performance in the software context. LLM Evaluation Metrics can be categorized into two main types: Intrinsic Metrics and Extrinsic Metrics. In this chapter, we would like to focus on these metrics.

5.1 Intrinsic Metrics

Intrinsic evaluation aims to measure the quality of embedding by assessing their performance on specific Natural language processing tasks. These tasks are related to the embedding space itself, such as word similarity, analogy, and classification. [Safjan] Intrinsic evaluation captures how well the model captures what it is supposed to capture on test sets from the corpus.

5.2 Extrinsic metrics

Extrinsic metrics evaluate a machine learning model based on its performance on specific tasks or real-world applications. These metrics are often used to assess how well the model will perform in the intended use case. The intrinsic metrics that are used to evaluate NLP systems are as follows:

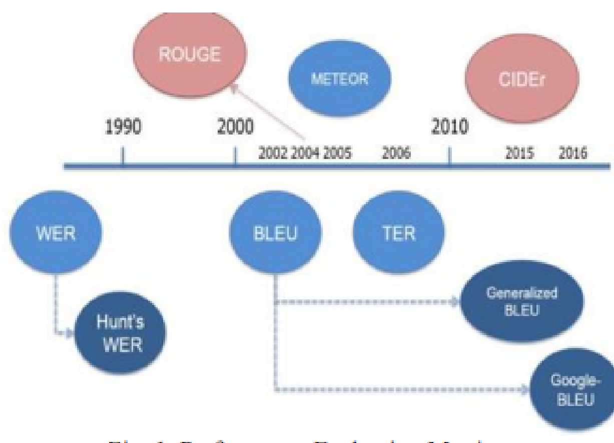

Figure 5.1: LLM evaluation metrics
[Yashaswini and Shylaja]

5.2.1 Accuracy

In classification jobs, the accuracy metric is employed to determine how closely a measured value resembles a known value. When the output variable is discrete or categorical, it is generally employed .
For instance, how often a sentiment classification algorithm is correct.

5.2.2 Precision

The accuracy metric would provide the proportion of labels with positive labels compared to cases with positive labels assigned by the classifier. For example, if identifying a cancer that is prevalent 1 per-chant of the time, a model that always spits out “negative” will be 99% accurate, but 0% precise.[Yashaswini and Shylaja]

5.2.3 Recall

Recall measures how well the model can recall the positive class. Recall value signifies the number of positive labels that the model has correctly identified as positive [Yashaswini and Shylaja]. Precision and Recall are complementary metrics that have an inverse relationship.[Afonja, 2017] If both metrics are equally important then the F1 score can be used to combine precision and recall into a single metric.

5.2.4 Bilingual Evaluation Understudy (BLEU):

The BLEU score evaluates the quality of text that has been translated by a machine from one natural language to another [Yashaswini and Shylaja].A measurement of performance for evaluating the effectiveness of machine translation models is the BLEU Score. It assesses how effectively a model interprets in different languages.

5.2.5 Perplexity:

Perplexity measures how well a language model predicts a given text. It quantifies the level of uncertainty or surprise of the model when predicting the next word. A lower perplexity score indicates that the model is better at predicting the next word and, therefore, has a better understanding of the language it is processing.

5.2.6 METEOR:

A precision-based metric for assessing the output of machine translation is the Metric for Evaluation of Translation with Explicit Ordering (METEOR). It avoids some of the BLEU score's drawbacks, like exact word matching when computing precision. Synonyms and stemmed words can be matched with a reference term using the METEOR score.

On the basis of stemmed words and meanings, the n-grams can be matched. METEOR calculates a score using unigram precision and recall.

5.2.7 ROUGE:

The evaluation metric Recall-Oriented Understudy for Gisting Evaluation (ROUGE) assesses recall. It is frequently employed in machine translation activities and for assessing the caliber of output content. However, as it assesses recollection, summary tasks are where it is most frequently utilized.

5.2.8 CHRF Score:

Character level n-grams are crucial for the automatic evaluation of more intricate metrics. The linguistically driven n-gram based F-scores, particularly those based on Part-of-Speech tags and morphemes outperform popular metrics like BLEU and TER in terms of correlating with human assessments [Yashaswini and Shylaja]

Chapter 6

Tools and Libraries for LLM Development in Software Companies

The development of language models involves complex algorithms, vast amounts of data, and powerful computational resources. These models are trained on extensive corpora of text data and fine-tuned to perform specific tasks. To achieve these goals, the companies needs to understand the resources, tools for further implementation in the companies.The following sections will delve into the specific tools and libraries that enable LLM development in software companies, providing a comprehensive guide for practitioners, researchers, and decision-makers in the field.

6.1 Frameworks for Deep Learning

The development of sophisticated language models requires powerful deep learning frameworks that provide the necessary tools, libraries, and functionalities. These frameworks enable researchers and developers to design, train, and deploy algorithms. Here's an in-depth look at some of the most prominent deep learning frameworks used in Language Model Development (LLM).

6.1.1 TensorFlow

Developed by Google, TensorFlow is an open-source deep learning framework known for its flexibility and extensive community support. It's widely used in both research and industry for various machine learning tasks including LLM. [wikipedia]

- TensorFlow supports high-level APIs, making it easy to develop machine learning models.
- As the input data set is massive, any mathematical calculations or estimations can be done easily.
- TensorFlow is packed with Machine Learning APIs which are a combination of low-level and high-level.
- TensorFlow offers model training and development of models on CPU and GPU.
- Google has incorporated several data sets and pre-trained models in TensorFlow, including mnist, ImageNet, coco, etc
- he Machine Learning models may be run on mobile and embedded devices with TensorFlow. Besides, Pre-trained models can also be used for production directly.
- TensorFlow is an open-source platform, free to use and allows developers and researchers to build and deploy Machine Learning models [tensorflow] [geeksforgeeks]

6.1.2 PyTorch

Created by Facebook's AI Research lab, PyTorch is a library which is popular among researchers for its dynamic computation graph and ease of use. It offers a more robust approach, making it accessible for those familiar with Python programming.

- Pytorch allows flexibility in building and modifying neural networks on the fly
- Pytorch comes with TorchScript which Enables seamless transition from eager mode to graph mode and facilitates optimization and deployment
- Pytorch provides native Support for Parallelism and simplifies training models on multi-GPU setup.

PyTorch's dynamic nature makes it suitable for experimentation and prototyping in LLM. Researchers often prefer PyTorch for exploring new algorithms and architectures for language modeling [pyt, 2016]

Keras

Keras is a high-level neural network API which runs on top of other deep learning frameworks like TensorFlow. It's known for its simplicity and ease of use, making it an excellent choice for beginners and experts alike.

- Keras provides a straightforward and intuitive interface for building and training neural networks
- Keras allows an easy assembling of standard components without deep knowledge of underlying mathematical principles
- Keras can run on top of TensorFlow, CNTK, or Theano and providing flexibility in implentation
- Keras offers a collection of pre-trained models; including popular language models for quick implementation

Keras is often used for rapid prototyping and development of language models. Its user-friendly nature allows developers to quickly build and experiment with different neural network architectures for LLM [Chollet et al., 2015]

Deepseed

DeepSpeed represents a comprehensive deep learning optimization library designed to simplify and enhance distributed training and inference processes. This software suite offers user-friendly tools to enable remarkable scalability and speed for both training and inference tasks. With DeepSpeed it is possible to:

- Train/Inference dense or sparse models with vast parameters counts
- ensuring exceptional system throughput scalable to numerous GPUs.
- Train/Inference on resource constrained GPU systems
- Achieve unprecedented low latency and high throughput for inference.
- Achieve extreme compression for an unparalleled inference latency and model size reduction with low costs.

It is recommended to try DeepSpeed on Azure as it is the simplest and easiest method. The recommended method to try DeepSpeed on Azure is through AzureML recipes.[Microsoft, 2023]

6.2 Pre-trained Models for Code Related tasks

6.2.1 Starcoder

StarCoder represents an expansive language model tailored for code, developed to excel in comprehending and generating programming code across nearly 80 programming languages, including Python. It surpasses other available code language models and fine-tuned variants, particularly in the domain of Python and various programming languages.

One of its iterations, StarCoderBase, boasts an impressive parameter count of 15.5 billion and a context length of 8,000, facilitating rapid large-batch inference. It has undergone training on an extensive dataset of 1 trillion tokens sourced from The Stack, which aggregates permissively licensed GitHub repositories.

A distinctive strength of StarCoderBase is its superior ability to produce valid code outputs while maintaining a notably low occurrence of insecure completions, particularly in cases where over 95% of the generated code is valid. It showcases proficiency in tasks like converting natural language descriptions into code, documenting code, and predicting type annotations.

Furthermore, StarCoder has been trained on natural language text, rendering it versatile for a range of natural language tasks, including reasoning.

It's essential to acknowledge that while StarCoder makes strides in data privacy, it might still generate personally identifiable information (PII). Measures have been implemented to identify and remove PII, yet tailored validation and refinement remain necessary for specific applications.

The StarCoder models are accessible to the public under a version of the Open Responsible AI Model license that encourages practical utilization and stimulates ongoing research and advancement in the domain.[Li et al., 2023]

6.2.2 CodeT5

CodeT5 stands as an advanced pre-trained encoder-decoder model tailored explicitly for tasks involving code comprehension and creation. Unlike its predecessors that treated code snippets much like natural language text, CodeT5 capitalizes on the distinctive attributes of programming languages, meticulously considering token types within code. It embraces a unified architecture accommodating both code understanding and generation tasks, thereby enabling efficient multi-task learning. This is facilitated through an innovative identifier-aware pre-training task, which equips the model to discern code tokens functioning as identifiers, enhancing its ability to capture semantic nuances in code.

Fine-tuning CodeT5 for specific tasks is facilitated through task-specific transfer learning or multi-task learning strategies. When applied to code generation tasks, CodeT5 can be adapted using its Seq2Seq framework, while for code understanding tasks, it explores methodologies such as generating labels as unigram target sequences or predicting them based on class label vocabularies. This versatile model has undergone fine-tuning across a spec-

trum of code-related tasks, showcasing its prowess in defect detection, clone identification, summarization, translation, and refinement.

Notably, CodeT5's efficacy has demonstrated significant advancements in both understanding and generation tasks spanning diverse directions like programming language-to-natural language, natural language-to-programming language, and even within programming languages themselves.[Wang et al., 2021]

Chapter 7

Practicalities For Fine tuning Open Source Pretrained Models

Fine-tuning open-source pre-trained models is a powerful practice in machine learning. The Process allows to leverage the knowledge captured in a model trained on a large data set and adapt it to a specific task or data set in any specific domain Pre -trained models, often trained on massive corpora, have demonstrated remarkable proficiency in capturing patterns, relationships, and representations from raw data. Fine-tuning takes advantage of this rich knowledge which allows practitioners to tailor models for their desired tasks. This chapter will introduce about practicalities to consider when fine-tuning a pre-trained open source models.

7.0.1 Step by Step Guide in Fine tuning

Here are some practical steps to consider when fine-tuning open source pre-trained models:

Selecting a Pre-trained Model

Pre-trained models are neural networks that have undergone extensive training on a large volume of data, typically for a general NLP tasks. They are able to translate complicated linguistic elements and patterns to other relevant tasks. Compared to developing a model from scratch, using pre-trained models can help to get better results faster and with fewer data.[Linkedin]. When selecting a pre-trained model for any down stream task, there are several factors to consider, such as the task and data.For instance for a text classification task, a model pre-trained on a large text corpus like BERT or GPT-2 might be a good starting point. Besides, for any code related tasks, choosing a model which was not trained to understand programming languages semantics may not be a good option.

Preparing Data-set

Before starting fine-tuning a pre-trained model, it's crucial to ensure that data-set is properly prepared. This involves several steps to clean and structure the data so that it can effectively be used for training the model. The general steps include handling missing values, noise removal, removing duplicates etc. Before using data in a model, the data needs to be processed into an acceptable format for the desired model. A model does not understand raw text, images or audio.Therefore, the inputs need to be changed into numbers and assembled into tensors. The primary tool for processing textual data is a tokenizer.A tokenizer works by splitting text into tokens according to a set of rules. The tokens are converted into numbers and used to build tensors as input. When using a pre-trained model, it's important to use the associated pret-rained tokenizer which ensures text is split the same way as the pretraining corpus, and uses the same corresponding tokens-to-index (known as **vocab**) during pre-training.A pretrained tokenizer can be loaded with `AutoTokenizer.from_pretrained("bert-base-cased")` as below :

```python
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
```

When processing a batch of sentences, they do not always have the same length. Tensors, the model's input, need to have a consistent shape, hence this is a difficulty. By including a unique padding token, padding serves as a way for guaranteeing that tensors are rectangular.
Furthermore, sometimes a sequence can be way too long for a model to handle. In that case, truncation is necessary for the sequence to have a shorter length.[?]

Hyperparameter Tuning

Hyper parameters are parameters that are set before the training process begins and control various aspects of the training process itself. They are not learned during training but rather set by the user.[Sathishkumar et al., 2023] Examples of hyper-parameters include learning rate, batch size, number of layers, number of hidden units, regularization strength, etc. Hyper parameter tuning involves selecting the best values for these parameters to achieve the best performance on the validation set.
Selecting the right set of hyperparameters is crucial for model's performance and accuracy. Unfortunately, there are no set rules on which hyperparameters work best nor their optimal or default values. We need to experiment to find the optimum hyperparameter set. This activity is known as hyperparameter tuning or hyperparameter optimization.
Hyperparameters can effect model structure, function, and performance. Hyperparameter tuning allows to tweak model performance for optimal results. This process is an essential part of machine learning, and choosing appropriate hyperparameter values is crucial for success.
For example,For the learning rate of the model as a hyperparameter, if the value is too high, the model may converge too quickly with suboptimal re-

sults. Again, if the rate is too low, training takes too long and results may not converge. An appropriate and balanced choice of hyperparameters results in accurate models and excellent model performance [Amazon Web Services]

Training

A pretrained model can be fine tuned with a deep learning framework of the developer's choice.The provided content is a guide through training transformer-based models using PyTorch and TensorFlow:

PyTorch

To commence the training process, an appropriate transformer model is loaded using the AutoModelForSequenceClassification module. This module enables the seamless integration of pre-trained transformer architectures tailored for sequence classification tasks.

A TrainingArguments object is constructed to observe the training process. This object encompasses a range of hyperparameters and training options that significantly impact the model's convergence and performance. By judiciously configuring these settings, developers can tailor the training process to the specific demands of task.

The essential elements of the training process are included in a Trainer object that is initialized. This object comprises the initialized model, the previously defined TrainingArguments, the training and evaluation datasets, and the metrics calculation function.

The actual model training is initiated using the **train()** method of the Trainer object. This step iteratively updates the model's parameters using the training data, iteratively refining its ability to make accurate predictions. By incorporating the hyperparameters, metrics, and training data, the model's adaptation process is managed in a controlled and optimized

manner.

TensorFlow

The initial phase of model training entails loading a data set.The **AutoTokenizer** function facilitates the generation of tokenizers tailored to specific transformer architectures. Tokenization is a pivotal preprocessing step that converts raw text data into manageable units, a prerequisite for model understanding.The utilization of NumPy arrays ensures efficient handling of tokenized data and labels.

Within the TensorFlow system, the **TFAutoModelForSequenceClassification** function seamlessly loads a pre-trained transformer model. Model compilation involves specifying optimization strategies, including parameters such as learning rates, enabling the network to adapt to the nuances of the provided data.

With tokenized data and labels prepared and the model compiled, the training phase begins. This pivotal stage is facilitated through the **model.fit()** function in TensorFlow. This process checks for the gradual achievement of domain-specific knowledge which results in enhancing the model's predictive abilities.

As the scale of datasets grows, considerations regarding memory efficiency and computational performance become difficult. In such scenarios, the adoption of **tf.data.Dataset** within TensorFlow offers a good solution. [?]

Native Pytorch
In some scenarios, researchers and practitioners may prefer a more tailored training loop for fine-tuning transformers models.Before starting on fine-tuning, it's advisable to optimize memory usage. This can be achieved by liberating resources, such as by removing previously loaded models, using

the following code snippet:

```
del model
del pytorch_model
del trainer
torch.cuda.empty_cache()
```

This process also comprises in tokenization, data loader, model initialization , optimization strategy etc. In this case, the AdamW optimizer, is utilized for fine-tuning. The device (GPU or CPU) is specified for training, ensuring optimal hardware usage. Tracking training progress is essential and the **tqdm** library provides a helpful progress bar.

Evaluation

An evaluation metric is defined using the evaluate library. For instance, accuracy can be chosen as a suitable metric to gauge the model's classification performance. This chosen metric serves as an indicator of the model's ability to correctly classify input sequences, contributing to an objective assessment of the training outcomes.

The compute metrics function is implemented to derive meaningful meaning from the predictions made during evaluation. This function processes the model's predictions, compares them with the ground truth labels, and calculates the specified evaluation metric. The accuracy, calculated using this function, offers insights into the model's effectiveness in producing correct classifications.

7.0.2 Tips for Fine tuning

Using a Smaller learning rate

When fine-tuning a pre-trained model, it is essential to use a smaller learning rate than the one used during pre-training. This is because the pre-trained model has already learned useful features, and we do not want to overwrite them with random initialization.

Data augmentation

Data augmentation is a technique used to increase the size of the training dataset by applying transformations such as rotation, scaling, and flipping. Data augmentation can help the model generalize better by exposing it to more variations of the input data

Freeze the Early Layers

The early layers of a pre-trained model are responsible for learning low-level features such as edges and corners. These features are useful for many tasks, and we do not want to overwrite them during fine-tuning. Therefore, it is recommended to freeze the early layers of the pre-trained model and only fine-tune the later layers

Regularization

Regularization is a technique used to prevent overfitting. Overfitting occurs when the model performs well on the training set but poorly on the test set. Regularization techniques such as L1 and L2 regularization can be used to prevent overfitting during fine-tuning.

7.0.3 Challenges and Issuses

Dataset Acquisiton

One of the primary challenges researchers and practitioners can face when fine-tuning pretrained models is acquiring or creating datasets that are tailored to their specific task. While pretrained models are versatile, they still require domain-specific and task-specific data for optimal performance. Acquiring such datasets can be a complex task, particularly for niche or emerging fields where labeled data might be scarce. Moreover, curating datasets that are representative and balanced poses its own set of challenges.

Documentation and Guidelines

The availability and quality of documentation play a pivotal role in the success of any implementation. While libraries like Hugging Face's Transformers have greatly simplified the process of fine-tuning, the documentation and guideline for fine-tuning pre-trained models can sometimes be inadequate or ambiguous. Clear and comprehensive documentation is essential for guiding users through the intricate process of selecting the right model, configuring hyper parameters, pre-processing data, and interpreting results. Insufficient documentation can lead to confusion, misinterpretations, and ultimately sub optimal outcomes.

Lack of Implementation Examples

Learning from examples is an effective way of skill development, and the realm of fine-tuning pre-trained models is no different. Insufficient illustrative examples that cover diverse tasks can hinder users' ability to effectively fine-tune models. Researchers often struggle with translating theoretical knowledge into practical implementations, and a dearth of real-world examples enhances this challenge.

The NLP community may collaborate to create common datasets that are tailored to particular objectives. Open-source platforms that encourage dataset contributions and crowdsourcing can help alleviate the scarcity of task-specific data.

Chapter 8

Optimizing LLMs for Software Company Use

In the dynamic landscape of software development, LLMs have emerged as transformative tools with much potential. However, as the scale and complexity of LLMs grow, the challenge lies in effectively deploying them within resource-limited environments. This chapter delves into innovative strategies to tailor LLMs for the software industry. We explore methods to ensure fairness, enhance interpretability, easier collaboration, and boost performance

8.1 Model Compression and Pruning for Efficient Deployment

Model compression techniques offer a promising avenue for addressing the challenges posed by the size and complexity of large language models (LLMs) when deploying them in resource-constrained environment[Magister et al., 2023]. By implementing model compression and pruning, software companies can reduce the size, memory usage, and computational costs of LLMs, enabling more efficient deployment in resource-constrained environments.

One such technique, known as pruning, presents an effective strategy to streamline LLMs. Pruning involves the removal of unnecessary or redundant components, like neurons, channels, or entire layers, while maintaining the

model's overall architecture. An advanced variant, structured pruning, takes a rule-based approach to eliminate entire structural components of the model, all the while preserving the global network structure.[Zhu et al., 2023].

A noteworthy example of structured pruning is the LLM-Pruner, which stands as a testament to the innovation in this field. This technique amalgamates a dependency detection algorithm with an efficient importance estimation method, resulting in a finely tuned pruning process.[Ma et al., 2023].

By integrating model compression and pruning techniques, software companies stand to achieve significant reductions in LLM size, memory utilization, and computational expenses. This, in turn, paves the way for the deployment of LLMs in resource-constrained environments with enhanced efficiency, marking a substantial advancement in the realm of software engineering.

8.2 Quantization and Distillation for Resource-Constrained Environments

Quantization is a model compression technique that converts floating-point numbers to integers or other discrete forms, reducing storage requirements and computational complexity .Quantization can be applied during the training process (quantization-aware training), fine-tuning of a pretrained model (quantization-aware fine-tuning), or after the model has completed training (post-training quantization)[Gholami et al., 2021]. Recent research has explored quantization methods for large language models (LLMs), achieving substantial model compression with minimal accuracy degradation.

Knowledge distillation is another technique used for model compression, where a smaller student model is trained to mimic the behavior of a larger teacher model .Layer-by-layer knowledge distillation, optimized quantization support, and hardware-friendly quantization schemes have been proposed to reduce weight and activation precision in LLMs [Zhu et al., 2023].

Techniques, such as GPTQ [Frantar et al., 2023] and ZeroQuant [Yao et al., 2022], aim to maintain accuracy while reducing the bit precision of

LLMs, making them suitable for resource-constrained environments .

8.2.1 QLORA and LORA:

QLORA is a quantization-aware fine-tuning technique for large language models (LLMs) that aims to conserve memory without compromising performance. It introduces innovative concepts like a new data type, double quantization, and paged optimizers to achieve state-of-the-art results on the Vicuna benchmark[Dettmers et al., 2023][Zhu et al., 2023].

LORA, on the other hand, is a low-rank factorization technique for LLMs that aims to approximate a weight matrix by decomposing it into smaller matrices with significantly lower dimensions. LORA and its variants have been widely adopted in the field of LLM research to fine-tune models efficiently [Hu et al., 2021].

8.3 Monitoring and Maintaining LLMs in Software Development Lifecycle

LLMs have versatile applications in software maintenance, including bug prediction, program repair, code review, debugging, and logging.

LLMs like BERT, CodeBERT, CodeT5, Codex, PLBART, T5 have shown remarkable capabilities in understanding programming languages and generating syntactically correct and contextually relevant code.

Integrating Large Language Models (LLMs) with emerging input modalities, such as spoken language, diagrams, and multimodal inputs, presents an opportunity to significantly broaden their capabilities in comprehending and handling a wide array of user needs. By incorporating these new input forms, LLMs can transcend textual limitations, enabling more effective communication and interaction across various domains and contexts [Hou et al., 2023].

Expanding the use of LLMs to under-explored areas like software requirements, design, and management can revolutionize how projects are managed.

Efficient deployment of Large Language Models (LLMs) necessitates evaluating their inference efficiency through accuracy, zero-shot ability, and inference scaling laws.

However, ensuring their real-world applicability demands ongoing monitoring and maintenance throughout the software development lifecycle. This entails adapting to evolving data distributions, detecting and correcting errors, mitigating biases, optimizing performance, and staying updated with LLM versions and updates. By integrating these measures, LLMs can consistently provide accurate, unbiased, and efficient responses in dynamic real-world environments. [Zhu et al., 2023]

Chapter 9

Future Trends and Research Directions

The field of large language models has experienced impressive development and innovation, opening the way for exciting next trends and research directions that are ready to reshape the software industry's landscape. LLMs have the potential to transform a number of facets of software engineering as they continue to develop. This section examines the significant effects of these developments, emphasizing the areas that need the most improvement, the difficulties that still lie ahead, and the moral principles that must underpin their incorporation into software engineering techniques.

9.0.1 Challenges in LLM Development for Software Companies

- To learn language patterns, large language models need a vast quantity of training data, and the results heavily depend on the training data. LLMs will increase any problems and biases or mistakes present in the training data, potentially resulting in models that behave with prejudice, such as providing biased recommendations. This means that errors might spread and that the performance and generalizability of the model can be greatly affected by the quality and representativeness of the training data. [Ozkaya, 2023]. For instance, language mod-

els that are applied to recommend code patterns have been found to carry security flaws forward which creates risks in not only generating buggy code, but also perpetuating immature implementation practices in software developers.[Perry et al., 2022]

- The explainability of deep learning and machine learning models is a significant topic of concern within the field of artificial intelligence. This concern is particularly relevant when dealing with complex models like Large Language Models (LLMs). Explainability refers to the ability to understand and interpret how these models make their decisions and predictions.In various AI applications, especially those involving business decisions, it's crucial to be able to explain why a model is suggesting a certain recommendation or making a particular decision. [Tantithamthavorn et al., 2023]
  Understanding an LLM's decision-making process can help developers diagnose issues and improve model performance.Explanations can reveal whether the model is making mistakes due to poor training data or other issues.

- Large Language Models (LLMs) are built using content created by various individuals, which might include private information and the unique creative styles of content creators. Training these models using patterns in the generated output raises concerns about plagiarism. While some content is repetitive and generating it accurately can enhance efficiency, differentiating content, including code, where individual contributions are important becomes challenging. [Ozkaya, 2023]

Another issue that needs to be addressed is the gap in collaboration between Software Engineering people and Machine learning scientist.

9.1 Future Directions of LLM and Software Engineering

- Future research can focus on creating Large Language Model (LLM) architectures that are specifically designed to address software engineering tasks. These tailored architectures can take into account the unique characteristics of coding languages, software design patterns, and development practices. By fine-tuning LLMs for software engineering, researchers can enhance their ability to generate accurate and contextually relevant code, documentation, and recommendations. [Hou et al., 2023]

- Incorporating domain-specific knowledge into LLMs can greatly enhance their performance in software engineering tasks. Future work can explore techniques to effectively integrate software engineering principles, patterns, and best practices into LLMs' training data and fine-tuning processes. This integration can ensure that LLM-generated content aligns with established coding standards and practices within the software development community. [Hou et al., 2023]

- LLMs are often seen as black-box models due to their complex internal workings. Enhancing the interpretability and explainability of LLMs is crucial for building trust and adoption in software engineering tasks.[Tantithamthavorn et al., 2023] Researchers can explore techniques to generate human-readable explanations for LLM-generated outputs, such as code explanations or reasons behind suggested code changes. This empowers developers to understand and trust the model's recommendations.

- The collaboration between LLMs and human developers holds immense potential for improving software engineering processes. Future research can focus on developing interactive interfaces and tools that enable seamless integration between LLMs and developers. These tools can allow developers to interact with LLM-generated suggestions, re-

fine them, and incorporate their domain expertise, resulting in higher-quality code and documentation

9.2 Truthfulness of LLMs

The assurance of aligning large language models (LLMs) with human intentions is paramount prior to their deployment in real-world contexts. The seven principal dimensions of LLM trustworthiness can be named as: reliability, safety, fairness, resilience against misuse, explainability and logical inference, adherence to societal norms, and robustness. Models exhibiting higher alignment with human intentions often demonstrate superior overall trustworthiness, though the effectiveness of alignment can diverge among various trustworthiness facets. It is worth noting that LLMs possess the capacity to fabricate and generate content that lacks connections to existing knowledge. Consequently, it becomes imperative for LLMs to uphold neutrality when addressing matters pertaining to political ideologies, public figures, events, or products. The credibility of an LLM hinges on its capability to elucidate its decision-making rationale and offer transparency into the process by which it generates content. This multifaceted approach underscores the importance of addressing alignment, accountability, and transparency in order to foster trust in the capabilities and outputs of LLMs.[Liu et al., 2023]

9.3 Ethical AI and Responsible LLM Development in Software Industry

Ethical considerations hold significant importance when it comes to the development and deployment of Large Language Models (LLMs) in the software industry. The training data used to teach these models can carry biases, and if not addressed, these biases can manifest in the outputs generated by LLMs[Ozkaya, 2023]. As a response to this, it has become essential to curate training datasets that are transparent, diverse, and well-balanced, ensuring that the resulting LLM outputs are free from undue biases and discrimina-

tion.

Responsible LLM development goes beyond addressing biases. It involves tackling challenges inherent to these models. For instance, the sheer size of LLMs can present deployment challenges, as they require substantial computational resources and memory. Moreover, LLMs often depend heavily on the quality and quantity of training data. Ensuring a consistent and reliable supply of relevant data is crucial for optimal performance.

In response to challenges, ongoing efforts are directed towards refining LLMs. One strategy involves reducing their sizes to improve efficiency and practical usability. Techniques like genetic algorithms are being explored to compress LLMs without compromising their performance.[Hou et al., 2023]

As LLMs find their place in various software engineering tasks, it's vital to consider the resources they demand. Storage, memory, and computational requirements must be carefully managed to ensure smooth integration. Organizations should strike a balance between the benefits of LLM utilization and the resources allocated to accommodate their usage effectively.

Chapter 10

Conclusion

At this point it is time to look back to the objective of the study and compare the objective to our findings. This chapter will summarize the findings from the whole study and recommend some healthy practice ideas to the practitioners and the researchers.

10.1 Summary of Key Findings for Software Companies

As outlined in the previous chapters, the versatile applications of LLMs in software development are evident across automated code review, bug detection, code completion, suggestion, software documentation generation, and customer support chatbots. The integration of LLMs in software industry can offer transformative potential, enhancing code quality, and enabling more efficient customer interactions. Besides these, the development of LLM also asks for knowldege and experience from software domain for their best results and performances.

However,these advantages are accompanied by a series of considerations and challenges. The meticulous selection of LLM architectures and techniques tailored to software engineering tasks is essential to ensuring optimal performance.

As these approaches are data driven approach, the ultimate results of

depends on the availability and quality of the data. We can not but accept the fact of scarcity of task specific labelled data set in the software domain.Moreover the careful management of data preparation, training techniques, evaluation metrics, and deployment strategies emerges as a critical factor in achieving successful LLM integration.Ethical concerns related to biases in training data and fairness metrics underscore the need for responsible LLM development.

As we peer into the future, the significance of model optimization for efficient deployment, exploration of edge device deployment, and the imperative to maintain LLMs throughout the software development lifecycle become increasingly apparent. The synthesis of these findings presents a roadmap for software companies, guiding them toward informed decisions and best practices as they navigate the realm of LLMs, fostering innovation while upholding ethical standards and achieving sustainable success in the evolving software landscape.

10.2 Recommendations for LLM Practitioners in Software Development

Training Language Models (LLMs) on well-curated and diverse datasets is crucial to reduce biases and improve their performance in software engineering tasks. Biases can emerge from biased training data, which can lead to unfair or unrepresentative outcomes. To mitigate this, training data should be carefully selected, and potential biases should be identified and rectified. [Hou et al., 2023]

The deployment of LLMs in software engineering raises important ethical considerations. LLM-generated outputs should be thoroughly reviewed to ensure they are fair, unbiased, and align with ethical guidelines. It's crucial to address issues related to fairness, transparency, and accountability in the deployment of LLMs. [Liu et al., 2023]

Developing LLM architectures tailored specifically for software engineering tasks can significantly enhance their effectiveness and efficiency. By un-

derstanding the unique requirements of software engineering, models can be optimized to generate more accurate and contextually relevant code, documentation, and other software related tasks.

Interpretability and explainability are crucial aspects, especially in critical domains like software engineering. Techniques should be explored to make LLMs more interpretable.

Creating interactive interfaces and tools that enable seamless interaction between LLMs and human developers is essential. Such interfaces can help developers integrate LLMs into their workflow, receive real-time suggestions, and provide feedback to improve the model's performance.

The performance of LLMs should be continuously evaluated and optimized. Factors such as model size, deployment challenges, and data dependencies should be considered. Regular model updates can ensure that the LLM remains effective and aligned with the evolving needs of the software engineering field.

Keeping up-to-date with the latest research and advancements in LLMs is crucial. The field of machine learning, including LLMs, is rapidly evolving. Staying informed about new techniques and approaches can help in harnessing the full potential of LLMs for optimizing software development processes.[Hou et al., 2023]

10.3 Final Thoughts

The emergence of Large Language Models (LLMs) has brought about a fundamental shift in the field of Software Engineering (SE). These advanced models possess remarkable capabilities in handling complex and extensive language-related tasks, promising to reshape SE practices in a substantial way. In this comprehensive study, we delve into the current landscape of how LLMs are being integrated into Software engineering.

Our exploration begins with an examination of the diverse LLMs that have found application in SE activities. We aim to unravel their unique characteristics and versatile uses within the domain.

Moving forward, we take a deep dive into the crucial procedures associated

with data collection, pre-processing, and utilization of LLMs. We emphasize the pivotal role of high-quality datasets that have undergone meticulous curation. .

Furthermore, our review sheds light on specific SE tasks that have experienced significant improvements thanks to the incorporation of LLMs. We highlight the tangible benefits and practical advancements that have been achieved through the integration of these models. From code generation to natural language understanding in SE, LLMs have demonstrated their ability to enhance the efficiency and effectiveness of various processes.

Finally, we look into the several methods used to evaluate and improve LLM performance for SE tasks.Additionally, we highlight the current difficulties and introduce a road map that identifies interesting future directions. For researchers and engineers investigating the application of LLMs in software engineering, this thorough overview offers crucial insights.

Bibliography

Pytorch: An open source machine learning framework. https://pytorch.org/, 2016.

Microsoft azure machine learning. https://azure.microsoft.com/en-us/products/machine-learning#x11b6ca2f2d3f43b098fc9b9aecdf5240, 2023.

Microsoft azure machine learning. https://cloud.google.com/ai-platform/docs/technical-overview, A.

A. Abbas. https://www.techopedia.com/5-ways-llms-can-empower-software-engineering.

T. Afonja. Model evaluation i: Precision and recall. 2017. URL https://towardsdatascience.com/model-evaluation-i-precision-and-recall-166ddb257c7b.

AIMultiple. Large language model training: Costs, times, resources, 2023. URL https://research.aimultiple.com/large-language-model-training/.

M. Alqarni and A. Azim. Low Level Source Code Vulnerability Detection Using Advanced BERT Language Model. *Proceedings of the Canadian Conference on Artificial Intelligence*, may 27 2022. https://caiac.pubpub.org/pub/gdhb8oq4.

I. Amazon Web Services. What is hyperparameter tuning? URL https://aws.amazon.com/what-is/hyperparameter-tuning/?nc1=h_ls.

S. Arakelyan, R. J. Das, Y. Mao, and X. Ren. Exploring distributional shifts in large language models for code analysis, 2023.

builtin. https://builtin.com/data-science/transfer-learning.

M. Chalokia. https://www.linkedin.com/pulse/time-based-splitting-determining-train-test-data-come-manraj-chalokia/.

X. Chen, M. Lin, N. Schärli, and D. Zhou. Teaching large language models to self-debug, 2023.

F. Chollet et al. Keras. `https://keras.io`, 2015.

M. Ciniselli, N. Cooper, L. Pascarella, A. Mastropaolo, E. Aghajani, D. Poshyvanyk, M. Di Penta, and G. Bavota. An empirical study on the usage of transformer models for code completion, 08 2021.

J. L. S. W. E. W. F. S. R. Z. W. Y. L. Z. Daniel Fried, Armen Aghajanyan and M. Lewis. A systematic evaluation of large language models of code. *Incoder: A generative model for code infilling and synthesis. CoRR,*, 2022.

T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer. Qlora: Efficient finetuning of quantized llms, 2023.

Z. Du, Y. Qian, X. Liu, M. Ding, J. Qiu, Z. Yang, and J. Tang. Glm: General language model pretraining with autoregressive blank infilling, 2022.

D. C. Eliane Maria, De Bortoli Fávero. Bert$_se : Apre - trainedlanguagerepresentationmodelforsoftwareengineering.$18/9/2020.

G. N. Frank F. Xu, Uri Alon and V. J. Hellendoorn. A systematic evaluation of large language models of code. *6th ACM SIGPLAN International Symposium on Machine Programming, San Diego, CA, USA*, pages 38–45, Online, 2022.

E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers, 2023.

J. Ge, S. Tang, J. Fan, and C. Jin. On the provable advantage of unsupervised pretraining, 2023.

geeksforgeeks. https://www.geeksforgeeks.org/why-tensorflow-is-so-popular-tensorflow-features/.

A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer. A survey of quantization methods for efficient neural network inference, 2021.

L. Gong, J. Zhang, M. Wei, H. Zhang, and Z. Huang. What is the intended usage context of this model? an exploratory study of pre-trained models on various model repositories. *ACM Trans. Softw. Eng. Methodol.*, 32(3), may 2023. ISSN 1049-331X. doi: 10.1145/3569934. URL https://doi.org/10.1145/3569934.

P. W. O. Greg Brockman, Mira Murati. Openai api. URL https://openai.com/blog/openai-api.

M. U. Hadi, Q. Al-Tashi, R. Qureshi, A. Muneer, M. Irfan, A. Zafar, M. Shaikh, N. Akhtar, J. Wu, and S. Mirjalili. Large language models: A comprehensive survey of its applications, challenges, limitations, and future prospects, 2023.

D. Hendrycks, K. Lee, and M. Mazeika. Using pre-training can improve model robustness and uncertainty. In K. Chaudhuri and R. Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 2712–2721. PMLR, 09–15 Jun 2019. URL https://proceedings.mlr.press/v97/hendrycks19a.html.

X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang. Large language models for software engineering: A systematic literature review, 2023.

E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen. Lora: Low-rank adaptation of large language models, 2021.

HuggingFace. https://huggingface.co/models. a.

HuggingFace. Summary of the models, b. URL https://huggingface.co/transformers/v3.1.0/model_summary.html#autoencoding-models.

ibm. https://www.ibm.com/topics/supervised-learning.

X. Jiang, Y. Dong, L. Wang, Z. Fang, Q. Shang, G. Li, Z. Jin, and W. Jiao. Self-planning code generation with large language models, 2023.

B. Kou, M. Chen, and T. Zhang. Automated summarization of stack overflow posts, 2023.

D. Li, Y. Shen, R. Jin, Y. Mao, K. Wang, and W. Chen. Generation-augmented query expansion for code retrieval, 2022.

R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, Q. Liu, E. Zheltonozhskii, T. Y. Zhuo, T. Wang, O. Dehaene, M. Davaadorj, J. Lamy-Poirier, J. Monteiro, O. Shliazhko, N. Gontier, N. Meade, A. Zebaze, M.-H. Yee, L. K. Umapathi, J. Zhu, B. Lipkin, M. Oblokulov, Z. Wang, R. Murthy, J. Stillerman, S. S. Patel, D. Abulkhanov, M. Zocca, M. Dey, Z. Zhang, N. Fahmy, U. Bhattacharyya, W. Yu, S. Singh, S. Luccioni, P. Villegas, M. Kunakov, F. Zhdanov, M. Romero, T. Lee, N. Timor, J. Ding, C. Schlesinger, H. Schoelkopf, J. Ebert, T. Dao, M. Mishra, A. Gu, J. Robinson, C. J. Anderson, B. Dolan-Gavitt, D. Contractor, S. Reddy, D. Fried, D. Bahdanau, Y. Jernite, C. M. Ferrandis, S. Hughes, T. Wolf, A. Guha, L. von Werra, and H. de Vries. Starcoder: may the source be with you!, 2023.

Linkedin. https://www.linkedin.com/advice/0/how-do-you-use-fine-tune-pre-trained.

Y. Liu, Y. Yao, J.-F. Ton, X. Zhang, R. Guo, H. Cheng, Y. Klochkov, M. F. Taufiq, and H. Li. Trustworthy llms: a survey and guideline for evaluating large language models' alignment, 2023.

ludwig. https://ludwig.ai/latest/configuration/preprocessing/.

X. Ma, G. Fang, and X. Wang. Llm-pruner: On the structural pruning of large language models, 2023.

L. C. Magister, J. Mallinson, J. Adamek, E. Malmi, and A. Severyn. Teaching small language models to reason, 2023.

A. Mastropaolo, N. Cooper, D. Nader, S. Scalabrino, D. Poshyvanyk, R. Oliveto, and G. Bavota. Using transfer learning for code-related tasks. *IEEE Transactions on Software Engineering*, PP:1–20, 01 2022. doi: 10.1109/TSE.2022.3183297.

medium. https://medium.com/data-science-365/random-vs-stratified-splits-5d3d528d445b.

Microsoft. Microsoft DeepSpeed. https://github.com/microsoft/deepspeed, 2023.

nltk. https://www.nltk.org/.

nvidia. https://blogs.nvidia.com/blog/2022/03/25/what-is-a-transformer-model/.

OpenAI. https://openai.com/blog/customizing-gpt-3.

I. Ozkaya. Application of large language models to software engineering tasks: Opportunities, risks, and implications. *IEEE Software*, 40(3):4–8, 2023. doi: 10.1109/MS.2023.3248401.

I. Ozkaya. Application of large language models to software engineering tasks: Opportunities, risks, and implications. *IEEE Software*, 40, no. 3,:4–8, May-June 2023.

N. Perry, M. Srivastava, D. Kumar, and D. Boneh. Do users write more insecure code with ai assistants?, 2022.

G. Recchia. Teaching autoregressive language models complex tasks by demonstration, 2021.

K. I. Roumeliotis and N. D. Tselikas. Chatgpt and open-ai models: A preliminary review. *Future Internet*, 15(6):192, May 2023. ISSN 1999-5903. doi: 10.3390/fi15060192. URL http://dx.doi.org/10.3390/fi15060192.

Ruth Brooks. The role of natural language processing in AI. https://online.york.ac.uk/the-role-of-natural-language-processing-in-ai/#:~:

text=Natural%20language%20processing%20(NLP)%20is,a%20lot%20of%20unstructured%20data.

Safjan. https://safjan.com/measure-quality-of-embeddings-intrinsic-vs-extrinsic/.

G. H. T. E. Sam Manning, Pamela Mishkin and E. Eisner4. A research agenda for assessing the economic impacts of code generation models. 3/3/2022.

V. Sathishkumar, A. Ramu, and J. Cho. Machine learning algorithms to predict the catalytic reduction performance of eco-toxic nitrophenols and azo dyes contaminants (invited article). *Alexandria Engineering Journal*, 72: 673–693, 2023. ISSN 1110-0168. doi: https://doi.org/10.1016/j.aej.2023.04.007. URL https://www.sciencedirect.com/science/article/pii/S1110016823002806.

M. Schäfer, S. Nadi, A. Eghbali, and F. Tip. Adaptive test generation using a large language model, 2023.

P. W. C. L. Sid Black, Leo Gao and S. Biderman. Gpt-neo: Large scale autoregressive language modeling with mesh-tensorflow. *https://github.com/kingoflolz/mesh-transformer-jax.*, 2021.

simform. https://www.simform.com/blog/completeguide-finetuning-llm/.

C. Tantithamthavorn, J. Cito, H. Hemmati, and S. Chandra. Explainable ai for se: Challenges and future directions. *IEEE Software*, 40(3):29–33, 2023. doi: 10.1109/MS.2023.3246686.

tensorflow. https://www.tensorflow.org/about.

J. Thanaki. ´python natural language processing : Leverage the power of machine learning and deep learning to extract information from text data. 30, 2017.

V. S. J. C. C. D. A. M. P. C. T. R. R. L. M. F. J. D. S. S. P. v. P. C. M. Y. J. J. P. C. X. T. L. S. S. G. M. D. Q. L. Thomas Wolf, Lysandre Debut and

A. M. Rush. Transformers: State-of-the-art natural language processing. *. In Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, 2020.

A. Tormos, D. Garcia-Gasulla, V. Gimenez-Abalos, and S. Alvarez-Napagao. When how to transfer with transfer learning, 2022.

O. L. E. Y. URI ALON, MEITAL ZILBERSTEIN. Code2vec: Learning distributed representations of code. 30/10/2018.

S. N. P. N. U. J. J. L. G. A. N. K. L. . P. I. Vaswani, A. Attention is all you need. neural information processing systems, 30, 5998–6008. 30, 2017.

Y. Wan, W. Zhao, H. Zhang, Y. Sui, G. Xu, and H. Jin. What do they capture? – a structural analysis of pre-trained language models for source code, 2022.

B. Wang and A. Komatsuzaki. Gpt-j6b: A 6 billion parameter autoregressive language model. *https://github.com/kingoflolz/mesh-transformer-jax.*, 2021.

Y. Wang, W. Wang, S. Joty, and S. C. H. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation, 2021.

wikipedia. https://en.wikipedia.org/wiki/tensorflow.

Wikipedia. Word embedding, 2023. URL `https://en.wikipedia.org/wiki/Word_embedding`.

B. Wodecki. 7 language models you need to know. pages 19–36, July 2, 2022.

F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn. A systematic evaluation of large language models of code, 2022.

Z. Yao, R. Y. Aminabadi, M. Zhang, X. Wu, C. Li, and Y. He. Zeroquant: Efficient and affordable post-training quantization for large-scale transformers, 2022.

S. Yashaswini and S. S. Shylaja. Metrics for automatic evaluation of text from nlp models for text to scene generation. *EJECE, European Journal of Electrical Engineering and Computer Science*, PP.

A. D. G. N. B. J. L. S. H. Yue Wang, Henry Hung Le. https://blog.salesforceairesearch.com/codet5-open-code-large-language-models/.

D. T. N. D. X. F. M. G. L. S. B. Q. T. L. D. J. M. Z. Zhangyin Feng1, Daya Guo. Codebert: A pre-trained model for programming and natural languages. 18/9/2020.

X. Zhu, J. Li, Y. Liu, C. Ma, and W. Wang. A survey on model compression for large language models, 2023.