PUCRS

ESCOLA POLITÉCNICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
MESTRADO EM CIÊNCIA DA COMPUTAÇÃO

OTÁVIO PARRAGA

ONE AGAINST MANY: EXPLORING MULTI-TASK LEARNING GENERALIZATION IN SOURCE-CODE TASKS

Porto Alegre
2023

PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
SCHOOL OF TECHNOLOGY
COMPUTER SCIENCE GRADUATE PROGRAM

ONE AGAINST MANY: EXPLORING MULTI-TASK LEARNING GENERALIZATION IN SOURCE-CODE TASKS

OTÁVIO PARRAGA

Master Thesis submitted to the Pontifical Catholic University of Rio Grande do Sul in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Prof. Dr. Rodrigo Coelho Barros

Porto Alegre
2023

Ficha Catalográfica

P259o Parraga, Otávio

One Against Many : Exploring Multi-Task Learning Generalization in Source Code Tasks / Otávio Parraga. – 2023.
90.
Dissertação (Mestrado) – Programa de Pós-Graduação em Ciência da Computação, PUCRS.

Orientador: Prof. Dr. Rodrigo Coelho Barros.

1. Aprendizado Multi-Tarefa. 2. Código-Fonte. 3. Transformers. I. Barros, Rodrigo Coelho. II. Título.

Elaborada pelo Sistema de Geração Automática de Ficha Catalográfica da PUCRS
com os dados fornecidos pelo(a) autor(a).
Bibliotecária responsável: Clarissa Jesinska Selbach CRB-10/2051

OTÁVIO PARRAGA

ONE AGAINST MANY: EXPLORING MULTI-TASK LEARNING GENERALIZATION IN SOURCE-CODE TASKS

This Master Thesis has been submitted in partial fulfillment of the requirements for the degree of Master in Computer Science, of the Computer Science Graduate Program, School of Technology of the Pontifical Catholic University of Rio Grande do Sul

Sanctioned on 29th March, 2023.

COMMITTEE MEMBERS:

Prof^a. Dr^a. Viviane Pereira Moreira (PPGC/UFRGS)

Prof. Dr. Rafael Heitor Bordini (PPGCC/PUCRS)

Prof. Dr. Rodrigo Coelho Barros (PPGCC/PUCRS - Advisor)

I dedicate this work to my parents and sister, my beloved family.

“There are more things in heaven and earth, Horatio, than are dreamt of in your philosophy.”
(William Shakespeare)

ACKNOWLEDGMENTS

First and foremost, I extend my most profound appreciation to my beloved parents, Sílvio Parraga and Nádia Rejane Parraga. Your unwavering love, support, and encouragement have been the driving force behind my achievements. You have always been there for me, providing guidance and inspiration. Your sacrifices and belief in me have fueled my determination, and I am forever grateful for your presence in my life.

To my dear sister, Laura Parraga, thank you for your continuous support and understanding. Your belief in my abilities has been a constant source of motivation. Your encouragement and words of wisdom have uplifted me during challenging times. I am grateful for our bond and your love throughout my entire life.

I express my appreciation to my advisor, Rodrigo Coelho Barros. Your guidance and expertise have been invaluable throughout my research. Your mentorship has made a difference in my journey, and I am grateful for the opportunity to work under your guidance.

A special thank you goes to my girlfriend, Thaís Fernandes. Your support, understanding, and love have been a constant source of strength. You have been my pillar of support, offering encouragement and lending an empathetic ear during challenging times.

I want to extend my gratitude to my colleagues in the research lab. Your collaboration, intellectual discussions, and shared experiences have enriched my research journey. The brainstorming sessions, support during challenging experiments, and shared moments of celebration have made this journey all the more memorable. I am grateful for the camaraderie we share and the collective growth we have achieved together.

I extend my heartfelt appreciation to all my friends, especially Davi Neves, Lucas Uszacki, Otávio Lessa, and Pietro Gasparin. Your friendship, understanding, and encouragement have been a constant source of joy and inspiration. Thank you for standing by my side, offering encouragement, and creating cherished memories together. Your support and understanding during my academic pursuits have meant the world.

UM CONTRA MUITOS: EXPLORANDO A GENERALIZAÇÃO DO APRENDIZADO MULTI-TAREFA EM TAREFAS COM CÓDIGO FONTE

RESUMO

Engenharia de software é um processo complexo que envolve vários passos, muitas vezes requerendo um investimento significativo de recursos. Como resultado, muitas ferramentas para suportar o desenvolvimento surgiram, com modelos de aprendizado de máquina se tornando cada vez mais populares para tarefas relacionadas. Recentemente, Transformers, uma classe de modelos, obteve um tremendo sucesso no processamento de linguagem natural e foi adaptado para trabalhar com código-fonte, com modelos como o CodeBERT treinado em texto e código. CodeT5, um desses modelos, emprega uma abordagem prompt multi-task durante o treinamento para garantir melhor capacidade de generalização para tarefas-alvo. No entanto, primeiro, é necessário esclarecer qual é o impacto dessa abordagem de multitarefa em um cenário Big Code. Nesta dissertação, estudamos as várias vantagens e desvantagens dessa abordagem de aprendizado para tarefas relacionadas a código-fonte. Usando modelos pré-treinados de ponta, comparamos métodos específicos de tarefas e de prompt multi-tarefa, analisando resultados de tarefas específicas para entender sua influência no desempenho. Também experimentamos diferentes combinações de tarefas para determinar quais são mais benéficas e se ajudam o modelo a entender melhor o contexto em que está sendo usado. Este trabalho lança luz sobre a aprendizagem de multitarefa prompt para tarefas de código-fonte, destacando como ela pode melhorar a eficiência de recursos e avançar a pesquisa em aprendizado multitarefa para Big Code.

Palavras-Chave: multi-tarefa, código-fonte, transformers.

ONE AGAINST MANY: EXPLORING MULTI-TASK LEARNING GENERALIZATION IN SOURCE-CODE TASKS

ABSTRACT

Software engineering is a complex process that involves several steps, often requiring a significant investment of resources. As a result, many tools to support development have emerged, with machine learning models becoming increasingly popular for related tasks. Recently, Transformers, a class of models, has achieved tremendous success in natural language processing and has been adapted to work with source code, with models like CodeBERT trained on both text and code. CodeT5, one such model, employs a prompt multi-task approach during training to ensure better generalization capability for target tasks. First, however, it needs to be clarified what impact this multi-tasking approach has on a Big Code scenario. In this thesis, we studied the various advantages and disadvantages of this learning approach for source-code-related tasks. Using state-of-the-art pre-trained models, we compared task-specific and prompt multi-task methods, analyzing results on specific tasks to understand their influence on performance. We also experimented with different task combinations to determine which are most beneficial and whether they help the model better understand the context in which it is being used. This work sheds light on prompt multi-task learning for source-code tasks, highlighting how it can improve resource efficiency and advance research in multi-task learning for big code.

Keywords: multi-task, source-code, transformers.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

AI – Artificial Intelligence

ANN – Artificial Neural Networks

AST – Abstract Syntax Tree

CNN – Convolutional Neural Network

CSN – Code Search Net

DL – Deep Learning

LLM – Large Language Model

ML – Machine Learning

MLM – Masked Language Modeling

MLP – Multi-Layer Perceptron

MTL – Multi-task Learning

NLCS – Natural Language Code-Search

NSP – Next Sentence Prediction

NMT – Neural Machine Translation

PL – Programming Language

QA – Question Answering

RNN – Recurrent Neural Network

UTCG – Unit Test Case Generation

CONTENTS

1. INTRODUCTION

Artificial Intelligence (AI) has emerged as one of the fastest-growing and most popular areas in Computer Science. Intelligent tools and technologies have become essential to modern life, automating several tasks and providing crucial assistance across many others [69]. One of the driving forces behind this advancement is Machine Learning (ML), which is widely used in various applications, including facial recognition [82], object detection [30], intelligent chatbots [13], and many more.

Software engineering is among the many areas that had significant improvements due to ML algorithms [7]. The process of documenting, writing, and testing code during the software development cycle is expensive and challenging, which motivates the application of many technologies to create productivity tools. A group of tasks such as Code Summarization [37], Natural Language Code Search [36], and Code Generation [48] had many advancements in the past few years due to the progress of ML tools that work with natural language, and more specific Artificial Neural Networks (ANNs) and Deep Learning (DL).

Since 2012 with AlexNet [40], ANNs and their numerous variations have had an expressive role in the recent achievements of AI. The use of GPUs to optimize the training process combined with large-scale datasets allowed the development of state-of-the-art models in many application domains [27].

An area deeply impacted by ANNs was Natural Language Processing (NLP), which studies methods to compute, process, and reason over textual data [22]. A flavor of ANN that became very popular in dealing with sequential data as text was the Recurrent Neural Network (RNN)[66]. RNNs store relevant temporal information in a hidden state, directly sent from a previous stage to the following, together with the network input [27].

A recent neural architecture that has become state-of-the-art for most NLP tasks is the Transformer [79]. This architecture is heavily based on the attention mechanism, and its capability to have parallelism during training allowed Transformers to replace the RNNs and rapidly take over the research landscape. What started as a method for machine translation was soon adapted to several other tasks and became the primary approach for DL in NLP [71]. Due to the significant advancements obtained by this architecture, many different Transformer-based models arrived and dominated the literature, such as BERT [21], BART [42], GPT [59], and T5 [61].

Those advancements brought by Transformers are not exclusive to NLP. Due to the naturalness hypothesis [7], we can assume that numerous natural language patterns are also present in programming languages, allowing us to use NLP techniques to solve source-code problems. That assumption allowed several Transformers [2, 25, 28] to be trained on bimodal data (code and natural language), permitting them to solve a large group of code-related tasks in the software development life cycle. Among bimodal models, CodeT5 [85] joined Multi-Task Learning (MTL) and Transformers for source code, adapting similar strategies from those applied in T5, but now to work with programming languages.

As seen in T5 and CodeT5, MTL is one of the implemented strategies to leverage the success of ML models [19]. The traditional framework to train a neural network uses only one task during the entire process, leading the model to become a specialist in that particular task. While effective, it differs enormously from human learning since we can extract information from more than one exclusive source [14]. Using more than one task during training can improve the model's generalization capability, using patterns learned from one task to improve performance in another [72].

Even with CodeT5 and other source code models, it is unclear how and when MTL can be a successful approach to Transformers in code-related tasks. The few studies [44, 85, 81, 56] that bind together these fields do not shed light on the consequences of MTL regarding single-task performance and generalization ability. Given the many released pre-trained models on programming language corpora [2, 18, 25, 28, 85], there is still an uncovered area of research regarding the benefits and struggles brought by MTL.

This work aims to compare single-task and MTL strategies when dealing with source-code tasks. We conduct several experiments to evaluate how different models perform in a prompt MTL setup for source-code-related tasks. Specifically, we focus on Natural Language Code Search (NLCS) and Unit Test Case Generation (UTCG) as evaluation tasks, which require a high level of code comprehension. We also use six other tasks to aid the learning process.

Our goal is to determine whether MTL can improve the performance of three different cutting-edge models in these specific tasks. Furthermore, we explore the influence of each task during training to discover interesting associations between datasets, tasks, and programming languages.

Although the multi-task models did not outperform the single-task models in most cases, they achieved comparable results, making them an exciting option for optimizing resource allocation. Moreover, our study revealed some curious behaviors, indicating that the pre-training process of the models has a heavy influence on the posterior adaptation of MTL with prompt modifications.

Through our study, we intend to clarify how MTL can be used to build better solutions for code-related tasks using deep models, understanding the consequences and relationships that can be established from different models in using multiple tasks during model training. We believe our findings contribute to developing more effective and efficient deep-learning models for source code-related tasks and to the research landscape of MTL.

This Master's thesis focuses on multi-task learning for source code-related tasks. Section 2 provides background information on software engineering, machine learning, and multi-task learning, highlighting the importance of these fields and their intersection. Section 3 presents related work, discussing the most relevant papers and projects in the area of multi-task learning for source code tasks. In Section 4, establish the research problem and our research questions to guide the experiments. Section 5 describes the methodology used in our experiments, detailing the datasets used, the models trained, and the evaluation metrics used to analyze the results. Section 6 presents the results of our experiments and discusses their implications, highlighting the advantages and disadvantages of the prompt multi-task approach for source-code tasks. Finally, in Section 7, we draw our conclusions, summarizing the contributions of our work and discussing avenues for future research in the area of multi-task learning for source code tasks.

2. BACKGROUND

2.1 Machine Learning

Machine Learning (ML) is a sub-field of Artificial Intelligence (AI), and according to Arthur Samuel, "Machine Learning (ML) is the field of study that gives computers the ability to learn without being explicitly programmed" [68]. Another interesting definition is from Tom Mitchell, who defines a learning problem as, "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E." [53].

This learning process is only possible with the help of many mathematical tools that allow algorithms to discover complex patterns. Such discoveries permit fitted models to predict behaviors or even describe events, becoming essential tools in the decision-making process [53].

The learning process with data-driven ML models is usually divided into two main steps: training and testing. At first, the algorithm learns patterns from a dataset, optimizing them to approximate the data distribution. During the test phase, we use a different data set that follows the same distribution from the training set since our objective is to validate if the learned patterns generalize knowledge rather than memorize what was seen.

If the model memorizes the training data, it becomes susceptible to model noise, thus performing poorly in test data. This phenomenon is called overfitting. The opposite occurs when the model cannot properly learn the data complexity, underfitting [76].

ML algorithms are usually divided into two main groups, defined by how the learning process occurs: supervised and unsupervised [76, 53] learning. The main difference between these two groups is the presence of a label (or target) that the model learns to associate with the features received as inputs. There are numerous other learning procedures besides those previously mentioned, like reinforcement learning [74], imitation learning [41], and others which we won’t explain in detail since they don’t relate to the theme of this work.

2.1.1 Supervised Learning

The objective of supervised learning is to predict the target variable (or label) y given an input x. During the training process, given a dataset $D = \{(x_i, y_i)\}_{i=1}^{N}$ the algorithm will learn how to correlate a set of inputs X to a group of corresponding target variables Y [57, 52]. Each instance x_i will commonly be a D-dimensional vector, and each vector value is called a feature or attribute. The idea behind supervised learning is that a teacher would carefully manage the learner, providing the correct answer for every input, thus guiding the model in the right direction to learn patterns that better predict the target variable based on the input features.

Two classical tasks commonly seen as supervised learning examples are classification and regression, where the main difference between them is the target variable data type [53]. For classification, each y_i will be a categorical value that belongs to a finite set of possibilities, $y_i \in \{1, ..., C\}$ [52]. Based on previous patient data, a classification model could predict whether a patient develops a specific disease. Regression is a slightly different task, where the target variable y_i is a real value, such as house prices or the temperature of some environment.

2.1.2 Unsupervised Learning

In unsupervised learning, the scenario is quite different from the previous one. There are no target variables, and the training process learns relations between the input data itself, $D = \{x_i\}_{i=1}^{N}$. Such algorithms are used for explanations and discovery scenarios, focusing on uncovering patterns that are not easily obtainable from more straightforward analysis methods. Due to that characteristic, a good knowledge of the problem context is desired to interpret the outputs correctly [57].

A classic example of this type of learning is the k-Means algorithm [47], which learns the best way to group the data into a specific number of clusters defined by the users following arbitrary criteria. Unlike supervised learning, there is no easy way to measure the quality of a model since there is no target label (ground truth) to compare the results with [53].

2.1.3 Self-Supervised, Few-Shot and Zero-Shot Learning

A significant problem in supervised learning is the need to label all data, a process that becomes very expansive when building large datasets. Considering the amount of available data that does not have target labels, an alternative to train them is by using a self-supervised learning approach. In this learning procedure, a model will learn the data distribution by comparing or predicting parts of the data using its data as the source [46]. A relevant example of self-supervised learning is language modeling. A model learns the token distribution (words or part of words) from a corpus and can suggest words based on the known distribution.

Another way to counter the lack of sufficient data to cover all cases in the dataset is to train the model to learn how to predict or do a task with little or no data. We call few-shot learning when a trained or fine-tuned model is expected to predict a group of instances based on a few labeled examples [73]. When no example is available during training, and we expect the model to generalize still and successfully predict the instance, we are dealing with zero-shot learning [89].

2.2 Artificial Neural Networks

Artificial Neural Networks are ML algorithms inspired by the functioning of the human brain. Introduced in 1943 by McCulloch and Pitts [49], the idea was to mathematically simulate a neuron, a main cellular component of the human brain. This idea gained life in 1958 with Rosenblatt's Perceptron [64], a simple structure that connects a set of input nodes to a group of output nodes [76]. Such nodes are commonly called neurons.

Each input node in the perceptron is connected to an output node by a weight w_i, where the weight w_0 is the bias term and $x_0 = 1$. The entering values in each neuron are multiplied by their respective weights and summed. The resulting value passes through a signal function that saturates to 0 for negative values or 1 otherwise. A mathematical notation for a single neuron perceptron can be seen in Equation 2.1:

$$h(x) = signal(w_0 + \sum_{i=1}^{n} w_i x_i). \tag{2.1}$$

A problem with the perceptron is the limitation of the algorithm to solve only linearly separable issues since it is only a linear combination of the inputs. Figure 2.1 exemplifies the linear decision boundary that makes it impossible for the model to solve problems like the logic XOR.

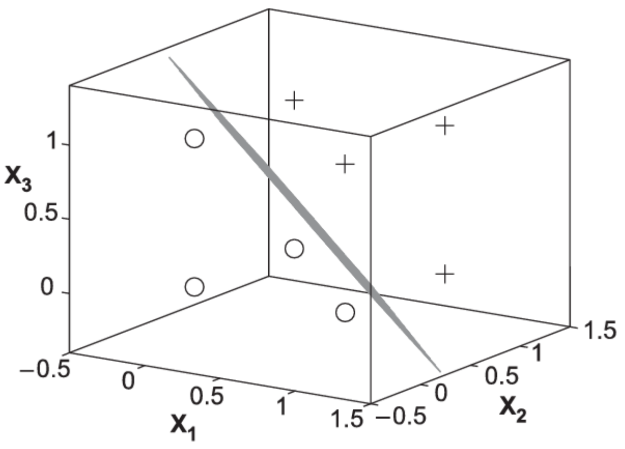


Figure 2.1: Perceptron's linear decision boundary [76].

Due to limitations of dealing with non-linear problems, the Multi-Layer Perceptron (MLP) arises, bringing hidden layers between input and output layers [52]. Besides new layers that already add some non-linearity to the model, another component was added: activation functions. These functions took the place of the signal function and replaced it with non-linear functions, helping the model to identify even more complex patterns in data. One of the most popular functions in contemporary architectures is the ReLU [1] function, shown in Equation 2.2, and those resulting from it, such as PReLU [31], GELU [33], CELU [10], and others. This multi-layered architecture with non-linear activation functions is usually called Feed Forward Network [27].

$$ReLU(x) = max(0, x) \tag{2.2}$$

The training process of a neural network is based on three components: loss function, forward pass, and backward pass. The loss function is responsible for guiding the optimization process of the network weights, changing accordingly to the more suitable function for the problem being learned. Usually, in a supervised learning scheme, the loss function will receive the predicted output $\hat{y}$ and compare it with the original labels y, estimating how far the model's prediction was from the real values. An example of a loss function for a regression problem is the Mean-Squared Error Loss (MSE), shown in Equation 2.3.

$$J(w) = \frac{1}{2}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 \tag{2.3}$$

The forward pass is when the model, given an input, uses the weights to try to predict the correct output for a specific task. With the predicted output, the loss function will evaluate how close to the objective the model is, and then, based on that evaluation, the resulting value is used to perform the backward pass.

The backward pass is the practical learning step, where the model uses the backpropagation algorithm to discover the weights' gradients for updating them, thus optimizing the loss function. Backpropagation is the step of computing the chain rule of multivariate derivatives, which is used to optimize the network weights, often in conjunction with an optimization procedure called Gradient Descent. The goal here is that one can minimize the loss function by iteratively making steps in the opposite direction of the gradient vector (vector of partial derivatives of the loss function with respect to each network's weight), thus reducing the difference between predictions and expected outputs.

2.3 Deep Learning

Deep Learning (DL) is the name of the sub-field of ML that studies large and complex neural networks that are used to build good feature representations during the training process [27, 11]. This field achieved high popularity due to the tremendous success of the so-called deep models in complex tasks with non-structured data such as image classification [32], object detection [30], and neural machine translation [9].

The main difference between DL and simpler neural networks, or other ML models, relies on generating features (or feature engineering), a process needed especially when working with non-structured data like images, audio, or text. Standard ML algorithms struggle when the input contains a high dimensional space due to the curse of dimensionality, which necessitates an efficient feature engineering [27] procedure. In DL, the network is responsible for the feature extraction phase, hierarchically discovering patterns in a composite way. For instance, in the first layers of a convolutional network, we see the emergence of more general representations, while in the deeper layers, the network learns more refined representations emerge [23].

Besides amplifying the number of hidden layers and making the network deeper, we got distinct architectures specified in DL to process different data. For images, convolutional networks achieved State-of-the-Art for a long period [32, 27]. These networks apply a sequence of filters (kernels) in the image, processing groups of pixels and extracting patterns from them, becoming a more elegant approach to many tasks with image processing.

2.3.1 DL for NLP

A common unstructured kind of data in which DL has excelled is natural language, becoming a valuable tool in the research area called Natural Language Processing (NLP). NLP is responsible for researching how machines can process and extract useful information from plain text data [22]. Usual Feed-Forward Networks cannot sufficiently represent sequential information, leading to problems when using them to understand text, where the order of words in the sequence is crucial for conveying the semantics of what is written. To overcome that problem, a different architecture called Recurrent Neural Networks (RNNs) [66] was proposed and started to be used for temporal series.

RNNs use parameter sharing to apply the update on the same group of parameters in different time steps, allowing them to generalize across multiple input forms and sequence lengths. By receiving a hidden state with the input, the network can keep the essential information from the past seen entries, building a structure similar to a memory. A problem emerges when we must keep long-term relations between the inputs due to the vanishing gradient problem, which limits RNNs to very short-term memory [34]. Two other recurrent architectures were proposed as alternatives to the vanilla RNN to solve the previously mentioned situation: Long-Short Term Memory (LSTM) and Gated-Recurrent Units (GRUs). Both architectures introduced different gates as alternatives to keep longer memories in the recurrent networks, achieving great success for various tasks such as text classification [4], neural machine translation [9, 16], and language modeling [51].

Another significant improvement in NLP was proposing a new vectorized way to represent words, word pieces, or characters in a sentence, namely Word Embeddings. They are dense vectors that, through a self-supervised learning approach, encode the semantic meaning of a token (usually a word or word piece) in a multi-dimensional space. Such representations can be used by neural networks when working with text, replacing other representation strategies such as Bag-of-Words (BoW) [22] or Term Frequency–Inverse Document Frequency (TF-

IDF) [67]. Two popular algorithms to generate word embeddings were proposed by Mikolov et al. [50]: Continuous Skip-Gram and Continuous Bag-of-Words. In contemporary architectures, these representations are trained together with the model itself in the embedding layer, usually seen among the initial layers of the model.

The difference between word embeddings and more straightforward feature extraction techniques is that instead of only counting the frequencies of terms, the algorithm tries to learn the semantics of a token by comparing how it relates to other tokens. By mapping the semantics, an exciting property arises from these dense vectors allowing us to analyze the relation between tokens arithmetically. With such properties, we can make analogies like $v_{king} - v_{man} + v_{woman} \approx v_{queen}$, where v_x represents the embedding of a word x [50].

2.4 Attention

Between the innovations to improve the performance of recurrent neural networks, the attention mechanism was one of the most prominent. It started as a way to solve the existing bottleneck between the encoder-decoder architecture, commonly used for the Neural Machine Translation (NMT) problem, where one network (e.g., an LSTM) would encode the textual representation and send the final hidden state to the decoder (e.g., another LSTM). In this architecture, the encoder is responsible for comprehending the data and compressing the valuable information of the input in one context vector. The decoder would use this vector to generate the output sequence sequentially [17].

Attention solves the previously mentioned bottleneck by measuring the relevance of each encoder's hidden state with the actual hidden state of the decoder. The decoder receives not only an input token and the previous hidden state but also a context vector c_i

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j \tag{2.4}$$

Where the value of α_{ij} is given by a softmax operation, evaluating how relevant each encoder's hidden state is for the decoder's current hidden state

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})} \tag{2.5}$$

$$e_{ij} = a(s_{i-1}, h_j) \tag{2.6}$$

In this case, a is an alignment function that computes the attention scores for the current decoder's hidden state s_{i-1} concerning each encoder's hidden state h_j. In the original paper, a is a feed-forward neural network [9].

The attention mechanism helped not only in matters of performance but also with the problem of the vanishing gradients since now the gradients can reach the early stages of the encoder network. Also, the attention scores can be used as an explainability tool to comprehend which words the model relied on when predicting a specific output word.

2.5 Transformer Networks

Entirely based on the attention mechanism, the Transformer network, introduced by Vaswani et al. [79], became a popular alternative to RNNs to work with sequential data. Initially proposed for NMT, the architecture relies mainly on attention blocks to identify the language patterns, allowing the model to leverage parallelism during training and thus replace the sequential RNNs. This difference not only allows the transformer model to train faster but also allows another way to capture language features.

The Transformer architecture is divided into two main blocks: an encoder and a decoder, and both are composed of smaller blocks called sub-layers. Each sub-layer output is given by $LayerNorm(x + Sublayer(x))$, where $LayerNorm$ is a normalization layer, and the summation with x is a residual connection. The encoder is responsible for extracting a rich feature representation from the input, and the decoder uses the extracted features to build the output.

The model receives as input a group of embedded tokens that are summed with the positional encoding, a function responsible for keeping track of the temporal aspect of the model while still allowing for parallel computation.

The authors call their attention function "Scaled Dot-Product Attention", which receives as input a group of query and key values with dimension d_k and a set of values with dimension d_v. The final equation is similar to the dot-product attention function but scaled by the factor $\sqrt{d_k}$, as can be seen below

$$Attention(Q, K, V) = softmax\Big(\frac{QK^T}{\sqrt{d_k}}\Big)V \tag{2.7}$$

This function is used in the Multi-Head Attention layers, which apply it to the input itself multiple times (different views of what can be learned).

$$MultiHead(Q, K, V) = Concat(head_1, ..., head_h)W^O \tag{2.8}$$

$$head_i = Attention(QW_i^Q, KW_i^K, VW_i^V) \tag{2.9}$$

Each parameter matrix W is learned during the train, and the number h is a hyper-parameter called attention heads [79]. The Masked Multi-Head Attention layer is similar to the previous one, with the difference that subsequent tokens are masked, and the last sub-layer is a simple feed-forward network. The complete Transformer representation can be seen in Figure 2.2.

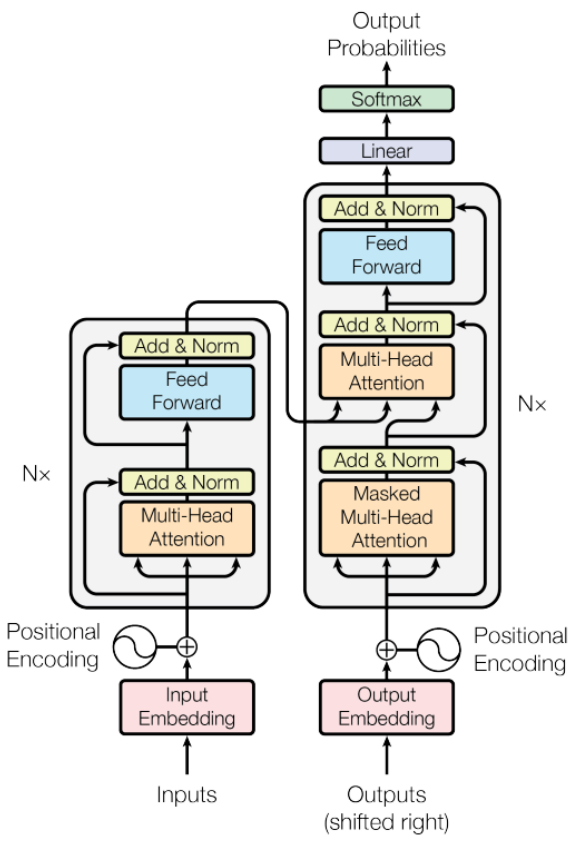


Figure 2.2: The Transformer's architecture is divided into an encoder and a decoder. Both parts comprise other functions, mainly based on the attention mechanism [79].

2.5.1 BERT, GPT, and T5

Besides the original transformer architecture, other Transformer-based architectures achieved success in a wide range of natural language tasks [12, 63, 61], adapting the NMT structure to new different forms of processing data. These new architectures brought the possibility of pre-training the model in large corpora and thus made it available for fine-tuning, following the transfer learning scheme [91].

Introduced by Delvin et al. [21], BERT stands for *Bidirectional Encoder Representations from Transformers*. BERT is a Transformer-like model that encodes a given sentence into a dense representation, becoming very useful for extracting rich textual features. BERT's architecture is based only on the encoder block of the original Transformer [79], and due to that architecture, observed in Figure 2.3, it is easily adapted to solve different natural language tasks after the appropriate pre-training process, becoming an excellent textual encoder.

BERT's main innovations were in the model's architecture, as already mentioned, and in the training process. Instead of the simple conditional language modeling, where the model should predict the next token for the given context, BERT is trained with Masked Language Modeling (MLM). In MLM, the model receives a group of WordPiece [88] tokens, and 15% of the tokens are replaced at each iteration. The objective of the model, then, is to predict the replaced words. But if every replaced token were replaced with a mask token, there would be a big difference between pre-training and fine-tuning, where the model does not have masked tokens. The alternative is not always to be replaced by a mask, from the 15%, 80% are masked as $[MASK]$, 10% are replaced by a random token, and the remaining 10% remain unchanged. With this approach, the model can learn bidirectional representations from the input itself, without limitations to processing the text only left-right or vice-versa [21].

Together with the MLM, Next Sentence Prediction (NSP) is also used during training to allow the model to understand the relationship among different sentences. A $[SEP]$ token is used in the input to indicate the division between the first and second sentences. The output then comprises a token to indicate whether the second sentence is the continuity of the first. This task is added to allow BERT to learn downstream tasks such as Question Answering (QA), where the model needs to receive two input sentences instead of one.

BERT also uses a different way to embed the tokens for the input. In addition to the composition of token embedding and positional embedding, already seen in the original Transformer paper, due to the NSP task, segment embedding is also summed to the dense representation. With that, the model can identify which tokens belong to the first sentence and which belong to the second [21].

Another famous Transformer-based architecture is the GPT family [59, 60, 12], composed only by decoder blocks. Released by OpenAI and distinct from BERT, GPT does not create bidirectional representations from words since it works

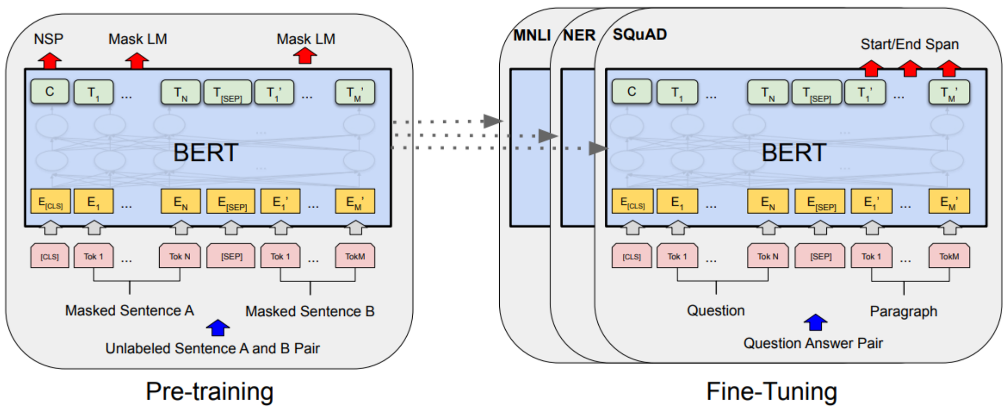


Figure 2.3: BERT has specific pre-training tasks that allow the model to learn general characteristics of the natural human language. After pre-training, the model can be adapted to a large number of different tasks by a fine-tuning process [21].

autoregressively. That characteristic makes it great at conditional generation tasks such as conditional language modeling or text generation. The most recent model following that architecture was GPT-3 [12], a large language model trained in massive corpora and responsible for exploring the limits of what language models can do.

Differently from both previous architectures, T5 [61] follows a more approximate architecture from the original Transformer, composed of a group of encoder and decoder blocks. The central aspect behind T5 is the natural capability of the model to be trained in a multi-task learning paradigm. Every task is treated as a sequence-to-sequence problem and distinguished by a prefix attached to the prompt input, as seen in Figure 2.4.

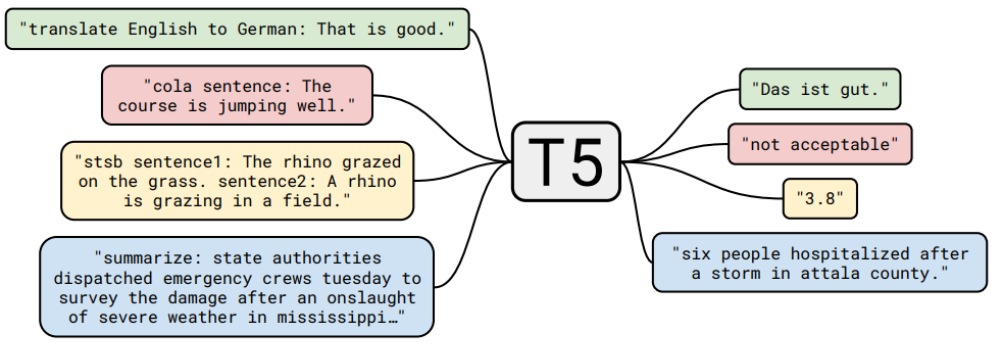

"translate English to German: That is good."
"cola sentence: The course is jumping well."
"stsb sentence1: The rhino grazed on the grass. sentence2: A rhino is grazing in a field."
"summarize: state authorities dispatched emergency crews tuesday to survey the damage after an onslaught of severe weather in mississippi…"
T5
"Das ist gut."
"not acceptable"
"3.8"
"six people hospitalized after a storm in attala county."

Figure 2.4: The text-to-text framework proposed to train T5 in the multi-task learning setup. The prefix identifies which task should be performed by the model [61].

2.6 Source Code

Every software developed follows a Programming Language (PL) in the form of source code. We can define source code as "any fully executable description of a software system. It is therefore so construed as to include machine code, very high-level languages, and executable graphical representations of systems" [29]. Thus, source code is responsible for defining the behavior of a computer program by putting together a group of logical arguments, operations, and rules in a static human-readable structured sequence of steps to build an algorithm. The term *big code* refers to performing statistical analysis or training machine learning models with huge programming language corpora [7, 5]. Most ML tasks in big code are related to software development necessities and productivity tools.

Although programming languages are written using natural language terms and can be analyzed as textual documents, there are some differences between both. Ren et al. [62] enumerates three major differences:

- Number of Words: We have a selected and limited number of words in a programming language, a direct contrast to natural language. When analyzing source code, a group of specific words has more meaning and relevance than others, which implies different learning and evaluation approaches.

- Document Structure: reading code is not as simple as reading any English text. Since we cannot simply go from left to right, we must obey the natural tree structure of code composed of variables, loops, and conditional commands. This tree structure is called the Abstract Syntax Tree (AST) [3].

- Semantic Structure: while the natural language may have ambiguous expressions, in source code, each instruction must be unique, objective, and specific on what should happen.

The AST representation holds code-specific information that cannot be understood by analyzing the tokens and their order in the source code. In an AST, each part of the code is detailed in a tree-like structure, where each node contains relevant details, such as whether the node is an identifier. Using this representation is said to improve the performance of a neural model [28, 8, 58, 85] since they can use not only syntax information but also semantics.

It is important to notice that even with those relevant differences, a hypothesis considers coding as a form of human communication. Therefore, a group of statistical properties in source code should be similar to natural language [7]. The Naturalness Hypothesis, as it is called, allows us to explore these patterns and extend many of techniques used with natural text to the programming domain without significant complications. Following the trend in NLP, the most recent approaches for source code tasks are Transformer-based [79], following the scheme of pre-training and fine-tuning.

A publicly-available benchmark for source code tasks is called CodeXGLUE [48], inspired by the original GLUE [80], a popular benchmark for NLP models. The idea of such benchmarks is to group tasks and corresponding datasets that can be used to evaluate how well a model can understand complex aspects of a specific area. CodeXGLUE splits 10 tasks into 4 distinct categories based on the input and output modalities, establishing and keeping a list of best-performing models for each.

2.7 Multi-task Learning

Unlike typical ML models commonly trained as task specialists, humans learn multiple tasks simultaneously. For instance, babies learn how to walk isolated from other activities and leverage information from other motor skills that are being developed simultaneously to help in the learning process [19]. Relying on that principle, Multi-Task Learning (MTL) is a learning paradigm where more than one task is used during training, expecting the model to generate better feature representation by using different knowledge sources.

MTL is a solid approach to improving the generalization capabilities of a model [65]. NotIt is not only an alternative to leverage performance but can be a tool for lowering the computational resource cost since instead of training one model for each task, we can use only one model for multiple objectives [72, 93].

A formal definition of MTL can be found in the work of Zhang and Yang [93], where:

> Given m learning tasks $\{T_i\}_{i=1}^{m}$ where all the tasks or a subset of them are related, MTL aims to learn the m tasks together to improve the learning of a model for each task Ti by using the knowledge contained in all or some of other tasks.

Recent work divides MTL into two categories: hard and soft parameter sharing [19]. In hard parameter sharing, all or the majority of the layers from the network are shared across tasks. A standard option with neural networks is to share the hidden layers and keep a task-exclusive head to generate the task-specific outputs [19]. Differently, soft parameter sharing is a paradigm where each task has its model with its weights. Then, regularization is applied to the model, inducing the parameters to be as similar as possible.

For Transformers, a relevant strategy to MTL that becomes popular with T5 [61] is to use the prompt to warn the model about which task should be executed for a specific input. With that premise, it adds a task prefix to the input signaling to the model in which the task should be executed. Manipulating the input's behavior to lead the model toward a specific output or objective has recently become popular, especially after GPT-3 [12] and the area of prompt engineering [43, 92].

A critical aspect to pay attention to when following an MTL paradigm in which tasks should be learned together. The tasks used during training must have some level of similarity; otherwise, that strategy can lead to poor performance and the phenomenon called negative transfer [72]. In fact, different details besides the relationship of the tasks need to be taken into account when training, such as the frequency at which each task will appear, which datasets to use, and which general strategy training will follow (hard or soft parameter sharing). Such points are necessary to avoid negative transfer and stop the knowledge obtained from one task harms another [72].

It is essential to distinguish between MTL [14] and transfer learning [91] since that, in both paradigms, more than one task is used. Besides being a widely adopted strategy to speed up and improve the generalization process [21, 25, 61, 48, 32], transfer learning has two different moments: pre-training and fine-tuning. During pre-training, a task such as language modeling is used to induce the model to learn general aspects of a particular field or context (e.g., general semantics and syntactic). The pre-trained model is then used in the fine-tuning step, which focuses on specifying the weights for solving a target task. In MTL, we focus on using more than one task in the training process without distinguishing the number of steps or methodology.

Some recent models are pre-trained with multiple tasks, such as BERT, which uses MLM and NSP [21]. In these cases, the literature does not commonly refer to the pre-training tasks as an MTL scenario, even with more than one task guiding the optimization of the training. T5 [61] and similar models are exceptions; once the pre-train with many tasks is the

Recently Google Research introduced Pathways Language Model (PaLM) [**?**], a 540-billion parameter, dense decoder-only Large Language Model (LLM) trained with Pathways [20], a distributed system to train machine learning models efficiently. The paper claims that the more parameters the LLM has, the more knowledge it can extract from the dataset, learning how to do more distinct tasks without explicit training. Unfortunately, the proportion of resources to train an LLM such as PaLM or GPT-3 [12] is unrealistic to most research laboratories or any practitioner’s available machinery, leading the community to search for ways to optimize the usage of available hardware.

3. RELATED WORK

3.1 Source Code Models

The first proposed models for source code tasks were RNN based, such as LSTMs [35], following the idea of adapting natural language solutions for programming languages. Code-NN [37] was one of the most relevant neural-based models proposed for source code tasks. The proposed model achieved interesting results in Code Summarization, where for a given code snippet, the model should output its corresponding natural language description. For example, in Natural Language Code Search (NLCS), given a natural language description, the model should find the most similar code snippet according to a score function and a previous database. Using a dataset based on StackOverflow[1] posts, they trained an LSTM with attention for each task as encoder and an LSTM as decoder for summarizing source code.

Code2Seq, proposed by Alon et al. [8] is another relevant work based on RNNs. In addition to the natural sequential information, an AST [3] representation is used to extract code-specific structural details. The proposed model follows the encoder-decoder framework, with a fully-connected layer in the middle. Each AST path is encoded by an LSTM and summed with the code embeddings, creating a combined representation used as input for the fully-connected layer. The output of the fully-connected layer will feed an LSTM decoder cell with attention. The authors prove that their model performs better than previous baselines in both tested tasks, Code Summarization and Code Captioning[2].

3.1.1 Transformer Models for Code

Following the general NLP research landscape, large pre-trained Transformer-based models like CodeBERT [25] and CodeGPT [48] achieved great success in source code tasks. They usually pre-train the models on a large code corpus collected from GitHub [3] using language modeling as the training objective to cap-

[1]https://stackoverflow.com/

[2]In this work, the authors treat Code Summarization as discovering the name of a function and Code Captioning as finding a natural language description from a code repository to match a code snippet.

[3]https://github.com/,

ture general code information and be successfully fine-tuned to a downstream task like Code Summarization or Code Generation.

CodeBERT [25] was the most relevant BERT-like model released for code tasks. It was pre-trained on the CodeSearchNet dataset (CSN) [36] with *masked language modeling*, and *replaced token detection* tasks using code and natural text as bimodal data, and only code as unimodal data. After pre-training, the model is fine-tuned to Code Summarization and Code Search, both tasks that require learning programming and natural language. Even without a code-specific representation such as AST, seen in Code2Seq [8], CodeBERT effectively learns how to extract meaningful dense representations out of code and text, allowing it for further adaptations.

As an improvement to CodeBERT, GraphCodeBERT [28] goes beyond the syntactic level and implements two data-flow pre-training tasks, capturing the semantics and code structure. With *edge prediction*, the model can learn about the relationship between the variables themselves, and with *node alignment*, the model learns how these variables behave on the code snippet. Another CodeBERT-based improved model is SynCoBERT [84], which also uses two pre-training tasks specific for code: *identifier prediction*, which tries to predict, for each code token, whether it is an identifier, and *AST edge prediction*, very similar to GraphCodeBERT's edge prediction.

Based on GPT-2 [60] architecture, CodeGPT [48] was released as an autoregressive model for Code Completion and Code Generation tasks. More recently, Codex [15] was released by OpenAI as an LLM for code, serving as the base for GitHub Copilot [4], a service to help programmers with code completion in daily tasks. Some other models were also released and evaluated for generation-focused tasks, some trying to beat Codex, and some released as experiments or tutorials like CodeParrot [90].

Following the original Transformer encoder-decoder structure, two proposed models were PyMT5 [18] and PLBART [2], based on T5 [61] and BART [42] architectures, respectively. Both models were pre-trained only on masked language modeling tasks, without any code-specific task, and then are fine-tuned on downstream tasks: PyMT5 in Docstring and Code Generation (defined by the prefix attached in the input) while PLBART in Code Summarization, Code Generation, Code Translation, and Code Classification.

[4]https://github.com/features/copilot

3.2 MTL for source code Tasks

Following the idea of using deep learning and MTL to increase productivity in development tools, CugLM [44] was proposed as the first Transformer-based code completion model. Three tasks are used during model pre-training. Bidirectional MLM and Next Code Segment Predicting are similar to the tasks used for BERT [21] pre-train but adapted to source code context, one for token level and another for sentence level. The third task is unidirectional LM, which is the same technique in autoregressive models like GPT-2 [60].

Interestingly, only identifier tokens with type information are masked for bidirectional MLM. Identifier tokens are variables or function names, often containing relevant information about the code snippet. The authors claim these tokens contain relevant information to help the model understand the source code. As in BERT, the loss of the model is a sum of the three cross-entropy losses.

A limitation of this work is that for weakly-typed programming languages, the first pre-training task becomes a regular masked language modeling task. Thus, the model loses all code-specific information it could understand, becoming just an adaptation of natural language tasks to a different context. Moreover, significant structural information is lost without a specific method to extract it.

Another MTL approach for source code tasks is MulCode [81], a model that combines a pre-trained BERT [21] and a Tree-LSTM [75] to extract code-based embedding and an AST representation, respectively, acquiring and using sequential and structural information. The model contains three layers: (i) a universal representation layer responsible for receiving the inputs and extracting the features to be used in the tasks; (ii) the task-specific input layer that uses two attention modules to give the necessary importance to sequential and structural information, which changes from task to task; (iii) and finally, a task-specific output layer to compute the outputs and losses in the training step.

MulCode itself is not pre-trained due to its general architecture, just a combination of other pre-trained models. The model is trained on three different downstream tasks:

- Comment classification (classifying whether a statement is good or not).
- Author attribution (discovering the author of a given code snippet).
- Duplicated function detection (identifying whether two functions have the same functionality).

Those tasks were selected due to the small size of the datasets available for each, an exciting scenario where MTL can improve the model's generalization capability by combining information from all datasets and tasks. Another important aspect is the usage of the AST representation, which provides the model with code-specific information for all tasks.

The authors tested the approach with three other distinct tasks: library classification, algorithm classification, and bug detection to show that the method generalizes well for source code. These three tasks were chosen given their dissimilarity in terms of complexity and dataset sizes.

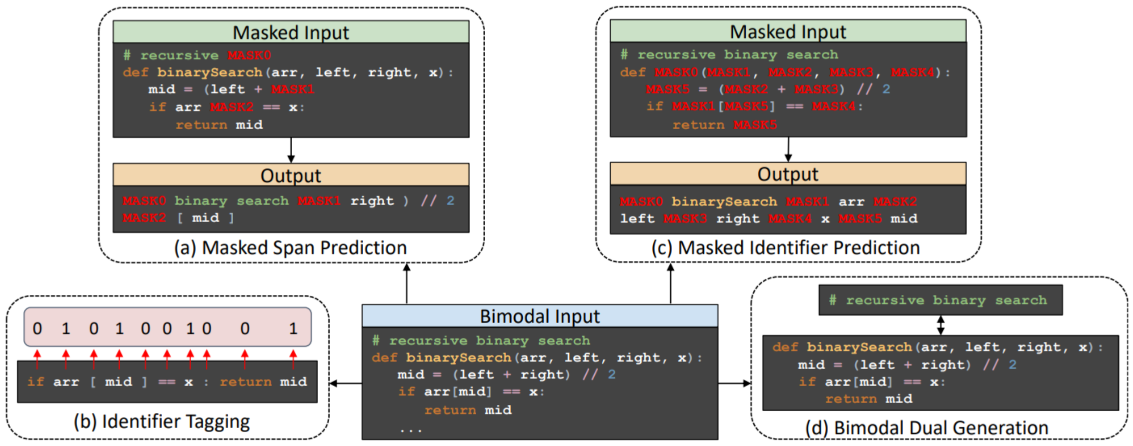
CoTexT [56] comes as the first multi-task approach to combine the CodeSearchNet dataset [36], and T5 [61] checkpoints as initialization configuration. They use a self-supervised pre-training strategy and then fine-tune the model on four downstream tasks from the CodeXGLUE benchmark [48]: Code Summarization, Code Generation, Code Refinement, and Defect Detection.

The MTL approach followed by the authors uses prefixes to specify in which language the input is, similar to T5 [61]. That method is only used for fine-tuning tasks that contain more than one language or possible configuration. Hence, for a task like Code Summarization on CodeSearchNet, the input got a prefix containing the name of the programming language, treating each different PL as another task. This multi-task approach was used for only two tasks between the four downstream previously discussed.

Another T5-based [61] model was CodeT5, proposed by Wang et al. [85], a transformer model trained for better code understanding and generation. Since it relies on T5 architecture, the authors claim that the encoder extracts useful source code knowledge into a dense representation while the decoder handles generation tasks. The main difference from CoTexT is the pre-training process, where besides the ordinary self-supervised learning method, they use a combination of source-code-focused tasks to enhance model comprehension and understanding. Further, the model is fine-tuned to a group of six tasks to evaluate its performance.

During pre-training, besides the usual MLM (a denoising objective), CodeT5 implements two tasks to help the model encode code-specific properties based on information extracted from the AST representation. The first is *identifier tagging*, where the model tries to discover which tokens are identifiers, and *masked identifier prediction*, where all identifiers receive a specific unique token, and the model needs to discover the tokens based on their occurrences in the code snippet. The combination of these three approaches together is called *Identifier-aware Denoising Pre-training*.

Another pre-training task tested in a few experiments is *bimodal dual generation*, which is the translation from code to comment and vice-versa. The entire pre-training scheme of CodeT5 can be seen in Figure 3.1.



```
# recursive MASK0
def binarySearch(arr, left, right, x):
    mid = (left + MASK1
    if arr MASK2 == x:
        return mid
```

```
MASK0 binary search MASK1 right ) // 2
MASK2 [ mid ]
```

```
# recursive binary search
def MASK0(MASK1, MASK2, MASK3, MASK4):
    MASK5 = (MASK2 + MASK3) // 2
    if MASK1[MASK5] == MASK4:
        return MASK5
```

```
MASK0 binarySearch MASK1 arr MASK2
left MASK3 right MASK4 x MASK5 mid
```

```
if arr [ mid ] == x : return mid
```

```
# recursive binary search
def binarySearch(arr, left, right, x):
    mid = (left + right) // 2
    if arr[mid] == x:
        return mid
    ...
```

```
# recursive binary search
```

```
def binarySearch(arr, left, right, x):
    mid = (left + right) // 2
    if arr[mid] == x:
        return mid
```

Figure 3.1: The complete pre-training scheme of CodeT5, bounding together all four tasks. Important to mention that Bimodal Dual Generation is not always used [85].

An MTL approach was tested and compared with the typical singular-task protocol for fine-tun downstream tasks. In the multi-task scenarios, the authors used the same strategy from CoTexT and T5 to add prefixes indicating to the model which task should be performed to get the correct output. A difference between CodeT5 and CoTexT is that were not only the PLs appended at the input but also the task that the model should perform. Especially for Code Summarization, the multi-task approach achieves the best scores.

4. RESEARCH PROPOSAL

Though sharing similarities with natural language, source code is an essentially different data modality, with specific traits and characteristics [7]. With the rise and success of large pre-trained Transformers that work with both programming and natural language modalities [2, 18, 25, 28, 85], one can study how the application of different training techniques and methods may affect the behavior of a model for helping in source-code tasks.

MTL is undoubtedly a powerful strategy to optimize resources and increase the generalization capability of ML models [19]. By learning a group of related tasks together, the model can leverage information from different perspectives and sources to enhance its overall performance. In recent years, the approach of adding prompt prefixes as a way to induce MTL in language models [61, 85, 56] has emerged as a simple yet effective way to boost model performance. Despite its benefits, there is still a lack of exploration about the possibilities of exploiting MTL for source-code-related tasks.

A good example is CodeT5 [85], which only applies MTL as an attempt to improve performance without considering a further analysis of the tasks or a more detailed exploration. That shows the current lack of understanding on how multi-task can increase or decrease model performance, especially when using multi-task via prompt modifications.

In addition to improving performance, MTL is a particularly attractive option for reducing the computational resources required for training models [14]. In today's era of large language models, practitioners without access to powerful hardware need to explore alternative strategies for optimizing available resources. One such strategy is to solve multiple tasks using a single model, which can significantly reduce the computational burden of training and deploying ML models.

Our focus in this work is to explore prompt multi-task learning for source-code tasks and analyze its effect on the generalization ability of a model. To achieve this goal, we will compare various neural architectures and evaluate our models on two specific tasks: Natural Language Code Search (NLCS) and Unit Test Case Generation (UTCG). Notably, UTCG has not yet been evaluated or experimented within the MTL approach, making this study a novel contribution to the field.

Our objective is to address the lack of exploration in using MTL for source-code tasks by adopting the idea of using a prefix attached to the beginning of the input to determine the model's intended task [85, 61]. We aim to investigate not only whether a multi-task model can surpass its single-task counterparts but also understand the relationship between tasks and datasets, and how each contributes to the overall training procedure. It is worth noting that due to computational constraints, some compromises will be made for model training, such as limiting the number of experiments and reducing batch sizes.

4.1 Research Questions

Aligned with the objective detailed above, we have built a group of research questions we answer in this thesis, followed by the methods used to achieve those answers.

- RQ1: Does prompt MTL improve the generalization ability of a model in UTCG and NLCS? If so, by how much?
- RQ2: Some tasks in our prompt MTL setup have a major influence on the training process. Is the same behaviour observed for both UTCG and NLCS and between the models?
- RQ3: Will a model trained with MTL have better zero-shot performance in UTCG and NLCS than models trained for single tasks?

To answer our research questions, we will follow a specific methodology. First, we will train three state-of-the-art architectures for big code learning in both multi-task and single-task approaches for two downstream evaluation tasks while comparing the scores on specific metrics for each task. This will allow us to determine whether there are direct improvements or undesired consequences achieved by using the multi-task learning paradigm. Throughout these comparisons, the single-task models will be considered as baselines.

In order to address RQ2, we will train models with different combinations of tasks using a multi-task learning approach and evaluating the impact of each task on the overall model performance. For example, we may train a model with only sequence-to-sequence tasks and compare its performance to that of a full multi-task model, analyzing whether tasks that do not follow such a framework have a bad influence in the training process. This will allow us to determine whether certain tasks can help the model encode code-specific information more effectively than others and shed light on the relationship between the tasks and the overall performance of the model.

Finally, for answering RQ3, we will first train the models in a MTL setup using all tasks described previously, except by the evaluation ones, and then without any other adjustment we will measure their zero-shot performance. Recall that zero-shot learning is the practice of using a model to infer or recognize patterns and classes not seen during training [89], and in NLP we can use zero-shot inference to have a good notion about the generalization capability of a model [12]. Our objective with this question is to check whether MTL improves the model overall knowledge about the domain.

5. METHODOLOGY

5.1 Evaluation Tasks and Metrics

Our evaluation will focus on being fair and resource-optimized. For such, we will focus on two downstream source code tasks, namely Unit Test Case Generation (UTCG) and Natural Language Code Search (NLCS). We will train baseline single-task models to ensure a fair evaluation. By limiting our focus to two tasks, we intend to facilitate our analysis at this moment of the research and also save us computational time.

UTCG is a fundamental part of software development. It can help prevent numerous problems and also the necessity of exhaustive refactoring. Nevertheless, it is a complicated and time-consuming task, mainly due to the nature of implementing the functions and attending to all points that need to be tested. Hence, the use of machine learning models to help in this process comes in handy. In UTCG, we aim to create unit tests for source code methods: given a function (method) coded in a particular programming language, return a functional unit test for that function.

To evaluate the performance of the generated tests, we will use two main metrics: Bilingual Evaluation Understudy (BLEU) [54], commonly used in Neural Machine Translation and also for other sequence-to-sequence tasks, and CodeBLEU [62], an adaptation of BLEU for source-code tasks.

In order to compute BLEU, a candidate sentence is compared with a group of references in different *n*-gram levels. Equation 5.1 demonstrates the general approach to compute the metric, where p_n is the modified precision obtained with the *n*-grams, N is the number of *n*-grams that will be used (commonly set to 4), and w_n is the weight (usually we use uniform weights, where $w_n = \frac{1}{N}$). As described in Equation 5.2 and used to penalize short translations, BP is the brevity penalty, where c is the size of the candidate and r is the size of the reference.

$$\mathrm{BLEU} = \mathrm{BP} \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right) \tag{5.1}$$

$$\mathrm{BP} = \begin{cases} 1 & \text{if} c > r \\ \exp\left(1 - \frac{r}{c}\right) & \text{if} c \leq r \end{cases} \tag{5.2}$$

Unfortunately, BLEU is not enough to measure the quality of the generated source code. Since BLEU only measures *n*-gram matches, it only considers the sequential structure of the evaluated text, thus not evaluating more relevant aspects of source code. To capture code-specific information and properly measure the quality of the generated tests, we will also use CodeBLEU [62], a metric that evaluates the tree structure and AST representation of code to measure quality. CodeBLEU uses, besides *n*-gram matching, a weighted version that considers relevant words more important than others, a syntactic AST matching procedure, and a data-flow match.

As seen in Equation 5.3, each criterion is weighted by a specific term that provides its importance. In this work, both α, β, γ, and δ are set to 0.25, which means we are giving equal importance to all terms.

$$\text{CodeBLEU} = \alpha \cdot \text{BLEU} + \beta \cdot \text{Weighted BLEU} + \gamma \cdot \text{Syntatic AST Match} + \delta \cdot \text{Semantic Data-flow Match} \tag{5.3}$$

In NLCS, given a natural language function description such as "*how to multiply matrices?*", the model should retrieve the corresponding code snippet that performs the described behaviour, or at least the most similar behaviour to that description. Since it is a retrieval task, a common metric used to evaluate ranking systems is the Mean Reciprocal Rank (MRR), which evaluates the correctness of ordered lists for a group of queries. Equation 5.4 details how MRR is calculated, with Q representing the queries and $rank_i$ the place where the most relevant value appeared on the list for the i-th query.

$$MRR = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{\text{rank}_i} \tag{5.4}$$

5.2 Datasets and Tasks

This study consists of eight tasks, each with its respective dataset. Two tasks will be used for training and evaluation, while the other six will serve as support for the training process, and we will not measure model performance on them. In the following, we will provide details about each dataset, the associated task, and general information such as size and structure.

Two large datasets will be used to train and evaluate most models for this work: CodeSearchNet (CSN) [36] and Methods2Test [77]. The CSN dataset was first introduced as a challenge for the code retrieval task. Given a natural language description of a function as input, the model should output a code snippet that performs that function. Since this dataset is the same one used to train CodeBERT [25], we can use the already cleaned version, available on the project's GitHub Repository[1].

```
# Train Input
def file_handle(fnh, mode="rU"):
   handle = None
   if isinstance(fnh, file):
      if fnh.closed:
         raise ValueError("Input file is closed.")
      handle = fnh
   elif isinstance(fnh, str):
      handle = open(fnh, mode)
   return handle

# Train Output
"""Takes either a file path or an open file handle, checks validity and
   returns an open
   file handle or raises an appropriate Exception.

   :type fnh: str
   :param fnh: It is the full path to a file, or open file handle

   :type mode: str
   :param mode: The way in which this file will be used, for example to
      read or write or
             both. By default, file will be opened in rU mode.

   :return: Returns an opened file for appropriate usage."""
```

Methods2Test is a dataset released by Microsoft that contains a group of focal methods and their respective unit tests. In addition to the focal methods, the dataset also contains the context in which the methods are inserted, so they can be used to improve results. The dataset is available on the GitHub Repository in https://github.com/microsoft/methods2test, and we exemplify one instance from the dataset in the source code below.

```
// Train Input
public static String getThreadId() {
   Instant now = Instant.now();
```

[1]https://github.com/microsoft/CodeBERT

```
    long nano = now.getNano();
    long remainder = nano - (nano / 1000 * 1000);
    if (remainder == 0) {
        int rndNano = random.nextInt(1000);
        nano = nano + rndNano;
    }
    return now.getEpochSecond() + "-" + nano;
}

// Train Ground Truth
@Test public void testGetThreadId() {
    String threadId = Utils.getThreadId();
    assertNotNull(threadId);
}
```

A fundamental cleaning step to prepare the datasets is the removal of duplicated instances. As seen in the work of Allamanis [6], the presence of duplicated code snippets can lead to an overestimation of the actual model performance, which is currently one of the major problems when evaluating very large models that are trained on big code. Also, for CodeSearchNet, we removed the docstrings present inside the source code snippets to avoid performance overestimation of the retrieval task.

Each of the six aiding tasks that were used exclusively for multi-task training has a distinct dataset with its own programming languages, input, and outputs. We use the same pool of tasks from CodeT5 [85], so the datasets are used as provided in the repository. Code Summarization is the only task for which we did not download a specific dataset; instead, we adapted the code to load different information from CSN, already used for NLCS.

We introduce a brief description of the datasets and the associated task below:

- Translation: this task involves translating a code snippet from one programming language to another. The dataset contains pairs of Java/C# code snippets, where one is used as source and the other as target for model training [85].

- Refinement: in this task, the model is given a buggy code snippet and should output its fixed version. The dataset contains pairs of buggy and fixed Java functions [78].

- Generation: the task is to generate a code snippet based on a given natural language description. The dataset includes natural language descriptions and contexts as input, and the output is a source code function [38].
- Defect Detection: this task involves analyzing whether a function is vulnerable to external attacks from other systems. The dataset includes C functions and their corresponding vulnerability labels (true or false) [94].
- Clone: the task is to determine whether two code snippets have the same semantic meaning. The dataset contains pairs of Java functions and a label indicating whether they are clones or not [83].

All datasets statistics are shown in Table 5.1.

Table 5.1: Statistics of each dataset used in the experiments, considering the absolute number of instances and the relative percentage. For Code Search Net datasets we consider their percentage doubled since they are used in two tasks (Code Summarization and NLCS).

Task Name	Train Size	% Train	Valid Size	% Valid	Test Size	% Test	Language(s)
Clone	901.028	25.59	415.416	69.89	415.416	68.05	Java
Code Search	167.288	9.50	7.325	2.46	8.122	2.66	Go
Code Search	164.923	9.37	5.183	1.74	10.955	3.59	Java
Code Search	58.025	3.30	3.885	1.31	3.291	1.08	Javascript
Code Search	241.241	13.71	12.982	4.37	14.014	4.59	PHP
Code Search	251.820	14.31	13.914	4.68	14.918	4.89	Python
Code Search	24.927	1.42	1.400	0.47	1.261	0.41	Ruby
Defect	21.854	0.62	2.732	0.46	2.732	0.45	C
Generation	100.000	2.84	2.000	0.34	2.000	0.33	Java
Methods2Test	624.022	17.73	78.534	13.21	78.388	12.84	Python
Refine	46.680	1.33	5.835	0.98	5.835	0.96	Java
Translate	10.300	0.29	500	0.08	1.000	0.16	Java/C#

5.3 Models and Baselines

To answer the research questions, we will primarily make use of three models detailed in the related work section: CodeBERT [25], GraphCodeBERT [28], and CodeT5 [85]. Only CodeT5 is naturally suited to be used in a multi-task learning setup by adding prefixes in the input. Therefore, we had to adapt the other models for the multi-task training setup.

To adapt both BERT-like models to an encoder-decoder setup, like CodeT5, we used a specific module from HuggingFace's hub [87] called "EncoderDecoder-Model". That module can receive two BERT models and use the first as encoder and the second as decoder, adding Cross-Attention layers [26] and a head that adapts the model from Masked Language Modeling to Causal Language Modeling. When using BERT-based models with the "EncoderDecoderModel" class, the weights of the added layers and modules are initialized from scratch.

We use more than one model in our experiments for two reasons. First, we want to analyze how generalizable the models are to receive prompt modifications and comprehend them as instructions, regardless of their architecture. Second, we want to validate the prompting multi-task learning framework in more than one architecture, analyzing how easily it can be adapted to other scenarios.

Regarding overall training configurations, we used the following strategy: all models were trained for 10 epochs with an early stop of two epochs with no improvement in validation loss, batch sizes of 12 for CodeT5 and 10 for both BERT-based models, tokenization max limit of 128 tokens, the learning rate of 2^{-5} with Adam [39] as the optimizer and scheduler to reduce the learning rate at each epoch. Some hyperparameters were set based on hardware or time limitations, while others on a small round of tests performed during the creation of the baseline models. We use the same Cross-Entropy Loss for all tasks and update only the weights involved in the forward process.

Unlike usual dataset configurations, we followed the same training procedure than CodeT5 [85]. Each epoch contains a limited number of iterations, and each iteration consists of a batch of samples from the same task, extracted from the task's respective dataset in random order. The number of iterations is configured when starting an experiment, and we used $100,000$ for all executed experiments. The training instances are sampled from the training split while training the model. For validation, we sample from the validation splits. At the end of training, we use the test split to report our results in data the model has never seen.

In this work, to serve as baselines, we train all three models for each one of the evaluation tasks using a single-task approach, using all available data for that given task. While for UTCG one model is enough to generate the data for all evaluation metrics, in NLCS we must train one model for each programming language. For instance, to obtain the CodeT5 baseline, we trained one model for UTCG and six models for NLCS, one for each programming language. We use the available pre-trained weights as the training starting points and then directly fine-tune them into the target tasks, extracting the required metrics. We achieve a fairer scenario by training the models in our computational environment since the same computational constraints will be equally applied to all models.

For both multi-task and single-task training procedures, we executed steps of hyper-parameter tuning. Our objective was to adjust our scenarios to a more realistic situation, where we would look for the best model that could solve a specific problem.

For our implementation, we relied on the PyTorch framework [55], which served as the backbone of our project. We also leveraged PyTorch Lightning [24] to train and evaluate all of our models, which streamlined our workflow and improved efficiency. To conduct our experiments, we utilized as hardware two Nvidia GeForce GTX 1080 Ti graphics cards, an AMD Threadripper with 32 cores, and 126GB of RAM.

6. RESULTS

6.1 Baseline Models

We trained baseline models for each task as explained in Section 5. These models were trained using a single-task modality and evaluated using the exclusive task dataset. We present the results for the baseline models in Table 6.1 for UTCG and Table 6.2 for NLCS. Additionally, we tested the attachment of a prefix to the prompt to inform the model which task to perform in UTCG. However, we only tried this approach in UTCG due to the high number of models required for NLCS.

Table 6.1: Baseline results for UTCG, with and without prefix.

Model	BLEU	WBLEU	ASTM	DFM	CodeBLEU
CodeBERT	7.47	8.54	34.62	32.66	20.82
GraphCodeBERT	7.26	8.19	34.54	**33.44**	20.85
CodeT5	8.35	9.20	36.09	31.89	21.38
CodeBERT+Prefix	8.36	9.24	35.40	32.45	21.36
GraphCodeBERT+Prefix	6.27	7.05	35.41	32.76	20.37
CodeT5+Prefix	**8.66**	**9.54**	**36.16**	31.86	**21.55**

Table 6.2: Result for baselines in all programming languages from CodeSearch-Net.

Model	go	java	javascript	php	python	ruby	overall
CodeBERT	0.894	0.781	**0.747**	**0.856**	0.695	**0.742**	**0.785**
GraphCodeBERT	**0.909**	**0.786**	0.570	0.847	**0.750**	0.704	0.763
CodeT5	0.865	0.772	0.710	0.807	0.720	0.646	0.753

In the following sections, note that we will refer to the best baselines as the one with highest score achieved for each metric, regardless of the model that achieved it.

6.2 RQ-1 Models

Our first research question focused on discovering whether a model trained with a prompt MTL procedure can achieve comparable results or surpass its single-task counterpart. As mentioned in Section 5, we trained three different architectures using all 8 tasks.

In addition to our preliminary experiments, we conducted two configurations to explore whether they could enhance model performance. The first configuration involved incorporating a language prefix along with the task prefix, which we used in all cases. We aimed to determine whether this could improve the model's understanding by indicating the task and language used. The second configuration assigned equal importance to all tasks using the same probability for all datasets. Note that this does not mean all datasets had an equal number of instances in the training dataset, just that the weights were the same for all tasks when sampling instances. Table 6.3 and Table 6.4 present the results of our experiments for UTCG and NLCS, respectively. When comparing a model with a baseline in answer to our second research question, we use the baseline model that matches the architecture and achieves higher results in overall MRR and CodeBLEU. We also refer to the best score obtained for each metric as the **best baseline**, regardless of the model it came from.

Table 6.3: Results for multi-task training in UTCG with all three models. SP stands for "same probabilities" and LP stands for "language prefix". In bold, are the best results for each metric, and underlined are the second-best results.

Multi-task Model	BLEU	WBLEU	ASTM	DFM	CodeBLEU
Best Baselines	**8.66**	**9.54**	**36.16**	**33.44**	**21.55**
CodeBERT	5.32	6.06	31.21	30.38	18.24
GraphCodeBERT	4.88	5.72	31.03	<u>32.72</u>	18.59
CodeT5	<u>6.59</u>	<u>7.73</u>	33.38	29.15	<u>19.11</u>
CodeBERT+SP	4.30	5.06	29.11	30.76	17.30
GraphCodeBERT+SP	4.55	5.09	30.15	27.86	16.91
CodeT5+SP	6.48	7.20	32.81	28.56	18.76
CodeBERT+LP	4.98	5.87	30.10	32.22	18.29
GraphCodeBERT+LP	4.93	5.70	31.35	31.66	18.41
CodeT5+LP	<u>6.59</u>	7.31	<u>33.40</u>	28.97	19.06
CodeBERT+SP+LP	4.51	5.10	28.77	29.07	16.86
GraphCodeBERT+SP+LP	4.46	5.00	29.84	28.72	17.01
CodeT5+SP+LP	6.47	7.18	32.76	28.60	18.75

In UTCG, none of the multi-task models achieved better results than our best baselines or their direct single-task counterparts. All the models significantly reduced their capabilities due to prompt MTL across all configurations we

Table 6.4: Results for multi-task training in NLCS. SP stands for "same probabilities" and LP stands for "language prefix". In bold, are the best results for each metric, and underlined are the second-best results.

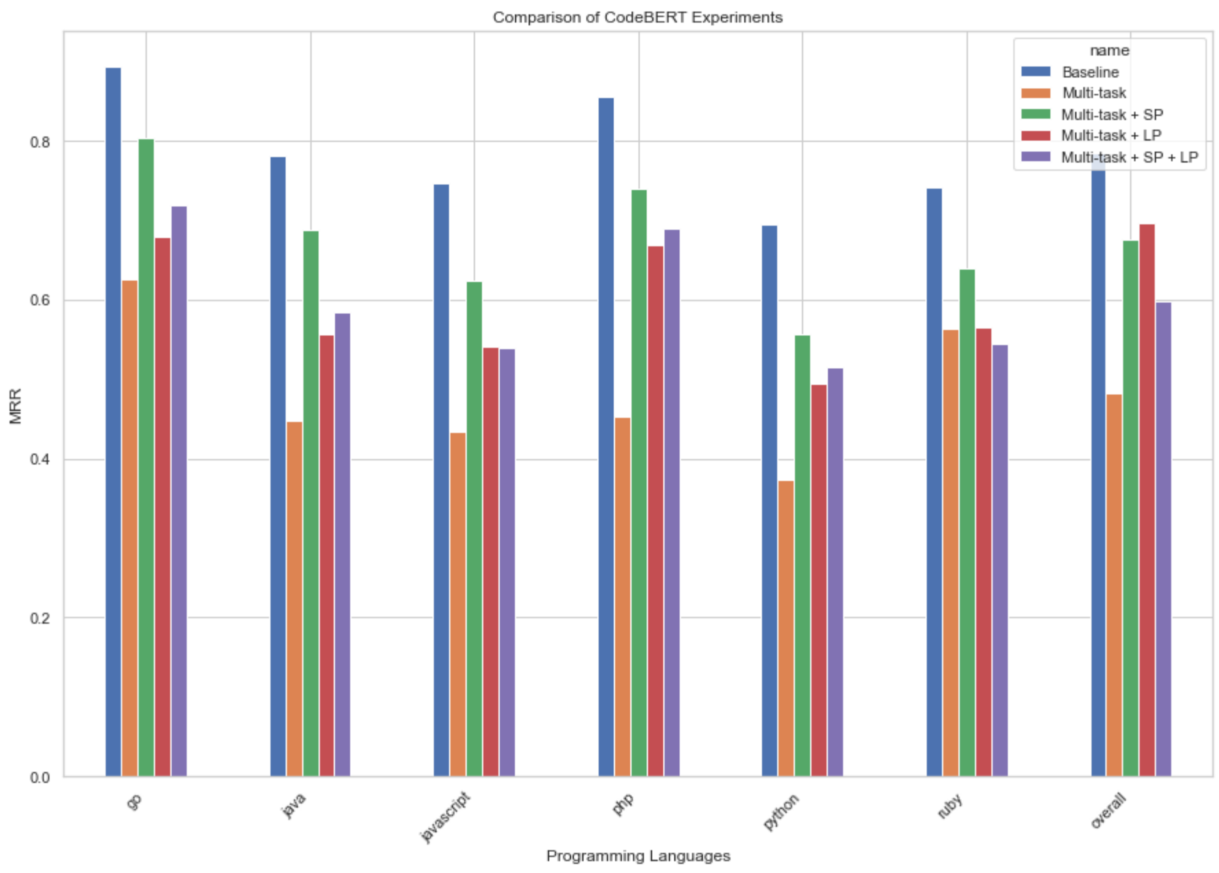
Model	go	java	javascript	php	python	ruby	overall
Best Baselines	**0.909**	**0.786**	**0.747**	**0.856**	**0.750**	**0.742**	**0.785**
CodeBERT	0.626	0.447	0.433	0.453	0.374	0.563	0.482
GraphCodeBERT	0.162	0.294	0.328	0.278	0.206	0.417	0.280
CodeT5	0.784	0.686	0.651	0.739	0.604	0.714	0.696
CodeBERT+SP	0.803	0.687	0.623	0.740	0.557	0.640	0.675
GraphCodeBERT+SP	0.034	0.046	0.103	0.058	0.015	0.076	0.055
CodeT5+SP	0.809	0.728	0.696	0.766	0.635	0.738	0.728
CodeBERT+LP	0.679	0.556	0.541	0.669	0.494	0.565	0.697
GraphCodeBERT+LP	0.240	0.028	0.024	0.016	0.008	0.066	0.063
CodeT5+LP	0.789	0.685	0.650	0.736	0.612	0.715	0.697
CodeBERT+SP+LP	0.719	0.584	0.539	0.690	0.515	0.544	0.598
GraphCodeBERT+SP+LP	0.103	0.047	0.047	0.051	0.186	0.062	0.082
CodeT5+SP+LP	0.809	0.708	0.694	0.748	0.631	0.738	0.721

tested. CodeBERT was the model that suffered the most, with a 21.07% reduction in CodeBLEU in the worst-performing model, while CodeT5 fared better with only an 11.32% reduction in the best model. This difference between CodeT5 and the other models in UTCG, a task that requires both an encoder and decoder, was already expected due to the model's architecture and natural ability to comprehend prompt modifications.

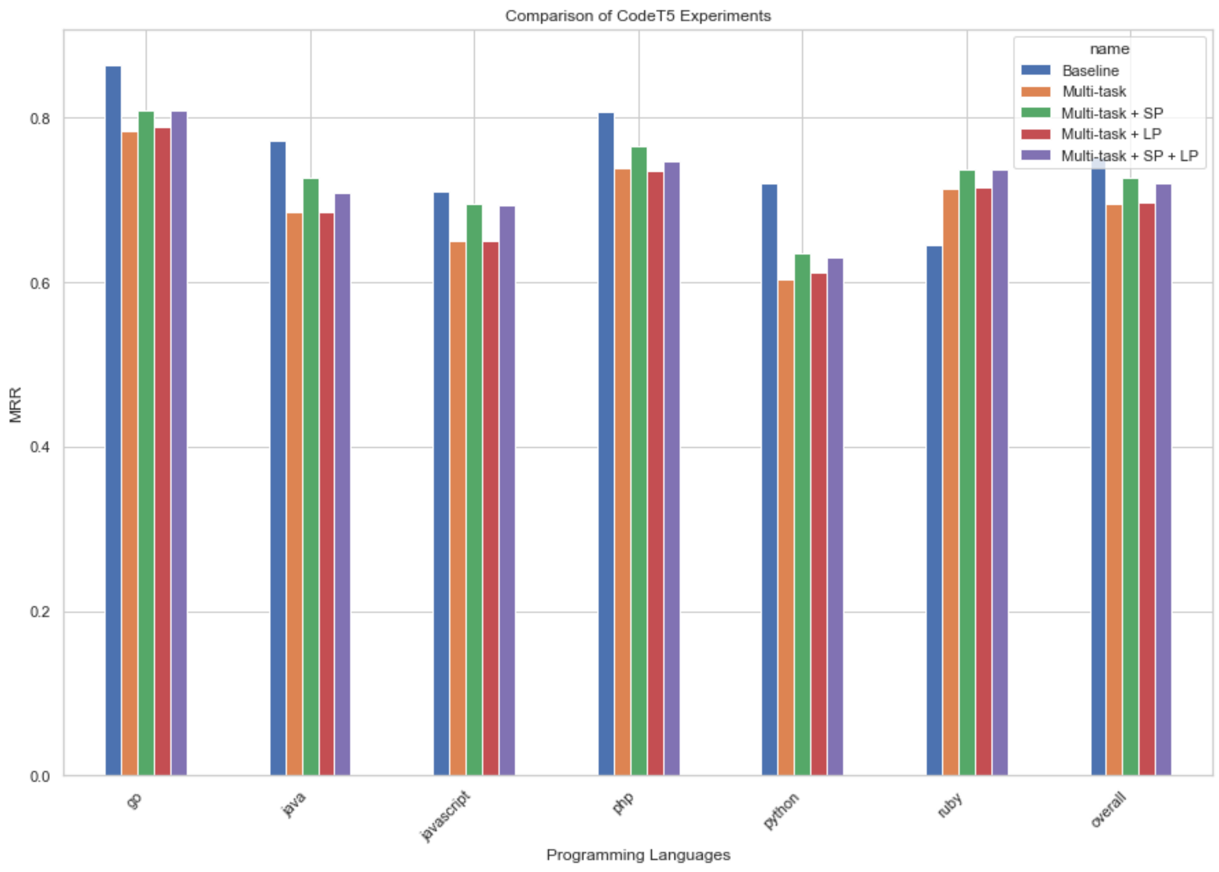
Upon closer examination of each term of CodeBLEU, it becomes apparent that no metric suffered a disproportionate loss in performance when compared to the others. This finding suggests that tvhe models do not selectively unlearn only code or natural language information. Another important outcome is that none of our tested configurations outperformed the base multi-task model, which solely employs the task prefix and probabilities based on dataset size.

Something relevant is that BERT-like models not trained with any prompt-specific technique learned how to extract knowledge in a prompt MTL setup. That finding is interesting because it demonstrates that MTL by prompt is a viable approach even for models not trained in that setup.

In NLCS, while most multi-task models did not perform better than their single-task counterparts or the baselines, some exciting observations should be made. Although none of the best baselines was surpassed by multi-task models, specific combinations of architectures and languages achieved better MRR scores than their direct counterparts. Notably, with CodeT5, using the same probabilities for all tasks in the training dataset resulted in multi-task models



(a) CodeBERT



(b) CodeT5

Figure 6.1: Comparison of multi-task learning methods in NLCS.

achieving MRR scores close to their single-task counterparts. Furthermore, all multi-task CodeT5 models for Ruby surpassed the single-task MRR score, as seen in Figure 6.1b. These results are significant because training only one multi-task model achieved closer results to training six specialists for NLCS.

The same cannot be said with CodeBERT, in which the single-task models invariably maintained the best MRR scores, as seen in Figure 6.1a. However, for some languages the results with CodeBERT (specifically with the same probabilities) got close to CodeT5, which again indicates that, under some configurations, models that are not naturally adapted for prompt MTL can learn how to extract useful information during fine-tuning. For the BERT-Like models, even after adding and training a new module, the model can still keep its capability of producing good feature representations.

In the case of Ruby, where all multi-task CodeT5 models were better than the baseline, we see a point where MTL shines by countering the small number of instances by extracting knowledge from other sources. A curious aspect is that BERT-like models could not leverage the numerous datasets to improve over ruby MRR scores. A possible explanation is the natural ability of CodeT5 to work with MTL, inherited from T5, which is a clear advantage against BERT-based models, in which decoders were not present in the original pre-training.

GraphCodeBERT was the only model among the tested ones that exhibited a significant forgetting problem. While in UTCG the model performance decreased similarly to the other models, in NLCS GraphCodeBERT lost almost all of its ability to perform the task. This is surprising, considering that GraphCodeBERT has the same architecture as CodeBERT but with a different pre-training strategy. One possible explanation for this behavior is that the pre-training tasks that GraphCodeBERT was trained on may have limited its ability to generalize in a multi-task setting. Unlike CodeBERT, which uses more generalist tasks, GraphCodeBERT leverages dataflow information to aid in code comprehension, which may restrict the model when creating the dense representations in the encoder. Constraining the model adaptability to specific tasks may have hindered its ability to adapt to multi-task scenarios and other training approaches.

It is important to remember that such a downgrade happened while using prompting MTL and attaching a decoder block to the model, which does not mean that the model cannot support other multi-task approaches.

In conclusion, after comparing the results obtained from both tasks and considering the most stable models, we have observed that it is possible to achieve results comparable to or even better than the single-task counterparts for NLCS. However, none of the multi-task models managed to surpass the baseline results. Despite this, multi-task CodeT5 stands out as a competitive model that can execute seven tasks simultaneously and perform similarly to the baselines that use the same pre-trained model. For example, by using CodeT5 with the same probabilities, we can obtain a single model with a performance loss as little as 12.93% for UTCG and 3.32% for NLCS instead of training seven separate models. Therefore, we emphasize the importance of CodeT5 due to its ability to achieve results competitive to the baseline models.

6.3 RQ-2 Models

To address our second research question, we conducted experiments that involved removing specific tasks or combinations of tasks from our general training procedures. Given the reliability and consistency observed in our previous experiments, we mainly relied on CodeT5 [85] for our experimentation but extended our analysis to CodeBERT [25] where possible. Our experimental scenarios were designed to consider common relationships between tasks, such as sharing the same programming language, using the same part of the model, or utilizing the same dataset.

6.3.1 Removing One Task

In our first experiment, we aimed to analyze the impact of each task, thus isolating them by testing six multi-task scenarios where we removed only one helper task from the training procedure. For comparison, we used the base multi-task models trained and analyzed in the previous research question. The results of each training scenario are presented in Tables 6.5 and 6.6 for CodeT5 and CodeBERT, respectively.

Table 6.5: Concatenated results from experimenting with how each task would affect the training procedure using CodeT5 model. We did not highlight the best results from the baseline once our focus with this experiment is the task removals.

Removed Task	NLCS Metrics							UTCG Metrics				
	go	java	javascript	php	python	ruby	overall	BLEU	WBLEU	ASTM	DFM	CodeBLEU
CodeT5 Baseline	**0.865**	**0.772**	**0.710**	**0.807**	**0.720**	0.646	**0.753**	**8.66**	**9.54**	**36.16**	**31.86**	**21.55**
None (Base Multi-Task Model)	0.784	0.686	0.651	0.739	0.604	0.714	0.696	6.59	7.73	33.38	29.15	19.11
Clone	0.827	0.736	0.694	0.779	0.643	**0.735**	0.735	6.60	7.32	33.38	29.28	19.15
Defect Detection	0.804	0.685	0.653	0.725	0.600	0.716	0.697	6.60	7.32	33.41	29.24	19.14
Generation	0.773	0.693	0.657	0.732	0.614	0.715	0.697	6.55	7.26	33.34	29.14	19.07
Refinement	0.800	0.704	0.647	0.738	0.609	0.716	0.702	6.58	7.29	33.40	29.18	19.11
Summarization	0.697	0.662	0.638	0.685	0.568	0.695	0.657	6.67	7.39	33.49	29.19	19.19
Translation	0.764	0.700	0.665	0.744	0.609	0.714	0.699	6.59	7.30	33.39	29.21	19.12

When analyzing the results obtained with CodeT5, we observe that the influence of each task during training is distinct. When we remove the “Clone” dataset, for example, it causes an increase in every NLCS metric. This effect is intriguing because the “Clone” dataset makes up a significant portion of the training dataset (25.59%), and its absence suggests that it could negatively impact the training. On the other hand, “Code Summarization” appears to be an essential task for multi-task training with the CodeT5 model, and its removal leads to lower performance in every NLCS metric. However, that same relevant task for NLCS in CodeT5 has the opposite effect on UTCG, where its absence improves the result of every term in CodeBLEU. This situation is counter-intuitive and illustrates the complexity of training multi-task models. For example, a task that works with sequence-to-sequence modeling is responsible for lowering the model capability in our evaluation task, which only uses the encoder, while simultaneously improving the results of NLCS.

Suppose we consider the model obtained by excluding “Clone” from the training procedure. In that case, we can diminish the difference between our baseline and a multi-task model, achieving an even higher overall MRR and CodeBLEU. That result exemplifies that, by finding the proper configuration, we can train a single multi-task model that is competitive with a group of specialist models. Also, we tested only excluding a single task at a time. Based on our results, where the absence of a task in most cases improves model performance, we hypothesize that some specific combination would achieve a potential optimal point for the problem. Unfortunately, in terms of available time for experiments, it was impractical for us to test every possible combination.

Table 6.6: Concatenated results from experimenting with how each task would affect the training procedure using CodeBERT model. We did not highlight the best results from the baseline once our focus with this experiment is the task removals.

Removed Task	NLCS Metrics							UTCG Metrics				
	go	java	javascript	php	python	ruby	overall	BLEU	WBLEU	ASTM	DFM	CodeBLEU
CodeBERT Baseline	**0.894**	**0.781**	**0.747**	**0.856**	**0.695**	**0.742**	**0.785**	**8.36**	**9.24**	**35.40**	**32.45**	**21.36**
None (Base Model)	0.626	0.447	0.433	0.453	0.374	0.563	0.482	5.32	6.06	31.21	30.38	18.24
Clone	0.815	0.742	0.648	0.785	0.629	0.681	0.716	4.78	5.47	31.17	30.82	18.06
Defect Detection	0.028	0.018	0.033	0.029	0.016	0.053	0.029	2.21	2.45	26.15	28.38	14.77
Generation	0.160	0.131	0.154	0.201	0.202	0.173	0.170	4.86	5.57	30.97	30.93	18.08
Refinement	0.768	0.709	0.650	0.760	0.579	0.664	0.688	5.39	6.11	31.38	30.49	18.34
Summarization	0.565	0.510	0.502	0.681	0.449	0.553	0.543	4.82	5.48	31.44	30.75	18.12
Translation	0.642	0.659	0.571	0.697	0.599	0.691	0.643	4.82	5.48	31.44	30.72	18.11

Upon comparing how both models react to the removal of the tasks, it becomes clear how each model captures knowledge differently. In fact, “Clone” removal outperforms the base multi-task model and all other task removals in most MRR values for the CodeBERT model. This suggests that the “Clone” task harms the NLCS task, similarly to what we saw in CodeT5. But, while for CodeT5 no task was essential to the multi-task training process (causing a strong decrease in performance), for CodeBERT the scenario was significantly different. The removal of “Defect Detection” had great importance during CodeBERT training, and its absence leads the model to inferior performance in both tasks, especially in NLCS. Such a model’s need to rely on “Defect Detection” to NLCS is also counter-intuitive if we consider that it is the second less-frequent task in the dataset and uses a programming language (C) that does not appear for any other task, thus being present in only a few instances. Also, “Generation” holds a high significance for the model performance in NLCS, but the same does not hold to UTCG. It seems that, for CodeBERT, some tasks are required to help the model in keeping the knowledge needed to create the dense representations used for NLCS.

6.3.2 Only Seq2Seq Tasks

Another hypothesis of ours was to use only tasks that require a similar level of behavior from the model. Since NLCS uses only the encoder to generate the dense representations used to align NL and PL snippets, optimizing only one part of the model for some instances can bring the results down on UTCG. With that in mind, we ran an experiment removing NLCS from the multi-task training procedure since all other tasks require the decoder portion of the model.

Another hypothesis we raised was that the model could perform better by using only tasks where the input and output were written in PLs. That way, we emphasize the model to focus on understanding source code and to not learn how to build representations for NL inputs or outputs. The results for both tests are shown in Table 6.7.

Table 6.7: Testing only tasks using both parts of the model (sequence to sequence tasks). We trained Seq2Seq with all tasks except NLCS, and Code2Code we trained with “Translation“, “Refinement“, and UTCG.

Experiment	BLEU	WBLEU	ASTM	DFM	CodeBLEU
Baseline	**8.66**	**9.54**	**36.16**	**31.86**	**21.55**
Multi-Task CodeT5	6.60	7.32	33.38	29.28	19.1
Seq2Seq	6.80	7.53	33.91	29.41	19.41
Code2Code	<u>7.72</u>	<u>7.99</u>	<u>34.56</u>	<u>29.93</u>	<u>19.93</u>

As observed, the focus on only using sequence modeling tasks can increase the performance of CodeT5 multi-task models for UTCG. The results do not outperform the baseline models but are better than the base multi-task model, although in this case we sacrifice the range of tasks to increase quality. We also analyze that training using only code-exclusive tasks provides better results than considering all sequence-to-sequence tasks. A possibility for that improvement may be because for some tasks, such as “Clone”, which are quite frequent in the dataset, the output is “True” or “False”, which is very different from the expected output for UTCG, a unitary test.

Among all experiments, including measuring UTCG, our Code2Code multi-task model achieved the highest scores, even though it could not reach the baseline model. We see by that experiment that a more limited group of tasks, with a more explicit focus, can indeed improve model performance.

6.3.3 Language-Exclusive Tasks

Another hypothesis we raised was reducing tasks according to the programming language. We tested using two languages shared by more than two tasks (which happens with all languages in the CodeSearchNet dataset): Java and Python. In the case of Python, we trained the model with “Code Search”, “Code Summarization”, and UTCG, while for Java, we trained with “Code Search”, “Code Summarization”, “Code Generation”, “Code Translation” and “Code Refinement”. Results are presented in Table 6.8

Table 6.8: Experiments training models only with tasks that share the same PL. While with Java we could only evaluate on NLCS, for Python we could obtain metrics for both our evaluation tasks.

Experiment	NLCS Metrics		UTCG Metrics				
	java	python	BLEU	WBLEU	ASTM	DFM	CodeBLEU
Baseline	**0.772**	**0.720**	**8.66**	**9.54**	**36.16**	**31.86**	**21.55**
Multi-task CodeT5	0.686	0.604	6.59	7.73	33.38	29.15	19.11
Java Tasks	<u>0.734</u>	-	-	-	-	-	-
Python Tasks	-	<u>0.654</u>	<u>7.01</u>	<u>7.76</u>	<u>34.20</u>	<u>29.66</u>	<u>19.66</u>

By grouping the tasks by PL, we can see that both models are better than the multi-task base model but still cannot outperform the baseline. In these cases, the number of tasks our model can handle is inferior to the base multi-task, but that reduction provides notable gains. For both languages, we achieved better MRR than the base multi-task CodeT5, but note that they have not become the best multi-task results.

6.3.4 CodeSearchNet-Exclusive Tasks

One last hypothesis we wanted to validate was to analyze the usage of different tasks sharing the same dataset. In that way, we can evaluate whether the model can perform better in both tasks by extracting and sharing different information obtained from the same source. Results are introduced in Figure 6.2 Our training set contained all instances available in the datasets for those models.

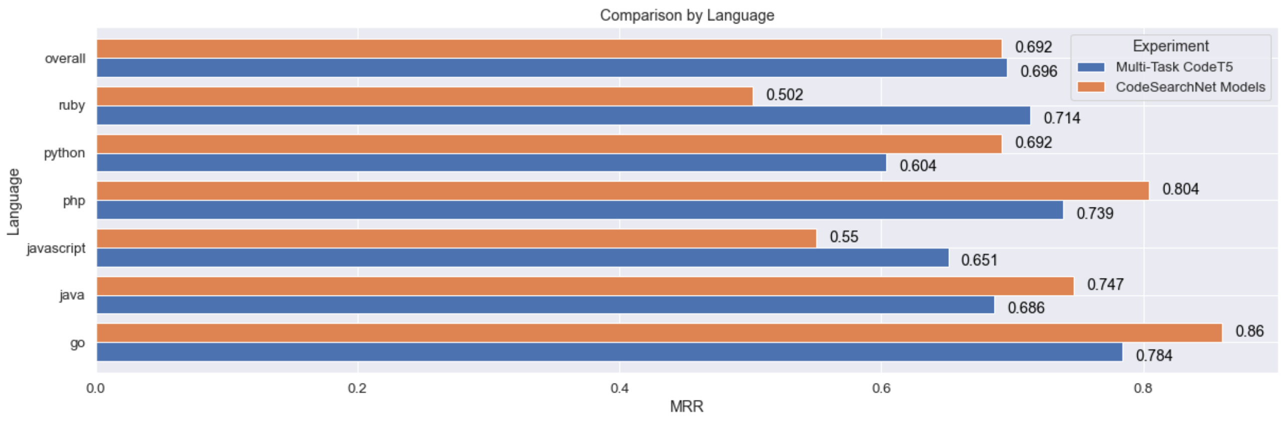


Figure 6.2: Comparison of our base multi-task CodeT5 and multi-task models trained solely using CodeSearchNet (Code Search and Code Summarization tasks).

In terms of performance, the CodeSearchNet models showed better results for Go, Java, PHP, and Python, while Multi-Task CodeT5 performed better in Javascript and Ruby. However, when considering the overall performance, CodeSearchNet Models achieved a lower score than Multi-Task CodeT5. Such behavior tells us that, except for Ruby and Javascript, the presence of more tasks tends to harm the model capability of learning good feature representations.

For the four programming languages with larger datasets, the results using only “CodeSearch” and “Code Summarization” were the best among all multi-task models from previous experiments, indicating that extracting information from the same source may be a better option than just aggregating more tasks from distinct sources.

By looking at such a scenario, it becomes even more apparent that combining tasks and datasets is not an easy thing. For Ruby and Javascript, adding knowledge from other sources helped the model improve its performance. This effect may be due to the small dataset size for those languages, which causes the model to benefit from other data sources to get better results. However, improving the dataset size is not a guaranteed solution, bearing in mind that Go is the second language with higher improvement (9.69%) and has a smaller dataset than PHP, which is almost double the size.

6.4 RQ-3 Models

To answer our last research question, we trained three language models with prompt multi-task learning, leaving the evaluation tasks out of the process. Our objective was to compare them with the base pre-trained models, as they are available in the Hugging Face Hub, by evaluating them without ever seeing the evaluation tasks. With this zero-shot modality, we expect to understand whether multi-task learning can help the models acquire knowledge about source code and generalize to completely new tasks within the same context. We show the results in Table 6.9.

Table 6.9: Results for the experiments comparing the base pre-trained models and the multi-task models in zero-shot learning. We see that while CodeBERT and CodeT5 benefit from the multi-task training, GraphCodeBERT suffers a downgrade. The evaluation tasks were not used during training.

Model	NLCS Metrics							UTCG Metrics				
	go	java	javascript	php	python	ruby	overall	BLEU	WBLEU	ASTM	DFM	CodeBLEU
Base CodeBERT	0.001	0.001	0.003	0.0008	0.001	0.007	0.0023	0	0	3.76	6.23	2.50
Base GraphCodeBERT	0.050	0.044	0.065	0.024	0.037	0.115	0.055	0	0	6.25	7.43	3.43
Base CodeT5	0.001	0.001	0.003	0.0006	0.0004	0.005	0.0018	0	0	3.19	11.84	3.76
Multi-Task Pre-Trained CodeBERT	0.002	0.001	0.005	0.004	0.0009	0.007	0.003	0.01	0.02	6.25	7.41	3.42
Multi-Task Pre-Trained GraphCodeBERT	0.002	0.009	0.009	0.005	0.003	0.012	0.006	0	0	3.76	6.23	2.50
Multi-Task Pre-Trained CodeT5	0.003	0.003	0.004	0.001	0.001	0.011	0.003	0.27	0.36	8.96	14.08	5.92

As we can see, for CodeT5 and CodeBERT, the multi-task models achieved a better score for most metrics, both for NLCS and UTCG. The improvements indicate that the models learn more about tasks not ever seen by acquiring knowledge from multiple sources. That result is similar to what is explored by Google with PaLM [20], where increasing the data used to learn the model also increases its generalization capacity over unseen tasks. The increase in NLCS shows that the models learn how to build better representations by the encoder. In contrast, the improvement in CodeBLEU terms shows that the models start learning how to formulate and interpret code snippets.

Again, GraphCodeBERT is the only model that shows different behavior. It starts as the base model with more prior knowledge about code, generating better dense representations, thus achieving the best overall score for NLCS among all experiments. The results for UTCG calls our attention, mainly because the decoder is a module not involved in the original model and which we adapted, causing some of the weights to be randomized in the base model. Nevertheless, it achieves relatively good results for ASTM and DFM. We believe it happens because its pre-training tasks mostly rely on understanding data flow, which is highly relevant for both terms.

6.5 Dataset Size Ablation

After analyzing GraphCodeBERT's performance in the first research question, we were puzzled by its significant drop in results despite its similarity to CodeBERT. This led us to conduct a series of experiments to analyze the point in the fine-tuning process where the model began to lose its ability to generalize for NLCS while also comparing its performance to that of UTCG. Our approach involved modifying the number of iterations in the same experiment configuration used to train our base multi-task models, thereby enabling us to determine how much fine-tuning was necessary for the model to perform poorly in NLCS. We also conducted the same experiments using CodeT5 to compare both models in the same process.

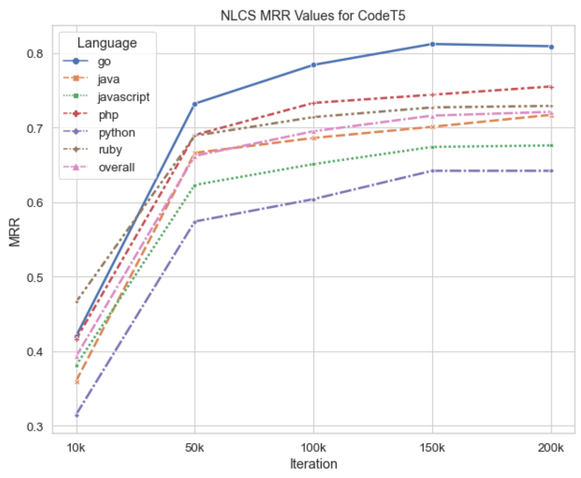


Figure 6.3: CodeT5 MRR for each PL in CodeSearchNet as we improve the number of training iterations.

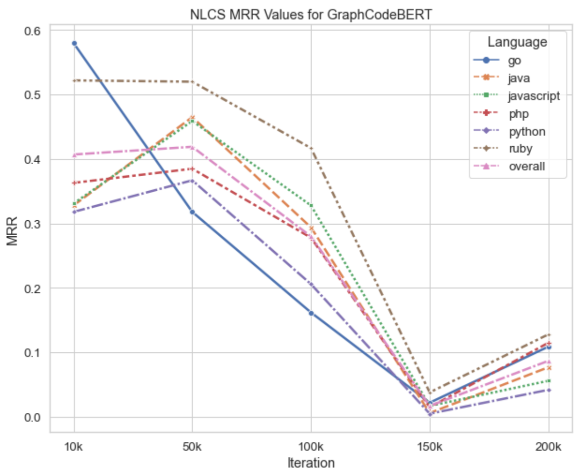


Figure 6.4: GraphCodeBERT MRR for each PL in CodeSearchNet as we improve the number of training iterations.

As we can see by comparing Figures 6.3 and 6.4, each model has a very distinct behavior as we increase the number of iterations in our experiment (analog to the dataset size) in the NLCS task. While CodeT5 almost always benefits from increasing the dataset size (and when it does not, the drop in MRR is not high), GraphCodeBERT has pretty inconsistent behavior. GraphCodeBERT

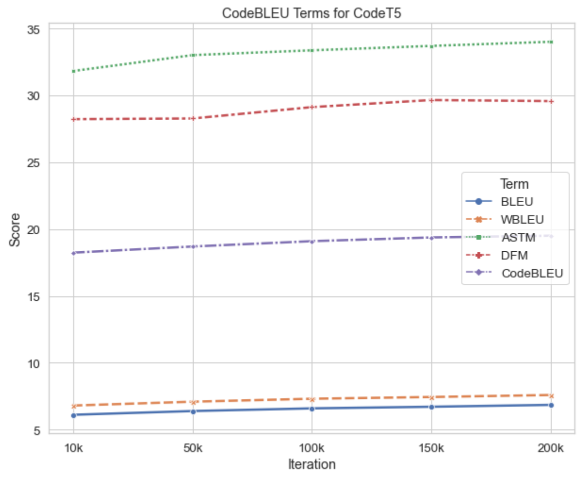


Figure 6.5: CodeT5 scores for each term in the CodeBLEU formula as we improve the number of training iterations.

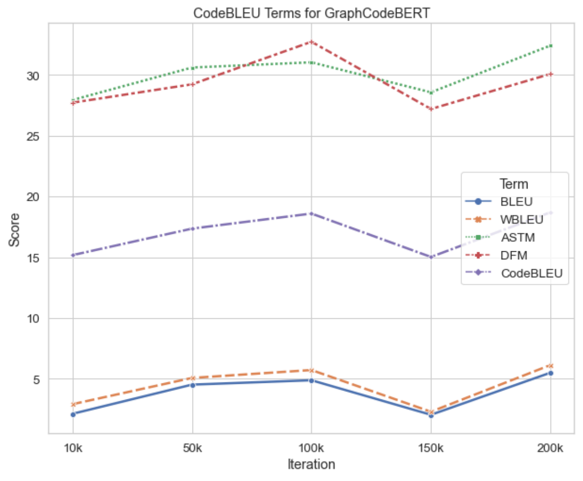


Figure 6.6: GraphCodeBERT scores for each term in the CodeBLEU formula as we improve the number of training iterations.

achieves the highest MRR values when training for less than $50,000$ iterations; from that point forward, the scores mostly decrease. We can see a trace of improvement in the final experiment, but we could not do further experimentation due to the cost related to increasing even more the dataset size.

Regarding UTCG, the scenario is more stable for GrahpCodeBERT and remains unchanged for CodeT5, as demonstrated in Figures 6.5 and 6.6. Again, CodeT5 tends to improve with more data, while GraphCodeBERT has one decreasing point at $150,000$ instances in the training dataset. For UTCG, the scenario is less inconsistent than for NLCS, not reaching the point of losing almost all capability of performing the task. In this aspect, our research reaches one of its limitations since we do not analyze the models behavior in other tasks to understand whether this phenomenon happens only in NLCS or also in other tasks.

6.6 Discussion

Previous experiments show that prompt multi-task learning is not a simple strategy to optimize results. While answering our first research question, we saw that no multi-task model could outperform the baselines, and only in one PL the multi-task models were better than the singular-task counterparts. Even with the performance degradation, we trained only one model for seven tasks, and in cases such as CodeT5+SP, we got very close to the CodeT5 baselines.

A fact that drew our attention was that while CodeBERT could handle prompt MTL, even when not pre-trained similarly, GraphCodeBERT with the same architecture could not achieve the same performance. That fact indicates that the tasks used during pre-training have significant relevance for posterior fine-tuning capabilities.

When comparing the tasks and their relationships along the training procedure, we saw in practice what the literature had quickly mentioned [72]. We observed that “Clone” has a negative impact in most metrics and for both models, and its absence can lead to better performance for both models we tested. But the relevance of “Defect Detection” and “Generation” for CodeBERT was surprising, mainly because they do not constitute a large proportion of the total dataset. Also, we observe that, for both models, “Code Summarization” is a relevant task for extracting good representations used in NLCS.

Upon analyzing the results obtained for validating all research questions, we can state that multi-task learning can increase the capabilities of the models to understand more about source code-related tasks. Unfortunately, such understanding seems to be affected by other aspects, such as the pre-training tasks used when adapting the models for MTL. In such a sizeable experimental landscape, we observe that some configurations can help the models to achieve better results. Good examples were the usage of the same probabilities to have a more balanced dataset, which leverages the results of CodeT5 in multi-task training, and the removal of “Clone”, which also substantially improved model performance.

7. CONCLUSION

As software engineering becomes increasingly complex, artificial intelligence (AI) is becoming an essential tool for automating tasks and improving the quality of software products. DL models are the main piece of that improvement, using robust neural networks to build solutions that address complex problems [37, 8]. With the rise of pre-trained language models, numerous tasks in NLP achieved a new state of the art, leading to the adaptation of that strategy to other modalities.

One of the modalities these transformer language models boosted was big code, where many software development-related tasks have started to be supported by applications that leverage these models [25, 85, 48]. These models are trained with textual and code data, helping them to extract valuable knowledge from both sources. A problem emerges when these models become larger and larger, and training, storing, and serving them become complex issues.

One approach to solve resource allocation problems and also attempt at improving generalization is MTL. By training a model with more than one task, we can extract different knowledge from multiple sources and possibly achieve better performance in the focused task [14, 93]. Among the many alternatives to implement MTL, using prompt engineering to induce behaviors becomes a reliable method for language models [61, 12, 85].

Leveraging the recent ability of language models to understand prompt entries, we explored multi-task prompting methods for source-code-related tasks using three cutting-edge models, CodeT5, CodeBERT, and GraphCodeBERT, which have yielded promising results. We have demonstrated that MTL can achieve results comparable to, if not better, single-task baselines. Furthermore, besides reaching competitive results, we also reduce the cost by training only one model instead of seven, demonstrating the benefits of MTL.

This research has also uncovered evidence that the pre-training task may lead the model to certain limitations during fine-tuning. These limitations are particularly apparent in the case of GraphCodeBERT. By better understanding the relationship and interaction between tasks, as well as through careful configuration tuning during training, we can address these limitations and improve the performance of multi-task models in the future.

When exploring the tasks and datasets relationship, we saw that a more restricted or focused group of tasks could be a better idea, where we see a trade-off between the number of tasks and performance. Unfortunately, as observed when comparing CodeT5 and CodeBERT, each model reacts differently to the multi-task training, creating many possibilities to be explored in order to find the better combination. Also, we showed that for some pre-trained models MTL could increase the capabilities of zero-shot inference, even if by small margins.

The research landscape is very large. MTL provides a lot of possibilities and methods that can be combined with the several pre-trained language models that are open-sourced. We could not cover all those possibilities for this work, but we uncovered some interesting behaviours and statements regarding prompting MTL and some recent state-of-the-art models.

Overall, this work provides good insight into the use of AI in software engineering. Automating and optimizing software development processes have become increasingly important as the demand for software grows. With its ability to learn from multiple tasks simultaneously, MTL represents an area that approximates the learning method to how humans actually learn. By continuing to explore and refine this approach, we believe we can develop more powerful and effective AI systems to support software engineering and other complex domains.

7.1 Limitations

Due to this work's vast research landscape, many limitations must be pointed out, which were responsible for reducing our scope. We enlist them next.

7.1.1 Evaluation tasks

In this work, we decided to evaluate the trained models in only two tasks, NLCS and UTCG, which are tasks that require a considerable understanding of source-code-specific traits. The main reason for that decision was the available time and resources to invest in implementing the code to evaluate our models in all available tasks. However, as done in other multi-task studies [61, 56, 85, 44], all tasks should be evaluated for a holistic and complete analysis. That interpretation is needed, especially due to the absence of a specific answer for how each task will influence each other. By analyzing only two tasks, we lose much of what we could learn about each models' generalization capability.

7.1.2 Evaluated models

The number of evaluated models is another limitation of this work. Due to scope constraints, we only evaluated three models: CodeT5, CodeBERT, and GraphCodeBERT. While those models have shown cutting-edge results in previous studies, they may only represent some possible models that could be evaluated for multi-task learning in software engineering.

Also, the hyper-parameters of the models were not extensively searched. Although we have explored some possible values for the hyper-parameters, many others could have been tested to improve the performance of the models. Furthermore, different combinations of hyper-parameters could lead to different results and conclusions. Therefore, future work should explore a more extensive hyperparameter search to determine the optimal configuration for the models in the evaluated tasks.

7.1.3 Multi-task learning methods

We tested only one of the multiple multi-task learning approaches, using prompt modifications to induce the model to learn all tasks. As described in Section 2, multiple ways to implement and train multi-task models are available, each with singular limitations and capabilities [19]. We recognize that those other multi-task learning methods could have further improved our research, but we could not test them due to time restraints. Therefore, we plan to expand future research to explore these methods and compare their effectiveness with the already tested multi-task prompting method.

7.1.4 Hardware and time limitations

Most transformer networks used in this work are expensive regarding computational resource cost. Given such limitations, our model configurations were directly affected, forcing us to reduce batch sizes or the max size in of the tokenizer. For instance, each epoch for our multi-task models took approximately 8 hours, one of the reasons that led us to use early stop. Also, such limitations prevented us to look forward to hyper-parameter tuning, which could be another approach to finding ideal multi-task models or better configuration setups.

Additionally, the limited computational resources for this work forced us to use smaller models and shorter training configurations, which may have impacted the final results. In future work, it would be interesting to explore the use of more powerful hardware to allow for larger models and extended training configurations, potentially leading to better performance in the tasks that were evaluated. Besides, other multi-task approaches such as adding extra layers for each task, require more memory to store the weights.

The hardware also implies time constraints that we had during the research. As mentioned, one epoch for a multi-task model takes about 8 hours, and complete training takes, on average, 40 hours (using early stop). Based on that factor, some of our explorations, especially for our research questions, could not be executed on time, leading us to focus on the most promising alternatives to explore the task relationships.

7.2 Future Work

This work begins an extensive study of multi-task learning methods for source-code tasks. As previously mentioned, the research landscape contains many variables that demand time, resources, and knowledge to be explored. We have already established some tasks and experiments as future steps of this research.

Studying different combinations of tasks for at least CodeT5 and CodeBERT, or using the same model but with other seeds, would give us an even better understanding of how each model behaves during the learning process. Our idea to study with different random seeds comes directly from the paper we published about models for Code Search [86], where we discovered that underspecification is a relevant problem for this task, leading to distinct behaviors when training with different seeds.

Another group of possible relations that we need to map or explore is how programming languages interact with each other during multi-task training. We plan to explore such relations, mainly due to the nature of the languages, where low-level languages like Java and C share more similarities than with Python or Javascript.

To fulfill one of the previously discussed limitations, we have plans to implement and evaluate the trained models in all tasks we consider during the training process. That way, we will obtain a complete and concrete overview of the capacities obtained (or lost) during multi-task training. Another limitation we want to fill is expanding the research landscape by including new models and multi-task learning approaches. Recently a large group of pre-trained language models focused on source code tasks has been released and studied [90], but they focus solely on language modeling capabilities.

Also, we tested only a single option of prompting method: to append specific words to the model's input. A new area of prompting engineering research is arising with recent language models [45]. Methods have been proposed and debated, especially with models such as GPT-3 [12], which can easily follow prompt instructions. A mechanism we already idealized to implement and test is using universal adversarial triggers [70]. These adversarial triggers are slight prompt modifications that can lead a model to specific outputs without any further architectural change or new training procedure. We hypothesize that since T5 models use small prompt indicators to induce a task, we could use a more intelligent approach to optimize performance in different tasks by discovering ideal triggers during the training procedure.

7.2.1 Published Work

During this Master's, we have had the opportunity to conduct research in our field of study and contribute to the scientific community by publishing and submitting papers. The following discussion will briefly overview these publications and submissions, highlighting their main contributions and findings.

International Joint Conference on Neural Networks (IJCNN 2022) (Qualis A2): We published the paper "COBE: A Natural Language Code Search Robustness Benchmark" that proposes a benchmark to evaluate Code-Search models regarding their robustness. The framework linearly adds noise to the model input and considers how such modifications change the output.

Brazilian Conference on Intelligent Systems (BRACIS 2022) (Qualis A4): We published the paper "Leveraging Textual Descriptions for House Price Valuation", which proves that textual data can significantly improve the quality of house price valuation. Also, we do a qualitative analysis to understand how textual information increases model performance.

7.2.2 Submitted Work

ACM Computing Surveys Special Issue on Trustworthy AI (2022): We submitted the paper "Debiasing Methods for Fairer Neural Models in Vision and Language Research: A Survey", a survey on fairness and bias in deep learning models. We reviewed metrics and methods to achieve fairer models, a relevant topic in modern data-based AI. The paper is also available on ArXiv[1].

[1]https://arxiv.org/abs/2211.05617

REFERENCES

[1] Agarap, A. F. “Deep learning using rectified linear units (relu)”, *arXiv preprint arXiv:1803.08375*, 2018.

[2] Ahmad, W. U.; Chakraborty, S.; Ray, B.; Chang, K.-W. “Unified pre-training for program understanding and generation”, *arXiv preprint arXiv:2103.06333*, 2021.

[3] Aho, A. V.; Lam, M. S.; Sethi, R.; Ullman, J. D. “Compilers: Principles, Techniques, and Tools (2nd Edition)”. Addison Wesley, 2006.

[4] Akbik, A.; Bergmann, T.; Blythe, D.; Rasul, K.; Schweter, S.; Vollgraf, R. “Flair: An easy-to-use framework for state-of-the-art nlp”. In: 2019 Conference of the North American Chapter of the Association for Computational Linguistics (Demonstrations), 2019, pp. 54–59.

[5] Allamanis, M. “The adverse effects of code duplication in machine learning models of code”. In: 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, 2019, pp. 143–153.

[6] Allamanis, M. “The adverse effects of code duplication in machine learning models of code”. In: 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, 2019, pp. 143–153.

[7] Allamanis, M.; Barr, E. T.; Devanbu, P.; Sutton, C. “A survey of machine learning for big code and naturalness”, *ACM Computing Surveys (CSUR)*, vol. 51–4, 2018, pp. 1–37.

[8] Alon, U.; Brody, S.; Levy, O.; Yahav, E. “code2seq: Generating sequences from structured representations of code”, *arXiv preprint arXiv:1808.01400*, 2018.

[9] Bahdanau, D.; Cho, K.; Bengio, Y. “Neural machine translation by jointly learning to align and translate”, *arXiv preprint arXiv:1409.0473*, 2014.

[10] Barron, J. T. “Continuously differentiable exponential linear units”, *arXiv preprint arXiv:1704.07483*, 2017.

[11] Bengio, Y. “Deep learning of representations for unsupervised and transfer learning”. In: ICML workshop on unsupervised and transfer learning, 2012, pp. 17–36.

[12] Brown, T. B.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; Agarwal, S.; Herbert-Voss, A.; Krueger, G.; Henighan, T.; Child, R.; Ramesh, A.; Ziegler, D. M.; Wu, J.; Winter, C.; Hesse, C.; Chen, M.; Sigler, E.; Litwin, M.; Gray, S.; Chess, B.; Clark, J.; Berner, C.; McCandlish, S.; Radford, A.; Sutskever, I.; Amodei, D. "Language models are few-shot learners", *arXiv preprint arXiv:2005.14165*, 2020.

[13] Canhoto, A. I.; Clear, F. "Artificial intelligence and machine learning as business tools: A framework for diagnosing value destruction potential", *Business Horizons*, vol. 63–2, 2020, pp. 183–193.

[14] Caruana, R. "Multitask learning", *Machine learning*, vol. 28–1, 1997, pp. 41–75.

[15] Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; de Oliveira Pinto, H. P.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; Ray, A.; Puri, R.; Krueger, G.; Petrov, M.; Khlaaf, H.; Sastry, G.; Mishkin, P.; Chan, B.; Gray, S.; Ryder, N.; Pavlov, M.; Power, A.; Kaiser, L.; Bavarian, M.; Winter, C.; Tillet, P.; Such, F. P.; Cummings, D.; Plappert, M.; Chantzis, F.; Barnes, E.; Herbert-Voss, A.; Guss, W. H.; Nichol, A.; Paino, A.; Tezak, N.; Tang, J.; Babuschkin, I.; Balaji, S.; Jain, S.; Saunders, W.; Hesse, C.; Carr, A. N.; Leike, J.; Achiam, J.; Misra, V.; Morikawa, E.; Radford, A.; Knight, M.; Brundage, M.; Murati, M.; Mayer, K.; Welinder, P.; McGrew, B.; Amodei, D.; McCandlish, S.; Sutskever, I.; Zaremba, W. "Evaluating large language models trained on code", *arXiv preprint arXiv:2107.03374*, 2021.

[16] Cho, K.; Van Merriënboer, B.; Bahdanau, D.; Bengio, Y. "On the properties of neural machine translation: Encoder-decoder approaches", *arXiv preprint arXiv:1409.1259*, 2014.

[17] Cho, K.; Van Merriënboer, B.; Gulcehre, C.; Bahdanau, D.; Bougares, F.; Schwenk, H.; Bengio, Y. "Learning phrase representations using rnn encoder-decoder for statistical machine translation", *arXiv preprint arXiv:1406.1078*, 2014.

[18] Clement, C. B.; Drain, D.; Timcheck, J.; Svyatkovskiy, A.; Sundaresan, N. "Pymt5: multi-mode translation of natural language and python code with transformers", *arXiv preprint arXiv:2010.03150*, 2020.

[19] Crawshaw, M. "Multi-task learning with deep neural networks: A survey", *arXiv preprint arXiv:2009.09796*, 2020.

[20] Dean, J. “Introducing pathways: Next-generation ai architecture”. [Online; accessed December-2022], Source: https://blog.google/technology/ai/introducing-pathways-next-generation-ai-architecture/.

[21] Devlin, J.; Chang, M.-W.; Lee, K.; Toutanova, K. “Bert: Pre-training of deep bidirectional transformers for language understanding”, *arXiv preprint arXiv:1810.04805*, 2018.

[22] Eisenstein, J. “Introduction to natural language processing”. MIT press, 2019.

[23] Epshtein, B.; Uliman, S. “Feature hierarchies for object classification”. In: Tenth IEEE International Conference on Computer Vision (ICCV’05), 2005, pp. 220–227 Vol. 1.

[24] Falcon, W.; et al.. “Pytorch lightning”, *GitHub. Note: https://github.com/PyTorchLightning/pytorch-lightning*, vol. 3, 2019, pp. 6.

[25] Feng, Z.; Guo, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, T.; Jiang, D.; et al.. “Codebert: A pre-trained model for programming and natural languages”, *arXiv preprint arXiv:2002.08155*, 2020.

[26] Gheini, M.; Ren, X.; May, J. “Cross-attention is all you need: Adapting pretrained Transformers for machine translation”. In: 2021 Conference on Empirical Methods in Natural Language Processing, 2021, pp. 1754–1765.

[27] Goodfellow, I. J.; Bengio, Y.; Courville, A. “Deep Learning”. MIT Press, 2016.

[28] Guo, D.; Ren, S.; Lu, S.; Feng, Z.; Tang, D.; Liu, S.; Zhou, L.; Duan, N.; Svyatkovskiy, A.; Fu, S.; et al.. “Graphcodebert: Pre-training code representations with data flow”, *arXiv preprint arXiv:2009.08366*, 2020.

[29] Harman, M. “Why source code analysis and manipulation will always be important”. In: 2010 10Th IEEE working conference on source code analysis and manipulation, 2010, pp. 7–19.

[30] He, K.; Gkioxari, G.; Dollár, P.; Girshick, R. “Mask r-cnn”. In: IEEE international conference on computer vision, 2017, pp. 2961–2969.

[31] He, K.; Zhang, X.; Ren, S.; Sun, J. “Delving deep into rectifiers: Surpassing human-level performance on imagenet classification”. In: 2015 IEEE International Conference on Computer Vision (ICCV), 2015, pp. 1026–1034.

[32] He, K.; Zhang, X.; Ren, S.; Sun, J. “Deep residual learning for image recognition”. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016, pp. 770–778.

[33] Hendrycks, D.; Gimpel, K. “Bridging nonlinearities and stochastic regularizers with gaussian error linear units”, *arXiv preprint arXiv:1606.08415*, 2016.

[34] Hochreiter, S. “The vanishing gradient problem during learning recurrent neural nets and problem solutions”, *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, vol. 6–2, apr 1998, pp. 107–116.

[35] Hochreiter, S.; Schmidhuber, J. “Long short-term memory”, *Neural computation*, vol. 9–8, 1997, pp. 1735–1780.

[36] Husain, H.; Wu, H.-H.; Gazit, T.; Allamanis, M.; Brockschmidt, M. “Codesearchnet challenge: Evaluating the state of semantic code search”, *arXiv preprint arXiv:1909.09436*, 2019.

[37] Iyer, S.; Konstas, I.; Cheung, A.; Zettlemoyer, L. “Summarizing source code using a neural attention model”. In: 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), 2016, pp. 2073–2083.

[38] Iyer, S.; Konstas, I.; Cheung, A.; Zettlemoyer, L. “Mapping language to code in programmatic context”. In: 2018 Conference on Empirical Methods in Natural Language Processing, 2018, pp. 1643–1652.

[39] Kingma, D. P.; Ba, J. “Adam: A method for stochastic optimization”, *arXiv preprint arXiv:1412.6980*, 2017.

[40] Krizhevsky, A.; Sutskever, I.; Hinton, G. E. “Imagenet classification with deep convolutional neural networks”, *Advances in neural information processing systems*, vol. 25, 2012, pp. 1097–1105.

[41] Kumar, N. “The past and present of imitation learning: A citation chain study”, *arXiv preprint arXiv:1412.6980*, 2020.

[42] Lewis, M.; Liu, Y.; Goyal, N.; Ghazvininejad, M.; Mohamed, A.; Levy, O.; Stoyanov, V.; Zettlemoyer, L. “BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension”. In: 58th Annual Meeting of the Association for Computational Linguistics, 2020, pp. 7871–7880.

[43] Li, X. L.; Liang, P. “Prefix-tuning: Optimizing continuous prompts for generation”, *arXiv preprint arXiv:2101.00190*, 2021.

[44] Liu, F.; Li, G.; Zhao, Y.; Jin, Z. “Multi-task learning based pre-trained language model for code completion”. In: 35th IEEE/ACM International Conference on Automated Software Engineering, 2020, pp. 473–485.

[45] Liu, P.; Yuan, W.; Fu, J.; Jiang, Z.; Hayashi, H.; Neubig, G. “Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing”, *arXiv preprint arXiv:2107.13586*, 2021.

[46] Liu, X.; Zhang, F.; Hou, Z.; Mian, L.; Wang, Z.; Zhang, J.; Tang, J. “Self-supervised learning: Generative or contrastive”, *IEEE Transactions on Knowledge and Data Engineering*, vol. 35–1, 2023, pp. 857–876.

[47] Lloyd, S. “Least squares quantization in pcm”, *IEEE transactions on information theory*, vol. 28–2, 1982, pp. 129–137.

[48] Lu, S.; Guo, D.; Ren, S.; Huang, J.; Svyatkovskiy, A.; Blanco, A.; Clement, C.; Drain, D.; Jiang, D.; Tang, D.; et al.. “Codexglue: A machine learning benchmark dataset for code understanding and generation”, *arXiv preprint arXiv:2102.04664*, 2021.

[49] McCulloch, W. S.; Pitts, W. “A logical calculus of the ideas immanent in nervous activity”, *The bulletin of mathematical biophysics*, vol. 5–4, 1943, pp. 115–133.

[50] Mikolov, T.; Chen, K.; Corrado, G.; Dean, J. “Efficient estimation of word representations in vector space”, *arXiv preprint arXiv:1301.3781*, 2013.

[51] Mikolov, T.; Karafiát, M.; Burget, L.; Černockỳ, J.; Khudanpur, S. “Recurrent neural network based language model”. In: Eleventh annual conference of the international speech communication association, 2010.

[52] Mitchell, T. M. “Machine learning”. McGraw-hill, 1997.

[53] Murphy, K. P. “Machine learning: a probabilistic perspective”. MIT press, 2012.

[54] Papineni, K.; Roukos, S.; Ward, T.; Zhu, W.-J. “Bleu: a method for automatic evaluation of machine translation”. In: 40th annual meeting of the Association for Computational Linguistics, 2002, pp. 311–318.

[55] Paszke, A.; Gross, S.; Massa, F.; Lerer, A.; Bradbury, J.; Chanan, G.; Killeen, T.; Lin, Z.; Gimelshein, N.; Antiga, L.; Desmaison, A.; Kopf, A.; Yang, E.; DeVito, Z.; Raison, M.; Tejani, A.; Chilamkurthy, S.; Steiner, B.; Fang, L.; Bai, J.; Chintala, S. “Pytorch: An imperative style, high-performance deep learning library”. In: *Advances in Neural Information Processing Systems 32*, Curran Associates, Inc., 2019, pp. 8024–8035.

[56] Phan, L.; Tran, H.; Le, D.; Nguyen, H.; Anibal, J.; Peltekian, A.; Ye, Y. “Cotext: Multi-task learning with code-text transformer”, *arXiv preprint arXiv:2105.08645*, 2021.

[57] Provost, F.; Fawcett, T. “Data Science for Business: What you need to know about data mining and data-analytic thinking”. Reilly Media, Inc., 2013.

[58] Rabinovich, M.; Stern, M.; Klein, D. “Abstract syntax networks for code generation and semantic parsing”, *arXiv preprint arXiv:1704.07535*, 2017.

[59] Radford, A.; Narasimhan, K.; Salimans, T.; Sutskever, I. “Improving language understanding by generative pre-training”, *OpenAI blog*, 2018.

[60] Radford, A.; Wu, J.; Child, R.; Luan, D.; Amodei, D.; Sutskever, I.; et al.. “Language models are unsupervised multitask learners”, *OpenAI blog*, vol. 1–8, 2019, pp. 9.

[61] Raffel, C.; Shazeer, N.; Roberts, A.; Lee, K.; Narang, S.; Matena, M.; Zhou, Y.; Li, W.; Liu, P. J. “Exploring the limits of transfer learning with a unified text-to-text transformer”, *arXiv preprint arXiv:1910.10683*, 2019.

[62] Ren, S.; Guo, D.; Lu, S.; Zhou, L.; Liu, S.; Tang, D.; Sundaresan, N.; Zhou, M.; Blanco, A.; Ma, S. “Codebleu: a method for automatic evaluation of code synthesis”, *arXiv preprint arXiv:2009.10297*, 2020.

[63] Rogers, A.; Kovaleva, O.; Rumshisky, A. “A primer in bertology: What we know about how bert works”, *Transactions of the Association for Computational Linguistics*, vol. 8, 2020, pp. 842–866.

[64] Rosenblatt, F. “The perceptron: a probabilistic model for information storage and organization in the brain.”, *Psychological review*, vol. 65–6, 1958, pp. 386.

[65] Ruder, S. “An overview of multi-task learning in deep neural networks”, *arXiv preprint arXiv:1706.05098*, 2017.

[66] Rumelhart, D. E.; Hinton, G. E.; Williams, R. J. “Learning representations by back-propagating errors”, *nature*, vol. 323–6088, 1986, pp. 533–536.

[67] Sammut, C.; Webb, G. I. (Editors). “TF–IDF”. Springer US, 2010.

[68] Samuel, A. L. “Some studies in machine learning using the game of checkers”, *IBM Journal of research and development*, vol. 3–3, 1959, pp. 210–229.

[69] Sejnowski, T. J. “The Deep Learning Revolution”. MIT Press, 2018.

[70] Sheng, E.; Chang, K.-W.; Natarajan, P.; Peng, N. “Towards Controllable Biases in Language Generation”. In: Association for Computational Linguistics: EMNLP 2020, 2020, pp. 3239–3254.

[71] Socher, R.; Bengio, Y.; Manning, C. "Deep learning for NLP". In: Tutorial at Association of Computational Linguistics (ACL), 2012.

[72] Standley, T.; Zamir, A.; Chen, D.; Guibas, L.; Malik, J.; Savarese, S. "Which tasks should be learned together in multi-task learning?" In: 37th International Conference on Machine Learning, III, H. D.; Singh, A. (Editors), 2020, pp. 9120–9132.

[73] Sung, F.; Yang, Y.; Zhang, L.; Xiang, T.; Torr, P. H.; Hospedales, T. M. "Learning to compare: Relation network for few-shot learning". In: IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2018.

[74] Sutton, R. S.; Barto, A. G. "Reinforcement Learning: An Introduction". A Bradford Book, 2018.

[75] Tai, K. S.; Socher, R.; Manning, C. D. "Improved semantic representations from tree-structured long short-term memory networks", *arXiv preprint arXiv:1503.00075*, 2015.

[76] Tan, P.-N.; Steinbach, M.; Karpatne, A.; Kumar, V. "Introduction to Data Mining (2nd Edition)". Pearson, 2018.

[77] Tufano, M.; Drain, D.; Svyatkovskiy, A.; Deng, S. K.; Sundaresan, N. "Unit test case generation with transformers and focal context", 2020.

[78] Tufano, M.; Watson, C.; Bavota, G.; Penta, M. D.; White, M.; Poshyvanyk, D. "An empirical study on learning bug-fixing patches in the wild via neural machine translation", *ACM Trans. Softw. Eng. Methodol.*, vol. 28–4, sep 2019.

[79] Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A. N.; Kaiser, Ł.; Polosukhin, I. "Attention is all you need". In: Advances in neural information processing systems, 2017, pp. 5998–6008.

[80] Wang, A.; Singh, A.; Michael, J.; Hill, F.; Levy, O.; Bowman, S. R. "Glue: A multi-task benchmark and analysis platform for natural language understanding", *arXiv preprint arXiv:1804.07461*, 2018.

[81] Wang, D.; Yu, Y.; Li, S.; Dong, W.; Wang, J.; Qing, L. "Mulcode: A multi-task learning approach for source code understanding". In: 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2021, pp. 48–59.

[82] Wang, M.; Deng, W. "Deep face recognition: A survey", *Neurocomputing*, vol. 429, 2021, pp. 215–244.

[83] Wang, W.; Li, G.; Ma, B.; Xia, X.; Jin, Z. “Detecting code clones with graph neural network and flow-augmented abstract syntax tree”. In: 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2020, pp. 261–271.

[84] Wang, X.; Wang, Y.; Mi, F.; Zhou, P.; Wan, Y.; Liu, X.; Li, L.; Wu, H.; Liu, J.; Jiang, X. “Syncobert: Syntax-guided multi-modal contrastive pre-training for code representation”, *arXiv preprint arXiv:2108.04556*, 2021.

[85] Wang, Y.; Wang, W.; Joty, S.; Hoi, S. C. “Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation”, *arXiv preprint arXiv:2109.00859*, 2021.

[86] Wehrmann, J.; Parraga, O.; Barros, R. C. “Cobe: A natural language code search robustness benchmark”. In: 2022 International Joint Conference on Neural Networks (IJCNN), 2022, pp. 1–8.

[87] Wolf, T.; Debut, L.; Sanh, V.; Chaumond, J.; Delangue, C.; Moi, A.; Cistac, P.; Rault, T.; Louf, R.; Funtowicz, M.; Davison, J.; Shleifer, S.; von Platen, P.; Ma, C.; Jernite, Y.; Plu, J.; Xu, C.; Scao, T. L.; Gugger, S.; Drame, M.; Lhoest, Q.; Rush, A. M. “Transformers: State-of-the-art natural language processing”. In: 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations, 2020, pp. 38–45.

[88] Wu, Y.; Schuster, M.; Chen, Z.; Le, Q. V.; Norouzi, M.; Macherey, W.; Krikun, M.; Cao, Y.; Gao, Q.; Macherey, K.; et al.. “Google’s neural machine translation system: Bridging the gap between human and machine translation”, *arXiv preprint arXiv:1609.08144*, 2016.

[89] Xian, Y.; Schiele, B.; Akata, Z. “Zero-shot learning-the good, the bad and the ugly”. In: IEEE Conference on Computer Vision and Pattern Recognition, 2017, pp. 4582–4591.

[90] Xu, F. F.; Alon, U.; Neubig, G.; Hellendoorn, V. J. “A systematic evaluation of large language models of code”. In: 6th ACM SIGPLAN International Symposium on Machine Programming, 2022, pp. 1–10.

[91] Yang, Q.; Zhang, Y.; Dai, W.; Pan, S. J. “Transfer learning”. Cambridge University Press, 2020.

[92] Zhang, N.; Li, L.; Chen, X.; Deng, S.; Bi, Z.; Tan, C.; Huang, F.; Chen, H. “Differentiable prompt makes pre-trained language models better few-shot learners”, *arXiv preprint arXiv:2108.13161*, 2021.

[93] Zhang, Y.; Yang, Q. “A survey on multi-task learning”, *IEEE Transactions on Knowledge and Data Engineering*, vol. 34–12, 2022, pp. 5586–5609.

[94] Zhou, Y.; Liu, S.; Siow, J.; Du, X.; Liu, Y. “Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks”. In: Advances in Neural Information Processing Systems, Wallach, H.; Larochelle, H.; Beygelzimer, A.; d'Alché-Buc, F.; Fox, E.; Garnett, R. (Editors), 2019.