Enriching Source Code with Contextual Data for Code Completion Models
An Empirical Study

van Dam, Tim ; Izadi, Maliheh; Deursen, Arie van

DOI
10.1109/MSR59073.2023.00035

Publication date
2023

Document Version
Final published version

Published in
Proceedings of the 2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)

2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR) | 979-8-3503-1184-6/23/$31.00 ©2023 IEEE | DOI: 10.1109/MSR59073.2023.00035

Enriching Source Code with Contextual Data for Code Completion Models: An Empirical Study

Tim van Dam
Delft University of Technology
Delft, The Netherlands
t.o.vandam@student.tudelft.nl

Maliheh Izadi
Delft University of Technology
Delft, The Netherlands
m.izadi@tudelft.nl

Arie van Deursen
Delft University of Technology
Delft, The Netherlands
arie.vandeursen@tudelft.nl

***Abstract*—Transformer-based pre-trained models have recently achieved great results in solving many software engineering tasks including automatic code completion which is a staple in a developer's toolkit. While many have striven to improve the code-understanding abilities of such models, the opposite – making the code easier to understand – has not been properly investigated. In this study, we aim to answer whether making code easier to understand through using contextual data improves the performance of pre-trained code language models for the task of code completion. We consider *type annotations* and *comments* as two common forms of additional contextual information that often help developers understand code better. For the experiments, we study code completion in two granularity levels; token and line completion and take three recent and large-scale language models for source code: *UniXcoder*, *CodeGPT*, and *InCoder* with five evaluation metrics. Finally, we perform the Wilcoxon Signed Rank test to gauge significance and measure the effect size. Contrary to our expectations, all models perform better if type annotations are removed (albeit the effect sizes are small). For comments, we find that the models perform better in the presence of multi-line comments (again with small effect sizes). Based on our observations, we recommend making proper design choices when training, fine-tuning, or simply selecting such models given the intended data and application. Better evaluations and multi-modal techniques can also be further investigated to improve the practicality and accuracy of auto-completions.**

***Index Terms*—Code Completion, Transformers, Pre-trained Language Models, Context, Empirical Software Engineering**

I. Introduction

Transformer-based pre-trained models [1] originally proposed in the Natural Language Processing (NLP) field have recently been extended to the source code domain [2–4]. Thanks to natural properties of source code [5] and also the modifications to tailor these models, they are currently top performers in many code-related tasks such as automatic code completion (hereafter called auto-completion) [6–9]. Auto-completion techniques complete source code statements by suggesting the next token(s) given the current development context. They help developers program faster by correcting typographical errors, decreasing the typing effort, and facilitating API exploration [10] making auto-completion one of the most prominent features in Integrated Development Environments (IDEs).

Auto-completion utilizes two information channels; the natural language and the algorithmic channel [11]. The former explains the context of a program, while the latter specifies computer execution. Comments are a common form of optional information that can help developers understand code better, however, they do not affect how programs are run. Type annotations are another form of auxiliary information to help developers generate and/or understand code better. They can increase auto-completions' accuracy as the type of variables often directly dictates how these variables can be interacted with. However, types are often not present in dynamically-typed or optionally-typed languages. Auto-completion models often focus on code tokens and lately also on some aspects of program structure to provide better completions. Most recently, comments have also been utilized in such models [6]. Researchers have also proposed models such as LambdaNet [12] and TypeBERT [13] for the task of type inference. However, to what extent contextual information embedded in source code in the form of comments and annotated types can impact the performance of recent large-scale pre-trained language models has not been investigated yet.

In this work, we address this knowledge gap by conducting an extensive empirical investigation of the performance of recent language models for source code. We consider the three publicly available models, namely UniXcoder [6], CodeGPT [14], and InCoder [7]. We perform auto-completion in two granularity levels; next-token prediction and line completion. Moreover, we report the results based on five evaluation metrics commonly used to evaluate NLP models.

To preserve the underlying semantics of a piece of code, we add/remove optional auxiliary contextual information, i.e., type annotations and comments and generate multiple variations of the same code. To this end, we first collect a dataset containing a total of 704 TypeScript repositories from the most starred public repositories on GitHub. We then use the TypeScript compiler to create multiple variants of the same TypeScript code. The first of these variants has all type annotations removed, while the second has type annotations added to the code, given that the types can be inferred. TypeScript, being a gradually-typed language, does not mandate the presence of type annotations. The TypeScript compiler is therefore equipped with a type inference system that can deduce the types of variables without type annotations, given that the value of the variable provides enough information. The three datasets are then further processed by varying the levels of comments. We consider 1) keeping comments as-is, 2) removing all comments, 3) keeping only single-line comments, 4) keeping only multi-line comments, 5) keeping only doc-blocks, This leads to 15 datasets containing semantically-similar code, however, with different amounts of type annotations and comments. Then, we use the three models to perform automatic *token* and *line* completion on equivalent versions of TypeScript code with different amounts of type annotations. This is to establish the effect of the presence (or lack thereof) of type annotations. Additionally, we investigate the effects of removing all comments and limiting the comments to only single-line, multi-line, and doc-block comments.

Our results show that all three models perform better on untyped code than on code with type annotations. To assess the significance of the outperformance, we conduct the Wilcoxon Signed Rank test. The p-values obtained indicate that the differences are significant for all models across all evaluation metrics, meaning the differences are not random. Note that the effect sizes are small, i.e., practical significance may be limited. Based on the above, and considering that all five evaluation metrics are mostly affected in the same way, further efforts to propose better evaluation metrics that assess the value to developers are required. Additionally, our results indicate that the presence of multi-line comments significantly contributes to auto-completion performance. Similar to the previous case, although the differences are significant, the effect sizes are small in this case as well. Interestingly, although doc-block comments are a type of multi-line comment, they do not have a meaningful effect on performance relative to the baseline.

Therefore, the community should take these factors into account when selecting the appropriate auto-completion model given their purpose and application. The main contributions of this work are:

- An extensive empirical assessment of the impact of type information on three large language models for both token and line completion in TypeScript code with various amounts of type annotations,
- A comprehensive empirical assessment of the effect of natural language text information in three formats (single-line, multi-line, and doc-block comments) on the performance of these code language models for both token and line completion,
- Our source code, dataset, and select fine-tuned models are publicly available. [1]

II. Motivating Example

Figure 1 shows an example code snippet from the Angular repository with and without type annotations and comments.[2] Lack of appropriate type information or documentation can make it harder for developers to use this function properly. For instance, one might provide a string as `value` but a tree of numbers as `node`. In statically-typed languages, this would lead to a compile-time error, however, in dynamically-typed languages, this would run perfectly fine, which can lead to bugs entering the code base. It is therefore important for auto-completion models to work well in situations where type annotations are lacking to prevent users from introducing bugs to their code through auto-completion. In theory, additional type information should boost auto-completion models, as it provides them with a more comprehensive description of the source code. The same can be said about comments, which are typically used to describe complete functions with doc-blocks, or used to annotate smaller parts of code with single-line 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. Note that doc-blocks do follow a form of structure, but do not have an order and can contain a wide range of information. Investigating the impact of types and comments and their relationship on the performance of pre-trained Language Models (LMs) for auto-completion in both dynamically- and gradually-typed languages with varying amounts of type annotations gives us an understanding into what elements of source code can be used to improve the performance of auto-completion approaches.

```typescript
// Return the path to the node with the given value using DFS
function findPath<T>(value: T, node: TreeNode<T>): TreeNode<T>[] {
    if (value === node.value)
        return [node];
    for (const child of node.children) {
        const path: TreeNode<T>[] = findPath(value, child);
        if (path.length) {
            path.unshift(node);
            return path;
        }
    }
    return [];
}
```

```javascript
function findPath(value, node) {
    if (value === node.value)
        return [node];
    for (const child of node.children) {
        const path = findPath(value, child);
        if (path.length) {
            path.unshift(node);
            return path;
        }
    }
    return [];
}
```

Fig. 1. Sample code snippet with and without type annotations and comments

III. Background and Related Work

In the following, we first provide background on pre-trained models, then we review the existing work on completing code using these approaches.

1) Transformers and Pre-trained Models: Transformer-based [1] models have recently shown great promise in the

[1] https://github.com/AISE-TUDelft/ContextualDataCodeCompletion; https://huggingface.co/AISE-TUDelft/CodeGPT-TS-Multi-Untyped; https://huggingface.co/AISE-TUDelft/UniXcoder-TS-Multi-Untyped

[2] https://github.com/angular/.../#L7

area of NLP. BERT [15], a bidirectional Transformer model, showed the value of considering both the left and right context for training LMs. Liu *et al.* improved further upon BERT with the RoBERTa [16] model, and aimed to show that the performance of BERT can be further be improved through optimizing different design choices. Raffel et al. [17] proposed a Text-to-Text Transformer (T5) which treated various NLP goals as a *text-to-text* (i.e., seq2seq) task. While BERT-based models use *Masked* Language Modeling (MLM), the Generative Pretraining Transformer (GPT) architecture utilizes *Causal* Language Modeling (CLM) and is suitable for generation tasks [18]. Nowadays, Transformers are being tailored to source code to solve software engineering tasks. For instance, Feng *et al.*'s *CodeBERT* [19], builds on top of the RoBERTa model introduced above. *CodeT5* is the corresponding T5 model for the source code fine-tuned on multiple code-related tasks such as code summarisation, translation, generation, and more [20, 21]. *Codex* is an LM for code that is based on the GPT-3 architecture [22, 23]. The authors show that Codex is capable of implementing full-function implementations from textual prompts and function signatures alone. Although most research on auto-completion is focused on single-token prediction, several studies aimed to complete entire statements or blocks of code [9, 24, 25]. For instance, *AUTOSC* combines program analysis and software naturalness and fills in a partially completed statement with frequent and valid recommendations [24]. *GPT-C* is a multi-lingual model based on GPT-2, for completing lines [9]. *CodeFill* is Multi-Task Learning-based approach also based on GPT-2 for completing lines for dynamically-typed languages [8]. The authors showed using the extra information from the structure representation is beneficial for the model.

2) Studies on Type Annotations and Comments: Several previous works have demonstrated the ability to infer the types of variables and functions in dynamically-typed languages depending on their context [12, 13, 26, 27]. However, whether type information can be used to improve code understanding has not been established. *DeepTyper* [27] shows how deep learning can be applied to infer types to ease the transition from untyped code to gradually-typed code. Similar to DeepTyper [27], Malik *et al.*'s NL2Type [26] shows that natural language information in comments, functions, and parameter names can be exploited to predict types in dynamically-typed languages. Wei *et al.*'s *LambdaNet* [12] shows that deep learning can be applied to provide untyped code with type annotations in gradually-typed languages like TypeScript and Python. LambdaNet is able to predict user and third-party types, while DeepTyper is only able to predict types from a fixed vocabulary. Similar to LambdaNet [12], *TypeBERT* [13] is able to infer user and third-party types but 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. Mastropaolo *et al.* compare T5 [17] to n-gram models on a comment completion task, showing T5 [17], leverages code context to complete partial comments [28]. As previously mentioned, *Codex* [22] has shown proficiency in generating function implementations from natural language descriptions and a function signature alone. This shows that natural language can be of value to LMs for code. Additionally, these models show that it is possible for LMs to infer type information from source code. However, the opposite, whether type information can be leveraged to facilitate code understanding, has not been established. UniXcoder is a Transformer-based model proposed by Guo *et al.* [6]. Guo et al. show that UniXcoder performs slightly better on the auto-completion task when considering comments. However, the impact of type annotations and different types of comments, i.e., single-line and multi-line comments are not considered individually. Similarly, Fried *et al.* [7] show good single-token prediction performance, but do not investigate the performance of line completion, nor do they consider the influence of comments or type annotations. Note that we adapt both these models to perform line completion (more details in the approach section).

3) Empirical Studies on Auto-Completion Models: Ciniselli *et al.* [29, 30] analyzed the performance of two language models for text namely, T5 [17] and RoBERTa [16], for completing code in three granularity levels; single-token, line, and block. The authors included two datasets, containing Java methods and Android app methods from open-source GitHub repositories. They showed that T5 performs better, 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 generalizable to dynamically- or gradually-typed languages. Similar to our study, the authors aim to assess Transformer models' performance, however, the angles of their study differ from this study. While they focus on investigating the prediction granularity levels for T5 and RoBERTa, we focus on the impact of comments and annotated types on the performance of state-of-the-art large-scale LMs specifically fine-tuned on *source code*. Chirkova *et al.* [31] analyze the performance of Transformers on several code-related tasks including auto-completion. This work analyzes how well Transformer models are able to perform tasks using solely syntactic information. They show that the auto-completion task uses all AST components and that omitting types in ASTs has a negative impact on this task.

IV. Study Design

We select three of the most recently released LMs for source code that are publicly available, namely UniXcoder, CodeGPT, and InCoder. Choosing three models leads to more generalizable results across the research questions. Moreover, these models support two different language modeling objectives; masked and causal.

UniXcoder is a pre-trained model that leverages multiple modalities to facilitate several code understanding and generation tasks [6]. In addition to source code, comments and flattened Abstract Syntax Trees (ASTs) were used during pre-training to improve understanding. UniXcoder was pre-trained using MLM [15, 32], Unidirectional Language Modeling (ULM) [33], and denoising objectives [17]. Initially, UniXcoder was trained on unimodal natural language data

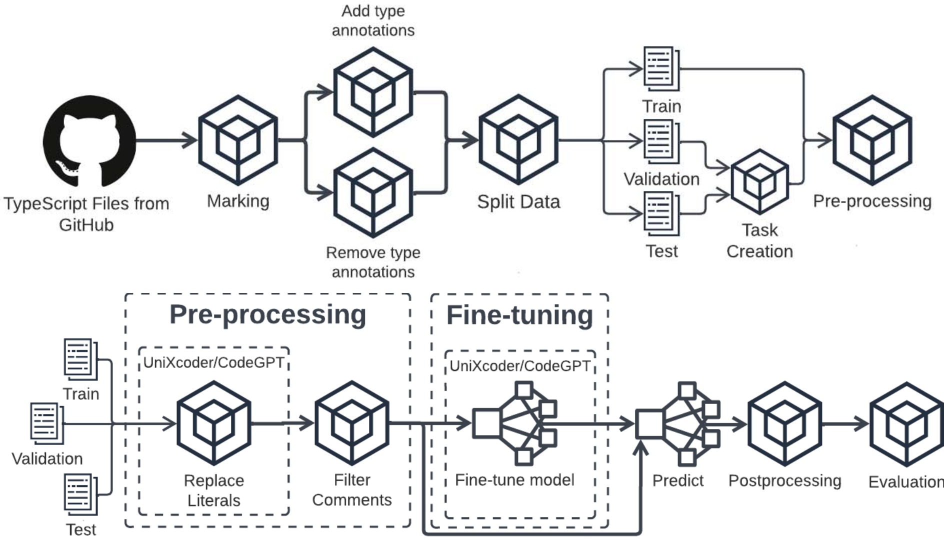

Fig. 2. Overview of the study pipeline

from the C4 [17] dataset. Afterward, it was trained on bimodal data in the form of text-code pairs from the CodeSearchNet dataset [34]. These pairs consist of function definitions and corresponding leading comments.

InCoder is a Transformer-based decoder-only model which is able to infill code based on both left and right contexts [7] using causal modeling [35]. It was trained on a large dataset consisting primarily of Python and JavaScript code from GitHub, GitLab, and StackOverflow. The training data from StackOverflow contains natural text from questions, answers, and comments in addition to code.

CodeGPT is a Transformer-based model based on the GPT-2 model. Lu *et al.* utilize the GPT-2 architecture to train several LMs for code-related tasks. The authors train four models, two for Python and two for Java: for each programming language, one model solely used the GPT-2 architecture, and one used the pre-trained text-based GPT-2 as a starting checkpoint. The latter methodology was used for auto-completion by Lu *et al.* and is consequently what is referred to as CodeGPT in this work.

Our study consists of seven main phases, namely *marking*, *type inference*, *data splitting*, *pre-processing*, *hyper-parameter tuning*, *fine-tuning*, and *post-processing*. During marking, we add comments to the TypeScript code in our dataset to determine where we will perform auto-completion. Afterward, we create equivalent marked datasets with 1) all type annotations removed, 2) all implicit types added, and 3) where type annotations are left as-is. For all three datasets, we divide the files into train, test, and validation sets according to an $80/10/10$ percent split. There are 5 pre-processing modes that all handle comments differently: 1) keep all comments, 2) remove all comments, 3) keep only multi-line comments, 4) keep only single-line comments, and 5) keep only doc-blocks. This results in 15 unique datasets. Next, we perform hyper-parameter tuning, after which we use these datasets to fine-tune 15 models for UniXcoder and CodeGPT. Afterward, we use the fine-tuned models to generate predictions for the test set. Finally, we post-process the predictions.

Note that we do not fine-tune InCoder: it does not have publicly available fine-tune code and has already been trained on data containing TypeScript code. This also has an effect on the pre-processing step for InCoder, as discussed later.

Figure 2 depicts the overall pipeline of our study. Note that different paths are taken depending on the auto-completion model in use. Next, we describe the steps in more detail.

a) Marking: When testing the fine-tuned models, marking indicates where auto-completion should be performed. Throughout a piece of code, we add placeholder comments in the form of `/*<marker:i>*/`, where i is a number that we use to identify markers within a file. These comments are later used to create the auto-completion tasks for the test and validation sets. We perform this task prior to other phases to ensure that each dataset has identical completion tasks: comments will not move as we transform code, hence we can use them to track a position in code even when transformations occur during type inference and pre-processing. For marking, we first randomly select 40% of non-empty lines. Then, for each selected line, we place a marker in front of an ECMAScript token that is not a *newline*, *whitespace*, *comment*, or *type annotation*. The code to the left of this marker will serve as input, and the code to the right of this marker up to the end of the line will serve as ground truth.

b) Type Inference: In order to ascertain the influence of the number of types on performance, we transform our TypeScript dataset in order to create two additional datasets: one of which makes all implicit types explicit, and one of which is without any type annotations. In the former case, we for instance transform `const x = 5 * 10;` to `const x:`

TABLE I
TUNED HYPER-PARAMETERS

Model	Hyper-param	Values	Best
UniXcoder	Learning Rate	{1e-5, 7.33e-5, 1.37e-4, 2e-4}	7.33e-5
	Batch Size	{2, 4, 8}	4
CodeGPT	Learning Rate	{1e-5, 7.33e-5, 1.37e-4, 2e-4}	1.37e-4
	Batch Size	{2, 4, 8}	2

`number = 5 * 10;`. The opposite transformation would happen in the latter case. This results in three datasets, with three different amounts of type annotations.

This step is more elaborately discussed in section V.

c) Data splitting: We split all TypeScript files into the train, test, and validation sets. All three datasets contain the same files, so we maintain one distribution for all datasets. This ensures that all models produce comparable results. The files are split according to an $80/10/10$ percent split.

d) Pre-Processing: We pre-process the data to prepare for fine-tuning and evaluation. First, we normalize spacing and linebreaks by replacing consecutive spaces and linebreaks with a single such character. Then, the following steps are applied to prepare the data for fine-tuning UniXcoder and CodeGPT: As a standard technique to reduce the vocabulary size in the literature [8, 9, 36, 37], we normalize the number and string literals by replacing them with special tokens, ⟨NUM_LIT⟩, and ⟨STR_LIT⟩, respectively. Then, we replace line breaks with the special ⟨EOL⟩ token. These pre-processing steps are not applied to the data that is fed to InCoder, as we do not fine-tune this model. Consequently, the InCoder expects raw code, without the special tokens that the pre-processing phase adds.

We then apply different variants of pre-processing regarding comments; 1) keep all comments as-is, 2) remove all comments, 3) keep only single-line comments, 4) keep only multi-line comments, and finally 5) keep only doc-blocks. This process results in 15 different collections of TypeScript files (three type variants, and five comment variants).

The pre-processed data is then stored in files to be used by the three models. To prevent large files from representing a large part of the validation and test sets, the maximum amount of code completion tasks per file is set at 15.

e) Hyper-parameter Tuning: We perform hyper-parameter tuning on 25% of the training and validation set of the dataset with the original types (TS704-OT) and unmodified comments to perform hyper-parameter tuning. We choose to use 25% of one dataset to limit the computational burden of our experiments. Note that fine-tuning *does* use 100% of all datasets. Table I reports the tuned hyper-parameters, the values that were tested, and the combination of hyper-parameters which led to the highest accuracy.

f) Fine-tuning: Next, we fine-tune UniXcoder and CodeGPT on all our datasets. That is, on each type-annotation variant and each comment-variant. This results in 15 different datasets, thus 15 different fine-tuned models for UniXcoder and CodeGPT. We use the hyper-parameters found during hyper-parameter tuning. Note that we do not fine-tune InCoder, as it does not have publicly available fine-tune code. Furthermore, it was already trained on TypeScript code, hence it is expected to be able to provide reasonable predictions on the test set without fine-tuning. We use a standard language modeling objective, predicting the next token given a context, and maximize the following likelihood. In Equation 1, m is the length of the predicted sequence of code token values and θ is the set of parameters that is learned through stochastic gradient descent optimization to model P [38].

$$L(V) = \sum_{i} \log P(v_i | c_0, ..., c_T, v_{i-m}, ..., v_{i-1}; \theta). \quad (1)$$

UniXcoder and CodeGPT treat the ⟨EOL⟩ tokens as the end of a sequence token, resulting in the fine-tuned models predicting up to the end of each line. To emulate this behavior in InCoder, we add a stopping criterion to the model that detects whenever a new line is among the generated tokens. If a new line is detected, we stop the model from generating more tokens.

g) Post-processing: After fine-tuning, the 30 fine-tuned models plus InCoder are used to generate predictions for the 15 test sets. These predictions are subsequently post-processed which consists of normalizing the spacing of code tokens (including line breaks), removing all comments, and replacing tokenized versions of literals with default literals. That is, ⟨STR_LIT⟩ becomes `""` (an empty string), and ⟨NUM_LIT⟩ becomes `0`. Since the data fed to InCoder does not contain tokenized versions of literals, we replace the raw string and number literals in the InCoder's predictions by these constants. To have a consistent and fair evaluation, this process is applied to both the prediction and the ground truth.

V. EXPERIMENTAL SETUP

We first present our Research Questions (RQ) and describe the datasets used for the experiments. Next, we introduce the evaluation metrics used to assess the models' performance in detail. Finally, we review the implementation details.

A. Research Questions

We aim to assess the impact of additional contextual information, e.g., explicit type annotations and textual explanations in the form of comments on the performance of three state-of-the-art large-scale pre-trained models for source code. For both of our RQs, we consider three code LMs, i.e., UniXcoder, CodeGPT, and InCoder, and two auto-completion tasks, namely token and line completion. Accordingly, we design our experiments to answer the following RQs.

- **RQ1: How is the performance of these models influenced by the ratio of available type annotations in code?** That is, whether these models perform significantly differently depending on the degree to which a piece of code is type-annotated. To provide a fair comparison, we use a TypeScript dataset, and create its equivalent code without type annotations using the TypeScript compiler. Moreover, to assess whether adding additional type annotations affect the performance of models, we add more

```
function solveQuadratic(a: number, b: number, c: number) {
    const d = b ** 2 - 4 * a * c;
    const denom = 2 * a;
    const sol1 = (-b + Math.sqrt(d)) / denom;
    const sol2 = (-b - Math.sqrt(d)) / denom;
    return [sol1, sol2];
}
```

```
function solveQuadratic(a: number, b: number, c: number): number[] {
    const d: number = b ** 2 - 4 * a * c;
    const denom: number = 2 * a;
    const sol1: number = (-b + Math.sqrt(d)) / denom;
    const sol2: number = (-b - Math.sqrt(d)) / denom;
    return [sol1, sol2];
}
```

Fig. 3. A sample code snippet before/after adding additional type annotations

annotations to the TypeScript dataset to make it more explicitly typed and run the same experiments on this new dataset.

- **RQ2: What is the impact of enriching source code context with textual information, i.e., comments?** This question explores how the performance of the three models is affected by the presence of different types of comments including single-line, multi-line, and doc-block comments. We compare these results against the results obtained from the respective datasets with no comments.

B. Datasets

To perform the experiments, we use publicly available GitHub repositories that predominantly consist of TypeScript code. **TS704-OT**, the first dataset, consists of a subset of the top-1000 starred repositories on GitHub. This dataset was retrieved by querying the GitHub Search API.[3] The GitHub API returned a total of 851 unique repositories. To prevent bias, we deduplicate our dataset against the repositories used for (pre-)training UniXcoder, CodeGPT, and InCoder. To answer the first RQ, we created an additional dataset based on the TS704-OT dataset where we remove all type annotations (**TS704-NT**). To do so, we use the TypeScript Compiler API to traverse the Abstract Syntax Tree of every TypeScript file, removing any type annotations that are encountered. Additionally, we created a dataset where we make all implicit types explicit using the TypeScript compiler (**TS704-AT**). As TypeScript is a gradually-typed language, types are not required but can oftentimes be inferred based on the types of other variables or constants. We use the TypeScript compiler to add type annotations when they can be inferred by the compiler, attempting to amplify a potential effect caused by the presence of type annotations. This process is displayed with a sample code snippet in Figure 3. TypeScript code may depend on type annotations that are defined in third-party dependencies. Hence, we first install all third-party dependencies using *npm* (Node Package Manager) to be able to infer these types. More specifically, we run the `npm install --ignore-scripts` command in each directory containing a `package.json` file (`package.json` files contain dependency information). We opt to ignore post-install scripts, as `npm` packages made for TypeScript ship with type declarations as-is, meaning that no additional scripts are required to retrieve all third-party types. This also significantly speeds up the installation process. Not all dependencies were available, hence we removed all projects with unavailable dependencies from all three datasets. This decreased the number of repositories in our datasets by 147. After having installed the dependencies, we locate all directories containing `tsconfig.json` files. These directories are at the root of TypeScript projects, and contain configuration options for the TypeScript compiler. We use the TypeScript compiler to load in these TypeScript projects and traverse the AST of each TypeScript file (files with the `.ts` extension), adding type annotations where possible using the TypeScript Compiler Type Checker. The resulting dataset The datasets are nearly identical: the sole difference is the number of type annotations.

TABLE II
DATASETS USED FOR FINE-TUNING AND EVALUATION

	TS704-OT	TS704-NT	TS704-AT
#Repositories	704	704	704
#Files	174,500	174,500	174,500
#LOC	26,115,719	25,548,595	26,115,719
Type Explicitness	30.95%	0.00%	95.62%

Table II shows the size of our datasets in terms of the number of repositories, files, and lines of code (LOC). Additionally, it displays the *type explicitness* of the code in these datasets. Type explicitness refers to the number of type annotations present in the code relative to the maximum amount of type annotations that can be present in the code.

C. Evaluation Metrics

We compare the predictions made by the models, fine-tuned on typed and non-typed code, against the ground truths using several evaluation metrics. In this study, we include a wide range of standard metrics commonly used for evaluating line completion solutions to provide a more comprehensive assessment [8, 9, 39]. As our focus is on assessing the performance of these models in various settings, we review these metrics in detail.

EM (Exact Match) compares ground truths with predictions and returns a boolean value. The EM score over an entire dataset is expressed as a percentage. Higher values are better.

ES (Edit Similarity) or Levenshtein 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. ES is a number in the range $[0, 1]$ that is computed by dividing the Levenshtein distance by the

[3] https://docs.github.com/en/rest/search?apiVersion=2022-1..repositories

length of either the prediction or the ground truth, depending on which is the 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 to the total amount of n-grams in the ground truth [40]. We apply BLEU-4 by treating each code token, as per the ECMAScript lexical grammar specification [41], as a unigram, and give each value of n an equal weight. A smoothing technique [42] is used to prevent division by zero when the prediction has fewer than four tokens.

ROUGE-L is a variant of the ROUGE (Recall-Oriented Understudy for Gisting Evaluation) metric which uses the Longest Common Subsequence algorithm 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 [43].

METEOR (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. Additionally, a penalty is applied based on how well the true unigram order is followed by the mapping. METEOR has been shown to be better at capturing human judgment over a complete dataset than BLEU [44]. Opposed to BLEU, METEOR puts more weight on recall, which has been shown to align closer to human judgment than precision [45]. We apply METEOR with parameters $\alpha = 0.9$, $\beta = 3.0$, and $\gamma = 0.5$.

D. Implementations and Configuration

We use `ts-morph` to interface with the TypeScript compiler API when removing or adding type annotations.[4] We use `js-tokens` for tokenizing TypeScript code during the line completion task creation, pre-processing, and evaluation phases to add and remove specific types of comments.[5]

We fine-tune UniXcoder using the fine-tune source code published by its authors.[6] We also fine-tune CodeGPT using the published code.[7] Slight tweaks were made to both scripts to make them compatible with our dataset files.

For UniXcoder, we use an initial learning rate of 7.33e-5 and a batch size of 4, as per the results of hyperparameter tuning (Table I). We set the maximum input sequence length to 936, the maximum output sequence length to 64, the beam size to 5, The remaining parameters are set to the default values as per the fine-tuning code published by the authors of UniXcoder. We then train UniXcoder for 10 epochs. For CodeGPT, we use an initial learning rate of 1.37e-4 and a batch size of 2 (Table I). The remaining parameters are set to the default values. We then train CodeGPT for 10 epochs.

We do not fine-tune InCoder, as it does not have publicly available fine-tune code. We use the InCoder model with 1 billion parameters, which is available on HuggingFace.[8] While InCoder is capable of providing completions given left *and right* context, we choose to only provide a left context as input, because UniXcoder and CodeGPT only support left contexts. For inference, we apply InCoder with a temperature of 0.2, and a $p = 0.95$ for top-p nucleus sampling.

TABLE III
RQ1: IMPACT OF TYPE ANNOTATIONS (LINE COMPLETION, DATA WITHOUT COMMENTS)

Model	Types	EM	ES	B4	RL	MR
UniXcoder	NT	**65.32**	**79.62**	**61.63**	**81.32**	**65.91**
	OT	58.85	73.68	58.25	75.83	63.05
	AT	59.93	74.41	58.82	76.39	63.56
CodeGPT	NT	**63.39**	**80.38**	**62.08**	**82.51**	67.03
	OT	60.29	78.00	61.30	80.63	66.80
	AT	61.32	78.74	61.86	81.13	**67.25**
InCoder	NT	**33.26**	**56.56**	**43.51**	**59.71**	**51.09**
	OT	32.00	54.99	42.70	58.15	50.04
	AT	32.01	54.76	42.36	57.82	49.77

We conducted the experiments with models on a cluster equipped with NVIDIA Tesla V100S GPUs, an AMD EPYC 7402 24C @ 2.80GHz CPUs. We ran hyper-parameter tuning, fine-tuning, and inference on this cluster using one GPU per process, four CPUs per process, and 48 GB RAM per process.

Running hyper-parameter tuning on UniXcoder and CodeGPT took around 34 hours per model. Fine-tuning these models took roughly 5 days per model. Inference on the test set took roughly 6 hours for every model, including InCoder.

VI. RESULTS AND DISCUSSION

A. RQ1: Impact of Type Annotations

To gauge the impact of type annotations on auto-completion performance, we run the three models against the test sets of datasets with different type explicitness ratios. We then report the results for each task, namely line and token completion. Next, to determine which performance difference is significant we perform the Wilcoxon Signed Rank test. In the following tables, we abbreviate BLEU-4, ROUGE-L, and METEOR metrics to B4, RL, and MR, respectively.

1) Line Completion: Table III presents how each model performed on comment-less code with and without type annotations. The displayed data is retrieved by running the models against the test set, computing every metric for each prediction, and averaging the metrics. We then use the Wilcoxon Signed Rank test to determine whether different type explicitness rates significantly impact auto-completion performance. Specifically, we compare TS704-NT with TS704-OT, and TS704-OT with TS704-AT. The test sets contain a total of 153,147 paired code completion tasks. We obtain paired samples from these tasks by pairing the metric values computed for predictions done by the two models under test. This is done for every metric. The family-wise significance level of the tests is $\alpha = 0.05$. Additionally, we compute Cliff's delta to measure

[4] https://www.npmjs.com/package/ts-morph/v/15.1.0

[5] https://www.npmjs.com/package/js-tokens/v/8.0.0

[6] https://github.com/microsoft/CodeBERT/tree/master/UniXcoder

[7] https://github.com/microsoft/CodeXGLUE/.../CodeCompletion-line

[8] https://huggingface.co/facebook/incoder-1B

TABLE IV
RQ1: Impact of Type Annotations (Line Completion, data without comments, Wilcoxon Signed Rank p-values, Cliff's Delta)

Model	Types 1	Types 2	Metric	p	δ
UniXcoder	NT	OT	EM	0.000	0.065
	NT	OT	ES	0.000	0.080
	OT	AT	EM	0.000	−0.011
	OT	AT	ES	0.000	−0.011
CodeGPT	NT	OT	EM	0.000	0.021
	NT	OT	ES	0.000	0.026
	OT	AT	EM	0.000	−0.010
	OT	AT	ES	0.000	−0.011
InCoder	NT	OT	EM	0.000	0.012
	NT	OT	ES	0.000	0.021
	OT	AT	ES	0.000	0.003

the effect size. Table IV reports partial results of the statistical tests. The B4, RL, and MR metrics are omitted for brevity, as these were significant in all cases.

Removing type annotations leads to the best performance for all three models. In all cases, this outperformance is statistically significant but quite small – especially for InCoder. This performance difference may be explained by the fact that UniXcoder and InCoder are pre-trained on corpora that include large amounts of JavaScript, which is nearly identical to TypeScript without type annotations. However, it should be noted that even CodeGPT, which does not have any source code in its pre-training data, also shows better performance on code without type annotations. This opposes the rationale that auto-completion performance can be enhanced by adding type annotations. A possible explanation could be that source code tokens are more valuable to the models than type annotations, and removing type annotations leaves more input space for source code tokens.

Adding type annotations to partially-typed code improves auto-completion, but not as much as removing type annotations altogether. For UniXcoder and CodeGPT, adding type annotations led to a statistically significant performance improvement for all metrics. For InCoder, a similar improvement was observed for all metrics except for Exact Match. In all cases the effect size is much smaller than the effect size caused by removing all type annotations, suggesting limited practical utility. While this can not be determined from our experiments alone, these findings may suggest that strongly typed languages are easier to interpret than gradually-typed languages such as TypeScript. However, the fact that removing type annotations led to a bigger performance increase once more suggests that source code may be a more important part of the input than type annotations.

TABLE V
RQ1: Impact of Type Annotations (Token Completion, Data without Comments)

Model	Types	EM	ES
UniXcoder	NT	**78.28**	**81.36**
	OT	70.80	73.88
	AT	71.81	74.74
CodeGPT	NT	**78.47**	**82.02**
	OT	75.35	78.98
	AT	76.42	79.93
InCoder	NT	**51.46**	**57.05**
	OT	50.94	56.61
	AT	51.15	56.64

TABLE VI
RQ1: Impact of Type Annotations (Token Completion, data without comments, Wilcoxon Signed Rank p-values, Cliff's Delta)

Model	Types 1	Types 2	Metric	p	δ
UniXcoder	NT	OT	EM	0.000	0.075
	NT	OT	ES	0.000	0.081
	OT	AT	EM	0.000	−0.010
	OT	AT	ES	0.000	−0.010
CodeGPT	NT	OT	EM	0.000	0.025
	NT	OT	ES	0.000	0.027
	OT	AT	EM	0.000	−0.011
	OT	AT	ES	0.000	−0.011
InCoder	NT	OT	EM	0.000	0.004
	NT	OT	ES	0.000	0.004
	OT	AT	EM	0.000	−0.002

2) Token Completion: Table V presents how each model performed on the token completion task on comment-less code. We only report Exact Match and Edit Similarity, as all other metrics are sequence-level metrics that do not work on single tokens. We perform the Wilcoxon Signed Rank test on these results, as displayed in Table VI.

Removing type annotations once again leads to the best performance. The performance gain is statistically significant for all models, with effect sizes similar to those found for line-level completion.

Adding type annotations improves auto-completion performance. This outperformance is statistically significant for all metrics for UniXcoder and CodeGPT, and statistically significant only for Exact Match for InCoder. The effect sizes are small across all models.

Overall, the results for token completion are similar to the results for line completion. Removing type annotations leads to a larger performance gain than adding type annotations. The reason for this is not clear from our experiments alone, however, we theorize that this could indicate that the tested models are better at interpreting source code than type annotations, making it worthwhile to remove type annotations to make more space for source code tokens.

Answer to RQ1: All models perform best on untyped code across nearly all metrics. The Wilcoxon Signed Rank test shows the performance difference between untyped (TS704-NT) and type-annotated code (TS704-OT) is significant. This suggests that type annotations do not necessarily enhance auto-completion models' ability to interpret and complete code. Additionally, performance on code with a high type explicitness (TS704-AT) is significantly better than performance on code with a normal type explicitness ratio (TS704-OT). This could suggest that the irregular nature of type annotations in gradually-typed languages may make it more difficult for language models to interpret optionally-typed languages. Comparing effect sizes indicates that it is a better option to strip (rather than add) type annotations from TypeScript code to improve auto-completion performance. We theorize that these observations suggest that source code provides more value to these models, rather than type annotations. Consequently, removing type annotations to allow more source code to be used by the models can lead to increased performance.

B. RQ2: Impact of Comments

We run UniXcoder, CodeGPT, and InCoder against the test sets of TS704-NT with different types of comments to determine whether specific types of comments influence auto-completion performance. Similarly, we use the Wilcoxon Signed Rank test to determine significance, and consider both line and token completion.

1) Line Completion: Table VII shows how each model performed on code containing different types of comments. In this table, and the tables thereafter, the *CMT* column indicates which types of comments are present in the code. Possible values are **NC** (No Comments), **SL** (Single-Line comments), **ML** (Multi-Line comments), **DB** (Doc-Block comments), and **AC** (All Comments).

We conduct the Wilcoxon Signed Rank test to determine whether any observations are significant. We use performance on comment-less data as a baseline, and compare it to performance on all other types of comments. We also compute Cliff's delta. The results are shown in Table VIII. The B4, RL, and MR metrics are once more omitted for brevity, as these were significant in nearly all cases, and show great correlation to EM and ES.

Preserving **All Comments or Multi-Line Comments** leads to the best performance for all models and all metrics. This implies that the information embedded in multi-line comments leads to the best understanding of the source code. UniXcoder, CodeGPT, and InCoder are all (pre-)trained on corpora containing English text, which could explain their outperformance on code with multi-line comments relative to their performance on code without such comments. The outperformance on these types of comments is statistically significant for all three models, albeit with small effect sizes. Keeping All Comments versus only multi-line comments have similar effects for UniXcoder and CodeGPT, but for InCoder specifically, multi-line comments have a larger effect size.

Preserving solely **Doc-Block Comments** does not lead to substantial nor significant performance gains, despite Doc-Blocks being a type of multi-Line comment. This reinforces that the natural language descriptions inside multi-line comments cause performance enhancements, rather than Doc-Block information such as argument types and purposes.

Keeping only **Single-Line Comments** generally does not cause performance enhancements like preserving only multi-line comments. This could be explained by the differences in comment length: single-line comments are generally much smaller than multi-line comments. For UniXcoder, Single-Line comments appear to cause a very small, yet statistically significant performance increase. For InCoder, however, these comments have a statistically significant negative effect, one more with a small effect size. Single-Line comments do not significantly impact the performance of CodeGPT.

TABLE VII
RQ2: IMPACT OF COMMENTS ON LINE COMPLETION, TS704-NT

Model	CMT	EM	ES	B4	RL	MR
UniXcoder	NC	65.32	79.62	61.63	81.32	65.91
	SL	65.84	80.30	62.16	82.03	66.41
	ML	**69.28**	82.99	63.78	84.35	67.65
	DB	65.17	79.56	61.55	81.24	65.78
	AC	69.27	**83.09**	**63.88**	**84.46**	**67.76**
CodeGPT	NC	63.39	80.38	62.08	82.51	**67.03**
	SL	62.85	79.97	61.72	82.08	66.68
	ML	**66.90**	**82.56**	**62.82**	**84.26**	66.95
	DB	63.65	80.42	62.10	82.48	67.02
	AC	66.45	82.29	62.64	84.01	66.78
InCoder	NC	33.26	56.56	43.51	59.71	51.09
	SL	32.30	55.86	43.09	59.05	50.68
	ML	**34.89**	**57.78**	**44.13**	**60.67**	**51.37**
	DB	33.27	56.49	43.49	59.56	50.99
	AC	33.99	57.02	43.64	59.91	50.88

TABLE VIII
RQ2: IMPACT OF COMMENTS ON LINE COMPLETION, TS704-NT
(WILCOXON SIGNED RANK p-VALUES, CLIFF'S DELTA)

Model	CMT 1	CMT 2	Metric	p	δ
UniXcoder	NC	SL	EM	0.000	−0.003
	NC	SL	ES	0.000	−0.005
	NC	ML	EM	0.000	−0.031
	NC	ML	ES	0.000	−0.036
	NC	DB	EM	0.000	0.003
	NC	DB	ES	0.003	0.003
	NC	AC	EM	0.000	−0.032
	NC	AC	ES	0.000	−0.039
CodeGPT	NC	ML	EM	0.000	−0.020
	NC	ML	ES	0.000	−0.022
	NC	DB	EM	0.019	−0.001
	NC	AC	EM	0.000	−0.020
	NC	AC	ES	0.000	−0.023
InCoder	NC	SL	EM	0.000	0.008
	NC	SL	ES	0.000	0.009
	NC	ML	EM	0.000	−0.017
	NC	ML	ES	0.000	−0.019
	NC	AC	EM	0.000	−0.010
	NC	AC	ES	0.000	−0.010

Overall, different types of comments have a statistically significant effect on line-level auto-completion performance, albeit with relatively small effect sizes. multi-line comments appear to have the largest positive effect, indicating that the three models are most capable at interpreting these types of comments.

TABLE IX
RQ2: IMPACT OF COMMENTS ON TOKEN COMPLETION, TS704-NT

Model	CMT	EM	ES
UniXcoder	NC	78.28	81.36
	SL	78.83	81.91
	ML	**82.35**	**85.23**
	DB	78.35	81.39
	AC	82.30	**85.23**
CodeGPT	NC	78.47	82.02
	SL	77.85	81.44
	ML	**80.99**	**84.31**
	DB	78.42	81.86
	AC	80.66	84.05
InCoder	NC	51.46	57.05
	SL	50.45	56.21
	ML	**52.47**	**58.06**
	DB	51.32	56.92
	AC	51.47	57.22

TABLE X
RQ2: IMPACT OF COMMENTS ON TOKEN COMPLETION, TS704-NT
(WILCOXON SIGNED RANK p-VALUES, CLIFF'S DELTA)

Model	CMT 1	CMT 2	Metric	p	δ
UniXcoder	NC	SL	EM	0.000	−0.004
	NC	SL	ES	0.000	−0.004
	NC	ML	EM	0.000	−0.032
	NC	ML	ES	0.000	−0.034
	NC	AC	EM	0.000	−0.033
	NC	AC	ES	0.000	−0.036
CodeGPT	NC	SL	EM	0.006	0.002
	NC	SL	ES	0.003	0.002
	NC	ML	EM	0.000	−0.017
	NC	ML	ES	0.000	−0.019
	NC	AC	EM	0.000	−0.018
	NC	AC	ES	0.000	−0.019
InCoder	NC	SL	EM	0.000	0.009
	NC	SL	ES	0.000	0.009
	NC	ML	EM	0.000	−0.011
	NC	ML	ES	0.000	−0.012
	NC	AC	EM	0.002	−0.002
	NC	AC	ES	0.000	−0.004

2) Token Completion: Table IX shows the token completion performance of all models on code with different types of comments. We only report Exact Match and Edit Similarity, as the other metrics are not applicable to single tokens. We once more perform the Wilcoxon Signed Rank Test on the results, as shown in Table X. Overall, the results are similar to the results observed for line completion. Preserving **All Comments or Multi-Line Comments** once more leads to the best performance. This outperformance is significant and has the largest effect size for all three models. Once more, UniXcoder and CodeGPT have similar effect sizes when preserving multi-line and all comments, whereas InCoder appears to benefit more when preserving only multi-line comments.

Preserving only **Doc-Block Comments** once again does not lead to significant performance improvements, despite doc-block comments being a type of multi-line comment. Keeping only **Single-Line** comments significantly affects performance for all models. For UniXcoder and InCoder there are slightly positive and negative effect sizes respectively, which was also the case for line completion. For CodeGPT, we observe a slight, statistically significant, negative effect, which was not observed during line completion.

Overall, the results for token completion are consistent with the results for line completion: different types of comments influence auto-completion performance in different ways, with small effect sizes. Multi-Line comments appear to have the largest positive effect for the three models.

Answer to RQ2: All three models perform best on a code containing either all comments or solely multi-line comments. The presence of doc-block comments (which is a type of multi-line comment) does not cause a significant performance increase, suggesting that the value of multi-line comments comes from the natural language embedded in them. Additionally, single-line comments do not always appear to have the same effect: UniXcoder slightly benefits from the presence of single-line comments, but CodeGPT and InCoder experience performance degradation when present. While code containing multi-line comments performs best, the effect size is relatively small. Nevertheless, these types of comments do provide some value to code completion models. The results suggest that the three code completion models can adequately interpret natural language descriptions contained in multi-line comments. Other comment types do not appear to further enhance auto-completion performance, suggesting that these comments can be omitted from the input without sacrificing performance.

C. Discussion and Recommendations

The experiments' results indicate that the observed differences in auto-completion performance are statistically significant, hence they are not random. However, the effect sizes are small, which can indicate limited impact in practice.

In this work, we focused on TypeScript. More studies are required to generalize our findings including investigating whether the observed results apply to other programming languages or not. Python3 is a suitable candidate, as it is an optionally-typed language. However, it lacks a type inference engine that is present in the TypeScript compiler, which makes this more challenging. Additionally, experiments with different splitting policies (e.g., repository-based instead of file-based), other datasets, and more strictly de-duplication policies could reinforce our findings.

The results hint at the removal of type annotations being beneficial, which may suggest that current LLMs do not need additional clues such as type annotations to aid code understanding. This is specifically important as the input size of LLMs is generally limited – removing type annotations can make space for more important information. Examples of alternative types of contextual information that could be of more use to these models are 1) available function signatures, 2) local folder structure, 3) previous (correct) predictions, and 4) simply more code context. Our results indicate that predominantly multi-line comments are of importance for code completions. Removing all other types of comments can similarly create extra input space for other types of useful information.

In general, we believe that helping recent LLMs understand the input syntactically may not be the best way of advancing state-of-the-art models with millions or billions of parameters. Instead, providing different types of contextual information may help widen the scope of code completion models. Future research is needed to confirm how different types of contextual clues may be valuable to LLMs used for code-related tasks.

D. Threats to the Validity

We categorize threats to the validity of our study into three groups, namely internal, external, and construct threats.

Threats to internal validity: relate to the parameters affecting the performance of the model, factors unintentionally influencing the results, and errors in the implementations. We intentionally changed type annotations and comments to test their relationship with auto-completion performance. Variables involved in this work are fine-tuning hyper-parameters and metric-related parameters. Hyper-parameter tuning was performed before fine-tuning UniXcoder and CodeGPT, after which the hyper-parameters that lead to the best accuracy were kept constant throughout the experiments. This ensures that no observed differences between fine-tuned models can be attributed to a difference in the fine-tuning configuration. The same applies to parameters relating to metrics; the tokenization of input sequences required for BLEU, ROUGE-L and METEOR was always done the same way, according to the ECMAScript lexical grammar specification [41]. Moreover, our analysis relies on pre-existing code published by Guo *et al.* and Lu *et al.* to make the evaluation of UniXcoder and CodeGPT reproducible [6, 14]. 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, we provide fine-tuned models for UniXcoder and CodeGPT for all three type explicitness settings, trained on data containing all comment types.

Threats to external validity: relate to factors that could affect the generalizability of our findings. In this study, the datasets used greatly impact the generalizability of results. We create our own dataset, *TS704-OT*, based on which we create two more datasets, *TS704-AT* and *TS704-NT*, through adding and removing type annotations. To prevent skewed results, we removed all repositories in our dataset that were also used to train UniXcoder, CodeGPT, or InCoder. However, our datasets themselves were not de-duplicated, which could lead to overly-optimistic results caused by the overlap of the train and test sets [46]. Our duplication measurements indicate that there is only 1% exact duplication within our dataset, and 7% near-duplication. We believe this duplication is relatively small and aligns with real scenarios as in practice developers tend to reuse code frequently. Additionally, the datasets were split by file, as opposed to by project, or by repository. This could cause artificially high measurements as projects generally share patterns or variable names [47]. Hence, different splitting policies should be explored in the future. Future work could also incorporate different optionally-typed languages, and different datasets to further support our findings.

Threats to construct validity: relate to the validity of the measurements performed. We use several commonly-used metrics in the NLP field [1, 6, 14, 20]. Together, these metrics give a broad perspective on the performance of models, as each metric measures performance in a different way. 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 make the metrics incomparable [48]. This highlights the importance of indicating exactly how each metric is used. The specification of datasets is similarly important [48] 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. To ensure that all data reported are comparable, we used the same distribution of files in the train, validation, and test sets for all experiments.

VII. Conclusion

Source code tokens are not the only sources of information for exploiting the context when using code LMs. Optional type annotations and natural language text in the form of comments can add valuable information to the source code. However, not all this extra information may be useful for improving language understanding capabilities of auto-completion models. In this work, we investigated the impact of these sources of additional information when leveraged by three recent LMs for source code. Our results show that not all optional information channels are valuable to the three models. Namely, type annotations are shown to negatively affect these models, whilst their performance is enhanced by the presence of multi-line comments. These observations are statistically significant for all models and nearly all metrics (albeit with small effect sizes). Hence, the community should take these findings into account when selecting the best model for their tasks. For instance, exchanging type annotations and non-multi-line comments in the input of these models for other more beneficial types of contextual information can be helpful to these models. Future research can investigate how different types of contextual clues can impact the performance of LLMs for code.

VIII. Data Availability Statement

Our source code, dataset, and select fine-tuned models are publicly available.[9]

[9] https://github.com/AISE-TUDelft/ContextualDataCodeCompletion; https://huggingface.co/AISE-TUDelft/CodeGPT-TS-Multi-Untyped; https://huggingface.co/AISE-TUDelft/UniXcoder-TS-Multi-Untyped

References

[1] 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.

[2] A. Al-Kaswan, M. Izadi, and A. van Deursen, "Stacc: Code comment classification using sentencetransformers," 2023.

[3] M. Izadi, P. R. Mazrae, T. Mens, and A. van Deursen, "Linkformer: Automatic contextualised link recovery of software artifacts in both project-based and transfer learning settings," *arXiv preprint arXiv:2211.00381*, 2022.

[4] M. Izadi, "Catiss: An intelligent tool for categorizing issues reports using transformers," in *2022 IEEE/ACM 1st International Workshop on Natural Language-Based Software Engineering (NLBSE)*, IEEE, 2022, pp. 44–47.

[5] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. Devanbu, "On the naturalness of software," *Communications of the ACM*, vol. 59, no. 5, pp. 122–131, 2016.

[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] D. Fried *et al.*, "Incoder: A generative model for code infilling and synthesis," *arXiv preprint arXiv:2204.05999*, 2022.

[8] M. Izadi, R. Gismondi, and G. Gousios, "Codefill: Multi-token code completion by jointly learning from structure and naming sequences," *arXiv preprint arXiv:2202.06689*, 2022.

[9] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "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*, 2020, pp. 1433–1443.

[10] S. Amann, S. Proksch, S. Nadi, and M. Mezini, "A study of visual studio usage in practice," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, IEEE, vol. 1, 2016, pp. 124–134.

[11] C. Casalnuovo, E. T. Barr, S. K. Dash, P. Devanbu, and E. Morgan, "A theory of dual channel constraints," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*, 2020, pp. 25–28.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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].

[17] 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].

[18] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019.

[19] Z. Feng *et al.*, *Codebert: A pre-trained model for programming and natural languages*, 2020. arXiv: 2002.08155 [cs.CL].

[20] 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.

[21] A. Al-Kaswan, T. Ahmed, M. Izadi, A. A. Sawant, P. Devanbu, and A. van Deursen, "Extending source code pre-trained language models to summarise decompiled binaries," in *Proceedings of the 30th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2023.

[22] M. Chen *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[23] T. Brown *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[24] S. Nguyen, T. Nguyen, Y. Li, and S. Wang, "Combining program analysis and statistical language model for code statement completion," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2019, pp. 710–721.

[25] Y. Yang, Y. Jiang, M. Gu, J. Sun, J. Gao, and H. Liu, "A language model for statements of software code," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2017, pp. 682–687.

[26] 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.

[27] 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.

[28] A. Mastropaolo, E. Aghajani, L. Pascarella, and G. Bavota, "An empirical study on code comment completion," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 2021, pp. 159–170.

[29] 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.

[30] 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].

[31] N. Chirkova and S. Troshin, "Empirical study of transformers for source code," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 703–715.

[32] 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.

[33] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, "Improving language understanding by generative pre-training," 2018.

[34] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "CodeSearchNet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.

[35] A. Aghajanyan *et al.*, "Cm3: A causal masked multimodal model of the internet," *arXiv preprint arXiv:2201.07520*, 2022.

[36] M. Izadi, A. Heydarnoori, and G. Gousios, "Topic recommendation for software repositories using multi-label classification algorithms," *Empirical Software Engineering*, vol. 26, no. 5, pp. 1–33, 2021.

[37] M. Izadi, K. Akbari, and A. Heydarnoori, "Predicting the objective and priority of issue reports in software repositories," *Empirical Software Engineering*, vol. 27, no. 2, pp. 1–37, 2022.

[38] H. Robbins and S. Monro, "A stochastic approximation method," *The annals of mathematical statistics*, pp. 400–407, 1951.

[39] M. Izadi and M. N. Ahmadabadi, "On the evaluation of nlp-based models for software engineering," in *2022 IEEE/ACM 1st International Workshop on Natural Language-Based Software Engineering (NLBSE)*, IEEE, 2022, pp. 48–50.

[40] K. Papineni, S. Roukos, T. Ward, and W. Zhu, "Bleu: A method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, ACL, 2002, pp. 311–318. DOI: 10.3115/1073083.1073135.

[41] S. Guo, M. Ficarra, and K. Gibbons, *12 ECMAScript Language: Lexical Grammar*, 2022. [Online]. Available: https://tc39.es/ecma262/multipage/ecmascript-language-lexical-grammar.html#sec-ecmascript-language-lexical-grammar.

[42] 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, 2004, pp. 501–507. [Online]. Available: https://aclanthology.org/C04-1072.

[43] 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.

[44] 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.

[45] 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.

[46] M. Allamanis, "The adverse effects of code duplication in machine learning models of code," in *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 2019, pp. 143–153.

[47] A. LeClair, S. Jiang, and C. McMillan, "A neural model for generating natural language summaries of program subroutines," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, IEEE, 2019, pp. 795–806.

[48] 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.