An Empirical Analysis on the Performance of UniXcoder

Tim van Dam
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

An Empirical Analysis on the Performance of UniXcoder

Tim van Dam
Delft University of Technology
Delft, The Netherlands

***Abstract*—Numerous papers have empirically studied the performance of deep learning based code completion models. However, none of these papers considered nor investigated whether good performance on statically typed languages translates to good performance on dynamically typed languages. A lack of available type information can make code completion more difficult, as many types are interacted with differently. However, natural language in the form of comments could compensate for a lack of available type information. This paper evaluates whether UniXcoder, a state of the NLP model, is able to perform code completion on both dynamically and statically typed languages with similar performance. Furthermore, the impact of the presence of type annotations and comments is assessed. We show that UniXcoder is able to utilize type annotations and comments in order to improve code completion performance, and that using only singleline comments yields better results than using all comments in the source code.**

***Index Terms*—code completion, type annotations**

I. Introduction

While many developers use dynamically typed programming languages for their relatively low barrier of entry and ease of use, dynamically typed languages are not always superior to statically typed languages. In fact, it is hard to prove that dynamically typed languages lead to faster development times at all – even when explicitly trying to find instances where this is the case [1]. Not only humans might encounter difficulties when dealing with dynamically typed languages: simple code completion models may be unable to provide valuable suggestions, as the type of variables often directly dictates how this variables can be interacted with. When such information is unavailable, the lack of type information might make for poor code completions. However, deep learning code completion models – which have become more mainstream recently – may not suffer from this issue. Deep learning has already shown competence in the area of type inference: models such as LambdaNet [2] and TypeBERT [3] are able to accurately predict the types of variables in JavaScript code. But whether code completion models are similarly able use context to make predictions that make sense given variables types in dynamically typed languages has never been studied.

Code completion models such as CodeT5 [4] and UniXcoder [5] are able to use comments in order to aid understanding and thus code completion. But whether information embedded in comments is competitive with the availability of type information, and how the presence of both influence performance has not been studied before. This paper addresses this knowledge gap by evaluating the performance of UniXcoder [5], a state of the art pre-trained model that can perform various programming language (PL) tasks. In addition to source code, UniXcoder considers natural language embedded in comments and the structure of the code through flattened Abstract Syntax Trees (ASTs) to improve code understanding. UniXcoder achieves state of the art results for the code completion task, beating models such as CodeT5 [4], CodeGPT [6] and GPT-2 [7] on both Exact Match and Edit Similarity for the PY150 [8] and JavaCorpus [9] datasets [5].

We collect three datasets containing a total of 2,232 TypeScript repositories. From this data we extract functions and use the TypeScript compiler to create equivalent JavaScript functions. Then, we use UniXcoder to perform line completion on equivalent partial TypeScript and JavaScript functions to establish the effect of the presence (or lack thereof) of type annotations and comments. Additionally, we investigate the effects of limiting the comments to only singleline and multiline comments.

Results show that a presence of type annotations improves the code completion performance of UniXcoder. This also applies to comments, as the presence of comments is shown to be beneficial for performance. Additionally, we show that using only singleline comments leads to better results than using all comments.

In short, the contributions of this paper are:

- We perform an empirical analysis of UniXcoder on dynamically and statically typed languages;
- We establish the relevance of comments and type annotations for UniXcoder;
- We publicly provide our datasets, source code, and fine-tuned models[1].

The paper follows the following structure: section II describes the problem, section III discusses the background of this work and summarizes related work, section IV describes the approach used to compare code completion on dynamically and statically typed languages, and section V explains how this approach is applied. Then, section VI will discuss the results of the experiment, after which section VII interprets these results. Potential threats to validity are reviewed in section VIII, after which section IX concludes the paper. Finally, responsible research concerns are discussed in section X.

[1] https://github.com/timvandam/rp

II. Problem Definition

Code completion models have shown their use in statically typed and dynamically typed languages alike. However, while these models have been empirically studied, there have been no attempts at analyzing how the presence of type information influences the performance of code completion models. Addressing this knowledge gap may outline how well code completion models are able to infer type information based on the context, or could indicate that current models need to be expanded in order to improve. Similarly, it may determine the effects that comments have on the performance of code completion on dynamically typed languages in contrast to statically typed languages.

In theory additional type information should be able to make code completion models more accurate, as they are given a more comprehensive description of the source code. The same can be said about comments, which are typically used to describe complete complete functions with docblocks, or used to annotate smaller parts of code with singleline comments. The difference between the two is that type annotations are placed in a structured manner, whereas comments are not guaranteed to follow a specific structure (docblocks do follow some structure, but do not follow an ordering and can contain a wide range of different information).

Establishing the relation between the presence of types and comments to code completion performance in dynamically and statically typed languages gives an understanding into what elements of source code can be used in order to improve code completion performance.

III. Background and Related work

A. UniXcoder

UniXcoder is a state of the art pre-trained model that leverages multiple modalities in order to facilitate several code understanding and generation tasks [5]. Similar to other modern code completion models, UniXcoder is transformer-based, using an encoder-decoder architecture for most tasks, and a decoder-only architecture for code completion. In addition to source code, comments and flattened abstract syntax trees (ASTs) were used during pre-training in order to improve understanding. To this end, UniXcoder was pre-trained using masked language modeling [10, 11], unidirectional language modeling [12] and denoising [13]. UniXcoder performs slightly better on the code completion task when considering comments, but different types of comments such as singleline and multiline comments are not considered individually [5].

B. Related Work

Using deep learning to facilitate natural language processing tasks is an active topic of research with widespread real-world applications. It is consequently no surprise that there exist a plethora of works surrounding code completion using deep learning. Especially transformer-based models have seen great discussion recently. The following section reviews transformer-based models, other studies into the effectiveness of code completion models, and alternative ways deep learning is applied to deduce variable types based on context.

1) Transformer Models: Recently, transformer-based [14] models have shown great promise in the area of Natural Language Processing (NLP).

BERT [10], a bidirectional transformer model, showed the value of being able to consider both the left and right context during natural language processing tasks, which previous models like ELMo [15] and GPT [12] could not do. Bidirectionality was achieved by using *masked language modeling* (MLM) for pre-training. Similar to Baevski *et al.* [11], tokens are randomly masked and predicted during pre-training, after which the model is tasked to predict these masked tokens using both the left and right context of tokens. This is valuable in languages where words later on in a sentence determine conjugations, but also in programming as it is not a purely linear activity.

BERT was fine-tuned and evaluated using the GLUE benchmark [16], where it outperformed all other models at the time. Similar results were achieved for the SQuAD v1.1, SQuAD v2.0 and SWAG benchmarks.

With BERT, Devlin *et al.* highlighted the value of bidirectional pre-training, and showed that pre-trained models can be specialized to suit a wide range of NLP tasks.

Liu *et al.* improved further upon BERT with RoBERTa [17], and aimed to show that the performance of BERT can be further improved by optimizing some design choices. The primary changes consist of using a larger dataset, training longer and on longer sequences, no longer training using next sentence prediction, and improving MLM such that it masks dynamically instead of determining the mask pattern once at the start. Overall, the results of this modified pre-training procedure showed that BERT-based models can be more effective than previously thought. Similarly to the original BERT [10], RoBERTa achieved state of the art results on several benchmarks including GLUE and SQuAD. BERT and RoBERTa can also play a role in automatic code completion, as shown by Feng *et al.*'s CodeBERT [18], which builds on top of RoBERTa.

Differently from BERT and RoBERTa, Raffel *et al.*'s Text-to-Text Transformer (T5) [13] is a transformer encoder-decoder. The aim when creating T5 was to analyze the current state of models that solve NLP tasks, and to use this information to build a state of the art model that combines this information. This was achieved by treating every NLP task as a *text-to-text* (i.e. sequence to sequence) task, where even the task is described in the input text. T5 achieved state of the art results on many benchmarks covering a number of NLP tasks, including GLUE and SQuAD. T5 was found to be comparable in performance with task-specific architectures, even though T5 itself is relatively general. As with BERT/RoBERTa, T5 has a programming analogue: CodeT5 [4].

2) Studies About Code Completion Effectiveness: Ciniselli *et al.* [19, 20] aimed to analyze the performance of state of the art transformer-based code completion models. T5 [13] and RoBERTa [17] were used for this analysis. Performance

was tested on several levels of granularity: single tokens (token-level), statements (construct-level) and even complete code blocks (block-level) on two datasets, containing Java methods and Android app methods from open-source GitHub repositories. While Android apps are also written in Java, taking only methods from such apps could help learning as the same API is used across all methods.

It was shown that deep learning code completion models are viable, and that especially the T5 model does well. However, the success of these models when tasked to predict longer sequences is limited. As only Java was used for evaluation the results are not completely generalizable – it is impossible to determine whether these successes are transferable other languages, including dynamically or gradually typed languages. Additionally, while this analysis established performance differences depending on the granularity level, it did not attempt to determine other variables that could affect performance, such as comments or variable names.

3) Type Prediction: Several previous works have demonstrated the ability to infer the types of variables and functions in dynamically typed languages depending on their context [2, 3, 21, 22].

DeepTyper [22] shows how deep learning can be applied to infer types in dynamically typed languages such as JavaScript and Python in order to ease the transition from untyped code to gradually typed code. DeepTyper uses a GRU-based RNN trained on predominantly TypeScript files sourced from the top 1,000 GitHub projects with the most stars. Before learning, types were removed from these files, after which inferred types were compared against the source TypeScript files during evaluation. DeepTyper's output includes both a predicted type and the confidence it has in its prediction. The precision of DeepTyper is lower than that of JSNice [23], a statistical code property prediction model, however JSNice often opts not to return a potential type when it has low confidence, lowering its recall. Hence Hellendoorn *et al.* find that DeepTyper qualitatively outperforms JSNice. Combining DeepTyper with JSNice (using the former only when the latter is unable to determine the type) performed significantly better than using either alone. Using multiple tools this way could prove useful when attempting to complete dynamically typed code: an accurate type prediction system could work together with a code completion model made for statically typed languages.

Similar to DeepTyper [22], Malik *et al.*'s NL2Type [21] shows that natural language information in comments, function and parameter names can be exploited in order to predict types in dynamically typed languages. This is achieved using a LSTM-based RNN that learns from type-annotated source code. Over 160,000 open-source JavaScript files were used for learning and evaluation, which resulted in both a high precision and a high recall. This approach outperformed previous state of the art models including JSNice and DeepTyper. This work shows that natural language information can successfully be exploited to accurately predict types in dynamically typed languages, and that type prediction tools can serve as a tool to detect inconsistencies between type annotations and natural language descriptions.

Similar to DeepTyper [22] and NL2Type [21], Wei *et al.*'s LambdaNet [2] shows that deep learning can be applied to provide untyped code with type annotations in gradually typed languages like TypeScript and Python. LambdaNet uses a GNN after first modeling the types in the source code as a *type dependency graph*. Like DeepTyper, This GNN is trained on TypeScript files sourced from the top 1,000 GitHub projects. However in contrast to DeepTyper, LambdaNet is able to predict user and third party types, while DeepTyper is only able to predict types from a fixed vocabulary. The results show that LambdaNet outperformed state of the art models, including both JSNice and DeepTyper, when comparing their accuracy over library types.

Like LambdaNet [2], TypeBERT [3] is able to infer user and third party types. However, it takes a much simpler approach by applying BERT-style pre-training, after which it is fine-tuned on a large set of TypeScript data. While TypeBERT does not outperform LambdaNet on user and third party types, it significantly outperforms LambdaNet on prediction the top-100 most common types, achieving an exact match accuracy of 89.51% and a top-5 accuracy of 98.51% [3]. This once more shows that type prediction tools can be an extremely powerful resource when transforming code without type annotations to code with type annotations.

Overall, these works show that a potential gap in code completion performance between dynamically and statically typed languages as a result of the presence of type annotations is not necessarily a problem, as type inference models can be used to add type information to dynamically typed code. Languages where type annotations are optional such as TypeScript and Python 3 can similarly benefit from these models.

IV. Methodology

UniXcoder is fine-tuned and tested on several datasets in order to establish whether there is a significant relation between code completion performance and the presence of natural language in the forms of type annotations and comments. As shown in fig. 1, the method consists of three main phases: *data transformation*, *pre-processing* and *fine-tuning*. During data transformation, functions are extracted from the TypeScript datasets and split into JavaScript and TypeScript train, development and test sets. During pre-processing the literals and specific comments are removed from the data, after which the model is fine-tuned and applied on the test set. The following section explains how these phases work in detail.

A. Data transformation

The data transformation phase is responsible for creating JavaScript and TypeScript train, development, and test sets. The reason why this phase occurs prior to pre-processing is because this phase relies on the TypeScript compiler to create the JavaScript sets, and the pre-processing phase leads to uncompilable code.

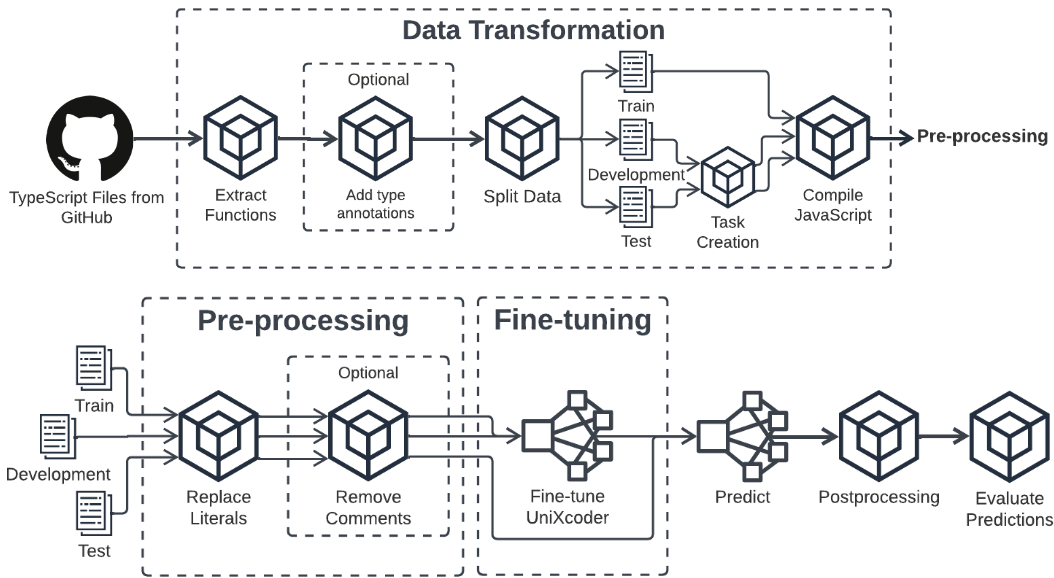

Fig. 1. Experiment Pipeline

First, functions – including potential leading comments – are extracted from the TypeScrript datasets using the TypeScript compiler. This data format is similar to the CodeSearchNet dataset [24], as used by UniXcoder during pre-training. As TypeScript does not require type annotations, there may be few type annotations in the code. In order to amplify the effects of type annotations, the TypeScript compiler is optionally used to add type annotations to these functions in places where they were not added, given that these type annotations can be inferred by the compiler. This process, for instance, converts `const x = 1;` to `const x: number = 1;`. Third party dependencies of the repositories in the dataset are installed prior to running this process, as code might contain type annotations defined in a third party dependency. This step was only executed for one of the datasets due to the high computation time required.

The functions are then split into a train, development, and test set according to a 75/5/20 split. The functions in the development and test set are used to create line completion tasks by first tokenizing the source code according to the ECMAScript lexical grammar specification [25]. Secondly, 20% of the tokenized lines are randomly selected, after which a `/*<mask>*/` comment token is placed in front of a randomly selected token in each of the selected lines. This comment token serves the purpose of indicating where we want to perform a line completion, and allows us to create an equivalent JavaScript version of this line completion task. The latter is achieved by feeding a detokenized version of the code (with mask comments) through the TypeScript compiler, resulting in JavaScript code with the same `/*<mask>*/` comments. Finally, the JavaScript and TypeScript code with mask comments is converted into line completion tasks consisting of inputs (i.e. left contexts) and ground truths by:

1) Splitting the masked code on mask comments, resulting in a number of substrings.
2) **Obtaining ground truths** by taking the first line of all but the first substrings. Note that the first substring is not considered as it occurs before any mask comments, while ground truths are situated behind mask comments.
3) **Obtaining inputs** by concatenating all substrings that occur before each ground truth.

This process results in line completion tasks for both the development and test set in JavaScript and TypeScript.

A JavaScript equivalent of the train set is generated by applying the TypeScript compiler directly to the functions in the train set.

B. Pre-processing

Pre-processing makes the previously created JavaScript and TypeScript sets suitable for use with UniXcoder. The same pre-processing process is applied to the train, development and test set.

First, number and string literals are normalized by replacing them with special tokens, ⟨NUM_LIT⟩ and ⟨STR_LIT⟩ respectively. Line breaks are replaced by the ⟨EOL⟩ token.

Each dataset is pre-processed several times, each with slight differences in the way comments are handled. There are four different ways comments are handled in order to test their effect on completion performance:

1) Remove all comments.
2) Do nothing – keep comments as-is.
3) Keep only singleline comments.
4) Keep only multiline comments.

C. Fine-Tuning

After pre-processing, we fine-tune UniXcoder on the JavaScript and TypeScript train sets individually using the

Adam optimizer with an initial learning rate of 2e-5 for 10 epochs using an NVIDIA V100S [26]. The epoch that leads to the highest Exact Match (EM) score on the development set is applied on the test set.

D. Post-processing

After the fine-tuned model has been applied to all the test data, its predictions are post-processed. Post-processing consists of normalizing the spacing of code tokens (including linebreaks), removing all comments, and replacing tokenized versions of literals with default literals: ⟨STR_LIT⟩ becomes `""`, and ⟨NUM_LIT⟩ becomes `0`. For consistency, this process is applied to both the prediction and the ground truth.

V. Experiment Design

A. Research Questions

The goal of this work is to provide an insight into the relation between code completion performance and the presence of comments and type annotations. Accordingly, it answers the following research questions:

RQ$_1$: *How is the performance of UniXcoder code completion affected by presence of type annotations in source code?*

This research question investigates whether UniXcoder benefits from the presence of type annotations. As mentioned, type annotations convey valuable information in a structured manner, which could increase code completion performance.

RQ$_2$: *To what extent is the performance of UniXcoder code completion affected by presence of comments?*

This research question explores how the performance of UniXcoder is affected by the presence of comments. The relevance of this research question lies in the fact that comments could function similarly to type annotations when it comes to aiding the understanding abilities of UniXcoder. Consequently, the extent to which comments improve performance warrant investigation.

RQ$_3$: *To what extent does presence of comments in contrast to presence of type annotations aid UniXcoder code completion performance?*

This research question explores how type annotations and comments compare when it comes to their effect on code completion performance.

B. Evaluation Metrics

The predictions by the fine-tuned UniXcoder models are compared to the ground truths using a number of metrics. The following metrics are considered: Exact Match, Levenshtein Similarity, BLEU-4, ROUGE-L, and METEOR.

Exact Match (EM) directly compares the ground truth to the prediction, resulting in a boolean value. The Exact Match score over an entire dataset is expressed as a percentage.

Levenshtein Similarity (Edit Similarity) compares the ground truth to the prediction on a character-by-character basis. Wrong characters (substitutions), too many characters (insertions) and too few characters (deletions) increase the levenshtein distance by 1. Levenshtein similarity is a number in the range $[0, 1]$ that is computed by dividing the levenshtein distance by the length of either the prediction or the ground truth, depending on which is longest.

BLEU-4 is a variant of BLEU (BiLingual Evaluation Understudy) that deals specifically with n-grams with n $\in [1, 4]$. BLEU compares the ground truth to the prediction by computing the ratio of n-grams that occur in both the prediction and the ground truth (not exceeding the total amount in the ground truth) to the total amount of n-grams in the ground truth. This is done for each n in some range, after a weighted sum of the results is computed to get the final BLEU score [27]. In this work BLEU-4 is applied by treating each code token – as per the ECMAScript lexical grammar specification [25] – as a unigram, and gives each value of n equal weight. A smoothing technique proposed by Lin *et al.* [28] is used to prevent division by zero when the prediction has fewer than four tokens.

ROUGE-L is a variant of ROUGE (Recall-Oriented Understudy for Gisting Evaluation). ROUGE-L uses the Longest Common Subsequence (LCS) algorithm in order to find the largest n-gram that occurs in both the prediction and the ground truth. ROUGE-L computes precision and recall and uses them to compute an F1 score [29]. As with BLEU, each code token corresponds to a unigram when ROUGE-L is applied.

METEOR, or Metric for Evaluation of Translation with Explicit ORdering, compares the ground truth and the prediction by mapping their respective unigrams, and computes a score over this mapping based on its precision and recall. METEOR has been shown to be better at capturing human judgement over a complete dataset than BLEU [30]. Opposed to BLEU, METEOR puts more weight on recall, which has been shown to align closer to human judgement than precision [31]. Once more, each code token is treated as a unigram.

C. Datasets

The experiment uses several openly available TypeScript datasets. The first dataset, *TS1K-18*, includes the top 1,000 starred GitHub repositories that predominantly consist of TypeScript code. This dataset is identical to that used by Hellendoorn *et al.* [22] and Wei *et al.* [2], and was created in 2018. However, as not all of these repositories are available anymore the dataset used in this paper is limited to 709 repositories.

An additional dataset, *TS1K-18-E*, was generated based on *TS1K-18* by making all implicit types explicit using the TypeScript compiler, as explained in section IV. Since not all repositories include a `tsconfig.json` file – which is used to configure the TypeScript compiler – not all repositories in the previous dataset are included in this dataset. Additionally, some repositories depend on third party dependencies which could not be installed, which made it impossible to infer third-party types. As a result this dataset is 33.63% smaller than *TS1K-18*.

Similar to *TS1K-18*, the third dataset consists of all TypeScript files in the top 1,000 starred GitHub TypeScript repos-

TABLE I
DATASETS USED FOR FINE-TUNING AND EVALUATION

	TS1K-18	TS1K-18-E	TS1K-22
#Repositories	709	523	1,000
#Functions	61,181	40,604	165,423
#LOC	1,261,692	913,773	3,465,559
Type Explicitness	48.66%	87.74%	45.79%

itories. This dataset was generated specifically for this study on June 9th 2022 using the GitHub API.

Table I shows the size of the datasets in terms of the amount of repositories, functions and lines of code (LOC) in these functions. Additionally, it displays the *type explicitness* of the functions in these datasets. Type explicitness refers to the amount of type annotations present in the code relative to the total amount of type annotations that are possible.

VI. RESULTS

The predictions made by UniXcoder on the various datasets were used to compute the aforementioned metrics as displayed in table II. The following section will highlight relevant results.

TypeScript models consistently outperformed JavaScript models by small margins, achieving the highest scores on each metric across all datasets. The largest discrepancy between JavaScript and TypeScript performance was found on the *TS1K-18-E* dataset, where TypeScript outperformed JavaScript by substantially higher margins than on the other datasets. Overall, TypeScript models trained on data with comments performed best.

Models trained on datasets with all comments did not achieve better performance than models trained exclusively on data with singleline or multiline comments. Across all datasets, models fine-tuned on data containing only singleline or multiline comments performed better on every metric relative to the baseline, which contains all comments. This is the case for both TypeScript and JavaScript.

Models trained on datasets containing exclusively singleline comments outperformed the baseline dataset models on almost all metrics. The only exception was *TS1K-18-E*, where the JavaScript model fine-tuned on the dataset with singleline comments performed worse than the baseline on exact match and BLEU-4. Both JavaScript and TypeScript showed the best performance on datasets containing only singleline or multiline comments.

Models trained on datasets containing exclusively multiline comments did not always outperform the baselines. In *TS1K-18*, the fine-tuned models trained on data with multiline comments even failed to beat the models trained on data without comments.

JavaScript models fine-tuned on datasets with comments outperformed TypeScript models fine-tuned on datasets without comments several times. Namely, for *TS1K-18*, the singleline comment JavaScript model achieved a higher score for edit similarity, for *TS1K-18-E* the multiline comment JavaScript model achieved a higher score for BLEU-4, and for *TS1K-22* the singleline JavaScript model achieved a higher score for edit similarity, BLEU-4 and METEOR. Note that in all these cases the outperformance by the JavaScript models was minimal.

VII. DISCUSSION

A. Type Annotations

The results support the rationale of type annotations leading to increased performance, as UniXcoder models fine-tuned on TypeScript code outperformed their JavaScript counterparts on every single metric and every single dataset. While the outperformance by TypeScript models was marginal, the *TS1K-18-E* dataset shows that an increased amount of type annotations can lead to a larger outperformance by the TypeScript models. As per table I, *TS1K-18-E* has a type explicitness of 87.74% compared to the 48.66% and 45.79% of *TS1K-18* and *TS1K-22* respectively. These facts suggest a positive correlation between the presence of type annotations and code completion performance.

B. Comments

Models fine-tuned on code with comments outperformed commentless code for each dataset, suggesting that comments do contribute to language understanding. Additionally, the results show that JavaScript models trained on commented code can come close to the performance of models trained on commentless TypeScript. This demonstrates the value of comments, but also shows that type annotations can be more capable at improving code completion performance.

Not all types of comments lead to the same performance improvement: each dataset showed that keeping only singleline or multiline tokens leads to better performance than keeping both. Especially singleline comments appear to improve UniXcoder performance, as models trained on data containing only singleline comments outperformed each baseline.

The contents or structure of comments might also have an effect on performance. For the *TS1K-18* dataset, models trained on commentless code sometimes even performed better than models trained on commented code. These uncertain results might be explained by lack of structure or cohesion of comments making it difficult to interpret. In the case of multiline comments this could be due to the verbose nature of JSDoc/TSDoc comments. These types of comments comments can become extremely large when code examples, explanations and URLs are added. Many different tags can be added to such comments, some of which might not provide valuable information. Combined with the fact that these tags can practically be placed in an arbitrary order may make them difficult to utilize for UniXcoder.

The best performance was observed on TypeScript models that were fine-tuned on data with comments, showing that comments can add value next to type annotations.

Overall, the outperformance by TypeScript models on every dataset indicates that type annotations are a valuable addition to code, and can help code completion models like UniXcoder in language understanding. This is strengthened by the fact that

TABLE II
UNIXCODER LINE COMPLETION RESULTS

Dataset	Language	EM	Edit Sim	BLEU-4	ROUGE-L	METEOR
TS1K-18	JavaScript	40.41	68.08	55.54	54.41	65.51
	TypeScript	40.56	68.26	55.62	54.85	65.78
- w/o comments	JavaScript	40.79	67.83	55.50	54.58	65.25
	TypeScript	40.99	68.29	55.85	55.28	65.79
- w/ multiline comments	JavaScript	40.15	67.60	54.92	54.23	64.77
	TypeScript	40.64	67.87	55.27	54.92	65.14
- w/ singleline comments	JavaScript	40.95	68.30	55.67	54.97	65.74
	TypeScript	**41.32**	**68.69**	**55.98**	**55.68**	**66.08**
TS1K-18-E	JavaScript	38.87	67.47	54.54	54.24	64.77
	TypeScript	39.49	68.50	55.27	55.98	65.93
- w/o comments	JavaScript	38.17	66.61	53.49	53.40	63.62
	TypeScript	39.49	68.03	54.75	55.61	65.19
- w/ multiline comments	JavaScript	39.22	67.50	54.76	54.49	64.90
	TypeScript	**40.10**	68.60	**55.85**	56.30	66.33
- w/ singleline comments	JavaScript	38.51	67.67	54.37	54.38	64.99
	TypeScript	40.08	**68.98**	55.61	**56.45**	**66.45**
TS1K-22	JavaScript	37.17	64.82	52.25	50.21	62.06
	TypeScript	37.78	65.61	52.88	51.39	62.86
- w/o comments	JavaScript	37.10	64.28	51.71	49.83	61.29
	TypeScript	37.67	64.84	52.09	50.83	61.79
- w/ multiline comments	JavaScript	37.27	64.40	51.76	50.01	61.32
	TypeScript	37.55	64.81	52.12	50.86	61.75
- w/ singleline comments	JavaScript	37.43	65.11	52.50	50.35	62.33
	TypeScript	**37.98**	**65.67**	**53.00**	**51.40**	**62.91**

the TypeScript models trained on a dataset with an increased amount of type annotations showed a larger margin of outperformance over respective JavaScript models. Additionally, while comments improve UniXcoder performance, comments are not as capable at improving language understanding as type annotations are. Not all types of comments lead to the same performance improvements, which indicates that the structure or content of comments might play an important role. However, future work into the effect of the content of comments is needed to confirm this hypothesis. The results consistently show that a combination of both type annotations and certain comments leads to the best result.

VIII. THREATS TO VALIDITY

Threats to internal validity relate to factors unintentionally affecting the results. In this work, type annotations and comments were intentionally changed in order to test their relation with code completion performance. Other variables involved in this work are fine-tune parameters and metric-related parameters. Fine-tune parameters remained the same throughout the entire experiment to ensure that no comparative observations can be attributed to a specific fine-tuning configuration. The same applies to parameters relating to metrics; the tokenization of input sequences required for BLEU, ROUGE-L and METEOR is always done the same way, according to the ECMAScript lexical grammar specification [25].

Threats to external validity relate to factors that could affect the generalizability of the findings. In this work, generalizability mostly depends on the datasets used. We use three datasets, of which *TS1K-18* has been used in previous studies [2, 22]. Additionally, the *TS1K-22* dataset, which contains over 2.7 times more lines of code as *TS1K-18*, was created specifically for the purpose of using a larger dataset, which improves generalizability. None of the datasets overlap with the pre-training data of UniXcoder, as UniXcoder was not trained on datasets containing TypeScript code. Further research is required to assure generalizability to other programming languages.

Threats to construct validity relate to the validity of the measurements performed. The evaluation in this work uses several well-known metrics that are commonly used in the field of natural language processing [4–6, 14]. Together, these metrics give a good perspective on the performance of models all from slightly different angles, as especially BLEU, ROUGE-L and METEOR are similar but differ in key ways. Additionally, each respective JavaScript and TypeScript model for some dataset used equivalent test sets, ensuring that the results are comparable between the two models.

IX. CONCLUSION

Type annotations and comments add valuable information to source code, and using this information is key to optimizing

language understanding. This work shows that UniXcoder is able to use both to facilitate this to an extent.

The performance of code completion with UniXcoder is positively affected by the presence of type annotations. Models trained and used on code with type annotations outperformed all models trained and used on code without type annotations. An increase in the amount of type annotations increased this performance gap, showing that the presence of type annotations can help UniXcoder.

The presence of comments has been shown to be beneficial to code completion performance. However, not all types of comments are equally effective at enhancing language understanding. The relatively unstructured or inconsistent nature of comments can make it more difficult for them to be correctly interpreted and used by UniXcoder. However, future research is needed to confirm whether adding more structure to comments improves performance (for instance by only using TSDoc/JSDoc with selected tags).

Type annotations were shown to be a more reliable way of enhancing performance than comments. As said, the relatively unstructured nature of comments can make them hard to exploit. Type annotations do not have this drawback, as they follow a fixed structure and generally provide information about tokens very close to the type annotations.

Future work could investigate if the effectiveness of code completion models can be enhanced by adding type annotations using tools like LambdaNet [2] or TypeBERT [3]. This work has shown that increasing the amount of type annotations can increase code completion performance, and using such tools present a great way to increase the amount of type annotations in both dynamically and statically typed languages.

Additionally, future work could generalize the findings to other programming languages. Python 3 is a suitable candidate, as it is a dynamically typed languages that optionally allows users to add type annotations.

X. Responsible Research

A. Reproducibility

This paper heavily relied on pre-existing code that was published by Guo *et al.* to make the evaluation of UniXcoder reproducible [5]. This perfectly demonstrates the importance of reproducibility. We assure reproducibility by publishing all resources required to conduct the experiment, including the source code and datasets. Additionally, all fine-tuned models – along with their train, development, and test sets – are published to allow the prediction and evaluation phase to be reproduced individually. This is beneficial when working with little computational power, as fine-tuning all models considered in this paper takes a long time. It also lowers the barrier of entry for people wanting to experiment with the fine-tuned models.

B. Transparency

BLEU, ROUGE-L and METEOR all require parameters or tokenization. Slight differences in the way these metrics are applied can wildly change results, and differences in parameters makes the metrics incomparable [32]. This highlights the importance of indicating exactly how each metric is used. The specification of datasets is similarly important [32], and plays a big part in reproducibility. For these reasons the processes applied to obtain the results are described in detail and transparently, such that it is clear how the data should be interpreted, and whether it is comparable to data provided in other studies.

XI. Acknowledgements

I would like to extend my sincere thanks to Maliheh Izadi for the guidance and availability for any questions during the creation of this work. Additionally, I would also like to thank Georgios Gousios for providing access to a powerful machine used extensively throughout creation of this paper, and to Frank van der Heijden for providing the code used to retrieve the top-1,000 starred repositories from GitHub to create the *TS1K-22* dataset. Lastly, I would like to thank Marc Otten, Jorit de Weerdt and Mika Turk for the support throughout this research.

References

[1] S. Okon and S. Hanenberg, "Can we enforce a benefit for dynamically typed languages in comparison to statically typed ones? a controlled experiment," in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, 2016, pp. 1–10. DOI: 10.1109/ICPC.2016.7503719.

[2] J. Wei, M. Goyal, G. Durrett, and I. Dillig, "LambdaNet: Probabilistic type inference using graph neural networks," in *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*, 2020. [Online]. Available: https://arxiv.org/abs/2005.02161.

[3] K. Jesse, P. T. Devanbu, and T. Ahmed, "Learning type annotation: Is big data enough?" *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021. DOI: 10.1145/3468264.3473135.

[4] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021*, 2021.

[5] 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.

[6] S. Lu *et al.*, "CodeXGLUE: A machine learning benchmark dataset for code understanding and generation," *CoRR*, vol. abs/2102.04664, 2021. arXiv: 2102.04664. [Online]. Available: https://arxiv.org/abs/2102.04664.

[7] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019.
[8] V. Raychev, P. Bielik, and M. Vechev, "Probabilistic model for code with decision trees," *SIGPLAN Not.*, vol. 51, no. 10, pp. 731–747, 2016, ISSN: 0362-1340. DOI: 10.1145/3022671.2984041. [Online]. Available: https://doi.org/10.1145/3022671.2984041.
[9] M. Allamanis and C. Sutton, "Mining source code repositories at massive scale using language modeling," in *2013 10th Working Conference on Mining Software Repositories (MSR)*, 2013, pp. 207–216. DOI: 10.1109/MSR.2013.6624029.
[10] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," *CoRR*, vol. abs/1810.04805, 2018. arXiv: 1810.04805. [Online]. Available: http://arxiv.org/abs/1810.04805.
[11] A. Baevski, S. Edunov, Y. Liu, L. Zettlemoyer, and M. Auli, "Cloze-driven pretraining of self-attention networks," *CoRR*, vol. abs/1903.07785, 2019. arXiv: 1903.07785. [Online]. Available: http://arxiv.org/abs/1903.07785.
[12] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, "Improving language understanding by generative pre-training," 2018.
[13] C. Raffel *et al.*, "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer," *arXiv e-prints*, arXiv:1910.10683, arXiv:1910.10683, Oct. 2019. arXiv: 1910.10683 [cs.LG].
[14] A. Vaswani *et al.*, "Attention is all you need," *CoRR*, vol. abs/1706.03762, 2017. arXiv: 1706.03762. [Online]. Available: http://arxiv.org/abs/1706.03762.
[15] M. E. Peters *et al.*, "Deep contextualized word representations," *CoRR*, vol. abs/1802.05365, 2018. arXiv: 1802.05365. [Online]. Available: http://arxiv.org/abs/1802.05365.
[16] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman, "GLUE: A multi-task benchmark and analysis platform for natural language understanding," *CoRR*, vol. abs/1804.07461, 2018. arXiv: 1804.07461. [Online]. Available: http://arxiv.org/abs/1804.07461.
[17] Y. Liu *et al.*, "RoBERTa: A Robustly Optimized BERT Pretraining Approach," *arXiv e-prints*, arXiv:1907.11692, arXiv:1907.11692, Jul. 2019. arXiv: 1907.11692 [cs.CL].
[18] Z. Feng *et al.*, *Codebert: A pre-trained model for programming and natural languages*, 2020. arXiv: 2002.08155 [cs.CL].
[19] M. Ciniselli *et al.*, "An empirical study on the usage of transformer models for code completion," *IEEE Transactions on Software Engineering*, 2021, ISSN: 1939-3520. DOI: 10.1109/TSE.2021.3128234.
[20] M. Ciniselli, N. Cooper, L. Pascarella, D. Poshyvanyk, M. Di Penta, and G. Bavota, "An Empirical Study on the Usage of BERT Models for Code Completion," *arXiv e-prints*, arXiv:2103.07115, arXiv:2103.07115, Mar. 2021. arXiv: 2103.07115 [cs.SE].
[21] R. S. Malik, J. Patra, and M. Pradel, "NL2Type: Inferring javascript function types from natural language information," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 304–315. DOI: 10.1109/ICSE.2019.00045.
[22] V. J. Hellendoorn, C. Bird, E. T. Barr, and M. Allamanis, "Deep learning type inference," ser. ESEC/FSE 2018, Lake Buena Vista, FL, USA: Association for Computing Machinery, 2018, pp. 152–162, ISBN: 9781450355735. DOI: 10.1145/3236024.3236051. [Online]. Available: https://doi.org/10.1145/3236024.3236051.
[23] V. Raychev, M. Vechev, and A. Krause, "Predicting program properties from "big code"," in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '15, Mumbai, India: Association for Computing Machinery, 2015, pp. 111–124, ISBN: 9781450333009. DOI: 10.1145/2676726.2677009. [Online]. Available: https://doi.org/10.1145/2676726.2677009.
[24] H. Husain, H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "CodeSearchNet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.
[25] S. Guo, M. Ficarra, and K. Gibbons, *12 ECMAScript Language: Lexical Grammar*, Jun. 2022. [Online]. Available: https://tc39.es/ecma262/multipage/ecmascript-language-lexical-grammar.html#sec-ecmascript-language-lexical-grammar.
[26] Delft High Performance Computing Centre (DHPC), *DelftBlue Supercomputer (Phase 1)*, https://www.tudelft.nl/dhpc/ark:/44463/DelftBluePhase1, 2022.
[27] 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.
[28] C.-Y. Lin and F. J. Och, "ORANGE: A method for evaluating automatic evaluation metrics for machine translation," in *COLING 2004: Proceedings of the 20th International Conference on Computational Linguistics*, Geneva, Switzerland: COLING, Aug. 2004, pp. 501–507. [Online]. Available: https://aclanthology.org/C04-1072.
[29] 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.
[30] 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.

[31] A. Lavie, K. Sagae, and S. Jayaraman, "The significance of recall in automatic metrics for MT evaluation," *Machine Translation: From Real Users to Research*, pp. 134–143, 2004. DOI: 10.1007/978-3-540-30194-3_16.

[32] M. Post, "A call for clarity in reporting BLEU scores," 2018. DOI: 10.48550/ARXIV.1804.08771. [Online]. Available: https://arxiv.org/abs/1804.08771.