A User Evaluation of UniXcoder Using Statement Completion in a Real-World Setting

Jorit de Weerdt
Supervisors: Maliheh Izadi, Arie van Deursen
EEMCS, Delft University of Technology, The Netherlands

June 19, 2022

A Dissertation Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering

A User Evaluation of UniXcoder Using Statement Completion in a Real-World Setting

Jorit de Weerdt
Delft University of Technology
Delft, The Netherlands

***Abstract*—State-of-the-art machine learning-based models provide automatic intelligent code completion based on large pre-trained language models. The theoretical accuracy of these models reaches 70%. However, the research on the practicality of these models is limited. Our paper will discuss the usefulness of UniXcoder, a machine learning-based cross-modal auto-completion model, in a normal environment through user evaluation. These models incorporate context around the requested completion and then return a prediction of code based on the context. To accomplish this, two plugins were made called 'Code4Me'. One for Visual Studio Code and PyCharm. These plugins work with a remote API that requires a segment of 3966 characters of the left and right context at the trigger point. The data collected consists of the inserted code completion, verification of the code completion, the IDE used, the trigger point, and the inference time. To evaluate the data the following metrics are used: BLUE 4, ROUGE-L, Exact Match, Edit Similarity, and the METEOR score. The results point out that developers accept one completion every 8 suggestions with an Exact Match for the suggestions of 62.5%. The user evaluation using a survey, albeit with limited responses, are favourable towards the model and Code4Me. The accuracy of UniXcoder, however, is lower in a real-world setting than when it is predicting source code. But overall, the usefulness of UniXcoder as an auto-completion model is apparent.**

***Index Terms*—code completion, statement completion, integrated development environment, user study**

I. Introduction

A code completion model is capable of predicting code. A state-of-the-art completion model would be able to increase the efficiency of development and in return save money and time. Some natural language models have a theoretical accuracy of 70% up to 4 token lengths [1]. This promised accuracy makes code auto-completion a highly sought-after code extension. The incredible promise of natural language model-powered code completion has piqued the interest of research institutes, universities, and businesses. They have invested time and money into either researching or creating a prediction model for programming languages. State-of-the-art models include CodeBERT [2], RoBERTa [3], GPT-2 [4], InCoder [5], UniXcoder [6], and many more. Each with its own improvements and accuracies.

However, the only metrics available to gauge the effectiveness of an auto-completion model are the accuracy on test source code and the potential latency of the model. Many state-of-the-art models have high accuracy scores once trained and evaluated on their source code datasets. But, this does not indicate how well such a model performs for a developer. One model could have an impeding latency for every auto-completion or it might only suggest straightforward predictions. Therefore, to properly evaluate a model the interactions between the model and the developer have to be analysed in a real-world setting. The analysis could provide insights into how well a model performs outside of source code tests. If analysis can accurately depict which model performs well or which changes have a more drastic impact on the developer's efficiency then it offers a second metric that can rank models and potential features more accurately. Furthermore, BLUE 4, ROUGE-L, Exact Match, Edit Similarity, and the METEOR score could provide more insights into the performance of the model.

To evaluate the relationship between the developer and the cross-modal auto-completion model UniXcoder, we will contribute two IDE plugins and a remote API (found on our github[1]) that will provide insights into **RQ1:** *What is the acceptance rate of suggestions by a developer?* Additionally, a manual evaluation by the developers will be analysed through the use of surveys. These two methods would allow for analytical and empirical data to answer **RQ2:** *How useful is the model from the perspective of its users?* Additionally, collected data could outline flaws in the model or plugin and inspire possible improvements for **RQ3:** *How can the acceptance rate of the model be improved for everyday coding use?*

The answers to these questions will provide more insight into the performance of models and provide a more realistic estimation of the viability of natural language models in real-world settings.

II. Problem Definition

The statistics of accuracy and latency are too limited to evaluate the practicality of a model. The metrics that currently measure how favourable a model is, are accuracy and latency. The accuracy is measured on test sets and the latency is the time it takes to suggest a code completion. These metrics measure theoretical usefulness but not practical usefulness. It could be that developers never make use of the model because of the high latency or it only predicts simplistic code suggestions. To, therefore, distinguish the models' usefulness to a developer two other metrics should be evaluated such as the frequency the developer uses the model and the accuracy of the code suggestions.

[1] https://github.com/code4me-me/code4me

III. Background

In the following sections, we look into the background work that is inherently established in this paper.

A. Language Models and Transformers

Natural language processing models grant computers the ability to read, speak, and understand human languages. Most of the state-of-the-art models make use of the architecture introduced by Transformers. The improvements of transformers are clear and outlined by Thomas Wolf et. al [7]. But, these models were not limited to human language and could also be expanded with abstract syntax trees to understand code and, in turn, predict code.

B. UniXcoder

UniXcoder [6] is a state-of-the-art pre-trained and unified cross-modal model for programming languages that supports code completion and works as an encoder-decoder [8] model. Cross-modal contents are (not limited to) flattened abstract syntax trees and code comments. The architecture of UniXcoder exists out of transformers and mask attention matrices with prefix adapters to control the behaviour [6].

C. Completion Types

There exist different kinds of code completion and action types. The code completion types determine what kind of suggestion is given. Token completion would be limited to a single code token. Statement completion which completes a line of code. Lastly, there is block completion which completes a method or function.

IV. Related Work

Code Completion is in high demand and research is sprouting up everywhere. So is the research around deep learning models and user evaluation.

The paper by Bibaev *et al.* [9] makes use of anonymous usage logs from the IDE to train a model that ranks completion candidates. This study provides four different action types: Explicit Select, Typed Select, Explicit Cancel, and Typed Cancel. For training, they used Explicit Select or Typed Select considering these have ground truths. Their trained model was only 366KB and the model adds less than 30ms for inference time. The findings were that their decision tree based model was superior in ranking candidates over the default heuristics-based ranking.

Another user study was carried out by Xu *et al.* [10]. This study also introduces a plugin for the IDE used. The plugin features a model with hybrid of code generation and code retrieval functionality that requires a natural language prompt. In contrary to automatic evaluation, the evaluation was done by users manually undertaking specific tasks and a survey. The evaluation of the users was positive but the creativity requirements of a fitting prompt was troublesome as some uses had trouble turning their ideas into natural language.

Furthermore, training these deep learning models is a computationally intensive task. Thus, Sharma *et al.* [11] compared three deep learning architectures Code-GPT, Roberta, and GPT2 using limited computational power. Before the comparisons, CodeGPT is finetuned on the given python dataset. The Roberta model is trained on four different CSharp datasets and the GPT2 model is trained on two CSharp domains, naive approach and domain-specific. After the training period of very few epochs (5) GPT2 reached an accuracy score of 0.7123.

Moreover, Svyatkovskoy *et al.* [12] achieves code completion with limited requirements too. In this paper, the memory requirements of a neural network are addressed. First, a new framework was created that allowed for better analysis of internal structures and how these internal structures influence accuracy, memory usage, and computational cost. Then a new STAN-based model was presented that re-ranks completion candidates by combining static analysis and granular token encodings. The new model requires significantly less memory. The BPE-based natural language model required x2-x38 more memory. It was also found that n-gram models tend to have sizes that exceed x1700 times the memory requirement of their model.

But the interest in code completion models has also opened up the interest in security-oriented research. It is shown by Schuster *et al.* [13] that neural code completion using different models can be manipulated to suggest malicious or vulnerable code completion options. This was demonstrated by 'attacking' the models themselves and attempting data poisoning. It was brought to attention that the current reliance on code completion could be a security risk since the bait of taking the wrong code suggestion is often taken reaching up to 100% pick rate. And it is stressed that there is currently a lack of proper security against poisoning models.

The increase in auto-completion models also marks the start of the race to get the best performing models. The authors Liu *et al.* [14] tackle two main limitations of statistical language modelling; the usage of static embedding and the poor performance on identifiers. To improve these drawbacks new multi-task learning-based pre-trained language model using transformers. During the training, the identifiers are marked with type information. Additionally, the CugLM model attempts to predict both the identifier and type. The type prediction aids with the token prediction. This model significantly outperformed Byte Pair Encoding based Neural Language Model (BPE NLM) in most Java and Typescript predictions. The CugLM model meagerly underperformed on punctuation in both languages.

In addition, an empirical study about transformers was conducted. This study done by Chirkova and Troshin [15] emphasizes how all components of an abstract syntax tree (AST) are essential for quality auto-complete suggestions. It was also discovered that grouping transformers increase accuracy.

Liu *et al.* [16] addressed the drawbacks of existing models. The neglect of syntactic constraints such as type in models and the inability to model the long-term dependency. To combat this, a new model was proposed that also makes use of multi-task learning focusing on modelling the relationship

between type and value of code elements. Furthermore, two distinct ways of learning were explored. Predict the type and value together or first predict the type and utilise it for code prediction. It was shown that the type-first approach yielded higher accuracies than predicting the type and value together. The gain in accuracy was minimal for both Java and Typescript.

Furthermore, another model was proposed by Liu *et al.*, in this paper [17] three different drawbacks of Multi-Task Learning models were addressed: The hierarchical structural information being underutilised, the lack of modelling the long-term dependency between semantic relationships, and the under-utilised data from related tasks. This new model uses a new 'path2root' encoder and combines it with an AST encoder. This allowed us to more easily predict the next node type and value. This model was trained on Python, Java, and JavaScript and showed massive accuracy improvements over both Nested N-gram and Pointer Mixture Network (PMN) for the prediction of the next node type and value.

A PMN can still be improved upon for aspects such as "out of vocabulary" (OOV) tokens. These tokens replace unknown words if one occurs. This was done by Li *et al.* [18] by making use of attention mechanisms (AST-based) for code completion. These mechanisms look at previous hidden states and calculate the importance of the next state. This is not the only technique that could improve OOV, as noted by Aye and Kaiser [19] the representation of OOV tokens was changed by combining character-level input with token output. This approach slightly improved accuracy compared to other state-of-the-art models.

V. Methodology

The completion suggestion list is triggered on a specific set of tokens called trigger points. These trigger points are intercepted by our plugin called "Code4Me" after which a suggestion request is sent to the remote API with the segment of context and the trigger point. This segment exists out of 3966 characters from both the left and right context of the cursor. The character count is taken by calculating the average token length in characters of a collected dataset[2]. Considering the plugin is required to support two models InCoder and UniXcoder the maximum-minimum required amount of tokens of both models are sent to the remote API.

The server then calls the pre-trained model with the context and the model returns a code suggestion. This is then prompted to the user and only upon acceptance, the plugin will verify the line after 30 seconds. This timeout was arbitrarily chosen as we deemed that if a suggestion was correct it should not be changed for a period of time. The timeout is also long enough to allow for a developer to make changes to the line of code if the accepted suggestion was inaccurate. The inserted line with potential modifications is sent back to the server and stored to allow for evaluation. Figure 1 showcases a general overview of the main three components.

[2]We calculated the token frequencies on the raw files used in the P1K-22 dataset.

> DOT, AWAIT, ASSERT, RAISE, DEL, LAMBDA, YIELD, RETURN, EXCEPT, WHILE, FOR, IF, ELIF, ELSE, GLOBAL, IN, AND, NOT, OR, IS, BINOP, WITH, ;, ,, [, (, {, ~, =

a) Server: The server contains the pre-trained UniXcoder model and functions as a language server with a remote API. The model runs on a The server is written in Python to make use of the HuggingFace library. Furthermore, all communication protocols use HTTPS.

b) Model: The model used is UniXcoder which is run by the server. The model requires the left context and will provide a suggestion. We implemented an early stopping condition; whenever the model suggests a newline token we halt the prediction. The stopping condition reduces the inference time and the model is evaluated using line completion. Therefore, we do not need more than one statement completion.

c) Code4Me: The IDE extensions as commonly referred to when related to VSC and 'plugin' when expressed in the Jetbrains environment. Both plugins embed the prediction of the model into the suggestion list for auto-completion. The suggestions from the models are displayed by a Code4Me icon in the suggestion list. The plugin edits the ranking of the suggestion list to the furthest of capabilities to increase the probability of the suggestion reaching the top of the list.

VI. Experiment Design

In this section, we outline our experiment in three main sections. The implementation of the experiment, the evaluation of the experiment and how the participants were recruited. Furthermore, we elaborate upon the research questions and explain how these questions will be answered using the mentioned metrics in the evaluation section.

A. Research Questions

We will answer the following research questions in this paper.

a) ***RQ1:*** *What is the acceptance rate of suggestions from the model when used by developers?:* The acceptance rate relates to the overall acceptance of suggestions by developers. The acceptance rate could differ among trigger points. Thus a discrepancy will be made between the overall acceptance rate and specific trigger point acceptance rate.

b) ***RQ2:*** *How useful is the model from the perspective of its users?:* The usefulness of the model could be influenced by several factors such as the inference time of a suggestion and the plugin itself. In addition, the usefulness of the model could be established by the acceptance rate but also downplayed by the fact that developers might implicitly accept the suggestion, for example, typing it over.

c) ***RQ3:*** *How can the acceptance rate of the model be improved for everyday coding use?:* Improving the acceptance right directly influences the usefulness of the model. This could be by improving the model itself or possibly the plugin that provides the completions. Evaluating both acceptance rate and usefulness could highlight flaws within the model or

plugin. Moreover, the metrics proposed (BLUE 4, ROUGE-L, Exact Match, Edit Similarity, METEOR score) could dispute if the theoretical accuracy matters for the model to be useful.

B. Implementation

To collect these metrics developers prompt the code completion manually or one of the aforementioned trigger points. The manual trigger for the code completion is bound to Visual Studio Code's `trigger suggest` command. This is by default defined in VSC to a certain keybind and chosen to ensure a native experience. It allows the plugin to seamlessly integrate within the VSC IDE. We deemed this to be the most user-friendly. The PyCharm plugin faced limitations and bound the keybind to `ALT+SHIFT+K`.

Upon a trigger point or manual activation, the plugin sends an HTTPS request with the trigger point (if any) and a segment of their python document. The server will then return a code suggestion and Code4Me will prompt the suggestion list. The server also documents the request to gather a usage frequency for every developer, the inference time, and the length of context receive[3]. Once the code is inserted a method is called that verifies if the code suggestion was used by checking if the line of code in the document corresponds to the suggested code by the model after a timeout of 30 seconds. Furthermore, the insert completion is traced throughout the changes made by the developer to make sure editing of the document does not influence the retrieval of the correct line. This is then sent back to the server using the remote API and documented for later evaluation.

C. Evaluation

To evaluate the data different metrics are proposed. We separate these metrics into two categories. Similarity metrics for approved code suggestions that have been edited from the original suggestion and the other more general metrics. Additionally, we provide metrics from a survey that developers took after having used the plugin.

a) Trigger Frequency: The frequency of each trigger allows us to focus on which part of the model could be the most useful and how the acceptance rate relates to those triggers. It would also allow us to verify if the expected high accuracies at certain trigger points are met.

b) Acceptance Rate: The acceptance rate directly correlates to the usefulness of the plugin. If a developer has a high acceptance rate then we know that the plugin was useful for them. However, the following metrics can disprove if the acceptance rate is significant enough to determine a model useful. If the acceptance rate is low but the Exact Match is high, the usefulness or lack thereof is influenced by other factors.

c) BLUE-4: BLEU-4 is a variation on BLEU (BiLingual Evaluation Understudy) and is an automatic evaluation of machine translation. The concept of BLUE is "the closer a machine translation is to a professional human translation, the better it is" [20]. BLUE 4 compares the tokenized ground truth to the prediction and computes a weighted sum of 1 to 4 N-grams. We used a uniform weight for 1-4 N-grams.

d) ROUGE-L: ROUGE-L is a variant of ROUGE (Recall-Oriented Understudy for Gisting Evaluation) [21]. ROUGE-L compares the tokenized ground truth to the prediction and evaluates it by looking at the Longest Common Subsequence (LCS) where each code token is a unigram. ROUGE-L computes precision and recall and these are used together to compute an F1 score.

e) Exact Match: The Exact Match metric compares if both the ground truth and the prediction are equal while ignoring leading and trailing whitespace. This returns a boolean value and is expressed as a percentage.

f) Edit Similarity: The Edit Similarity calculates the number of character edits required for the ground truth to be an exact match of the prediction. It also introduces three techniques used for editing: insertion, deletion, and replacement. A penalty of 1 for wrong characters, too many characters, and too few characters, respectively. The similarity score is calculated by dividing the Levenshtein Distance by the length of the longest candidate. This results in a number between 0 and 1.

g) METEOR: The METEOR (Metric for Evaluation of Translation with Explicit ORdering) score uses both precision and recall to evaluate the ground truth and prediction by aligning their unigrams. Precision indicates the present unigrams in the prediction that are also in the ground truth. Additionally, recall indicates the present unigrams in the ground truth that are also in the prediction. Moreover, METEOR is deemed better at simulating human judgement than BLUE [22]. METEOR weighs precision over recall by tenfold. It is shown by Lavie *et al.* [23] that recall approaches human judgement More than precision.

D. Participants

We recruited participants for the Code4Me plugin through several programming-oriented online community networks (including but not limited to LinkedIn, Reddit, Programming Fora, Discord, and personal networks). All users were made aware of the data collection and age requirement to partake in the study. We decided to loosen the restraints on who has access to the plugin to increase our chances of attaining participants. We, therefore, published both plugins on their respective IDE marketplaces. This, however, allows unregulated access to the plugin. Furthermore, all survey respondents were recruited through the usage of the extension by the use of a pop-up every 50 suggestions. It is not possible to fill in the survey without having used Code4Me as it required their unique anonymous identifier.

Lastly, we posted a Facebook advertisement. The advertisement was shown to people who fit within our participant requirements. It resulted in 268 clicks but a negligible increase in active users of the plugin. The ad was cancelled shortly after its launch (<48 hours).

[3]This can be less than the preferred minimum context if the completion is requested at the start of a file.

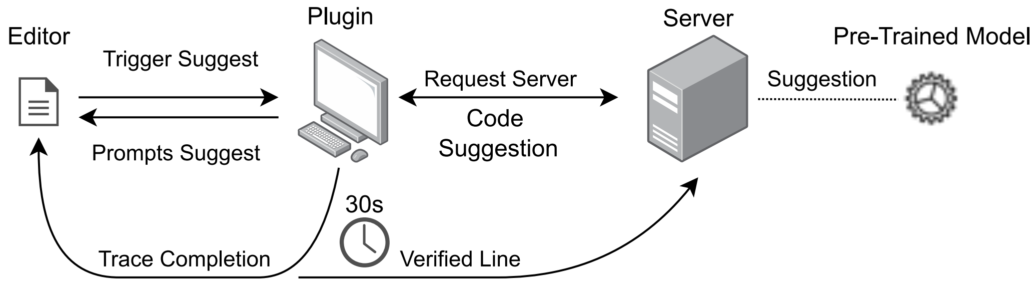

Fig. 1. Overview of the communication between the IDE, extension, and server.

TABLE I
METRICS PER COMPLETION KIND

	Metric	Total	Keybind	Trigger Point
	Occurrence	1552	39	1513
	Exact Match	22.49	35.90	22.14
	BLUE~4	34.89	51.64	34.46
ROUGE-L	Precision	40.56	57.35	40.13
	Recall	42.09	57.05	41.70
	F1 Score	39.65	56.66	39.22
	Edit Similarity	49.81	64.35	49.44
	METEOR	39.10	55.32	38.68

TABLE II
METRICS PER SELECTION TYPE

	Metric	Explicit Select	Not Selected
	Occurrence	200	1352
	Exact Match	62.50	16.57
	BLUE~4	67.09	30.13
ROUGE-L	Precision	78.70	34.92
	Recall	78.01	36.77
	F1 Score	76.92	34.14
	Edit Similarity	84.88	44.62
	METEOR	75.42	33.73

VII. RESULTS

The plugins were downloaded for a combined count of over 450 downloads. The VSC plugin received 54 downloads while the Jetbrains equivalent was downloaded 416 times. Out of all these downloads, 194 users used Code4Me. These users were not limited to using Code4Me for Python files. The following results are retrieved from the 68 users that used Code4Me for Python. Of these users, 32 were assigned to the UniXcoder model.

The results gathered over the course of two weeks are displayed in table I, II, IV, and III. Table I displays how the users interacted with Code4Me. Table II features the metrics selection type (explicit selection, not selected). The bundled performance of completions per token length of the suggested completion can be found in table IV. The top-10 used trigger points (ordered from left to right) are disclosed in Table III. All results were tokenized using our own version of the official Python tokenizer. The main difference is that ours is able to tokenize non-compiling code. The average inference time of the completions is approximately 150ms.

Besides usage statistics, users were prompted with a survey. This survey was answered 13 times. The quality of the suggestions was rated positively by 76.9% and 23.1% perceived the quality as neutral. Potential time saved by Code4Me was positively received by 92.3% of the respondents. 92.3% of the participants deemed the suggestion from Code4Me to be an improvement upon their IDE's default suggestions. Lastly, all participants from the survey were overall satisfied with the plugin and mentioned they would keep using it.

VIII. DISCUSSION

The majority of completions are triggered by the trigger points as shown in table I. The lack of occurrences for keybind triggered completions hinders us in interpreting these results with confidence. Thus, we will be directing our focus on the trigger point results. The general indication of trigger points shows that human evaluation would deem the completions subpar as simulated by the meteor score which is rather low. Moreover, the model shows about a 22.5% overall Exact Match accuracy without taking human intervention into account. But if we do include the developer's actions, the statistics favour the model far more.

Whenever a completion is explicitly selected by the user the results are great. The Exact Match rises up to 62.7% and this combined with the Edit Similarity metric of 85.1% can explain the overall appreciation by the survey respondents of the plugin. A developer appreciates if a completion is accurate over an inaccurate completion. Rejecting a completion does not impact the developer's time considering the inference time of the model is low. The model only performed well enough for the developer to select it 13.1% of the time out of all suggestions. We can conclude that the developer deemed the completion, when not selected, incorrect. This finding is also supported by the severe drop in scores when the code suggestion is not selected.

Evaluating the metrics per token length shows a downwards trend in Exact Match the longer the completion is. The other metrics follow this downward trend slightly but remain unfazed until a token length of 8. UniXcoder, thus, performs rather consistently until an average character com-

TABLE III
METRICS PER TOKEN LENGTH OF TRIGGER COMPLETION

		Token Length										
	Metric	1	2	3	4	5	6	7	8	9	10	10
	Occurrence	245	337	122	215	164	84	73	95	27	55	135
	Exact Match	35.92	30.27	33.61	16.74	21.34	10.71	20.55	14.74	3.7	9.09	2.22
	BLUE~4	24.58	41.5	44.88	41.72	42.72	31.83	35.36	37.63	19.49	29.97	12.49
ROUGE-L	Precision	50.41	38.87	46.72	45.76	42.49	38.26	37.36	45.41	20.8	38.87	15.1
	Recall	49.15	34.28	46.83	45.18	44.72	44.88	41.44	51.03	27.65	46.39	29.8
	F1 Score	49.23	35.44	45.97	43.89	41.87	39.35	37.79	45.8	22.78	40.62	17.52
	Edit Similarity	51.45	54.13	59.25	52.46	50.29	45.28	47.03	54.87	35.42	53.3	24.95
	METEOR	22.59	42.43	49.93	43.36	46.85	40.6	42.76	50.81	30.48	44.3	23.22

TABLE IV
METRICS OF TOP 10 TRIGGER POINTS

		trigger point									
	Metric	=	(	.	if	[	,	or	+	in	
	Occurrence	404	359	135	81	78	72	59	58	46	44
	Exact Match	28.96	16.99	38.52	4.94	26.92	16.67	10.17	5.17	41.3	11.36
	BLUE~4	30.06	35.3	51.28	24.47	45.12	30.58	32.24	21.1	49.6	35.16
ROUGE-L	Precision	33.61	33.41	55.67	31.66	47.87	34.61	43.95	64.52	55.9	50.89
	Recall	35.42	34.02	57.45	34.5	50.5	38.81	42.11	70.16	57.21	51.52
	F1 Score	33.38	31.07	55.53	31.48	46.65	35.55	42.54	64.47	54.87	48.67
	Edit Similarity	46.61	43.66	65.7	43.02	57.73	49.44	54.04	44.48	64.78	47.92
	METEOR	31.75	40.02	58.89	27.13	49.53	37.71	42.39	25.98	58.26	37.87

pletion length of 30 is reached. Edit Similarity and METEOR remain dominant with the highest score followed by ROUGE-L (recall).

In addition, the model performs better on certain trigger points. It is not surprising that =, `(`, and `.` are the highest occurring trigger points. The Exact Match being high for the period is likely because it can retrieve more context of already known functions and parameters. On the contrary, the lower performance of `(` as a trigger can be correlated with the fact that there could be a limited amount of context for the model to use, for example, whenever a new method is created. Moreover, the model predicts indexing and the in operator well. These two trigger points likely have more context too as the object, array, or map have likely been declared before the trigger point is called. While context lengths were stored later during the data collection period, the amount of data produced between that point and the end of the period does not warrant confidence. The data was too sparse and, therefore, also excluded from the tables.

Finally, the overall scores are lower than what is listed in the findings by Guo *et al.* [6] for UniXcoder, but when a suggestion is explicitly selected the scores surpass the findings of Guo *et al.*

A. Threats to Validity

The results of the study can be undermined because of the following potential threats to validity.

a) Internal Validity: One of the causes that might compromise the validity is the development of the plugins themselves. These plugins could contain bugs where something might go wrong and a trigger or completion might be lost. Additionally, it might be that the plugin makes more requests than required. This was discovered at the end of the data collection of the VSC plugin. If a user launches VSC and triggers Code4Me while the python interpreter of VSC is still initialising, it has a chance to queue multiple requests while it technically is only one. Once the python interpreter is initialised this behaviour was not seen again.

Lastly, the plugin sometimes produced incorrect syntax when inserting a suggestion. This was often noticed with closing brackets and braces. It will have influenced the Exact Match score.

b) External Validity: Code4Me has been published on the VSC and Jetbrains marketplace to ensure easy accessibility to users. This, however, opened up the opportunity for malicious intent by users. To prevent this, users were rate-limited to 1000 requests an hour but it does not prevent intentional poisoning of the data. To evaluate the data better, a focus group could be introduced that works with the plugin for an hour and then the statistics of those users could be compared. This could potentially improve the measurements of the results. If the focus group has significantly different results then it might be that the public users faced issues with Code4Me. Besides that, most of the results come from a smaller selection of users that have more completions. This can introduce bias in the data. This is also noticeable in the survey results where the users that received InCoder as a model outnumbered the users that received UniXcoder. We took the survey as a general reception as the perception did not differ among users of the models. Lastly, as aforementioned, some data was not stored until later in the study such as context length resulting in sparse and inconsistent data. If this would have been added at the start

of the study it could have explained the causal relationships between trigger points and their scores better.

c) Construct Validity: The tokenizer used influences the evaluation done by the metrics. A different tokenizer produces a different result. Furthermore, the Exact Match metric is incredibly strict and even low values can already be impressive. The metric should mostly be considered when looking at the token length of individual completions. It is, after all, significantly harder to get an Exact Match on a lengthier completion.

IX. Conclusion and Future Direction

In this paper, we developed two IDE extensions/plugins to provide developers with UniXcoder predictions. The 32 active users produced 1200+ data points and these were evaluated using our tokenizer. We provide five different metrics to assess the quality of the model. The metrics indicate an acceptance rate of 13.1% which means that 1 in 8 suggestions have been accepted. Combining the results and acceptance rate, we can deem the model useful to the users even though the Exact Match and Edit Similarity scores were found to be lower in a real-world setting than when predicting source code. Furthermore, this conclusion is reflected in the opinion of the (limited) survey respondents. The usefulness of Code4Me and thus UniXcoder is likely increased by the non-intrusive implementation of the plugins. We believe that if users are required to solely rely on the suggestions of UniXcoder, the model would be perceived as less useful.

Finally, for future improvements the model acceptance rate of the model could be improved by allowing left and right contexts like the generative model InCoder. Moreover, adding the possibility to use the filename as context could introduce better results. Additionally, it could be beneficial to extend the study period. And lastly, the plugins could be improved upon by providing multiple suggestions per completion which could increase the acceptance rate.

X. Responsible Research

The study involves human research. To ensure that we did this responsibly, we filed a request to the Human Resource Ethics Committee of the Delft University of Technology. In this request, we outlined what we intended on collected and how we ensured that this was done responsibly. The main risk identified were the age group, location of the users, and possible bias. The minimum required age was set to 18. The location was limited to the EU and possible bias was removed by broadening the search space for potential volunteers. We waited with collecting data until we received approval from the committee.

Secondly, we had to adhere to the GDPR. We clearly informed users that anonymised data collection took place and that partaking in the study (be it using the plugin or filling in the survey) is completely voluntarily. Additionally, the users were informed what their data was used for and when it would automatically be destroyed. The data will not be shared with third parties and is secured on the servers of the TU Delft.

Finally, the repository, that holds the code to the server and both plugins, is open-source.

XI. Acknowledgements

I would like to express my sincere gratitude to Georgios Gousios for providing hardware to run the server and models. Furthermore, I would like to thank Maliheh Izadi for her guidance throughout the study and Stefan Weel for generously creating the promotional material. Additionally, my appreciation goes out to Tim van Dam and Mika Turk for their general support. And lastly, I would like to thank Frank van der Heijden and Marc Otten for creating the Jetbrains IDE version of Code4Me.

References

[1] M. Izadi, R. Gismondi, and G. Gousios, "Codefill: Multi-token code completion by jointly learning from structure and naming sequences," Feb. 2022.

[2] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, *Bert: Pre-training of deep bidirectional transformers for language understanding*, 2018. DOI: 10.48550/ARXIV.1810.04805. [Online]. Available: https://arxiv.org/abs/1810.04805.

[3] Y. Liu *et al.*, *Roberta: A robustly optimized bert pretraining approach*, 2019. DOI: 10.48550/ARXIV.1907.11692. [Online]. Available: https://arxiv.org/abs/1907.11692.

[4] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2018. [Online]. Available: https://d4mucfpksywv.cloudfront.net/better-language-models/language-models.pdf.

[5] D. Fried *et al.*, *Incoder: A generative model for code infilling and synthesis*, 2022. DOI: 10.48550/ARXIV.2204.05999. [Online]. Available: https://arxiv.org/abs/2204.05999.

[6] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, *Unixcoder: Unified cross-modal pre-training for code representation*, 2022. DOI: 10.48550/ARXIV.2203.03850. [Online]. Available: https://arxiv.org/abs/2203.03850.

[7] T. Wolf *et al.*, "Huggingface's transformers: State-of-the-art natural language processing," *CoRR*, vol. abs/1910.03771, 2019. arXiv: 1910.03771. [Online]. Available: http://arxiv.org/abs/1910.03771.

[8] A. Vaswani *et al.*, *Attention is all you need*, 2017. DOI: 10.48550/ARXIV.1706.03762. [Online]. Available: https://arxiv.org/abs/1706.03762.

[9] V. Bibaev *et al.*, *All you need is logs: Improving code completion by learning from anonymous ide usage logs*, 2022. DOI: 10.48550/ARXIV.2205.10692. [Online]. Available: https://arxiv.org/abs/2205.10692.

[10] F. F. Xu, B. Vasilescu, and G. Neubig, *In-ide code generation from natural language: Promise and challenges*, 2021. DOI: 10.48550/ARXIV.2101.11149. [Online]. Available: https://arxiv.org/abs/2101.11149.

[11] M. Sharma, T. K. Mishra, and A. Kumar, "Source code auto-completion using various deep learning models under limited computing resources," Complex Intell. Syst., 2022. DOI: s40747-022-00708-7.

[12] A. Svyatkovskoy, S. Lee, A. Hadjitofi, M. Riechert, J. Franco, and M. Allamanis, "Fast and memory-efficient neural code completion," *arXiv preprint arXiv:2004.13651*, 2020.

[13] R. Schuster, C. Song, E. Tromer, and V. Shmatikov, "You autocomplete me: Poisoning vulnerabilities in neural code completion," in *30th USENIX Security Symposium (USENIX Security 21)*, USENIX Association, Aug. 2021, pp. 1559–1575, ISBN: 978-1-939133-24-3. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/schuster.

[14] F. Liu, G. Li, Y. Zhao, and Z. Jin, "Multi-task learning based pre-trained language model for code completion," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '20, Virtual Event, Australia: Association for Computing Machinery, 2020, pp. 473–485, ISBN: 9781450367684. DOI: 10 . 1145 / 3324884 . 3416591. [Online]. Available: https://doi.org/10.1145/3324884.3416591.

[15] N. Chirkova and S. Troshin, "Empirical study of transformers for source code," *CoRR*, vol. abs/2010.07987, 2020. arXiv: 2010.07987. [Online]. Available: https://arxiv.org/abs/2010.07987.

[16] F. Liu, G. Li, B. Wei, X. Xia, Z. Fu, and Z. Jin, "A unified multi-task learning model for ast-level and token-level code completion," 91, vol. 27, Empir Software Eng, 2022. DOI: 10.1007/s10664-022-10140-7.

[17] ——, "A self-attentional neural architecture for code completion with multi-task learning," IEEE, Institute of Electrical and Electronics Engineers, 2020, pp. 37–47. DOI: 10.1145/3387904.3389261. [Online]. Available: https://dl.acm.org/doi/proceedings/10.1145/3387904,%20https://conf.researchr.org/home/icpc-2020.

[18] J. Li, Y. Wang, M. R. Lyu, and I. King, "Code completion with neural attention and pointer networks," in *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence*, International Joint Conferences on Artificial Intelligence Organization, Jul. 2018. DOI: 10.24963/ijcai.2018/578. [Online]. Available: https://doi.org/10.24963%2Fijcai.2018%2F578.

[19] G. A. Aye and G. E. Kaiser, "Sequence model design for code completion in the modern IDE," *CoRR*, vol. abs/2004.05249, 2020. arXiv: 2004.05249. [Online]. Available: https://arxiv.org/abs/2004.05249.

[20] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: A method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, Philadelphia, Pennsylvania, USA: Association for Computational Linguistics, Jul. 2002, pp. 311–318. DOI: 10.3115/1073083.1073135. [Online]. Available: https://aclanthology.org/P02-1040.

[21] C.-Y. Lin, "ROUGE: A package for automatic evaluation of summaries," in *Text Summarization Branches Out*, Barcelona, Spain: Association for Computational Linguistics, Jul. 2004, pp. 74–81. [Online]. Available: https://aclanthology.org/W04-1013.

[22] S. Banerjee and A. Lavie, "METEOR: An automatic metric for MT evaluation with improved correlation with human judgments," in *Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*, Ann Arbor, Michigan: Association for Computational Linguistics, Jun. 2005, pp. 65–72. [Online]. Available: https://aclanthology.org/W05-0909.

[23] A. Lavie, K. Sagae, and S. Jayaraman, "The significance of recall in automatic metrics for MT evaluation," in *Proceedings of the 6th Conference of the Association for Machine Translation in the Americas: Technical Papers*, Washington, USA: Springer, Sep. 2004, pp. 134–143. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-540-30194-3_16.