Evaluating and Improving Generative Pre-Trained Transformers for Code Completion

Master thesis by Marcel Ochs
Date of submission: January 31, 2022

1. Review: Prof. Dr.-Ing. Mira Mezini
2. Review: Dr. Krishna Narasimhan
Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

Computer Science Department

Software Technology Group

Erklärung zur Abschlussarbeit gemäß §22 Abs. 7 und §23 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, Marcel Ochs, die vorliegende Masterarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Fall eines Plagiats (§38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung gemäß §23 Abs. 7 APB überein.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, 31. Januar 2022

M. Ochs

Abstract

Code completion is one of the core features of any modern integrated development environment (IDE). Usually IDEs make use of static code analysis in order to provide autocompletions and code suggestions. As code completion could be regarded as a variant of text generation, language models could be used to fulfill this task. GPT-2 for instance is a language model based on the transformer architecture. It was originally designed for natural text based tasks such as machine translation, summarization and more. The original GPT-2 model was trained on large bodies of crawled web texts such as news articles or social media posts. However, the most obvious hurdle using such natural language based language models for code generation is that programming languages and natural language follow different rules: Programming languages have structural and semantic patterns and do not provide as much freedom as natural language does. These restrictions can be leveraged: The strict programming language ruleset can used to parse alternative representations such as abstract syntax trees (ASTs) containing additional structural information on the source code.

Previous work by Facebook Research (Kim et al. [9]) implemented and evaluated a GPT-2 model called *TravTrans* which was trained on abstract syntax trees rather than on natural texts. The *TravTrans* architecture is very close to the original GPT-2 architecture and only introduces a slight overhead. Using alternative source code representation such as AST provides the already potent language model with an additional layer of information, resulting in a well performing model [5] for the increasingly popular task of intelligent source code completion.

This thesis is a deeper analysis of the proposed *TravTrans* model and tries to identify and solve potential architectural or technical downsides of transformer models for source code completion in an experimental setup, establishing a solid base on which future code completion research can build upon.

Contents

1 Introduction

1.1 Background and Motivation

In modern integrated development environments (IDEs), code completion is the most widely used and integral feature [9, 17, 13]. Beside reducing developing time by offering quick completions for developers, the code completion feature allows inexperienced developers to find their way around unknown frameworks and libraries by supplying them with relevant predictions.

The "classic" approach of code completion, *static code completion*, makes use of static code analysis, for example by keeping track of variables in a small database. After special marker characters such as points or brackets, the relevant objects or variables are presented to the developer. One downside of this approach is the ordering of the suggestions: Oftentimes the suggestions are sorted alphabetically rather than sorted by relevance, forcing the developer to scroll through a list of suggestions, nullifying the speed advantage code completion usually aims to achieve.

Another way to make code predictions is *intelligent code completion*: This approach makes use of machine learning in order to train a model to make relevant and context based code predictions. This thesis will shed some light on the usage of modern natural language models such as transformers for intelligent code completion.

Current state-of-the-art natural language models rely on the transformer architecture [18] which was originally designed for natural language tasks. Transformer implementations such as the *generative pre-trained transformer* (GPT) or the *bidirectional encode representations from transformer* (BERT) were also developed and tailored for natural language tasks such as text generation, question answering and machine translation [15].

The naive approach on using machine learning for source code prediction would be using a state-of-the-art natural language model which performs well on natural text generation

and train it on source code. This would work well if natural language and programming languages were following the same rules and structures. However, this is not the case: Programming languages follow a much stricter rule set than natural language. While there are numerous ways to describe a certain idea in natural language, there is only a limited number of ways to describe a process in a programming language. For example, changing the word orderings or even omitting entire words from a sentence can still get a message across. For programming languages however, even the slightest changes such as punctuation or missing brackets could instantly result in a non-compiling or error throwing program.

The advantage of such a strict rule set in programming languages is that source code can be parsed into alternative and structured code representation such as *abstract syntax trees* (AST). In addition to the source code itself, ASTs contain additional structural information on every part of the source code.

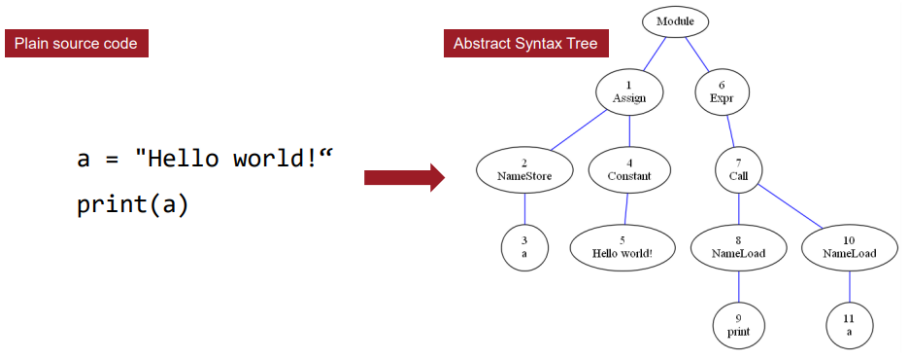

Figure 1.1: Abstract syntax tree as alternative source code representation

For example a language model could interpret variable `a` in line two of the source code in figure 1.1 as anything: It could be a variable, a string, a method call or anything else. The language model would have to learn to interpret the fields true meaning during training time and with sufficient training data. When parsing source code to ASTs however, the `a` is augmented with additional information in the parent nodes: The model can now clearly learn that in this case, `a` is a NameLoad in a method call and the string "Hello world!" was previously assigned to it.

ASTs extend the information provided by plain source code and therefore makes it easier for language models to train a properly working model.

The model presented in the research by Facebook Research Group [9] called *TravTrans* makes use of this finding: *TravTrans* is a slightly modified transformer model (GPT-2 to be exact) which was adjusted such that it can be trained on ASTs rather than on natural texts. This is a highly promising approach as the is architecture close to the state-of-the-art GPT-2 architecture with minimal modifications and introduces minimal additional overhead and complexity. Another advantage is that *TravTrans* is exclusively trained on parsed ASTs instead of natural language: Language models trained on large bodies of natural texts such as news articles, Wikipedia entries or social network posts are prone to learn biases and stereotypical ideologies [12] which may be damaging. This issue is circumvented by limiting training data to source code only.

Although there are numerous large models trying to tackle the intelligent code completion task, most of them (GPT-3 [3], GitHub Copilot[1], IntelliCode Compose[2]) are enormous models which require vast amounts of resources for training and are hard to deploy [17]. This limits the number of researchers capable to handle the model size who are interested in intelligent code completion. Due to its simplicity and good performance [5], *TravTrans* is the ideal language model for this thesis code completion scope. The goal is to predict the next field, just like an IDE would. Larger models such as GitHub Copilot may be capable of generating larger source code blocks but are less accessible and much harder to train and deploy personally.

This thesis will investigate how well the *TravTrans* model performs and find potential risks and solutions. The *TravTrans* model by Facebook Research acts as baseline model and was implemented in an experimental setup. The goal of this thesis is to establish a good understanding of the underlying model in order to provide a solid base for future research to build and extend upon.

1.2 Research Questions

During the investigation of the *TravTrans* models some questions emerged, the most interesting ones making up this thesis research questions.

RQ1: How does *TravTrans* perform when reproduced? Facebook Research published their *TravTrans* model on GitHub. However, they did not provide the accompanying

[1] https://copilot.github.com/

[2] https://www.microsoft.com/en-us/research/publication/intellicode-compose-code-generation-using-transformer/

training and evaluation scripts, which means that they have to be implemented from scratch. This research question aims to re-implement *TravTrans* on an experimental setup and afterwards train and evaluate on custom training and evaluation routines. The evaluation score of the re-implemented model will deal as baseline score for the following research questions.

RQ2: Can TravTans be improved by adjusting architectural settings? The *TravTrans* model architecture contains several properties whose values are not justified by the authors, such as the number of decoder layers or embedding sizes. This research question aims to look into these settings and to evaluate whether or not there are better settings.

RQ3: Should additional metrics be tracked for model evaluation? The scores generated during *TravTrans* model evaluation are grouped into several categories. These categories are capable of answering questions such as "How well does the model perform on parameter name predictions?" or "How well does the model perform on string predictions?". This research question aims to introduce additional categories which may be necessary in order to allow the comparison of *TravTrans* to other models.

RQ4: Can the out-of-vocabulary issue be reduced by using an alternative tokenizer? The tokenizer used for *TravTrans* makes it susceptible to out-of-vocabulary problems, meaning that the model cannot train on less frequent source code tokens. This research question tries to tackle this problem by introducing an alternative sub-word tokenizer which splits less frequent tokens into more frequent sub-tokens.

RQ5: Which impact does the overlap size have on model performance? During the process of flattening ASTs into sequences, *TravTrans* uses a sliding window if an AST is larger than the model input window size and therefore has to be split up. It is not made clear why the authors chose a specific overlap size which is why this research question aims to figure out the impact of smaller or larger overlap sizes on overall model performance.

RQ6: Does positional embedding improve the performance of the model? Recent research implies that the positional encoding layer in transformer models may be damaging to the model performance. The authors of *TravTrans* have therefore removed the positional encoding layer from the baseline GPT-2 model. This research question tries to find out if this implication holds true by re-introducing the positional encoding layer.

2 Theoretical Background

2.1 Artificial Intelligence Aspects

2.1.1 Neural Networks

The following sections will go into detail on how language models work from a technical perspective. In order to cover the neural network basics, this chapter will give an introduction to simple feed forward neural networks.

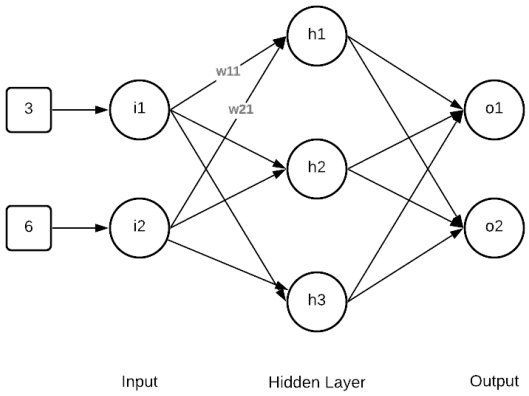

Figure 2.1: Simple feed forward neural network with one hidden layer

A feed forward neural network such as shown in figure 2.1 is a simple neural network architecture. It consists of input nodes and output nodes which are interconnected via one or more hidden layers. An input is fed to the input nodes which will then pass through the entire network until it reaches the output nodes. The resulting output is then compared

to the desired output and the error (also known as *loss*) is used to update the model parameters.

In the example shown in figure 2.1 there are two numbers passed to the model, 3 and 6. In a forward pass, input node 1 receives input value 3 and sends it towards the hidden layer. Each connection between two nodes contains a *weight*, which is a parameter updated during training. In this case, hidden layer node `h1` is connected to both input nodes `i1` and `i2` with the weights `w11` and `w21`. The value at `h1` is computed by summing the value of the previous nodes with the weight of their respective connection. In this example it means that the value at `h1` is: $3 * w1 + 6 * w2$. Afterwards another function (*activation function*) is applied on the value at `h1` before it's passed to the next node.

Finally, the generated output is compared to the desired output by using a *loss function* which is able to measure the distance or similarity between two results. This loss is then fed back through the model via *backpropagation* which computes the impact of every single weight on the loss in order to update them accordingly. The goal is that the loss of the next forward pass is lower due to the updated weights.

2.1.2 Natural Language Processing

Code completion can be regarded as a text generation problem: Given an input sentence, our code completion model should generate the next word. There are language models which achieve great results [15] in text generation. Language models are part of natural language processing (*NLP*), a large domain on the computer science field.

NLP deals with analyzing and processing texts for the purpose of achieving human-like language processing for a wide range of tasks or applications [11]. While the entire NLP topic can be divided into the sub fields *natural language processing* and *natural language generation*, this thesis focuses on the latter topic as the main objective is the generation of text or, more specifically, source code. Language models are trained on large amounts text which enables them afterwards to generate probability distributions for the next word of a sentence. This means that given an input sentence, the model could generate a probability distribution for every word in its vocabulary, the word with the highest probability being the most likely word to continue the given sentence.

In contrast to the classic natural language generation that generates natural texts for books and articles, the language model this thesis deals with is trained on the task of completing a flattened abstract syntax tree which was parsed from source code.

Additional applications of NLP include question-answering, text summarization, machine translation, dialogue systems and more [11].

The next section deals with sequence-to-sequence (Seq2Seq) models which are machine learning models often used for NLP tasks.

2.1.3 Seq2Seq Models

Sequence-to-sequence or *seq2seq* models can be very successful at machine translation, speech recognition, image and video captioning, text summarization and dialogue modeling tasks [7]. Given an input sequence of arbitrary length which may be text or even images, the model will generate an independently sized output sequence, hence the name sequence-to-sequence model.

The seq2seq approach is straightforward: An input sequence will be encoded by an *encoder*. It generates a vector representation of the input sequence which contains encoded information about the input sequence. This vector representation is then fed to the *decoder* which generates an output sequence given the vector representation. This allows the seq2seq model to be applied on numerous tasks.

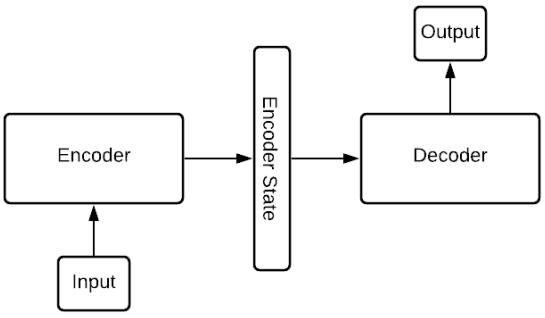

Figure 2.2: Basic seq2seq architecture

For example, an image could be fed into the encoder, which generates a hidden vector representation of the image. The decoder will then use the vector representation in order to generate a sequence of words which describe objects visible in the image in human readable text. Figure 2.2 visualizes the basic architecture of a seq2seq model.

Recurrent Neural Networks

The most basic implementation of the encoder or decoder blocks shown in 2.2 are Recurrent Neural Networks (RNNs). As the name suggests, a RNN is a neural network, consisting of a single recurring layer rather than having several hidden layers. The depth of the model depends on how many times the RNN feeds its output back to itself. At each step, the RNN will calculate a hidden state h and pass it to itself alongside with the next word as input, which will then have an impact on the next hidden state. Figure 2.3 shows an unrolled version of a RNN: Feeding the first word of the input x_1 to the model results in hidden state h_1. This hidden state will then be fed to the model together with the next word x_2 to generate hidden state h_2 and so on.

$$h_t = f_W(h_{t-1}, x_t) \tag{2.1}$$

Equation 2.1 shows how the hidden state for time step t is calculated by inserting the hidden state from the previous time step h_{t-1} and the input of the current time step x_t into a function f with weight parameters W.

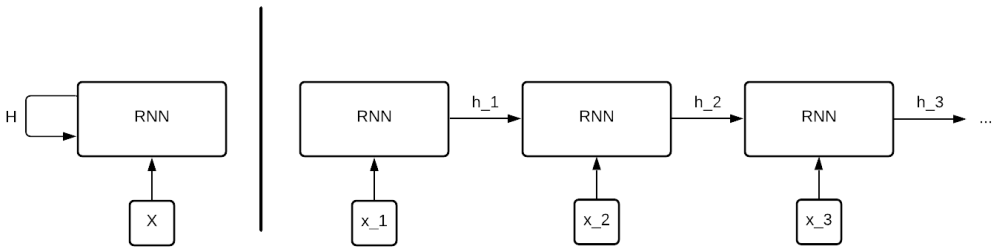

Figure 2.3: Left: RNN, Right: Unrolled RNN over 3 steps

Advantages RNNs work well with input sequences of arbitrary length. In order to cope with longer input sequences, the model just has to repeat the recurring self-feeding routine until the entire input sequence is processed.

For example: In order to process an input of length 3, the model will repeat the self-feeding process three times, each iteration processing the next element of the input sequence (see figure 2.3). This routine applies to an input of length 100, $1,000$ or any

other arbitrary number: The model would simply have to repeat the previously mentioned process for n times, n being an arbitrary number of elements in the input sequence.

Disadvantages The ability to accept long inputs comes with a disadvantage called *exploding or vanishing gradients*. The RNN can be regarded as a very deep neural network with many hidden layers if it was fed a very long input sequence. The chain rule equation during backpropagation will therefore be long as well, as it calculates the gradients through every layer. As the chain rule is a series of multiplications, if there are a lot of parameters that are smaller than 1, the gradients become exponentially small (for example when multiplying with 0.1 over and over again). As result, the gradients for the first weights are so small that the weight update at the first layers and therefore the first words of the input sequence has almost no impact. This is called vanishing gradients. On the other hand, if a lot of parameters in the chain rule equation are greater than 1, the gradients become exponentially large such that it could come to numerical errors. This is known as exploding gradients.

Another downside of RNNs is that the input sequence can only be fed into the model sequentially (word-by-word), as a word can only be processed by the RNN as soon as the model has computed the hidden state of its predecessor.

Yet another downside is the *long term memory* capability of RNNs: As the encoder tries to store information about the entire input sentence in a fixed length hidden vector, the longer the sentence becomes, the more information will be lost. Information on earlier words is more likely to be forgotten in a large input sentence. The impact of these issues can be severe.

For example: Assuming that a long sentence should be translated from English into another language, there could be a subject introduced at the beginning of a sentence. At the sentence's end, the correct gender of said subject should be generated. However, if the subject's gender was forgotten during the encoding process, the correct pronoun can hardly be predicted, resulting in a poor performing machine translation model. One word at the end of the sentence referring to a word at the beginning of the sentence is called a *long range dependency*. Both words are relevant to each other despite being wide apart. This issue will also be a big problem for intelligent code completion: A variable initialized somewhere at the beginning of the source code should still be remembered and relevant towards the end of the source code.

In order to avert the issue of exploding or vanishing gradients and short memory, Long Short-Term Memory models or LSTMs introduced special gates to control the information

flow.

Long Short-Term Memory

Long Short-Term Memory models or LSTMs are an improved version of RNNs. They contain *control gates* which allow more fine-grained information flow inside of LSTM cells: The input gate i decides which values will be updated, the *forget gate* f decides which values will be omitted and the *output gate* which decides which information will flow to the next step.

LSTMs can prevent the previously mentioned vanishing and exploding gradients as there is a direct link between layers, the *cell-state*. This allows the gradients to flow back freely through the cells without the risk of vanishing or exploding.

The downside that still remains is the sequential processing of the input sentence. Just like RNNs, LSTMs process input sequences in order, word-by-word and do not make use of parallelization. Additionally, due to their increased complexity they also require more training time than RNNs.

This disadvantage can be avoided by using a transformer model which is introduced in the next section. Transformer models are able to process entire sequences in one go.

Transformers

A novel mechanism called *Attention* was introduced by Bahdanau et al. [1] in order to alleviate the bottleneck of a fixed length vector between encoder and decoder in sequence-to-sequence models. The attention mechanism was implemented in the transformer model introduced by Vaswani et al. [18]. Not only should it retain long-range dependencies better than RNNs and LSTMs, but also the aspect of slow training due to sequential processing of input sentences is addressed, as the entire input sequences can be processed in a single step.

Architecture Figure 2.4 is a visualization of the transformer model architecture introduced by Vaswani et al. [18]. It follows the design of the general sequence-to-sequence architecture shown in figure 2.2. The left side of the transformer model consists of multiple (Nx) stacked encoder blocks while the right side consists of multiple stacked decoder blocks.

The general functionality of a transformer looks as follows: Initially, the input sequence is fed through the left encoder stack, labeled “Inputs” in figure 2.4. Words generated by the model are added to the sequence labeled “Output” which is fed through the right decoder stack. The actual prediction the model makes is a probability distribution for each word in the model’s vocabulary, which can be seen in 2.4 labeled as *Output Probabilities*: A word with a high probability will be more likely be the next word of the sentence, while a word with a lower probability will be less likely to continue the sentence. This is especially useful for the intelligent code completion task, as an IDE could use the probabilities to sort its predictions: Higher ranked predictions could have a higher probability of continuing the source code than lower ranked predictions which should be less relevant.

Machine Translation Example: An English sentence should be translated into French by using a trained transformer model. The input sentence of n words is inserted as *Inputs*. Initially, the *Outputs* section is empty as the model has yet to generate any words.

1. The input sentence is fed through the encoder stack which generates matrices that can be imagined of as the hidden vector representation in the basic sequence-to-sequence model in figure 2.2.
2. In the next step the (initially empty) *Outputs* sequence is fed through the decoder stack which incorporates the matrices generated in the first step. As an output the model will return a list of probabilities for each word of the model’s vocabulary. A word with a high probability is more likely to continue the sequence than a word with low probability. Following a greedy approach the word with the highest probability can be selected which is then inserted into the *Outputs* sequence.
3. Step 2 keeps repeating until the model generates a special symbol that indicates the end of the sentence. Ideally, the *Outputs* sequence should resemble a French translation of the English input sentence.

The following paragraphs go into more detail on how transformer encoder and decoder blocks are designed and how the attention mechanism works.

Encoder Block The left side of the transformer in figure 2.4 consists of multiple stacked encoder blocks. An encoder block mainly features two layers, an attention layer and a simple feed forward layer. The attention layer computes an attention score for each word of the sentence. An attention score for word i contains information about how relevant other words in the sentence are for word i. More details on the attention mechanism can be found in paragraph 2.1.3. Afterwards the attention scores are fed through a feed

forward layer which transforms the outputs from the previous attention layer such that they can be fed to the next encoder block.

Decoder Block The right side of the transformer in figure 2.4 consists of multiple stacked decoder blocks. A decoder block features three main layers: A masked attention layer, an (*encoder-decoder*) attention layer and a feed forward layer. The masked attention layer is a slight modification from the attention layer found in encoder blocks. This adjustment "masks" future words during the attention score calculation meaning that the attention for a word at position i will only include information on words at positions $< i$ and "mask out" words at positions $> i$. This masking prevents the model from "cheating" during the training process.

Example: Assuming that the model should be trained on an English-to-French translation task, an English sentence is fed to the model as well as the desired French translation. The French translation is required to compare if the words generated by the model are correct or not. In this example, the English training sample is "the car is very red" and the French training sample is "la voiture est très rouge". Transformer models can process entire sentences, so both of these sentences are fed to the model in one step. The model's main goal should then be to correctly generate the French sentence word-by-word. The problem is: The model's task is to generate the French sentence while already having access to the desired output. This means that during the attention process, the model could "peek" to the right and get the desired word instead of actually learning how to generate the correct word. In this example, assuming that the French word "la" was already predicted, the next step would be to predict "voiture". But during the attention process the model compares all of the words in the French sentence, including the ones that are yet to be predicted like "voiture". In order to force the model to learn and prevent cheating, the masked attention mechanism was designed.

Another difference between encoder blocks and decoder blocks is the encoder-decoder attention layer. Usually, the attention layer processes the inputs passed by the underlying encoder block. The encoder-decoder block however incorporates one part from the underlying decoder block and another part which was generated by the topmost encoder block. Other than that the computations performed by the attention mechanism are equal to the base attention mechanism found in encoder blocks. The encoder-decoder attention layer basically computes attention scores for the encoded input sequence on the generated output sequence.

Finally, these attention scores are fed through a feed forward layer in order to prepare them for the next decoder block.

Attention Mechanism RNNs sequentially process given input sequences word-by-word. Each word is passed into the RNN together with the hidden state from the previous word and results in a new hidden state which ideally contains some more information on the previously processed words. Transformers on the other hand don't make use of such a hidden state: Each word of the input sentence is compared to every other word in regard to relevance. Figure 2.5 shows the attention for one particular word, "it". The attention is not only calculated for the word "it" but on every word in the sentence. As can be seen in figure 2.5 the attention contains information on which words "it" may refer to, in this case, it is most likely "the animal". This means that instead of having a hidden state, each word in a transformer has its own attention value which contains information on how relevant other words of the input sentences are for the current word.

In order to calculate the attention for the ith word of the input sentence, we'll first require three vectors called "query", "key" and "value". Those vectors are calculated by embedding word i and multiplying it with the respective query, key and value weight matrices. These weight matrices are computed during the model training process and each encoder or decoder layer has its own weight matrices. Concluding the first step, word i gets embedded (see the input embedding layer in figure 2.4) and then multiplied with the query weight matrix, the key weight matrix and the value weight matrix. This results in three vectors, query, key and value vectors.

Afterwards, every word j in the sentence receives a score which is the dot product of the query vector of our current word i that we're currently looking at and the key vector of j. This results in n score values, one score for other word in the sentence which is n words long. The scores are then divided by 8 which is the square root of the key vector dimension. This step is to prevent vanishing gradients [18]. Finally, a softmax function is applied such that all scores add up to 1. Now the value vector of each word j gets multiplied with its respective softmax value. Afterwards the attention score for word i is computed by adding up all softmax-value products.

The vector names query, key and values make it easier to understand what happens during the attention process: As we're interested in the attention of word i, we take the query vector of word i and use it as query against the key value of all words in the sentence. This results in a score which can tell the how well the query of word i matches with the key of some other word j. If the score is high, i and j match up well together, a low score

means that both words do not add up well. By applying a softmax function on all scores we compress the scores such that all scores together are between 0 and 1 and sum up to 1. This can be imagined as assigning each word a percentage of importance: A higher softmax score is more important than a lower one. Finally, the softmax score is multiplied with the value of its respective word and these products are then summed up to form the final attention value for word i. At the end we were interested in each words value vector. We've scaled the value vector down by its importance which the softmax score resembled, meaning that important words keep most of their value while lesser important values are scaled down. Figure 2.6 visualizes the entire attention calculation. In this example it's obvious that the word "Thinking" has a high score towards itself, meaning that 88% of the value will be kept, while only 12% of the value for "Machines" will be kept.

One large advantage over RNNs or LSTMs is the ability to compute the attention scores for each word in parallel by making use of matrix multiplications. Instead of multiplying a single word embedding with the weight matrices for query/key/value, all word embeddings can be put into a single matrix. This matrix will then be multiplied with the query/key/value weight matrices which result in a query/key/value matrix containing the values for each word of the sentence. All following operations such as dividing or applying the softmax function can be applied on entire matrices. Finally, the attention output is a matrix of attention values for each word in the sentence. Instead of processing each word one-by-one the entire sentence is processed in one step.

Multi-Head Attention The previously introduced attention mechanism can be improved by using the *multi-headed attention*. It prevents the model to give high attention scores for one word on itself: That a word has a high relevance on itself is obvious. Due to this the model should rather focus on other words of the input sentence. Another addition is that multi-head attention includes multiple query/key/value weight matrices (Vaswani et al. [18] specified 8 by default) rather than a single one. All weight matrices are initialized independently and with random weights. This means that the previously described "general" attention mechanism approach is applied several times with different matrices. Instead of applying the attention mechanism just once with one query/key/value weight matrix, it will be computed n times, n being the number of attention heads. The resulting attention matrices are concatenated into one large matrix. As the matrix size won't be compatible with the following feed-forward layer, another weight matrix is introduced which reduces the dimensionality in order to keep the compatibility with the feed-forward layer.

Encoder-Decoder Multi-Head Attention Encoder-Decoder Multi-Head Attention is used in the second layer of decoder blocks. It works just like the basic Multi-Head Attention layer found in encoder blocks but instead of having query/key/value matrices, the Encoder-Decoder Multi-Head Attention layer only receives the query matrix from the previous decoder block. The key and value matrices come from the topmost encoder block. This means that the query comes from an output word and is compared to the keys from the input words. Instead of computing the relevance between words of the same sentence, the Encoder-Decoder Multi-Head Attention layer computes the relevance between words of different sentences, input and output.

Linear and Softmax Layer The last two layers of the transformer model are the linear and the softmax layer. The output from the last decoder block has to be turned into a matrix which contains a score for each word of the models vocabulary and therefore could be very large. This is done by adding the linear layer which is a simple fully connected neural network. In order to turn the attention scores scores into probabilities, a softmax function is applied. This results in scores between 0 and 1 which sum up to 1.

Generative Pre-Trained Transformer GPT-2

Generative Pre-Trained Transformers or GPT were initially introduced by Radford et al. [14], followed by revisions GPT-2 by Radford et al. [15] and the most recent GPT-3 in [3]. As the name suggests, GPT models are based on the original transformer architecture by Vaswani et al. [18] and introduce architectural changes. Figure 2.7 shows the architecture of a 12-layer GPT model.

Comparing figure 2.7 with figure 2.5 shows the main difference between the GPT architecture and the original transformer architecture: The GPT model solely consists of stacked decoder blocks and completely omits encoder blocks. The original transformer model allows two inputs, one for the actual input sentence called *input* and another one for all words previously generated by the model called *output*. GPT on the other hand is an *autoregressive*[1] model which can generate random coherent texts even without a given input. An input sentence can be provided (but is not required), which is fed to the model seen in 2.7 at *Text & Position Embed*. Generated words are also added to the input

[1] An autoregressive language model adds its generated output to its input and is therefore capable of iteratively generate large sentences

sequence and therefore influence the results for the next word prediction. GPT can train on large bodies of unlabeled data making it an *unsupervised autoregressive language model*.

2.2 Programming Language Aspects

2.2.1 Abstract Syntax Trees

One core intention of this thesis is to investigate the capabilities of NLP models for code completion tasks. However, most NLP models were designed for tasks dealing with natural texts such as books or articles rather than source code.

Usually a model like GPT-2 would be trained on large amounts of unlabeled text corpora of internet text. If source code was treated like natural texts, large amounts of source code from sources like for example GitHub could be directly fed to the model. However, there are additional structures in programming languages which are not present in natural texts and which could be leveraged in order to introduce additional structural information to the model and therefore achieve overall better scores. The structure investigated in this thesis is the *abstract syntax tree* (AST). This is a type of source code representation which displays source code in a tree data structure.

Example: An exemplary Python source code could look like this:

```
a = "Hello world!"
print(a)
```

If this source code snippet was treated like a natural text it could now be processed by a tokenizer which splits the input into single words. These words would then be fed to the transformer model. Alternatively, the source code could also be parsed into an abstract syntax tree, containing additional structural information which cannot be found in the plain source code. Figure 2.8 shows the AST representation of the source code above.

Comparing the plain source code with the abstract syntax tree shows that additional structural information is stored in the tree. For example, the model would have to learn from the plain source code that the equation symbol “=” is used for an assignment for variables. If such an equation symbol were now to appear in a string, the model could assume that its purpose is an assignment, while the real purpose may be a string value.

This is due to the ambiguity when treating source code as text. These confusions could be avoided by using abstract syntax trees. As can be seen in figure 2.8 there is no mention of an equation symbol as it's parsed into an "Assign" node. Finally, using ASTs leverage additional structural information about source code which can hardly be found by looking at plain source code. In addition to that, using ASTs instead of raw source code reduces ambiguity which could result from processing source code as words.

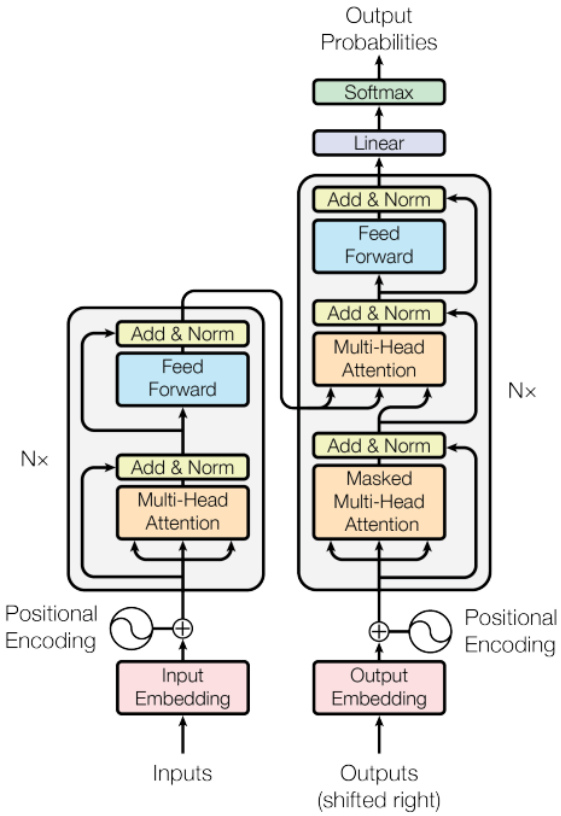

Figure 2.4: The original transformer architecture taken from [18] (Attention is all you need)

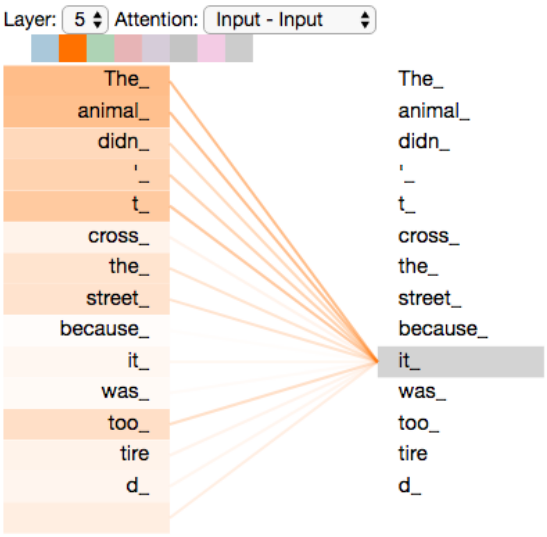

Figure 2.5: Exemplary visualization of attention on the word “it”, image from Jay Alammar’s “The Illustrated Transformer” https://jalammar.github.io/illustrated-transformer/

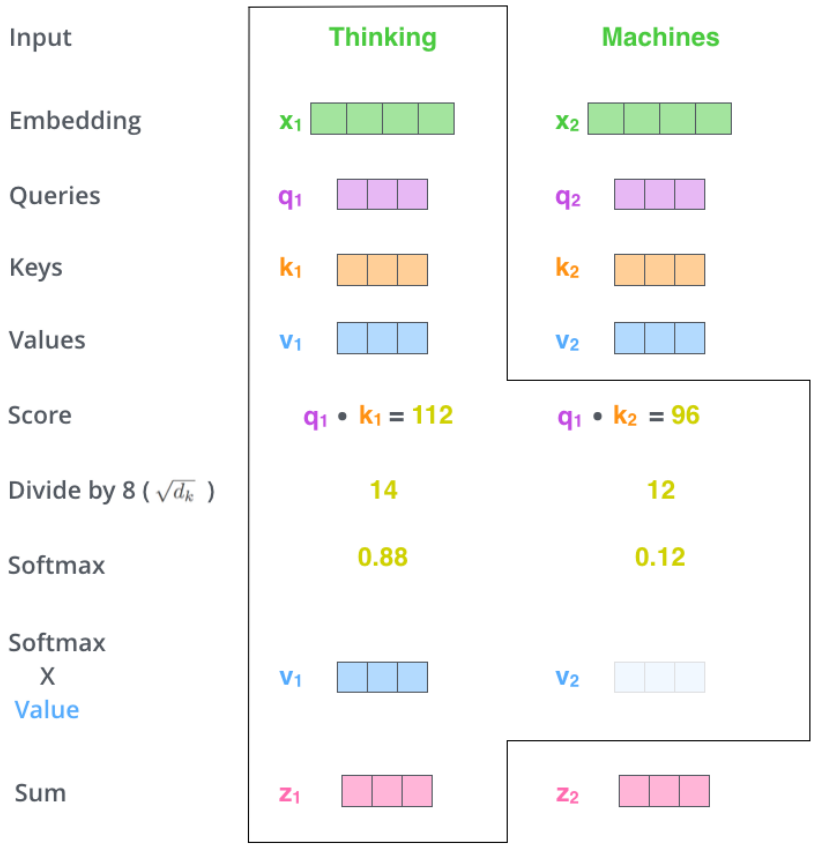

Figure 2.6: Calculating attention for the word “Thinking”, image from Jay Alammar’s “The Illustrated Transformer” https://jalammar.github.io/illustrated-transformer/

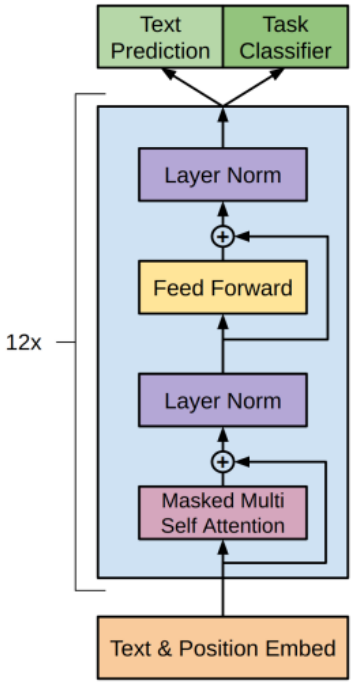

Figure 2.7: Generative Pre-Trained Transformer architecture from [14]

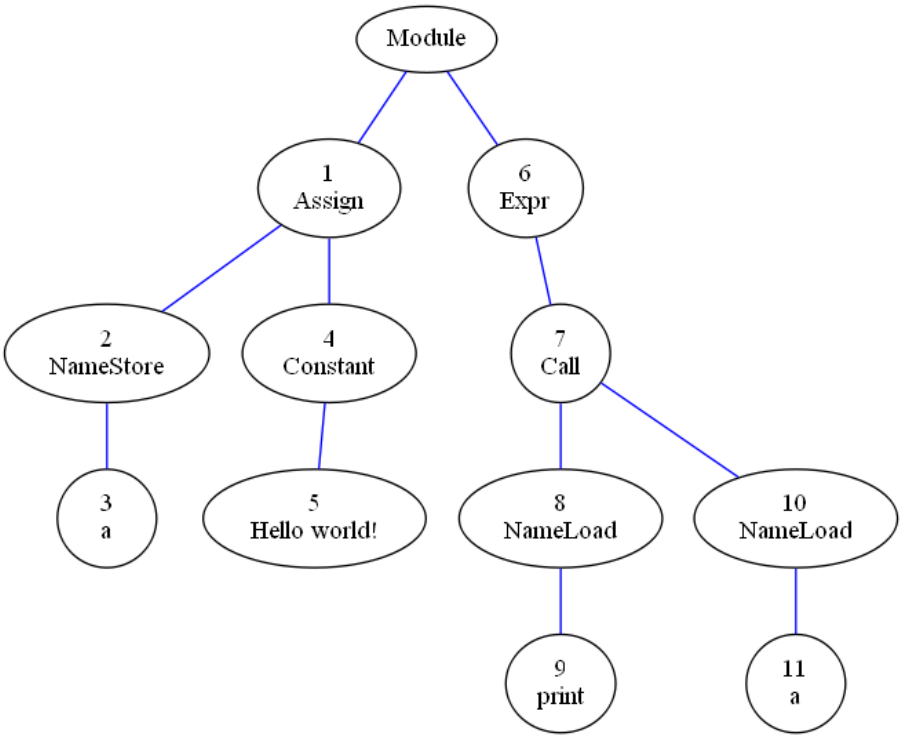

Figure 2.8: Abstract syntax tree representation of a sample code

3 Study Setup

This chapter deals with architectural and implementation details of the *TravTrans* model, as well as the data pre-processing pipeline which deals with the preparation of the dataset. Basically, the *TravTrans* model architecture is a modified *GPT-2 small*[1] model with 6 layers and instead training the model on natural language, it is trained on flattened abstract syntax trees.

3.1 Model Architecture

3.1.1 Baseline Model

As baseline model the *TravTrans* model by Kim et al.[9] is used as it's a simple yet powerful model for code completion tasks [5]. At the core of the stands a slightly adjusted GPT-2 model trained from scratch on the *150k Python Dataset*[2] consisting of $150,000$ abstract syntax trees parsed from Python source code.

The *TravTrans* model was selected as baseline model as its architecture is very close to the original GPT-2 architecture while achieving overall good scores on code completion tasks[9, 5] as can be seen in a comparison with alternative code completion models in table 3.1.

TravTrans achieves similar results to the *TravTrans+* model also introduced by Kim et al. [9]. However, *TravTrans+* replaces the "normal" attention mechanism with a tree relative attention mechanism that incorporates additional structural abstract syntax tree information by making use of paths between tree roots and leaf nodes. This adds overhead

[1] There are multiple variants of the GPT-2 model. They vary in the number of decoder blocks and therefore the amount of weight parameters.
[2] https://www.sri.inf.ethz.ch/py150

	Prior work		Our work	
Applications	Deep3	Code2Seq	PathTrans	TravTrans
Attribute access	38.5%	26.4%	41.5%	**44.7%**
Numeric constant	46.5%	45.4%	56.1%	**61.5%**
Name (variable, module)	41.0%	31.2%	48.0%	**50.7%**
Function parameter name	50.6%	39.3%	52.1%	**53.3%**
All leaf nodes	31.6%	31.0%	40.8%	**43.9%**

Table 3.1: *TravTrans* compared to alternative models

to the model training and additional complexity for a mere 1% performance increase regarding leaf node prediction [9]. As the additional overhead does not justify the small performance increase, the *TravTrans* model is selected as baseline model instead of *TravTrans+*. A comparison between *TravTrans* and *TravTrans+* can be seen in table 3.2. Additionally, Chirkova et al. [5] compared different approaches to source code completion with transformers and found out that *TravTrans* performs well overall for code completion tasks.

Applications	TravTrans	TravTrans+
Attribute access	60.5%	**61.2%**
Numeric constant	**63.5%**	**63.5%**
Name (variable, module)	66.6%	**67.7%**
Function parameter name	**67.2%**	67.0%
All leaf nodes	58.0%	**58.8%**
All internal nodes	87.3%	**91.9%**

Table 3.2: *TravTrans* compared to *TravTrans+*

3.1.2 Architecture

As *TravTrans* builds upon the GPT-2 architecture, figure 2.7 can be used to represent its architecture with the differences that the default number of layers is 6 instead of 12 and the position embedding layer is omitted.

Additionally, the embedding size used in *TravTrans* was set to 300 by Kim et al. [9], this choice is investigated in *RQ2* in section 5.1.2.

The original GPT-2 model introduced by Radford et al. [15] includes a positional embedding layer. However, *TravTrans* omits said layer, as trials suggest that it may be more damaging to the model performance than helpful [9]. This assumption is part of *RQ6* in section 5.3.1 and will therefore be investigated more thoroughly in the evaluation chapter.

Setup Modifications: The following research questions modify the aforementioned architectural setup in order to investigate how separate model properties such as embedding size, number of layers or tokenizer affect model performance.

- **RQ2** ("Can the baseline model be improved by adjusting architectural settings?") will evaluate the impacts of modifying model parameters, namely *embedding size* and *number of layers*. This research question will try to improve the baseline model by adjusting said parameters. There is no justification on how these values were selected despite them being an integral part of the *TravTrans* model which is why a further investigation promises to be insightful.
- As **RQ4** ("Can the out-of-vocabulary issue be reduced by using an alternative tokenizer?") deals with the introduction of an alternative tokenizer. The vocabulary size will therefore vary from the default $100,000$. The new tokenizer splits words into subwords and thus the vocabulary size is smaller.
- **RQ6** ("Does positional encoding improve the performance of the model?") investigates the impacts of a positional encoding layer in transformer models. Hence, it will introduce an additional layer, the positional embedding layer, which will be located just after the word embedding layer. The original architecture *TravTrans* is based on, GPT-2, did include such a positional encoding lever but research by Kim et al. [9] implied that it wasn't just obsolete but even damaging to the model performance. This claim will be investigated in this research question.

The remaining research questions not mentioned in the list will be using *TravTrans* with its default parameters specified in the Architecture chapter in 3.1.2

3.2 Data Pre-Processing

Data pre-processing is performed by a pipeline that takes the raw input data and modifies it to make it compatible with the baseline transformer model. The original training data *py150k* is in JSON format which has to be pre-processed and traversed in pre-order sequence. This turns the initial tree structure into a sequential structure which can be

fed to the baseline model. The following example is an excerpt of the training dataset in JSON format. It contains AST nodes, their respective values and tree node children.

```
[
{
    "type": "Module",
    "children": [
        1,
        3,
        5,
        7,
        9,
        11
    ]
},
{
    "type": "Expr",
    "children": [
        2
    ]
},
{
    "type": "Str",
  "value": " Provides ``mapping`` of url paths to request handlers.\n"
},
...
```

3.2.1 Dataset

The dataset used for training and evaluation is the *Python 150k dataset*[3]. It contains $150,000$ Python Abstract Syntax Trees that were parsed using a script which is included alongside the dataset. By default, the dataset is split $66:33$, which means that the training dataset contains $100,000$ samples and the evaluation dataset contains $50,000$ samples. However, the distribution was manually changed to a $70:15:15$ split: $105,000$ training

[3] https://www.sri.inf.ethz.ch/py150

samples, $22,500$ validation and $22,500$ test samples. The shuffling and splitting into training, evaluation and test data was performed by a custom Python script. The validation samples are used to evaluate and compare models with different hyperparameters whereas test samples are used to score the model with the best performing hyperparameters.

Each line in the dataset resembles a JSON object representing an abstract syntax tree. The entire dataset has a total $150,000$ lines of ASTs in JSON format.

3.2.2 Vocabulary

A language model requires a fixed sized vocabulary which acts as a database for every word or subword the model is able to interpret. In *TravTrans*, the vocabulary is a Python dictionary which maps the $100,000$ most frequent AST node values to an integer ID. Node values which occur frequently in the training dataset receive a low ID (minimum 0) while less frequent node values receive a higher ID (maximum $99,999$).

There are two different types of nodes in the py150k AST dataset which can be seen in figure 3.1: *Internal* nodes (yellow) and *leaf* nodes (red).

In order for the language model to make a useful prediction for an IDE, it has to predict a leaf node as well as the according parent internal node. Figure 3.1 shows why both prediction are required: If the model was to predict the “a”, the IDE wouldn’t know what “a” is about: It could be a variable name, a constant, a string, an integer etc. If the model predicted “NameStore” alone, the IDE would know the type but not the user-facing value. The same holds true for the next leaf node containing the string “Hello world!”. The IDE would require the type and therefore internal parent node of this string in order to present it accordingly.

Generating the vocabulary consisting of the top $100,000$ tokens, a script runs through the entire dataset and counts how often a specific token (internal *types* or leaf *values*) occurs. These $100,000$ tokens are then stored in a dicitionary which is then pickled[4] and stored.

In addition to the $100,000$ most frequents tokens there are two so-called “special” tokens. One is a padding token, that doesn’t contain any information and is just used to pad an input sequence such that it fills out the entire models input window. The transformer model is set up to ignore this type of token. The other special is used for tokens which are out-of-vocabulary: If a node type or value is processed that does not belong to the top

[4] https://docs.python.org/3/library/pickle.html

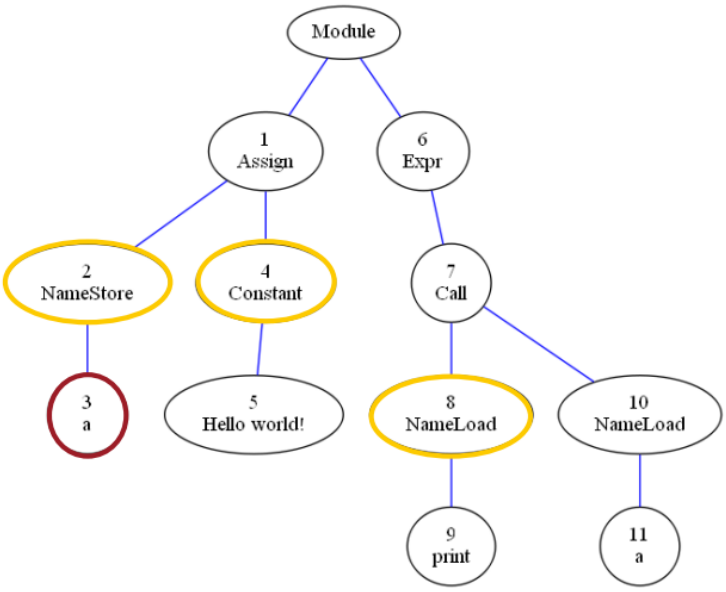

Figure 3.1: Internal nodes vs. leaf nodes

$100,000$ most frequent tokens in the vocabulary, it is deemed "out-of-vocabulary". Instead of being mapped to its own custom ID the node type or value is mapped to "unk_token", a special token assigned to every node not present in the vocabulary. The model will process this special token just like any other token. However, the original information of the node is lost as it can not be recovered from the unknown token. Hence, the *TravTrans* will not be able to predict out-of-vocabulary tokens. Instead, it would simply return an unknown token which could resemble any arbitrary token not present in the $100,000$ token vocabulary.

3.2.3 Tokenizer

Tokenizers deal with the process of splitting up sentences into words and assigning each word to an ID and vice versa. The *TravTrans* tokenizer is implemented as a simple *Word Level Tokenizer* which means that no splitting is performed. The tokenizer treats each AST node as a single token. Even if a leaf node contains a string consisting of several words,

this entire string is treated as one single token and therefore it will receive its own ID if the token fits into the vocabulary. Due to their uniqueness, string values and variable names are not guaranteed to receive their own slot in the vocabulary. It is improbable that the same exact string occurs frequently enough inside the entire dataset to find itself under the top $100,000$ most frequent words. Therefore it's likely that a lot of string values or variable names will be out-of-vocabulary and therefore mapped to the unknown token ID. For example: A string could hold a very random and unique value:

```
some_string = 'This is a very unique and interesting
    string, 1234567890'
```

In AST representation, this string value would be stored in a single leaf node. As each leaf node is treated as a single token, this random string would have to appear repeatedly throughout the entire dataset in order to be among the most frequent $100,000$ nodes. The likeliness of this happening is very low due to the uniqueness of the string.

Such a Word Level Tokenizer is easy to implement and understand but the downside is that unique or rare tokens such as string values or variable names are hard to predict due to the aforementioned out-of-vocabulary issues. During training string values or variable names are mapped to the ID of the special unknown token. If the model was expected to predict a very unique token (for example a unique string) it will most likely return the learned unknown token. The prediction of an unknown token is unusable for the end user of an IDE as it doesn't hold any information. This out-of-vocabulary problem limits the model's prediction capabilities.

RQ4 deals with the out-of-vocabulary issue and evaluates how this problem can be prevented by using a different tokenizer. A *Word Piece* algorithm was applied which splits up less frequent tokens into more frequent subwords. Every input can therefore be split up into such subwords and thus no unknown tokens are required.

3.2.4 Pre-Processing Abstract Syntax Trees

The training and evaluation data comes from the Python 150k dataset[5] which contains abstract syntax trees parsed from Python source code. The data files are in *JSONL* (JSON

[5] https://www.sri.inf.ethz.ch/py150

Line) format and every AST is represented as a JSON object. Each line in the dataset contains one JSON object of one abstract syntax tree.

These abstract syntax tree nodes contain the following items:

- Type, for example *Str*, *NameLoad*, *Expr* and others
- Value, for example strings, integers or variable names
- Sometimes Type **and** value
- children, contains the ids of all children nodes of the current node

As the *TravTrans* expects the trees nodes to only contain type **or** value, the ASTs have to be pre-processed. This means that a script will traverse the entire dataset and look for nodes that have type as well as value entries. As soon as such a node was found, the script will remove the value and add it back as a child node at the 1st position.

As a result all AST nodes contain only type **or** value information.

After being processed, the trees have to be split up. As the *TravTrans* context size[6] is limited to $1,000$, it can only deal with input sequences that are within that range. However, the dataset contains ASTs which are larger than the context size. Those trees will have to be split up into smaller trees of size *max_length* (by default $1,000$), which is done in a script provided by Kim et al. [9]. To retain information between slices, an overlap of size $max\_length/2$ is added between neighboring slices.

Example: An AST with $1,700$ nodes and *max_length* being $1,000$ will be split up into the following slices:

- Slice 1: $0 - 1000$, $u = 0$
- Slice 2: $500 - 1500$, $u = 1000$
- Slice 3: $700 - 1700$, $u = 1500$

The u saves the information until which node the current tree has already been traversed. This detail is required during the training process as the loss for the overlapping and therefore double data should only be calculated once. The slices are stored in a new *JSON* file.

RQ5 ("Which impact does the overlap size have on model performance?") investigates which impact the overlap size has on overeall model performance. By default, authors of

[6]The amount of elements which can be passed to the model in one iteration

TravTrans [9] set the overlap size to 50%. This research question tests alternative overlap sizes and compares the resulting trained model performances.

3.2.5 Traversing Abstract Syntax Trees

For now, the trees are still in JSON format (see the sample JSON in 3.2). However, *TravTrans* expects a sequence of token IDs rather than trees as JSON objects. In order to achieve this, the tree slices are traversed in pre-order sequence.

Pre-order traversal is a tree traversal method that first visits the root node and afterwards the child nodes from left to right. This traversal method suits the code completion task: With a given context up until a certain point, the task is to predict the following token. Conveniently the path up until the point of prediction can be rebuilt by pre-order traversal.

The resulting sequence is then stored in a comma separated text file, one line representing one sequenced AST slice.

3.2.6 Converting AST tokens to IDs

The values that were previously stored in a text file have to be converted into their respective token IDs by the tokenizer in order to make them compatible with the transformer model. The conversion routine reads the previously generated text file and replaces every token with its corresponding ID by looking up the tokens in the vocabulary dictionary. If the token does not exist in the vocabulary because it's less frequent than the top $100,000$ tokens, it will receive the ID $100,000$, a special ID for unknown tokens, which, converted back to a token, reads *<unk_token>*. As all out-of-vocabulary tokens are mapped to the same ID, the token value is lost. The model is able to predict a token with ID $100,000$ but converting it back to a token results in *<unk_token>* which is of no real use.

3.2.7 Capturing AST node IDs

The evaluation process clusters all tree nodes into a handful of evaluation categories in order to be able to distinguish between certain prediction types such as attribute access predictions, function call predictions or string predictions. Otherwise, there would only be a single score for the entire model, preventing a fine-grained view on model performance.

In order to allow such a classification of prediction types, the evaluation script needs two files: File one (called datapoints or *dps.txt*) contains the actual token IDs from the previous step. File two (called node ids or *ids.txt*) contains a lookup table which returns the evaluation category of any node. During evaluation, the evaluation script traverses through *dps.txt* and computes a score by comparing the desired output with the model prediction. Afterwards, the script checks in *ids.txt* in which evaluation category it should store the score:

For example, if the current token was a function call, the score should be stored in the function call category. This step is repeated for each token in *dps.txt*. Finally, all scores are assigned to their respective category.

The file *ids.txt* is generated by traversing through the pre-processed ASTs and storing each node index depending on the node type.

For example, if the 120th node in a pre-processed AST is of type "call", 119 will be added to *ids.txt* under the category *call_ids* which contains all IDs of function calls (119 is used instead of 120 as 0 is the first index).

3.3 Implementation

3.3.1 Framework

The model was implemented with PyTorch (v1.9.0) in Python (v3.8.X) and the original model can be found in the authors (Facebook Research) GitHub repository[7]. Their implementation is a fork of a basic GPT-2 PyTorch implementation [8]. PyTorch is a Python based machine learning framework which allows developers to quickly implement and distribute models such as neural networks or transformers. The *TravTrans* model implementation is publicly available at the Facebook Research Teams GitHub repository. Training and evaluation routines were implemented from scratch as they were not publicly available and afterwards published in an open source GitHub repository[9].

[7] github.com/facebookresearch/code-prediction-transformer
[8] github.com/graykode/gpt-2-Pytorch
[9] https://github.com/derochs/code-prediction-transformer

3.3.2 Hardware

All models were trained on the TU Darmstadt STG provided “Romulus” and “Remus” servers. Both run on dual Intel Core i9-7900X with 10 cores each and 128 GB of RAM. The GPUs used for training and evaluation are CUDA capable *NVIDIA GeForce GTX 1080 Ti* with 12 GB of memory respectively.

4 Model Training and Evaluation

This chapter introduces the training and evaluation routines which were implemented from scratch as they were not provided with the *TravTrans* model from Kim et al. [9]. In the machine learning field the structure of training and evaluation procedures are oftentimes very similar.

The training loop feeds training data to a machine learning model which will update its weights according to the errors it makes and therefore "learn" from the training data.

The evaluation loop on the other hand uses the trained model to make predictions which are then compared to the desired output. Both values, the prediction and the expected value, are then compared by calculating a score according to a specific metric. Despite the known general structure of training and evaluation routines, there are several training parameters which were not clearly communicated by Kim et al. [9] and therefore could lead to different results. For example, it remains unknown if and how the dataset was split into training/test/evaluation datasets, whether or not the dataset was shuffled before splitting and when model weights are optimized during training.

The following paragraphs describe the training and evaluation processes in more detail.

4.1 Model Training

4.1.1 Dataset

Datasets in machine learning scenarios are oftentimes very large and unwieldy. PyTorch provides a *Dataset* object, which simplifies the process of handling the entire dataset. It is used as an intermediary between the actual dataset file and the model: The Dataset class can fetch certain desired parts from the dataset file and pass it to the model, without the developer having to create dataset slices manually.

4.1.2 DataLoader

During the training process, training data is fed to the model. The model takes the input and generates an output. Afterwards, an error (*loss*) between prediction and desired target value (*label*) is calculated. Finally, the model parameters (*weights*) are updated in order to improve the model's prediction capabilities. This routine is applied on every element (*sample*) of the training dataset.

In case of *TravTrans* the training data consists of $105,000$ abstract syntax trees. Instead of feeding the model separate ASTs one-by-one, the model can make use of matrix operations in order to process multiple ASTs in parallel. In theory the entire dataset could be fed to the model in one iteration. However, in real world applications the GPU memory would be a limiting factor. The dataset subset size (*batch size*) should be large enough in order to process as many samples in parallel as possible but small enough not to cause any out-of-memory errors due to hardware limitations. A *DataLoader* deals as an intermediary between the entire dataset consisting of tens of thousands of ASTs and the model. It allows the specification of the batch size to be fed to the model.

Such a DataLoader is included in the PyTorch framework and can be used for training and evaluation purposes or whenever a subset should be sampled from a large amount of data. A batch size of 4 was selected for the training and evaluation scripts as the hardware used couldn't provide enough memory for larger batch sizes. This means that instead of $105,000$ training iterations processing one AST per iteration, the model processes 4 ASTs per iteration resulting in a total of $26,500$ iterations.[1]

In addition to allowing the sampling of batches the DataLoader provides the ability to define *collate functions* which allow the transformation of the sampled batch data before being fed to the model. In case of *TravTrans*, the collate function transforms the training data into four different files:

1. The input sequence consists of AST samples of which the last element is missing. An AST of n elements is turned into an AST of length $n-1$ by omitting the last element.

2. The target sequence consists of AST samples of which the first element is missing. Just like the input sequence, an AST of n elements is turned into an AST of length $n-1$ by omitting the first element.

Example: The original AST sequence

[1] One iteration means the transport of n samples through the model during training, each iteration resulting in a weight update

```
 1. Module, 2. Assign, 3. NameStore, 4. a, 5. Constant,
6. Hello world!, 7. Expr, 8. Call, 9. NameLoad, 10. print,
 11. NameLoad, 12. a
```

would be collated into input sequence:

```
 1. Module, 2. Assign, 3. NameStore, 4. a, 5. Constant,
6. Hello world!, 7. Expr, 8. Call, 9. NameLoad, 10. print,
 11. NameLoad
```

and target sequence:

```
1. Assign, 2. NameStore, 3. a, 4. Constant, 5. Hello world!,
 6. Expr, 7. Call, 8. NameLoad, 9. print, 10. NameLoad,
 11. a
```

This simplifies the comparison between model prediction and target at each index of the AST tree: At position 1 the model makes a prediction. The prediction should ideally be equal to the next word. As the target sequence is the input sequence shifted to the left by 1, the following word can be found in the target sequence at index 1 as well. This process repeats for each index in the entire AST: The model makes a prediction at index i of the input sequence which is then compared to index i in the target sequence.

3. The overlap sequence which is generated by the collate function contains information on overlapping AST slices. This helps the model to identify which indices are part of overlapping ASTs and therefore duplicates. In order to prevent the repeated loss computation for overlapping indices, the overlap sequence informs the model which indices are to be ignored during the weight update process (*optimization*).

4. The evaluation ID sequence contains the information which node ID belongs to which evaluation category. This sequence is used during evaluation in order to categorize scores into specific groups such as for example “String prediction” or “NameStore” predictions. This process is explained in more detail in section 4.2.

4.1.3 Training Implementation

Training during machine learning describes the process of a model making a certain prediction, comparing it to the desired output by calculating the distance and "learning" by adjusting its weight parameters such that the next prediction introduces a smaller error.

The data pre-processing pipeline introduced in the previous chapter 3.2 prepares the two necessary files for training, *dps.txt* containing token IDs and *ids.txt* containing AST node IDs. These are then fed to a dataloader which generates the four files: Input sequence, target sequence, overlap sequence and evaluation ID sequence.

The actual training process consists of several nested loops: The outermost loop handles *epochs*: One epoch represents the process of feeding the entire dataset to the model, computing the error (distance between model prediction and target value, or *loss*) and adjusting the model weights. If a model was trained for 10 epochs it means that the model has processed the entire training dataset for a total of 10 times.

The next level contains the batch loop. Instead of processing the dataset line-by-line (which would be equal to sub-tree-by-sub-tree), a dataloader can load several entries from the dataset at once. Using a larger batch size has a positive impact on the training time but at the same time requires more GPU memory.

During each batch loop iteration input sequence, target sequence, overlap sequence and evaluation ID sequence are loaded onto a PyTorch device, a GPU in this case. As mentioned in the previous chapter they contain the model input, the target that should be matched, an additional extra information to prevent double loss calculation on equal data due to the sub-tree overlaps and ids for evaluation purposes. The input sequence makes one forward pass through the transformer model which returns a vector of token predictions. Afterwards, the *loss* is calculated. The loss function used is called *cross entropy loss*. Cross entropy loss is a frequently used when comparing two probability distributions [10]. As the model output is a probability distribution over the possible next words it is reasonable to this kind of loss metric.

The loss is required in order to determine how much the prediction differs from the actual label (the expected or desired output). This calculated loss value is afterwards propagated back through the entire model in order to compute gradients on the models weights. These gradients indicate a weight how to adjust such that the loss will be reduced in the next iteration. This process of calculating the loss and propagating it back through the model in order to fine-tune the weights is called *backpropagation*.

One uniqueness of the *TravTrans* training is that the loss is not always computed for the entire input sequence. As mentioned before there are potential overlaps in sub-trees. If the loss was calculated on two neighboring subtrees it would calculate the elements in the overlap section twice. In order to avert this issue the *overla sequence* tells the model which parts of subtrees to ignore during the loss computation.

Finally, the model parameters are stored. This allows the model to be reconstructed with the exact same parameters at any time later on.

4.2 Model Evaluation

4.2.1 Evaluation Metrics

For the purpose of evaluating the correctness of a predicted number of solutions, the "mean reciprocal rank" (MRR) is an adequate metric [10]. It is oftentimes used in related work, for example in research by Bodden et al. [2] and Karampatsis et al. [8]. It makes sense to keep this metric such that comparisons between models in the same field can be performed with the same type of metrics.

The score is calculated by taking the reciprocal value of the correctly predicted tokens rank:

$$MRR = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{rank_i}$$

where n is the total number of calculated ranks and $rank_i$ is the ith rank of n.

MRR Example: The following example shows how the MRR is computed. In the first sequence, the given word is "Hello" and the expected following word (target label) is "World". However, the model makes four predictions, "Ship, Boat, World, House", the correct one being at rank 3. This means that the reciprocal rank is $\frac{1}{3}$. The reciprocal rank is summed up over all examples and finally divided by the number of total reciprocal ranks in order to receive the mean reciprocal rank.

Sequence	Predictions	Target Label	Reciprocal Rank
Hello	Ship, Boat, World, House	World	$\frac{1}{3}$
Welcome to the	Jungle, Forest, School	Jungle	$\frac{1}{1}$
The quick brown	dog, fox, bear	fox	$\frac{1}{2}$
		Mean Reciprocal Rank:	$\frac{\frac{1}{3}+1+\frac{1}{2}}{3} = 0.61$

MRR is a good measurement for the code completion task. Instead of simply checking for a correct prediction it will also take into consideration how well the correct prediction was ranked.

RQ3("Should additional metrics be tracked for model evaluation?") deals with the model evaluation and introduces additional evaluation categories which could be interesting for future model evaluation implementations. Calculating MRR scores for these new categories could give additional insights into the model performance and answer questions which were previously unanswered, one being "How well does the model perform on String prediction?" which is highly relevant for **RQ4**.

4.2.2 Evaluation Implementation

The evaluation routine is similar to the training routine. Pre-processing on the evaluation data is performed with the same pipeline as for training data, resulting in two files: *dps.txt* for the token IDs and *ids.txt* for evaluation purposes. The evaluation data stems from the initial dataset split described in section 3.2.1.

Figure 4.1 visualizes how an input sequence and model output may look like in a simplified setup. Each column i contains the word at index i and the predictions the model makes for the next index $i + 1$ (model output). As the transformer model processes all columns at once, the entire model output is available after a single iteration. As shown in the figure the final model output is a matrix with dimensions $m \times n$, m being the vocabulary size and n being the length of the input sequence.

This setup allows the MRR to be calculated at each index i in order to compute a single overall model score. However, the scores could also be categorized into specific prediction types. This is where the *ids.txt* file is created for: It contains the IDs of each predefined evaluation category such as "Call IDs" for function calls, "Assign IDs" for variable assignments and more.

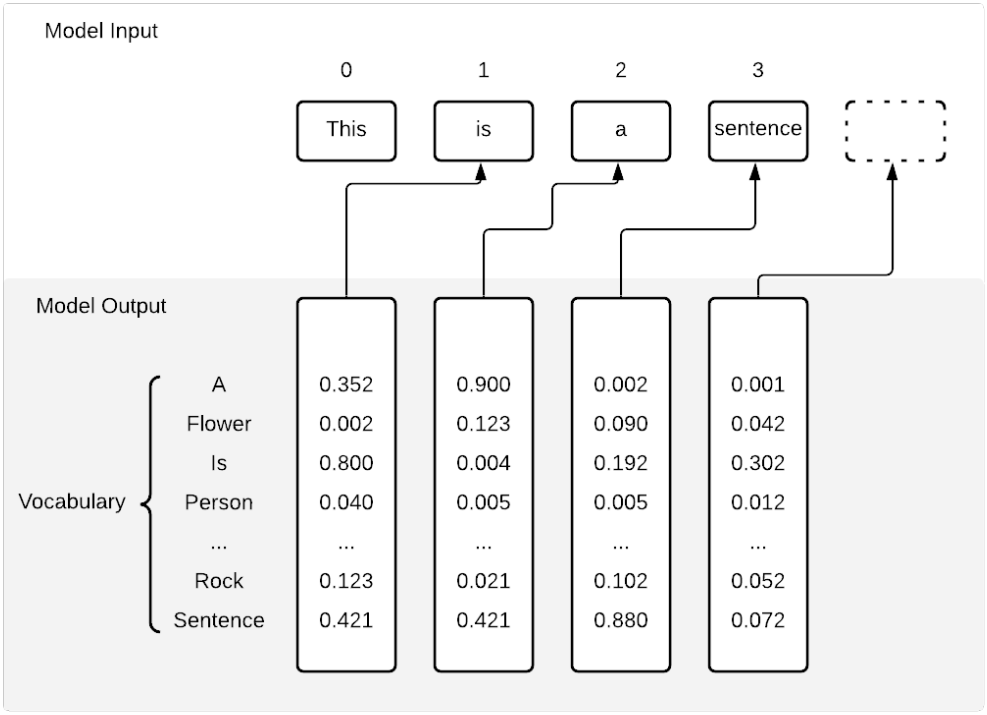

Figure 4.1: Visualized model output

For example, the *ids.txt* file could specify that IDs 0, 1 and 2 are of prediction type "Call IDs". As soon as the evaluation script is at index 0 it will read the information from *ids.txt* that at the current position a prediction for a "Call ID" will be made. The evaluation script will therefore compute the MRR and store it in the category for "Call IDs". After the entire evaluation process the mean of each category is taken in order to receive the overall "Call ID" score. Finally, this allows a general overall score to be split up into specific prediction types such as attribute access, numeric constant, variable name/module name, strings and more. Instead of comparing overall model performance one could now compare how the model performs on specific prediction types and answer questions such as "How well does the model perform on string prediction?"

Internal node predictions are assigned to the following categories:

- Function calls
- Assignments
- Return
- List
- Dictionary
- Raise

Leaf node predictions are assigned to the following categories:

- Attribute access
- Numeric constant
- Variable name, module name
- Function parameter name
- String

4.3 Model Usage

The main objective of the transformer model investigated in this thesis is to complete source code. Ideally the model should be found in the backend of some IDE that wants to make use of the code completion capabilities such as VisualStudio Code by Microsoft. But when it comes to the actual usage of the model, there are several aspects which are relevant to the practical code generation task.

As the trained model not only generates a single prediction for the upcoming word but rather generates a probability distribution for the entire vocabulary, the first aspect is the selection of the right word: Given a specific context, the model outputs a list of probabilities for each word in the model vocabulary. There are different types of approaches on how to choose the best word.

The naive and greedy approach would be to simply select the word with the highest probability. The advantage is that this task does not require any complex computation. However, a downside of this greedy approach is that the word predictions may end up in loops as there is no notion of randomness involved during the word generation process.

At some point the model may suggest that the next word is “X” and the word afterwards is “Y”. The following word with the highest probability may be “X” again, and the model would end up generating a sentence consisting of “X Y X Y X ...”. Such a loop can not be broken by the greedy approach as it will always select the next word to be the one with the highest probability.

These loops can be broken by **randomly sampling** the next word rather than simply taking the word with the highest probability. This can be imagined as putting each word of the vocabulary into a bucket, the probability of each word indicating how many times the word occurs in the bucket. A word with high probability is therefore more likely to be drawn than a word with a low probability. Words with high probabilities will still be preferred, however there is still a chance for a lower probability word to be selected and thus break the aforementioned loops.

Yet another approach is the **beam-search** which is useful when generating longer sentences rather than single words. Beam search requires a number of beams `n_beams` and the search depth `d`. Instead of just looking at the probability of the next word, the beam search algorithm generates entire sentences of length `d`, computes the overall probability of this sentences and compares them in order to figure out which sentence has the highest overall probability.

Beam Search Example:

Figure 4.2 shows an exemplary beam search. The beam search in this setup has 2 beams and a search depth of 2. At first a context is fed to the model, in this case it’s the word “The”. The model processes the context and produces next word predictions out of which it only keeps the 2 with the highest probability. The 2 stems from the 2 beams the model should generate. In the next step the model will concatenate the previous context with the word in every beam which results in two different contexts: “The dog” and “The nice”. Using these new contexts the model will again make next word predictions which result in “The dog has” and “The nice woman”. The process stops as the search depth of 2 is reached. Finally, the probabilities of each beam are multiplied together. The beam with the higher probability is the one that will be returned to the user, in this case it would be “The dog has” while the other beam is omitted.

Finally, there are numerous ways on how to choose a prediction from a probability distribution. As the goal of this thesis code completion task is the generation of the immediate next token or word, the random sampling approach seems to be the most feasible: It prevents the looping issue from the greedy approach and doesn’t require as

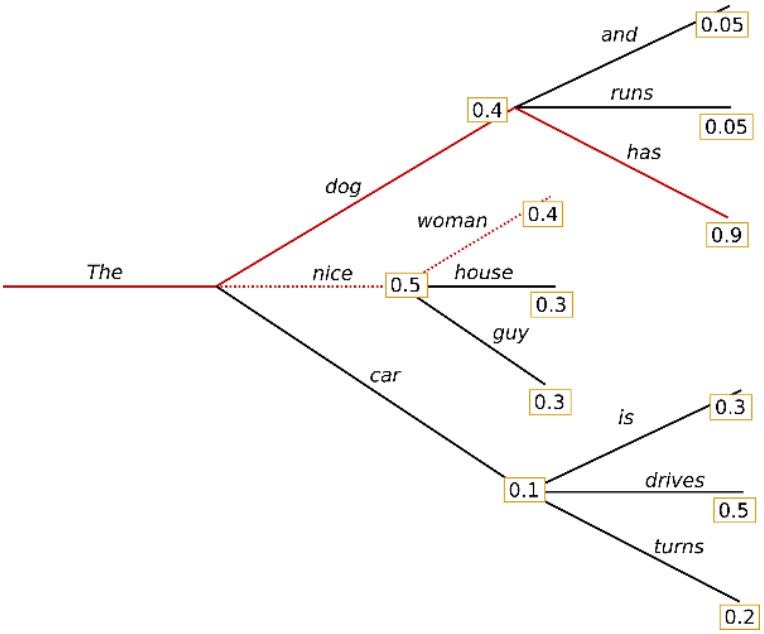

Figure 4.2: Beam Search Example from https://huggingface.co/blog/how-to-generate

much overhead as beam-search does, which was designed for longer text predictions anyways.

5 Study Results

5.1 Research Questions regarding Baseline Model Properties

TravTrans introduced by Kim et al. [9] is a well performing transformer model, especially when it comes to code completion [5]. This is made possible by using a state of the art transformer model introduced by Radford et al. [15] and optimizing it for abstract syntax trees, introducing as little overhead as possible. This section deals with research questions investigating the capabilities of *TravTrans* in detail and evaluating downsides and potential improvements of the model. Finally, the results should provide a solid base for future work in the intelligent code completion field and offer easily accessible starting points for extensions.

5.1.1 RQ1: How does the baseline model perform when reproduced?

As training or evaluation scripts were not made publicly available by the work of Kim et al. [9], they had to be implemented from scratch. Not having access to the original training and evaluation routines means that the custom implementations will be different and therefore most likely end in different results compared to the ones presented alongside the *TravTrans* model.

Instead of using the results provided by Kim et al. [9] as baseline results, this thesis will treat the results that were reproduced by the self implemented evaluation scripts in this research question as baseline results.

According to the evaluation implementation introduced in 4.2.2 the evaluation script is able to calculate scores for each evaluation category mentioned in the list shown in 4.2.2. Both training (4.1.3) and evaluation routines (4.2.2) vary from the implementation by Kim et al. [9] due to unavailable of the original code and so do the results.

Two different prediction types were presented in 4.2.1 which are special to the *TravTrans* model: *Leaf node predictions* and *internal node predictions*. Internal nodes contain syntactic types and leaf nodes contain token values [9] which are both required to make sensible code predictions.

For example, if an input was "x = " and the model is expected to complete it to "x = 2", it is not sufficient to predict the value "2" only. The "2" could be of type string, variable name or any other type. That's why the model has to predict the parent node of "2" which determines its structural type. In this case, the parent node should be of type "Num".[1]. Any leaf node that is predicted requires an additional prediction of its parent node in order to make a valid code completion. Internal node predictions on the other hand do not require such overhead. If the model were to predict a list operation like "ListLoad" or "ListStore" it just has to correctly predict this node type.

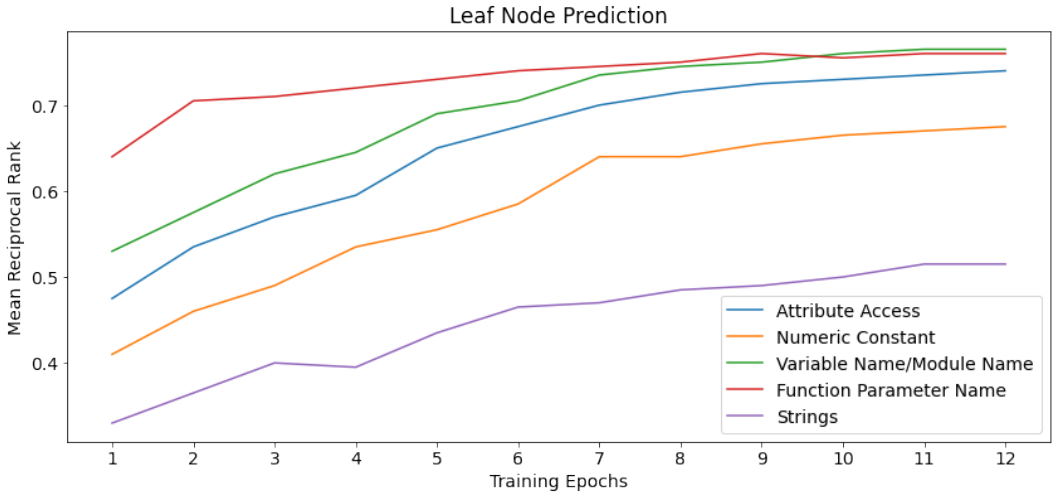

Figure 5.1: Scores for leaf node prediction

Figure 5.1 shows how the accuracy for leaf node predictions grows over increasing training time. The y-axis shows the MRR score for each leaf node prediction. The x-axis shows the current training epoch[2]. As mentioned before, leaf node predictions actually require two predictions, meaning that there are actually two scores. Figure 5.1 visualizes the mean of both scores. The results for internal node predictions can be seen in 5.2.

[1] Example provided by Kim et al. [9]

[2] During one training epoch the entire training dataset is fed through the model. Having 12 epochs means that the entire training dataset was fed through the model 12 times

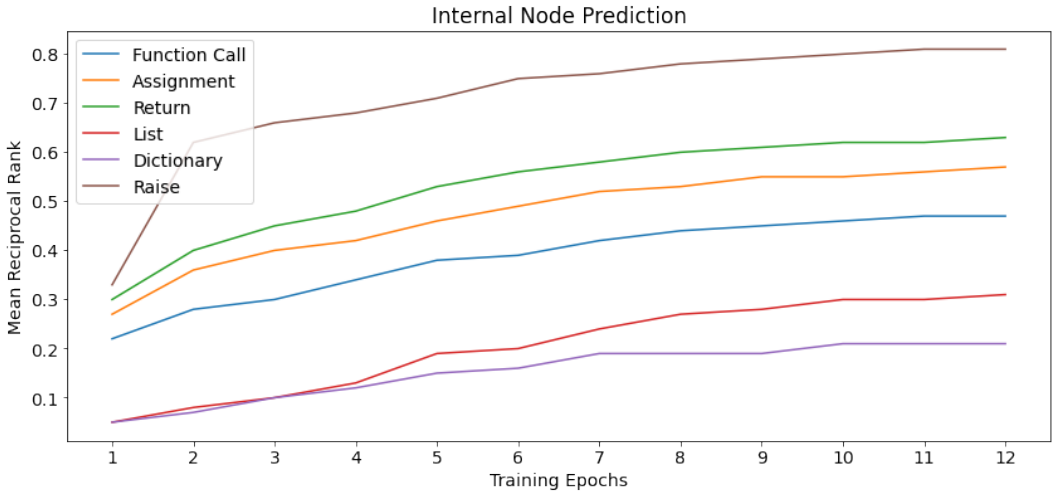

Figure 5.2: Scores for internal node prediction

According to Kim et al. [9] the original *TravTrans* model was trained for 11 epochs. Our *TravTrans* implementation on the other hand was trained for a total 12 epochs in order to have a broader view on the training process. We assumed that training can be stopped if a single epoch doesn't improve the model accuracy over 1%. Figure 5.3 shows that we could stop training after 10 epochs which is close to the original work by Kim et al. [9]. Most of the metrics fall below the 1% improvement mark after the 10th epoch. The same applies to the relative score improvements of internal node predictions which are visualized in figure 5.4.

As conclusion to this finding, 10 epochs is found out to be our default setting during training routines which means that all of the following *TravTrans* model alternatives will be trained for 10 epochs as well.

Finally, tables 5.1.1 and 5.1.1 show the final scores of the reproduced model compared to the original values from *TravTrans* presented by Kim et al. [9]. While most of the scores and trends are similar (for example internal node prediction for "Dictionary" are both rather low compared to the other values), there are a few outliers. The reproduced model seems to perform slightly worse than the original model on internal node predictions, but for leaf node predictions it's the other way around. There are several aspects that could cause these differences:

Scoring leaf node predictions and internal node predictions: For once, it is unclear how

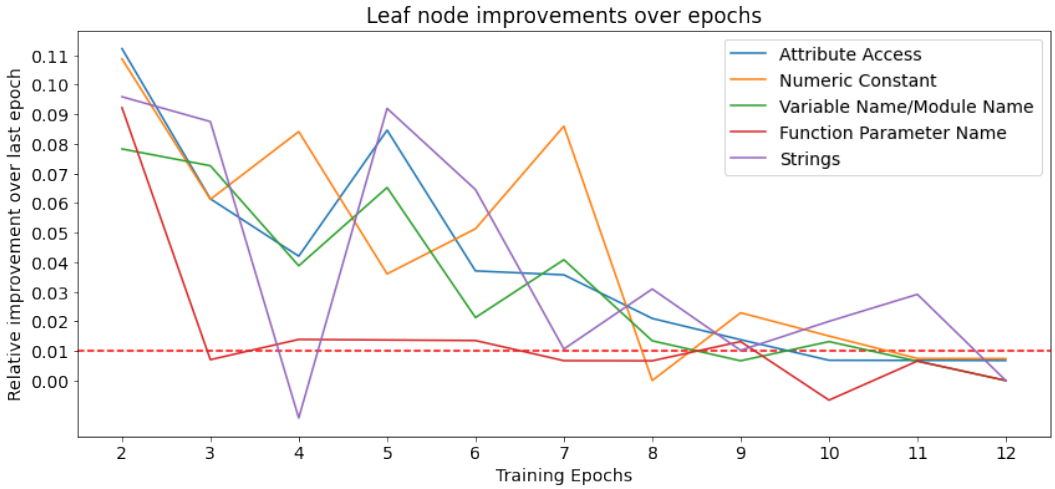

Figure 5.3: Relative leaf node score improvements over epochs

the original work merged the leaf node prediction score and the parent node prediction score. While this thesis used the mean of both values, the work by Kim et al. [9] could've used a different measure.

Alternative model parameters: Another reason for dissimilar results could be model parameters: Instead of a learning rate of $1e-3$, the reproduced model used a learning rate of $1e-5$ and instead of an Adam optimizer, the reproduced model uses the AdamW optimizer which handles weight the decay slightly different. The changes were applied as the settings presented by Kim et al. [9] resulted in even worse results.

Dataset preparation: Additionally, it remains unclear if and how the original *py150k* dataset was split into train/test/evaluation datasets. It is also unclear if the dataset was shuffled before the splitting process which is a common procedure during data preparation.

All of the aforementioned aspects have impact on model training and could contribute to the different results. Finding out the reason on why the results differ as much as they do is hard without having access to the entire training and evaluation routines used by Kim et al. [9]. However, despite the different results, one should keep in mind that the results of the following research questions can be applied on the original *TravTrans* model.

For example: If a change in the *TravTrans* model architecture results in a model performance increase of 10% it can be expected that the original *TravTrans* model trained with

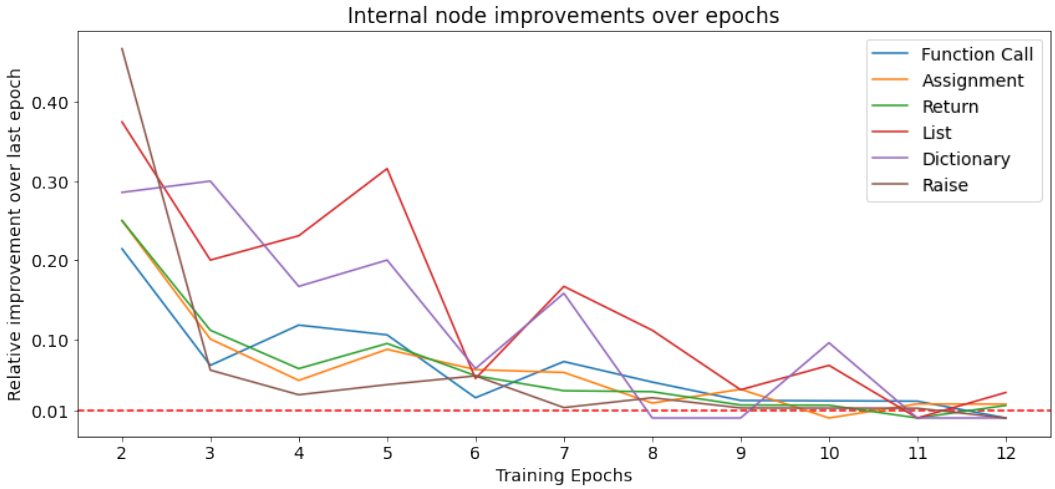

Figure 5.4: Relative internal node score improvements over epochs

the unpublished training scripts would experience a similar improvement. This is due to the fact that the only difference between the "original" *TravTrans* and the "custom" *TravTrans* is that they've been trained on a different routine. However, the architecture is identical and therefore should react equally to changes.

	Internal Node Predictions	
Application	Original Model	Reproduced Model
Function call	0.88	0.47
Assignment	0.78	0.57
Return	0.67	0.63
List	0.76	0.31
Dictionary	0.15	0.21
Raise	0.63	0.81

Table 5.1: Baseline model type prediction after training for 12 epochs

	Leaf Node Predictions	
Application	Original Model	Reproduced Model
Attribute access	0.6	0.74
Numeric constant	0.57	0.68
Variable name, module name	0.63	0.77
Function parameter name	0.65	0.76
String	N/A	0.52

Table 5.2: Baseline model value prediction after training for 12 epochs

RQ1 Takeaway:
Despite the fact that training and evaluation routines had to be implemented from scratch, the results from the work by Kim et al. [9] and our implementation are similar but vary in some places. This is to be expected as using an alternative training due to lack of information results in an alternative model. In addition to reproducing the model a good training time of 10 epochs was determined by checking for the relative model performance improvement after each epoch. In this case if the model did not improve more than 1% compared to the previous epoch, the model training would stop.

5.1.2 RQ2: Can the baseline model be improved by adjusting architectural settings?

A transformer is a model consisting of many parameters which can be tweaked individually, such as number of layers, embedding size, context size and number of attention heads to name a few. In order to determine the optimal parameter values a lot of training and testing has to be performed. The authors of *TravTrans* made certain choices for their *TravTrans* model without justification on why specific values were selected.

This research question aims to investigate some parameters of the model architecture in order to discover whether the default *TravTrans* values work well or if there are alternative settings able to improve the overall model performance. The scores from RQ1 are treated as baseline scores and will be used for comparison.

The two parameters investigated for this research question are the **number of decoder blocks** and the **embedding size** used in the transformer model.

Looking back at the GPT architecture in figure 2.7 the number of layers represents how many times the blue box in the center (*decoder*) is repeatedly stacked on top of itself.

Radford et al. [15] chose a number of 12 decoder blocks while *TravTrans* uses 6 decoder blocks. More decoder blocks introduce more parameters and therefore imply that training could result in a more fine-tuned model.

The second parameter investigated is the embedding size: An embedding size is the size of the internal vector representation of a token in the transformer model. In case of *TravTrans* each AST node is encoded into an integer ID by the tokenizer. Afterwards this integer is fed to the model which maps the ID to a fixed size vector. The vector size is also known as embedding size and contains information on the encoded AST node tree. A larger embedding size implies that it could hold more information.

The impact of the number of layers was investigated by training and evaluating the baseline model with a different number of layers, ranging through 1, 3, 6, and 9 layers, 6 being the default value. The other aspect implements the baseline model with different embedding sizes: 120, 240, 300 and 540, 300 being the default embedding size. It is to be expected that a larger embedding size or a larger number of decoder blocks will increase training time while also increasing model performance.

Figure 5.5 shows side-by-side how much training time is required and which impact the two aspects mentioned above have on the model score average. The red vertical line marks the default values selected by Kim et al. [9]. The following sections will go into more detail on both charts.

Number of layers

The left chart in figure 5.5 compares the impact of the number of layers on the overall model score and overall training time. Close attention should be paid on the proportions between both y-axis: The left y-axis represents the overall model score (in green) and ranges from 0.4 to 0.7, an increase of over 75%. The right y-axis shows the required training time (in blue) and ranges from 200 to $3,500$, an increase of over $1,600\%$. The first takeaway is that a small change in model performance will result in a large change in training time.

With an increasing number of layers the model performance in green seems to quickly raise up until 6 layers. The model relative performance increase when using more layers (9 layers) is rather small compared to the previous improvements. However, the training time doesn't seem to be impacted as it seems to increase linearly with an increase of number of layers.

The red dotted line shows the default number of layers (being 6) which seems to be a good trade-off between performance and training time: Using more layers, for example 9, would result in a performance increase of 5% while increasing training time by 20%.

Embedding size

The right graph displays the impact of embedding size on the overall model score. Similar to the previous graph the score increases in large steps up until the default value of 300. Using the larger embedding size of 540 improves the model performance by 5% but requires more than 55% more time during training. It seems reasonable to choose the value suggested by Kim et al. [9] as the large trade-off for a slightly better score is not easy to justify.

Concluding, the architectural settings suggested by [9] seem to be reasonable although no reasoning was given on the specified numbers such as number of layers or embedding size. There's always room for improvement when it comes to the model performance but most likely those improvements will come with a trade-off such as exponentially increasing training time or other issues such as overfitting[3].

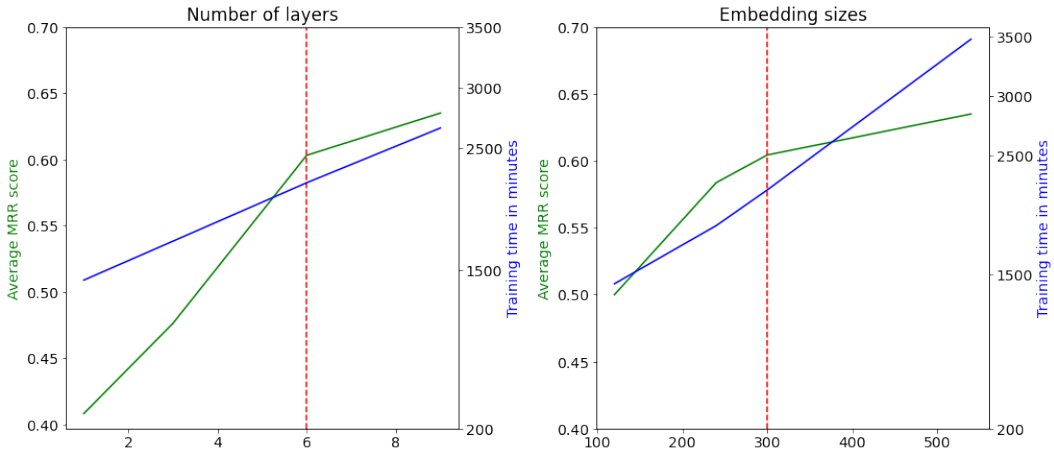

Figure 5.5: Alternative number of layers and embedding sizes

[3] If a model is trained for "too long" or on too little repetitive training data it starts to adapt to the training set. The model may perform increasingly better on the training data but increasingly worse on real world data as it's "overfitted" on the training data.

RQ2 Takeaway:
Increasing embedding size or number of layers will most likely improve the overall model performance. However, these changes come at a cost: An increase of embedding size or number of layers will also greatly increase training time. Increasing the layer count from 6 to 9 increases the model performance by 5% and training time by 20%. Increasing the embedding size from 300 to 540 increases model performance by 5% and training time by 55%. A trade-off between model performance and training time has to be found which Kim et al. [9] have achieved by using the optimal settings (6 layers, embedding size 300) as default. This finding could even be applied to alternative dataset sizes: In order to find optimal model parameters, various parameters should first be tested out on a a smaller subset of the dataset. Afterwards the best performing parameters could be applied tested on the original dataset.

5.1.3 RQ3: Should additional metrics be tracked for model evaluation?

Background

TravTrans assigns model scores to specific categories. This enables the evaluation to answer questions like "How good does the model perform on string prediction?" or "How good does the model perform on constant number prediction?". This essentially allows a more fine-grained comparison between different models for code-completion tasks.

For example, instead of comparing the overall MRR between two models the evaluation categories allow the comparison of prediction accuracies on specific prediction types such as strings, constants or variable names. Table 5.3 shows which node types are tracked in the baseline *TravTrans* evaluation process.

Internal nodes	Leaf nodes
Call	attr
Assign	Num
Return	NameLoad/NameStore
ListComp/ListLoad/ListStore	NameParam
DictComp/DictLoad/DictStore	
Raise	

Table 5.3: AST nodes tracked in baseline evaluation

Extension

The types mentioned in table 5.3 are only a small subset of all node types that can be found in the *py150k* dataset. Table 5.4 shows additional node types which are not considered in the baseline evaluation. Nodes which make up less than 0.1% of the entire dataset are excluded from the table. They are less relevant as they resemble rare corner cases and apparently don't show up too often in source code.

Internal nodes	Leaf nodes
AttributeLoad/AttributeStore	Str
body	keyword
Expr	FunctionDef
decorator_list	alias
arguments	ImportFrom
args	ClassDef
defaults	
Index	
If/orelse	
SubscriptLoad	
TupleLoad/TupleStore/TupleDel	
Return	
Dict	
CompareEq/CompareIn/CompareIs	
BinOp(Add/Mod/Mult/Sub/Div/Pow/Bit)	
For	
bases	
Import	
SubscriptStore	
UnaryOpNot	
Assert	
Module	
ExceptHandler	
TryExcept	
handlers	
Slice	
type	

Continued on next page

Table 5.4 – continued from previous page

Internal nodes	Leaf nodes
BoolOpAnd/BoolOpOr comprehension	

Table 5.4: Untracked AST nodes

The node types contained in 5.4 were previously not tracked during evaluation but some of them are worth adding, especially string predictions as these are relevant for RQ4.

Listing 5.1: Sample source code

```
person.age = 7
person.dimensions = (0.5, 1.8)
if person.age == 7:
    return 'Person age is 7'
```

Listing 5.1 shows an exemplary source code. With the original *TravTrans* evaluation categories, the model could not be evaluated on the largest part of the listing. The following list introduces new categories which are able to capture more of the code during evaluation.

AttributeLoad/AttributeStore
As the names imply, "AttributeStore" and "AttributeLoad" are used when storing and loading attributes. In listing 5.1 `person.age = 7` assigns a constant 7 to an "AttributeStore" of attribute "age" of name "person". Adding "AttributeStore" and "AttributeLoad" to the list of tracked node types could allow additional insight on how well the model performs on predicting attribute loads and stores.

If/orelse
Another interesting topic are conditionals: The "if" condition in listing 5.1 compares the person age with another integer. The original *TravTrans* evaluation categories would not be able to evaluate how well the model performs on generating conditionals such as "if" and "orelse".

CompareEq/CompareIn/CompareIs
Additionally to the previous conditionals, several node types can be added which deal with comparisons: "CompareEq", "CompareIn" and "CompareIs" could allow the scoring

of `==`, `in` and `is` operator prediction. Listing 5.1 contains such a comparison in the "if" conditional.

TupleDel/TupleLoad/TupleStore
While *TravTrans* tracks lists and dicts, it does not track tuples. This could be easily implemented by adding tracking for "TupleLoad", "TupleStore" and "TupleDel" node types. Tuples can also be found in the person dimension in listing 5.1.

The following list contains leaf node types from table 5.4 which are worth tracking as well:

String
The baseline evaluation does not evaluate string predictions. However, this thesis added the string prediction in every research question evaluation as it is highly important, especially for RQ4 which deals with the impact of using a subword tokenizer. As the out-of-vocabulary issues are most apparent in string predictions, this type is one of the most important ones to be tracked in its own evaluation category.

Results

Adding additional node types to be tracked during model evaluation allows deeper insights about its prediction accuracy. Figures 5.6 and 5.7 visualize the newly suggested evaluation types introduced by RQ3 alongside the baseline evaluation types.

RQ3 Takeaway:
There are several node types which are interesting and could be implemented into the evaluation script. Regarding internal node types, AttributeLoad/AttributeStore, If/orelse, CompareEq/CompareIn/CompareIs and TupleDel/TupleLoad/TupleStore can give additional insights on internal node prediction capabilities. When it comes to leaf node types, string types are very important to add, especially for RQ4 which deals with out-of-vocabulary issues most commonly found with string values.

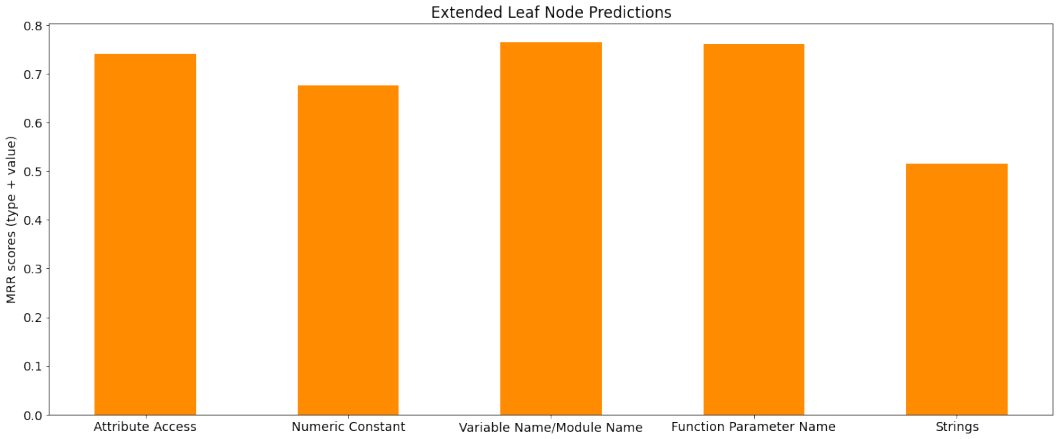

Figure 5.6: Model evaluation with new leaf node types introduced in RQ3

5.2 Research Questions regarding Data Pre-processing Adjustments

5.2.1 RQ4: Can the out-of-vocabulary issue be reduced by using an alternative tokenizer?

Background

The *TravTrans* model uses a tokenizer on *WordLevel* basis. This means that it maps words or in this case tokens to the ID assigned to them in the vocabulary. In the context of this thesis a token is the value of an AST node. This could be an integer, a string containing multiple words and special characters or a node type such as “NameLoad”. The *TravTrans* vocabulary is limited to $100,000$ tokens meaning that only the top $100,000$ most frequent tokens can be converted into IDs. Therefore, the IDs range from 0 to $99,999$. All tokens which are out-of-vocabulary will be converted into the unknown token ID $100,000$ which converts back to *<unk_token>*. This means that out-of-vocabulary tokens will lose their value and cannot be predicted by the model. The model may generate an *unk_token* which could stand for basically anything and therefore holds no true value.

The advantage of a WordLevel tokenizer is its simplicity and predictability: One AST node can be encoded into one integer ID and vice-versa. It can be implemented by a

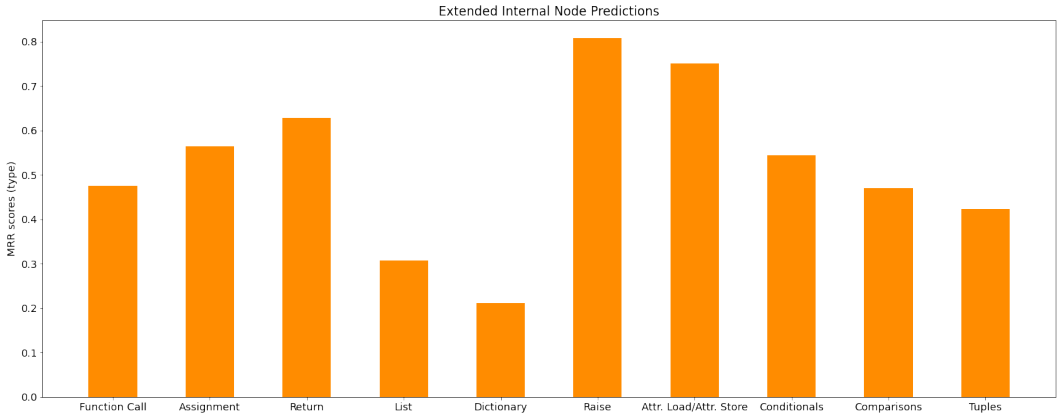

Figure 5.7: Model evaluation with new internal node types introduced in RQ3

simple data structure such as a dictionary and brings very little complexity to the data pre-processing phase.

The out-of-vocabulary issue[9, 19] is one big downside of the WordLevel tokenizer: A token that is less frequent than the top $100,000$ tokens will be mapped to the unknown token and therefore lose its information. In the code completion task an AST node could contain custom values such as strings and variable names. The likelihood of the exact same string or custom variable name appearing multiple times in the dataset and therefore being in the top $100,000$ most frequent tokens is very low. Strings are therefore prone be out-of-vocabulary. Variable names oftentimes follow coding conventions and therefore are more likely to be in-vocabulary than strings. However, custom variable names are at risk of being out-of-vocabulary as well.

As the proposed model is intended to make code predictions in an IDE it would be beneficial if it could predict variable names and maybe even strings. Therefore it makes sense to improve the prediction performance for those types by tackling the out-of-vocabulary issue.

The investigated approach replaces the WordLevel tokenizer with a *WordPiece* tokenizer [19]. A tokenizer based on WordPiece splits up tokens into the most frequent subwords and stores those instead of entire words. In this case a string would not be mapped directly to an integer ID but split up into subwords, which are then mapped to IDs.

The following example shows an input sentence and how it would be tokenized by (1)

the WordLevel tokenizer and (2) the WordPiece tokenizer.

```
Input: 'The quick brown fox jumped over the lazy dog'

(1) Output WordLevel tokenizer:

    TOKEN      TOKEN_ID
'<unk_token>' (100,000)

(2) Output WordPiece tokenizer:

TOKEN  TOKEN_ID
'The ' (2519)
'##quick' (16901)
'## br' (19582)
'##own' (819)
'## f' (399)
'##ox' (1582)
'## j' (12052)
'##ump' (3318)
'##ed ' (423)
'##over the ' (23004)
'##lazy' (20668)
'## do' (3333)
'##g' (140)
```

While the WordLevel tokenizer completely loses all of the string data by mapping it to an unknown token ID, the WordPiece tokenizer encodes it into subwords which can be concatenated in order to retrieve the original sentence. Subwords can be identified by the double hash symbol prefix ##. The proposal is that a GPT-2 model trained on data tokenized using the WordPiece instead of the WordLevel algorithm could predict strings by generating subwords instead of simply returning an unknown token that holds no specific value.

The disadvantages of the WordPiece tokenizer are *complexity* and *determinism*: Encoding one AST node could result either in a single token or numerous subwords. This makes it harder to calculate and fit tokens into the model context window. Using the baseline WordLevel tokenizer, one could count $1,000$ AST nodes and they would fit perfectly into

the model with context size $1,000$. The WordPiece tokenizer on the other hand does no longer encode tokens in a ratio of $1:1$ as one token could result in numerous outputs (in the example above a single string token is tokenized into 13 subwords). Therefore, the entire dataset has to be tokenized and then split into subwords. Afterwards the subwords can be counted in order to fit them into the model context size.

The second problem is determinism: If the WordLevel based model predicts one token, it's assured to be the next AST node. However, when predicting tokens using the WordPiece approach, the generated prediction could be the correct following token or just a part of it. In any case an additional check has to be performed in order to assure that the prediction is complete or if more subwords will follow. The model could also get stuck in a loop generating subwords which could introduce problems during training and evaluation.

Implementation

A new tokenizer pipeline is built using Huggingface's Tokenizer library[4] which introduces a modular tokenizer pipeline framework, allowing us to tailor a custom tokenizer for our needs. The pipeline consists of a Pre-Tokenizer and the actual tokenizer model. A Pre-Tokenizer is used to detect different words, for example by declaring that words are separated by whitespace or punctuation. A so-called *CharDelimiterSplit* Pre-Tokenizer was used, which detects words at given delimiters. As delimiter the comma character (",") was selected, as all tokens were separated via commas during the data pre-processing step. This means that the Pre-Tokenizer will iterate through the entire dataset and treat everything between two commas as a "word"; Despite the name "word" it actually holds the entire value of an AST node which could even be strings. This approach is similar to the baseline WordLevel tokenizer which also treated one AST node as token. The actual tokenizer model selected for the pipeline is WordPiece, a subword tokenizer that splits less frequent words into frequent subwords [16].

Training the Tokenizer

At first, the entire *py150k* dataset is prepared by using the same pre-processing routine from *TravTrans* described in section 3.2.4 which ensures that each AST node only contains either a type or a value but never both. Afterwards, every comma character has to be escaped from the tree nodes (such as strings), as commas are used for the tokenizer as a

[4] https://huggingface.co/docs/tokenizers/python/latest/index.html

delimiter between nodes. Raw node values and types are exported into a new file which is $150,000$ lines long. Each line resembles a pre-order traversed AST and consists of its node types and values, separated by the delimiter of choice, in this case commas. This file is used to train the tokenizer, using the trainer script provided by the tokenizer framework. The vocabulary size is set to $30,000$. The trained tokenizer is then exported as a JSON file for later use.

Training the Model

For training, the py150k dataset is split up into a train dataset containing $105,000$ samples. Similar to the dataset for the tokenizer training, the ASTs are pre-processed in order to ensure that each AST node contains either type or value but never both. In the baseline *TravTrans* data pre-processing, the next step would be splitting up ASTs larger than $1,000$ nodes into subtrees. With the subword tokenizer however, a single node could be converted into multiple subwords, meaning that a tree with $1,000$ nodes could end up in a collection of far more tokens and therefore wouldn't fit into the model context size of $1,000$. This means that before splitting up the ASTs, the types and values of each AST have to be encoded by using the previously trained tokenizer first. The output of tokenizing an AST is a list of integers, each integer being the ID of a subword. As the length of the integer list is final and will not increase, the splitting process can finally be applied. The converted ID sequence is split into chunks of length $1,000$ with an overlap of 50%.

Finally, the GPT-2 baseline model is trained from scratch on the dataset based on the new tokenizer implementation. The model was trained over 10 epochs with a batch size of 4.

Results

Figures 5.8 and 5.9 compare the leaf node and internal node prediction scores between *TravTrans* with WordLevel tokenizer and *TravTrans* with the WordPiece tokenizer. The hypothesis established that a subword tokenizer could reduce the out-of-vocabulary issue. This means that the prediction of less common and therefore out-of-vocabulary words should be improved.

One important candidate for out-of-vocabulary issues are string predictions: It is unlikely that there are sufficient identical strings in the dataset to make it into the vocabulary of the top $100,000$ most frequent words. Strings are oftentimes very unique and oftentimes don't follow any conventions. The baseline WordLevel approach would simply map the

string to an unknown token and the string value would be lost. The subword tokenizer however is able to split up strings into frequent subwords and iteratively generate large bodies of strings by concatenating those subwords.

The results in 5.8 show that the string prediction (leaf prediction) accuracy doubled by using the subword tokenizer. Variable name prediction (leaf prediction) did also experience a slight accuracy increase. Other leaf node predictions show very similar results to the WordLevel tokenizer approach, without any notable differences.

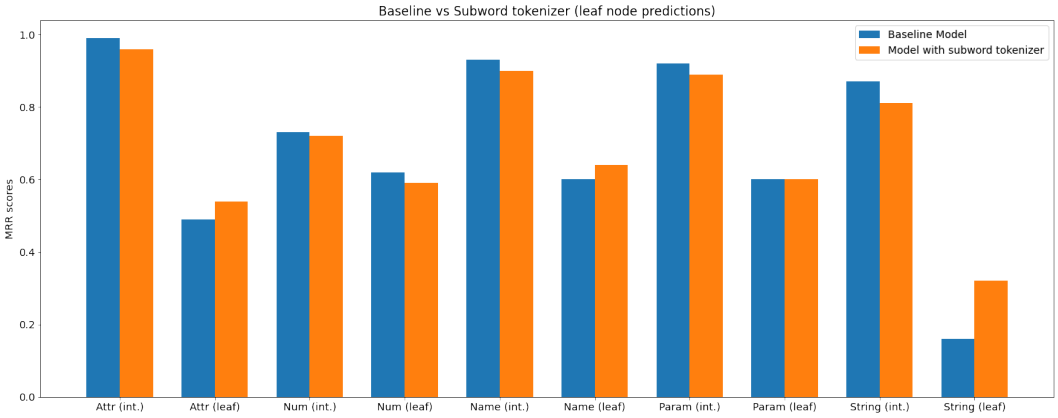

Figure 5.8: Comparison between baseline model and baseline model with subword tokenizer for leaf node prediction

What stands out is that the subword tokenizer seems to perform poorly on some internal node predictions, namely “Return”, “Raise” and especially “Dict”. Possible reasons on why these predictions may perform worse with the subword tokenizer are discussed in the following paragraphs:

Dict predictions: The evaluation category for “Dict” contains the following node types: “DictComp”, “DictLoad” and “DictStore”. Despite their similar naming, the subword tokenizer splits “DictLoad” and “DictStore” into two subwords (“Dict” and “##Store” for example). “DictComp” on the other hand is not split during tokenizing process, implying that this type is more frequent in the dataset and therefore doesn’t have to be split up. When the model tries to make a “Dict” prediction, it’ll most likely be successful predicting “DictComp”. However, if the model is supposed to predict “DictStore” or “DictLoad” the model is more likely to have a lower accuracy as it would have to make two correct predictions: First “Dict” and afterwards “Load” or “Store”. This issue could explain the low accuracy for “Dict” predictions.

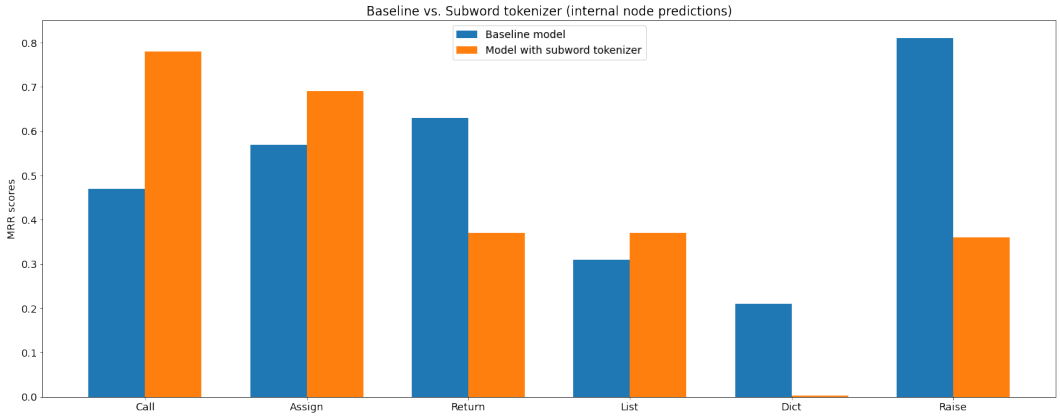

Figure 5.9: Comparison between baseline model and baseline model with subword tokenizer for internal node prediction

Raise predictions: Regarding the low accuracy of "Raise" predictions it should be mentioned that "Raise" nodes are relatively rare in the python150k dataset. From $92,758,587$ AST nodes in the entire dataset, only $156,416$ nodes are of type "Raise" which means that there are only 0.1% "Raise" samples in the entire dataset. This makes it hard for the model to train the correct usage of this specific node type and therefore results in a lower prediction accuracy. The WordLevel tokenizer assigns the word "Raise" the ID 58. A lower ID means that the word occurs more often in the dataset and the baseline tokenizer vocabulary contains $100,000$ words. The WordPiece tokenizer introduced in RQ4 on the other hand assigns it the ID 660 from a total of $30,000$ words. The rankings of both vocabularies can differ as they have different sizes and in addition to that the subword tokenizer vocabulary stores, as the name implies, subwords.

Finally, "Raise" is lower ranked in the subword tokenizer vocabulary and the subword tokenizer vocabulary is smaller than the baseline tokenizer vocabulary. This may result in the worse prediction accuracy for this node type.

Return predictions: Similar to the "Raise" type, the "Return" type is ranked lower in the subword tokenizer vocabulary than in the baseline vocabulary which could have an impact on the prediction accuracy.

When it comes to the code completion task, the leaf node predictions are more important than internal node predictions as the user of the IDE will most likely expect suggestions which are stored in AST leaf nodes, for example string values and variable names. This

makes the subword tokenizer a well performing alternative to the baseline tokenizer used in RQ1.

RQ4 Takeaway:
The subword tokenizer introduced in this research question is able to reduce out-of-vocabulary (OOV) issues. It turns out that the model using the subword tokenizer doubles the prediction accuracy for strings as well as variable name predictions which were mainly affected by the OOV issue. A few internal node predictions perform worse but leaf node predictions are more important for code completion tasks. Ideally, a combination of two models, one using a WordLevel tokenizer for internal node predictions and the other one using a WordPiece tokenizer for leaf node predictions, could greatly improve the overall prediction performance by merging the advantages of both models.

5.2.2 RQ5: Which impact does the overlap size have on model performance?

Background

During the baseline data pre-processing routine larger ASTs sequences (traversed in pre-order sequence) are split into smaller slices with a maximum size of $1,000$ tokens and a sliding window of 50%. This research question aims to figure out why an overlap of 50% was chosen by Kim et al. [9] and which impact a larger or smaller overlap size would have on the training time or model performance respectively. In order to answer those questions a new data pre-processing routine has been implemented which allows the specification of the overlap size in contrast to the fixed overlap size of 50% used in the original *TravTrans* approach.

Example

Figure 5.10 shows three simplified scenarios of tree splitting with different overlap sizes. In this setup the maximum length of a sub-tree is set to be 10 (the real world *TravTrans* model specifies a length of $1,000$ tokens).

The original AST can be seen at the very top, it contains 15 tokens.

The overlap size has direct impact on the step size of the sliding window which can be seen in the formula $step\_size = max\_length * (1 - overlap)$. Applying this formula on the three different examples (75%, 50% and 30%) in figure 5.10 result in the step sizes 2,

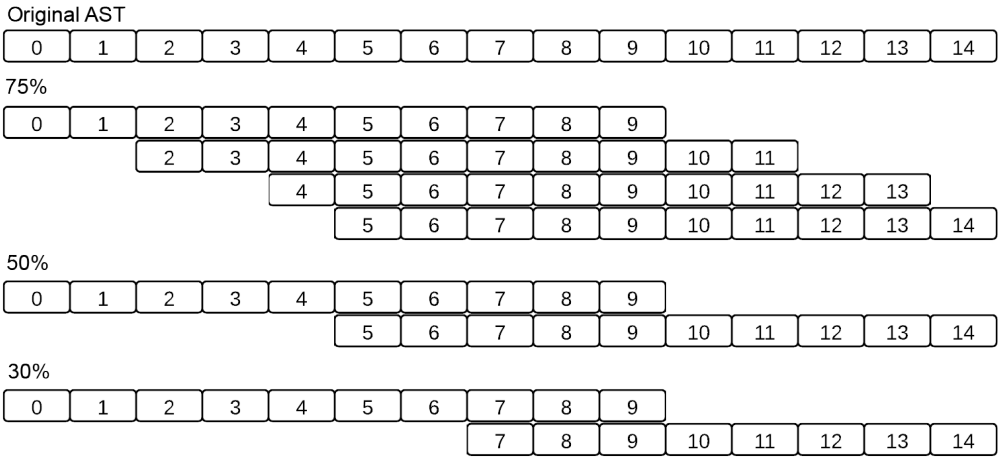

Figure 5.10: Simplified representation of how an AST is split with different overlap sizes

5 and 7 respectively. A first takeaway from this example is the fact that a larger overlap results in a smaller step size and therefore more subtrees.

An overlap of 75% and maximum length of 10 results in four subtrees for a source tree of 15 nodes.

In the same scenario an overlap of 50% and 30% results in two subtrees.

Implementation

In order to evaluate the impact of different overlap sizes an alternative data pre-processing script was implemented which allows the specification of overlap size. The overlap sizes which were investigated are 75%, 50%, 30% and 10%. This resulted in four differently sized training datasets which were then used to train four baseline transformer models over a span of 10 epochs. Finally, the four trained models were evaluated and compared.

Results

The example in figure 5.10 suggests that using a larger overlap size results in a larger number of subtrees. This assumption can be validated by comparing the number of lines of all four training datasets. As training datasets contain one subtree per line, more subtrees mean more lines and a larger overall file size. Table 5.2.2 compares the different sizes of training datasets generated with different overlap sizes. The comparison shows that a smaller overlap size results in less AST slices and therefore a smaller dataset size. A large overlap on the other hand contains more redundant data, more AST slices and therefore a larger dataset size, which will presumably negatively impact training time, which is evaluated in in the following paragraphs.

Overlap Size	Training dataset size	Training dataset line count/AST slice count
10%	1.4GB	174,870
30%	1.6GB	189,882
50%	1.9GB	220,162
75%	3.0GB	321,724

Table 5.5: Comparison of training dataset sizes generated with different overlap sizes

After having proved that a larger overlap size results in a larger dataset size, the following task would be to analyze the impact of the dataset size on the training time. As expected, a larger dataset results in an increased training time.

This relation is visualized in the training time vs. training file line count comparison in figure 5.11. It shows the expected direct correlation between training time and line count with slight deviations on the 30% and 50% overlap model setups. The teal colored plot shows how the training time increases exponentially with an increasing overlap size, while the orange plot shows how the training dataset size increases accordingly.

Finally, in order to evaluate the actual impact of different overlap sizes on the overall model performance, the four models trained on different training datasets were evaluated on the same evaluation dataset. The scores computed during evaluation are plotted in figure 5.12. The teal colored plot shows the training time required by models using different overlap sizes. The orange plot resembles the overall average model scores.

While the training time seems to increase exponentially with increasing overlap sizes, the relative model accuracy improvements seem to act as the inverse of the training time plot: Smaller overlap sizes strongly increase model performance while only slightly increasing

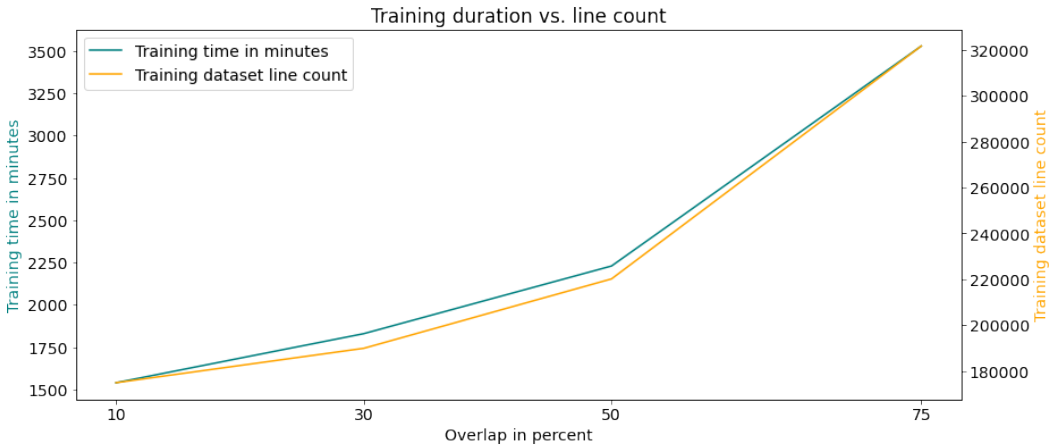

Figure 5.11: Training time compared to training dataset size

training time. Larger overlap sizes on the other hand result in great increases in training time while only slightly increasing model performance. This means that an optimal point between both extremes regarding overlap sizes has to be found in which training time is relatively low and model performance is relatively high.

The red dotted line marks the default overlap value chosen by Kim et al. [9] for the baseline model, 50%.

Increasing the overlap size above this line results in a huge training time increase of over 58%, while the model score experiences a relatively small improvement of less than 3%. The selected default value of 50% is a good candidate for a balanced trade-off between training time and model accuracy. Even a smaller overlap size of 30% would seem feasible which would reduce training time by almost 22% while the accuracy would only decrease by 3%. Depending on the scenario the overlap size could be adjusted accordingly: Rolling out the model for production use would require a high accuracy. Training would probably be only required once, which justifies a single long training time. In another scenario in which many models are trained and evaluated numerous times on expensive hardware (for example for research purposes), a smaller overlap size would be advantageous in order to speed up the training process while only slightly lowering accuracy.

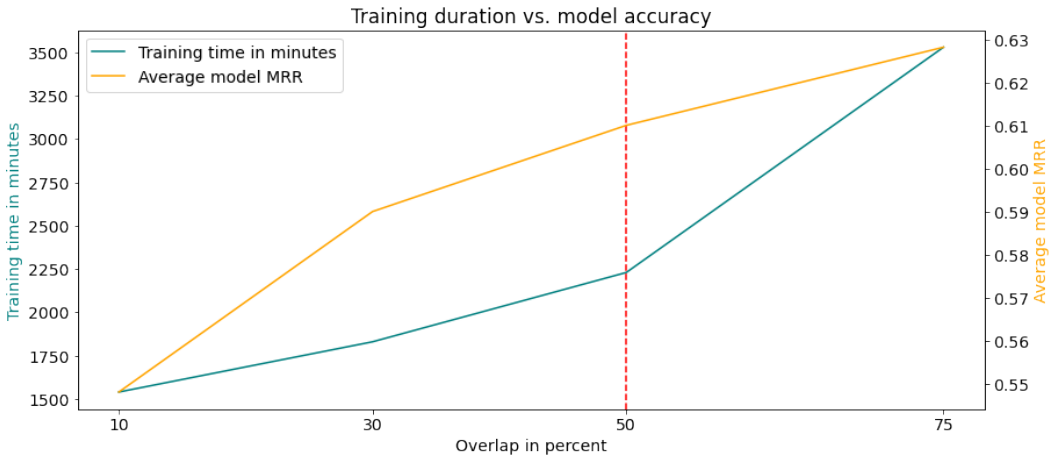

Figure 5.12: Training time compared to model accuracy

RQ5 Takeaway:
A larger overlap results in a larger dataset and therefore increases training time. However, a larger overlap also results in a higher model accuracy. The *TravTrans* authors Kim et al. [9] found a good balance between training time and model performance by using a default overlap size of 50%. However, an even better choice would be to reduce the overlap size to 30% which would reduce training time by almost 22% while only costing a decreased accuracy of 3%. An even higher model accuracy can be achieved with overlap sizes larger than 50%, however this comes with the downside of exponentially increasing training times.
Depending on the use case the overlap size can be adjusted to either reduce training time during development or increase model performance on a production environment.

5.3 Research Questions regarding Model Adjustments

5.3.1 RQ6: Does positional embedding improve the performance of the model?

Background Research by Irie et al. [6] and Kim et al. [9] implies that the additional positional information added by positional embedding or positional encodings are not required in transformers and could even be damaging to the overall model performance.

Kim et al. [9] took this detail into account for *TravTrans* and removed the positional embedding layer from the otherwise largely unmodified original 6-layer GPT-2 model. In order to evaluate the implication by Irie et al. [6] the positional embedding layer was re-implemented for this research question into the *TravTrans* model and compared to the baseline model without positional embedding.

TravTrans is a fork from a public GitHub project of a PyTorch GPT-2 implementation with slight modifications such as the removal of the positional embedding layer. The original positional embedding implementation has been re-introduced into the baseline model by fetching the implementation from the original GitHub project[5].

Results

Figure 5.13 and 5.14 visualize the leaf and internal node prediction accuracy of the baseline model with and without positional embedding respectively. It can be seen that the model with positional embedding achieves overall slightly worse results than the baseline model without positional embedding. There is no instance in which the model with positional embeddings outperforms the baseline model and therefore there is no use case for the code prediction task in which it would make sense to implement the positional embedding layer.

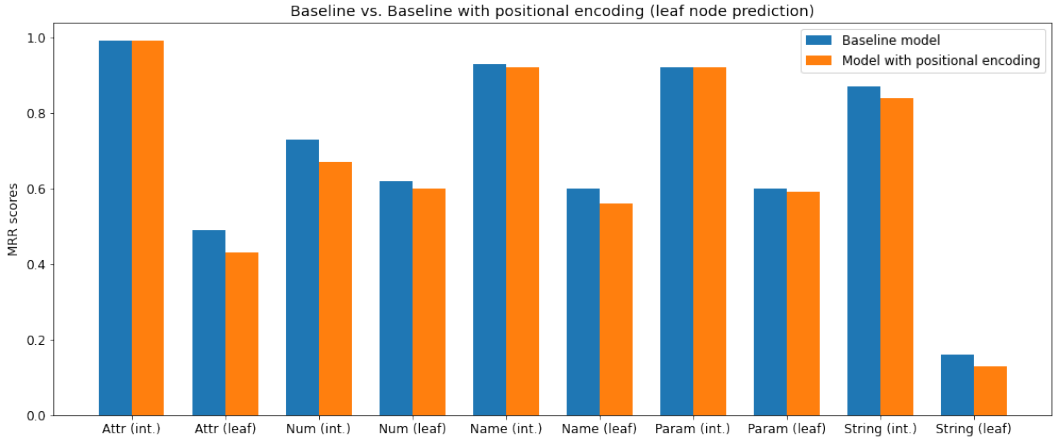

Figure 5.13: Baseline model vs model with positional encoding leaf node prediction

[5]https://github.com/graykode/gpt-2-Pytorch

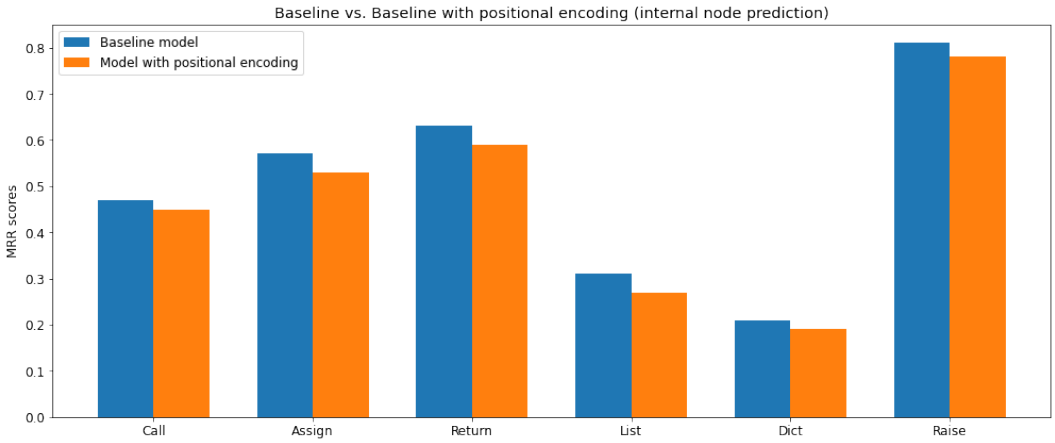

Figure 5.14: Baseline model vs model with positional encoding internal node prediction

Irie et al. [6] assume that due to the architecture of autoregressive[6] models such as GPT-2 in which one word is provided to the model in each time step (model repeatedly generates one word, adds it to inputs and generates next word) the model should be able to recognize this structure and therefore receive some notion of position. It should learn that the input sequence grows from left to right as new words are appended to the right. The authors proved this assumption by training and comparing two models, one with positional encoding and another one without.

L	Positional encoding	Params in M.	Perplexity		
			Train	Dev	Test
12	Sinusoidal	243	61.8	63.1	66.1
	None		55.6	**60.5**	**63.4**
24	Sinusoidal	281	55.6	58.0	60.8
	None		52.7	**56.6**	**59.2**
42	Sinusoidal	338	51.2	55.0	57.7
	None		50.5	**54.2**	**56.8**

Table 5.6: Comparison from Irie et al. [6] between models with and without positional encoding. Lower perplexity indicates a better prediction quality.

[6] An autoregressive model generates an output which is appended to the input for the next iteration. This allows an autoregressive model such as GPT-2 to iteratively generate entire sentences and paragraphs.

Table 5.6 shows how models without positional encoding perform slightly better. The paper also provides an answer on why positional encoding is not required but may even be damaging to the prediction quality as can be seen in 5.6: The first layer of a model with positional encoding focuses on the next 2 to 3 words while the model without positional encoding focuses on the next upcoming token and even learns to ignore functional words (examples from Irie et al. [6] are “the”, “and” and “to”).

This demonstrates that a model without positional encoding focuses more on the next word while models with positional encoding add a slight “blur” to their view.

RQ6 Takeaway:
The assumption that positional encoding hurts the model performance was confirmed by re-introducing the positional embedding layer from the original GPT-2 model. The model without positional encoding learns to automatically focus on the next word while the model with positional encoding is forced to focus on the first few words and therefore has a slight “blur” which may be the reason for its inferior performance. While positional encoding may be useful for forcing the attention of the language model on a certain part of the input sequence, it is not required for the task of text generation as the attention is already focused on the correct part of the sequence.

6 Threats and Future Work

This thesis investigated the usage of a natural language model, GPT-2 in particular, for code completion tasks. Instead of feeding raw source code as text to a transformer model, the code was first parsed into abstract syntax trees which contain additional structural information. Afterwards the trees were traversed in pre-order sequence in order to feed them to the model which only accepts sequential inputs. This approach opens up possibilities for future work which will be discussed in this section.

Feeding alternative source code representations to transformers The initial work by Kim et al. [9] shows that a transformer model trained on abstract syntax trees achieves superior results in comparison to a transformer model that treats programming languages as natural language by training on "plain" source code.

Natural language models were primarily designed to work on natural language tasks such as machine translation or text summarization of news articles or books. However, natural language is oftentimes ambiguous and a single sentence can be understood in different ways. Source code on the other hand is unambiguous and the actual syntactical structure can be represented as an abstract syntax tree [4]. Feeding this type of source code representation to a transformer rather than treating source code as natural language allows it to learn a programming language model which is closer to reality, contain less inaccuracies that may origin from language ambiguities and finally achieve better results.

Future work could try to find the ideal source code representation which outperforms abstract syntax trees:

Applying static code analysis such as control flow graphs could be a potentially viable extension to abstract syntax trees. A control flow graph is a graph in which the vertices represent code blocks and the edges represent the control flow between code blocks. Control flow graphs may therefore introduce even more dependencies which may not be present in abstract syntax trees.

Another source code representation may be so-called "CodeGraphs" which show dependencies between code entities such as methods and classes[1]. One major hurdle would be the conversion from a graph into a sequence as transformer models usually expect sequential inputs.

What type of source code representation performs best and the best way to turn source code graphs into sequences remains to be evaluated in future work.

Alternative tree traversal methods Abstract syntax trees have to be flattened into a sequence because transformer models usually expect a sequential input. In order to achieve this the authors of *TravTrans* [9] suggested the ATSs to be traversed in pre-order sequence. Future work could investigate if alternative tree traversal methods such as post-order or in-order traversal may be feasible for alternative tasks.

TravTrans underlying GPT-2 model usually generates words at the context end. This makes the traversal order crucial for generating code predictions. Traversing an abstract syntax tree in post-order sequence for example would mean that the last node in the sequence is the trees root node which is always of type "Module" and will therefore probably have little to no impact on the next word prediction. Post-order traversal would therefore seem inappropriate for generating tree nodes because sentence of the tree fed to the model would be in reverse.

If such an implementation is actually damaging to model performance remains to be evaluated as possible future research.

[1] https://github.com/xnuinside/codegraph

7 Conclusion

In the course of this work several aspects of *TravTrans*, a transformer for code completion, developed by Kim et al.[9] was re-implemented, adjusted and modified in order to discover potential weaknesses and possible improvements. By testing alternative settings to the ones selected by Kim et al. [9] it showed that the default values are a fair trade off between good model accuracy and short training times. This trade off also reveals that the model could achieve slightly better results at the cost of exponentially increased training times. This finding is applicable on the model architecture properties such as number of layers and embeddings size investigated in RQ2 as well as on the sliding window overlap size investigated in RQ5.

RQ4 shows that the usage of an alternative tokenizer, a subword tokenizer, greatly improves the prediction accuracy of highly unique values such as attribute names and strings. Due to their uniqueness these values are most likely to be omitted from the model vocabulary in order to make space for more frequent words. By introducing a subword tokenizer however, such values can be split up into common subwords which can be learned by the model. Finally, this allows the model to produce strings by generating and concatenating subwords. Previously the model could only generate string values and attribute names which are under the top $100,000$ most frequent words in the vocabulary.

Most of the findings show that there is no single optimal model for the task of code completion: Certain adjustments greatly improve the overall model accuracy at the cost of increased training time. In another case, using completely different components (such as the WodPiece tokenizer in RQ4) enable increased accuracy for very specific tasks such as string predictions. This takeaway reveals the possible necessity of a customizable and modular transformer model which, depending on its current task, can be equipped with the best suiting tools in order to achieve optimal scores.

Finally, artificial intelligence plays an increasingly large role in the field of code generation which cutting edge projects such as GitHub's Copilot show. However, the many risks

and problems that may be introduced by the relatively new field of “intelligent” code completion remain to be seen.

Bibliography

[1] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. *Neural Machine Translation by Jointly Learning to Align and Translate*. URL: http://arxiv.org/pdf/1409.0473v7.

[2] Eric Bodden et al., eds. *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. New York, NY, USA: ACM, 2017. ISBN: 9781450351058. DOI: 10.1145/3106237.

[3] Tom B. Brown et al. *Language Models are Few-Shot Learners*. URL: http://arxiv.org/pdf/2005.14165v4.

[4] Casey Casalnuovo, Kenji Sagae, and Prem Devanbu. *Studying the Difference Between Natural and Programming Language Corpora*. URL: http://arxiv.org/pdf/1806.02437v1.

[5] Nadezhda Chirkova and Sergey Troshin. "Empirical Study of Transformers for Source Code". In: *In Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece* (2020). DOI: 10.1145/3468264.3468611. URL: http://arxiv.org/pdf/2010.07987v2.

[6] Kazuki Irie et al. "Language Modeling with Deep Transformers". In: *Interspeech 2019*. ISCA: ISCA, 2019, pp. 3905–3909. DOI: 10.21437/Interspeech.2019-2225.

[7] Navdeep Jaitly et al. "An Online Sequence-to-Sequence Model Using Partial Conditioning". In: *Advances in Neural Information Processing Systems*. Ed. by D. Lee et al. Vol. 29. Curran Associates, Inc., 2016. URL: https://proceedings.neurips.cc/paper/2016/file/312351bff07989769097660a56395065-Paper.pdf.

[8] Rafael-Michael Karampatsis et al. "Big code != big vocabulary". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. Ed. by Gregg Rothermel and Doo-Hwan Bae. New York, NY, USA: ACM, 2020, pp. 1073–1085. ISBN: 9781450371216. DOI: 10.1145/3377811.3380342.

[9] Seohyun Kim et al. *Code Prediction by Feeding Trees to Transformers*. URL: http://arxiv.org/pdf/2003.13848v4.

[10] Li Li, Milos Doroslovacki, and Murray H. Loew. "Approximating the Gradient of Cross-Entropy Loss Function". In: *IEEE Access* 8 (2020), pp. 111626–111635. DOI: 10.1109/ACCESS.2020.3001531.

[11] Elizabeth D. Liddy. "Natural Language Processing". In: *Encyclopedia of Library and Information Science, 2nd Ed. NY. Marcel Decker, Inc.* (2001).

[12] Li Lucy and David Bamman. "Gender and Representation Bias in GPT-3 Generated Stories". In: *Proceedings of the Third Workshop on Narrative Understanding*. Ed. by Nader Akoury et al. Stroudsburg, PA, USA: Association for Computational Linguistics, 2021, pp. 48–55. DOI: 10.18653/v1/2021.nuse-1.5.

[13] Sebastian Proksch, Johannes Lerch, and Mira Mezini. "Intelligent Code Completion with Bayesian Networks". In: *ACM Transactions on Software Engineering and Methodology* 25.1 (2015), pp. 1–31. ISSN: 1049-331X. DOI: 10.1145/2744200.

[14] Alec Radford and Karthik Narasimhan. "Improving Language Understanding by Generative Pre-Training". In: 2018.

[15] Alec Radford et al. "Language Models are Unsupervised Multitask Learners". In: 2018.

[16] Mike Schuster and Kaisuke Nakajima. "Japanese and Korean voice search". In: *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2012, pp. 5149–5152. DOI: 10.1109/ICASSP.2012.6289079.

[17] Alexey Svyatkovskiy et al. "IntelliCode compose: code generation using transformer". In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Ed. by Prem Devanbu, Myra Cohen, and Thomas Zimmermann. New York, NY, USA: ACM, 2020, pp. 1433–1443. ISBN: 9781450370431. DOI: 10.1145/3368089.3417058.

[18] Ashish Vaswani et al. "Attention is All you Need". In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[19] Yonghui Wu et al. *Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation*. URL: http://arxiv.org/pdf/1609.08144v2.