When Code Completion Fails: a Case Study on Real-World Completions

Vincent J. Hellendoorn
Department of Computer Science
UC Davis
Davis, USA
vhellendoorn@ucdavis.edu

Sebastian Proksch
Department of Informatics
University of Zurich
Zürich, Switzerland
proskch@ifi.uzh.ch

Harald C. Gall
Department of Informatics
University of Zurich
Zürich, Switzerland
gall@ifi.uzh.ch

Alberto Bacchelli
Department of Informatics
University of Zurich
Zürich, Switzerland
bacchelli@ifi.uzh.ch

***Abstract*—Code completion is commonly used by software developers and is integrated into all major IDE's. Good completion tools can not only save time and effort but may also help avoid incorrect API usage. Many proposed completion tools have shown promising results on synthetic benchmarks, but these benchmarks make no claims about the realism of the completions they test. This lack of grounding in real-world data could hinder our scientific understanding of developer needs and of the efficacy of completion models. This paper presents a case study on 15,000 code completions that were applied by 66 real developers, which we study and contrast with artificial completions to inform future research and tools in this area. We find that synthetic benchmarks misrepresent many aspects of real-world completions; tested completion tools were far less accurate on real-world data. Worse, on the few completions that consumed most of the developers' time, prediction accuracy was less than 20% – an effect that is invisible in synthetic benchmarks. Our findings have ramifications for future benchmarks, tool design and real-world efficacy: Benchmarks must account for completions that developers use most, such as intra-project APIs; models should be designed to be amenable to intra-project data; and real-world developer trials are essential to quantifying performance on the least predictable completions, which are both most time-consuming and far more typical than artificial data suggests. We publicly release our preprint [https://doi.org/10.5281/zenodo.2565673] and replication data and materials [https://doi.org/10.5281/zenodo.2562249].**

***Index Terms*—Code Completion, Benchmarks, Language Models**

I. Introduction

Numerous studies have developed tools that can assist programmers in common, but taxing tasks, such as fault localization, program patching, type annotation, and API recommendation [1]–[4]. One of the most clear-cut ways in which tools can reduce the effort of software development for programmers is through code completion. Many tools that suggest plausible completions of the current token, statement, or even function have been proposed [5]–[8] and are part of most Integrated Development Environments (IDEs).

Nearly without exception, code completion tools are evaluated against *synthetic benchmarks*, typically produced by adding "holes" to existing code (e.g., removing an identifier) and tasking the model with predicting the correct expression for that location using the remaining context [4], [6], [9]–[12]. Nevertheless, there is reason to be suspicious of synthetic benchmarks: Proksch *et al.* show that development that happens locally takes place in a very different order and context from what is eventually committed, so that simulations in published code probably do not reflect the developer's working context [12], [13]. Code completion tool design could greatly benefit from real-world insights; Robbes & Lanza demonstrated this by using observations from a developer study to improve their code completion tool [5].

An empirical foundation is needed to establish the characteristics of real-world code completion usage and the objectives that code completion tools should meet to have real-world efficacy. Our case study provides the first step towards such a foundation by analyzing over 15,000 completions that were applied by 66 developers in Visual Studio in a pre-existing dataset. We compare these completions with artificial ones, first in terms of their characteristics and then by replicating them with state-of-the-art code completion models, including structured recommenders, Recurrent Neural Networks, and dynamic n-gram models. We answer the following questions:

RQ1: How do code completions in synthetic benchmarks compare to (applied) real-world completions?

We find that real-world completions are different from typical artificial ones on both trivial factors, such as primarily addressing tokens with longer names and local variables, and more complex ones, such as often focusing on method invocations and field accesses from within the same project.

RQ2: Do these different characteristics affect code completion models?

We show that state-of-the-art models have a much harder time completing real-world queries than those on randomly sampled tokens with similar characteristics. We outline promising factors and outstanding challenges; e.g., models that dynamically integrate local data fare much better than static ones, but intra-project queries remain the hardest category.

RQ3: What characteristics of real completions are overlooked in code completion models?

Inter alia, our analysis shows that most completions are both applied quickly and fairly predictable, apparently geared at reducing typing effort, but the small remainder (typically involving APIs) consume most of the developers' time. Existing models perform very poorly on the second category, suggest-

ing that they are optimized for repeating typical patterns but may not provide novel insights.

Our findings inform concrete recommendations for improving code completion benchmarks, tool design, and real-world efficacy. Benchmarks should first and foremost aim to address completions following their frequency of use, which we characterize, showing for example that intra-project API completions are surprisingly relevant. Code completion tools should similarly aim to be flexible, able to integrate local information and be aware of contexts in which they can be (im)precise. Finally, real-world efficacy is increased precisely by focusing on the least accurate predictions in synthetic data: these are far more common and time-consuming in real-world data. Crucially, our work shows that we need more benchmarks of real code completion engines and developer trials to promote impactful research in this field.

II. Experimental Setup

Synthetic code completion benchmarks are typically created by taking a complete program and removing a random token, such as an identifier or method invocation [4], [9]–[12], [14]. We want to contrast such synthetic completions with real code completions. To this end, we base our case study on a public dataset containing code completions invoked in Visual Studio and simulate artificial completions on the same data. This section details our data collection and modeling setup. Our processed and replicable code completion data, as well as our model implementations, are publicly available [15].

A. Code Completion Data

Several tools have been developed to track developer as they interact with the Integrated Development Environment (IDE), such as Blaze [16], DFlow [17] and FeedBaG++ [18]. The latter is most appropriate for our purposes; FeedBag++ captures event streams in Visual Studio that include invocations of the code completion tool, as well as events for other actions such as opening new projects, using version control and editing a file. Change related events include snapshots of the file in which the edit/completion took place, stored in the form of simplified syntax trees (SST) – an AST with fully qualified type annotations. We produce our *benchmark data* by processing a public dataset of developer interactions that were recorded with this tool [19]. In addition, we use a large dataset of C# repositories as the *training data* [20]. Both datasets have been cleaned and pre-processed as described below to allow replication of a variety of models, including deep learning models, cache-based models, and AST-based models, all of which we include in this work. A full description of the datasets is available in the original papers.

1) Benchmark Data: Our benchmark dataset contains event streams from 86 developers (of which 66 had reproducible completions) with varying levels of experience as they interact with C# projects in VisualStudio. This dataset includes many code completion events and general edit events. We extract all completion events and their accompanying code contexts, which contain a syntax tree of the state of the file in which the developer triggered the completion, with a placeholder at the current completion location. We extract the *prefix*, the characters that were already written towards the completion, and the ultimately selected correct completion. To be compatible with all the models in this study, we only aim to predict the identifier portion (e.g., the method name) of the completion.

Across these developers, we find nearly 200,000 completion events. Of these, ca. 56% were terminated by "filtering" (meaning the developer narrowed down the prefix by typing more characters), 20% were canceled and 24% were ultimately applied. This latter category is most interesting to us, as it presents reproducible completion events. We cannot make strong claims about the reasons for the cancellation events, but it is safe to say that they include at least some cases in which Visual Studio was unable to provide a useful completion (we discuss the ramifications of this in Section IV-A).

Not all completion events contain all the information needed to create reproducible completions. Among the applied cases, ca. 53% did not include locality information about where the completion took place and 32% did not store the selected completion, but rather a generic 'LookupObject' that could not be decoded into the corresponding token after the fact. Accounting for overlap between these categories, we extract 15,245 valid, applied completions belonging to 66 developers for which we can fully replicate the context.

For each completion event, we extract all the surrounding tokens from the stored syntax tree. Ideally, we would also have access to the tokens in the other files in the project, but these are understandably not stored in the dataset. We do have access to any file in which an edit event or completion event takes place, so we approximate the content of the surrounding project by storing the latest snapshots of edited files incrementally, thus creating a restricted view of the project for each developer that improves over time.[1]

In addition to using the recorded completions as a benchmark, we also simulate a synthetic benchmark on the same data; instead of completing the real missing token, we remove another token at random from the file in which the completion takes place and ask each model to infer it. To allow a more fine-grained comparison, we run this simulation many times per file and store the randomly selected token's characteristics (e.g., token length, syntactic type); this allows us to approximate several forms of synthetic benchmarks after the fact, specifically: completing all tokens, completing just identifiers, and completing only API calls. Several other types of synthetic data creation have been proposed (e.g., removing sets of related API calls) and may be studied similarly; we selected options representative of most work on code completion.

2) Training data: We use a public dataset of 340 C# GitHub repositories, comprising ca. 42M lines of code, to train the different completion tools [20]. This dataset is released with the benchmark data and its code is stored in the same representation (simplified syntax trees). We use a built-in method

[1]We have ca. two continuous weeks of observations for each developer, demarcated by "sessions" of IDE use

that comes with the dataset to transform it back into C#, with small adaptations to inline previously nested expressions again and reverse the flattened representations of the simplified syntax trees. This made it easy for us to extract lexical token sequences corresponding to the syntax trees.

The resulting token sequences can be used by lexical token-based models. To make training with deep learners (see Section II-B) feasible in reasonable time, we select 10% of the files in this dataset at random to create a corpus of ∼16M tokens. A static vocabulary is estimated on the training data and all events seen less than 10 times are treated as a generic "unknown" token, to produce a vocabulary of 75,913. Both the corpus and the vocabulary are comparable to prior work using deep learners on source code [14] and we use them to train both the n-gram models and deep learners.

B. Selected Code Completion Models

Our aim is not to compare and rank different completion engines, but we are interested in understanding how the discrepancies between real and artificial completion data impact real code completion models and tools. Thus, we want to select representative and accurate models from diverse backgrounds.

Approaches to perform code completions span a wide spectrum of techniques, but, virtually, all intelligent code completion models attempt to relate the *context* at the site of a required completion to a context that has been observed in some large training corpus. Completions that were applied in similar contexts in the training data can then be recommended in the present context. The difference between approaches often comes down to two aspects: (1) the type of context that is extracted, and (2) the way this context is related to contexts seen during training. One demarcation relates to the way information is extracted: on one side are tools that focus on extracting rich structural features from the surrounding source code, such as types that are being used close by. Approaches on the other side do not explicitly select structural features at all but instead exploit the naturally recurring patterns in source code using text mining techniques from which useful features can naturally emerge [9]. We select performant representatives from both ends of this spectrum, summarized in Table I.

A grey area exists in between these extremes, including approaches that select specific structural features of interest, but use text mining techniques to infer a model over these models; e.g., Bielik *et al.* develop a domain-specific language over allowed contexts and learn a statistical model to predict which context to use at test time [21]. Other models have been proposed, both using natural language models (e.g., [4], [22], [23]) and traditional feature selection (e.g., [5]–[7]); we leave the analysis of such models to future work and provide our dataset to support such efforts.

1) Structural Feature Selection: A traditional approach to code completion, especially API recommendation, is to describe the current editing context through a set of feature types, such as the surrounding method invocations. These feature types are manually designed by leveraging domain-knowledge about source code. The vocabulary of available, concrete features is then established by analyzing source-code repositories. A context is typically described by a vector that encodes the existence of all context features from the vocabulary, e.g., all method invocations from the vocabulary would be mapped to 1 if they exist in the context and 0 otherwise. The same vector is created for a query to the recommender and the proposals are inferred by relating it to vectors seen in the training data. A well-known example of this is the Best Matching Neighbor algorithm by Bruch *et al.* [6], which uses the Euclidean distance of the present context vector to those seen at training time to identify likely completions (i.e., those that occurred in similar contexts).

TABLE I
MODELS INCLUDED IN THIS STUDY WITH CHARACTERISTICS: HOW THEY REPRESENT CODE, WHETHER THEY CAN DYNAMICALLY LEARN NEW PATTERNS, AND WHAT TYPES OF COMPLETIONS THEY ADDRESS.

Model	Format	Dynamic?	Completions
BMN# [6], [7]	AST	No	Members*
RNN [23], [24]	Lexical	Limited	All**
n-gram [14]	Lexical	Yes	All**

(*) We extended the implementation to support member completion
(**) Only unqualified references, no signatures.

An extension of this model was proposed by Proksch *et al.*, who extended the set of structural feature types and introduce Pattern-based Bayesian networks (PBN) as a generalization of the binary contexts (i.e., present/not-present) to Bayesian networks [7]. This improves inference speed and memory consumption while yielding similar completion accuracy and enables the integration of much more contextual information. As the original BMN and PBN approaches were created for Java, several extensions needed to be made for our C# training data to allowing completions on other reference types such as fields or properties. BMN was easier to extend for this purposes, so we created an improved BMN variant, BMN#, which (1) includes all structural features that have been introduced by PBN, (2) widens the completion support from method calls to all possible C# member references, and (3) supports the completion of repeated method calls. Although BMN and PBN are designed to propose fully qualified recommendations, BMN# merges its proposals by identifier to be compatible with our other models, making it an identifier completion engine.

We consider the following features to describe how a particular type is used in the context: the *enclosing class* and *enclosing method* name provide locality information; we store the *type* and *definition* of the object to describe how it was created (e.g., which constructor was called) or where it was defined (if outside the current method, e.g., as a field, method parameter, etc.), and we distinguish between three types of object *usage*: method calls on the object, method calls in which the object is a parameter, and accesses to one of its type-members (i.e., events, fields, methods, and properties).

2) Linguistic Models of Code: Another class of completion models has been inspired by computational linguistics. These models attempt to learn a useful statistical model of a large

body of source code by capturing its most prevalent and informative statistical patterns – typically those that allow them to accurately predict each next token (e.g., word) given the previous tokens. A good model is characterized by low "surprisal" upon seeing each token, which is often measured by its cross-entropy.[2] This entropy tends to be much lower than might be expected from the range of possible word orderings, because natural languages group words together in meaningful and restricted patterns (think of adjectives before nouns, stylistic conventions). Work on the so-called "naturalness" of source code has demonstrated that such models also work well for source code, suggesting that developers treat it at least somewhat like text [9]. This idea has been reinforced by many studies, including fMRI scans [25]. The design of language models to maximize predictability of the next token makes them highly applicable to code completion. In this work, we include two such models.

a) n-gram Model: n-gram models are count-based models that consider only the previous $n-1$ tokens (we use $n=6$) when making a prediction and have often been used as baseline models in natural language processing due to their scalability and surprising effectiveness. In source code, these models were shown to be highly effective at mixing many kinds of dynamically available information (such as tokens in neighboring files) [14]. We thus consider both a "static" variant, which is simply trained on the training data, and a "dynamic" variant, which is also allowed to integrate various sources of information that are present at test time, including: previous completions ("completion cache"), surrounding tokens in the same file ("file cache") and tokens in surrounding files ("project cache"). We study the impact of these sources in Section III-B. The "project cache" is necessarily limited to files that were edited in the past (see Section II-A1), so it is less beneficial than the nested models as in prior work.

b) Deep Learning: these models extract latent representations of text and optimize many parameters to match each next token's latent representation. This potentially allows them to capture patterns with a much longer range than n-gram models, although training these models requires substantial regularization and their memory capacity is somewhat limited in practice. We use a recurrent neural network (RNN) model that is similar to those used in natural language processing and prior work on modeling source code [14], [23]. Our RNN has 300-dimensional embeddings, two recurrent layers with 650-dimensional GRU nodes (with drop-out regularization of 50%) and a dense projection layer onto the vocabulary of tokens. We train this model with an Adam optimizer, implemented in CNTK.[3] The resulting model has ca. 49M parameters, ranking it among the larger ones trained for source code; training required ca. 8 hours across 39 epochs on a GTX 1080 Ti GPU. We also include a variant that learns from prior completions at test time by training on each file after predicting the missing token, although its ability to do so is limited; we refer to this version as "dynamic" and to the standard deep learner as "static". For brevity, we will refer to these models as RNNs.

Even though RNNs often outperform n-gram models in typical natural language settings, we include both, because n-gram models are sometimes a better choice for modeling source code. Specifically, Hellendoorn & Devanbu laid out three software-specific challenges to address in order to properly model source code: handling arbitrarily large and changing vocabularies, integrating events from restricted and changing localities, and incorporating structural information from source code [14]. n-gram models can be taught these things effectively (using structures such as caches [10] or nested architectures [14]), whereas conventional RNN architectures struggle under these conditions.

C. Completion Attributes

All our completions aim to predict an identifier (without qualified type information) from a lexical or AST context (depending on the model) and an optional prefix. When analyzing our results, we consider several attributes of the completion events in our dataset.

Prefix: the characters that are already written towards narrowing down the set of possible completions. This varies widely in length, from empty in many cases (ca. 40%), to the full identifier in others.[4]

Selections: each selection that the developer hovered over in the presented list of completions is recorded, ranging from one to dozens. The final selection is the applied completion.

Duration: the duration characteristics of both the entire completion and of each selection considered are stored. Most completions take less than a second, but some completions are considered for long periods (see Section III-C).

Completion Type: each completion in our benchmark data concerns an identifier. We distinguish between them based on the identifier's type (project-internal, C# core library, or third-party) and its syntactic category (class-name, method-name, field, parameter or local variable). Primitive types are grouped in their own category.

D. Evaluation Metrics

We evaluate the accuracy of our completion models according to several standard metrics. We mainly consider Mean Reciprocal Ranking (MRR), a summary metric for top-K accuracies that averages the inverse of the ranks of each completion. MRRs range from 0 to 1, where values closer to 1 imply that the correct completion was often near the top of the suggestion list.

Each model assigns a probability to all its completions in a context, favoring some events more than others. If a model's top suggestion is offered with a high probability, this can be taken as an indication of high "confidence' in this

[2]This reflects the amount of information needed to encode a token given its context and the trained model

[3]https://cntk.ai

[4]The latter case may still be useful to a developer; e.g., API completions often include a template with some arguments set to default values.

suggestion. Vice versa, if even its top recommendation has a low probability, the model is likely uncertain. We can simulate a completion tool with a developer-tunable confidence threshold: if the model's top probability exceeds this threshold, its suggestion would be presented to the developer, otherwise it is ignored. This allows us to compute precision and recall values for various thresholds.

III. Analysis

Throughout this section, we identify nine findings on the differences between synthetic data and real-world completions, in terms of intrinsic data characteristics (RQ1), impact on code completion performance (RQ2), and lessons for real-world efficacy of code completion tools (RQ3).

A. RQ1: How do code completions in synthetic benchmarks compare to (applied) real-world completions?

In this section, we simulate different forms of artificial completions on the same data as our real completions by removing different types of tokens at random to complete. Our goal here is to characterize the completions that this would lead to. For increased stability, we ran 10 such simulations and averaged the results. We also include some n-gram model results to give an early indication of completion accuracy in this artificial setting; we study the contrast with real-world accuracy in greater detail in Section III-B and Section III-C.

1) Token Types: We start with synthetic datasets that consider every token in the file as targets for completion. Table III characterizes the tokens at issue in these artificial completions; more than two-thirds of artificial completions did not refer to identifiers in the source code. Instead, the majority (57%) concerned punctuation-like tokens (e.g., operators, braces), followed by identifiers (30.4%), keywords and numerals (10.8% and 1.8% respectively). In the real-world data, we found only completions pertaining to identifiers.[5]. Identifier completions are also harder than other tokens, as is evident from the n-gram MRRs in the final two columns, especially for a static model. Thus, we already see a strong skew between artificially synthesized benchmarks and real code completions. Fortunately, most studies involving code completion have recognized this and focused exclusively on predicting identifiers (or even specifically API usage), so we will focus on this type of synthetic data next.

> *Finding #1:* Most tokens are much easier to complete than identifiers, but these completions are not used.

2) Identifier Types: If we instead simulate just identifier completions, we can start by categorizing these along two axes: 1) the syntactic type of the identifier that is to be completed (e.g., method invocation, parameter, local variable), and 2) the origin of the declared type of this identifier (i.e., primitive type, or reference to an object from the same project, the C# core library, or a third-party library). Figure 1 shows the distribution characteristics of the syntactic categories, for both artificial (left) and real (right) completions. Real code completions overwhelmingly favor method invocations, followed by field accesses, both related to API usage. Core and external library method invocations are particularly underrepresented in artificial data, occurring twice as often in real life. Artificial completions show a more diverse pattern, comparatively involving local variables and type (class) names much more often, because those identifiers are commonly used.

[5]Although Visual Studio does not offer to complete punctuation and numerals, it is highly unlikely that such a feature would benefit developers, as most of these tokens are very short (averaging ca. one character each) compared to identifiers (averaging nearly 10 characters)

TABLE II
Characteristics of artificial completions in terms of their relative proportion, the average number of characters of the token to complete, and the MRR according to a Static and Dynamic n-gram model.

			n-gram MRR	
Type	**Proportion**	**#Chars**	Static	Dynamic
Punctuation	57.1%	1.05	61.0%	89.6%
Identifier	32.1%	9.61	15.8%	67.4%
Keyword	10.8%	4.27	45.4%	72.9%
Numerals	0.5%	1.02	48.5%	82.1%

The second characteristic of interest is the origin of the declared type of the identifier to be completed. Primitive type related identifiers (e.g., `bool`, `string`) proved the main differentiator here, spanning 12% of completions in the artificial dataset, but a negligible portion in real-world data. Identifiers with these types are also the shortest of any category (think of loop variables like '`i`'), suggesting that developers both choose shorter names for simpler types and are unlikely to use code completion on these names.

Among object types, however, we observe a high similarity between the artificial and real completions: both concern project-internal types ca. 50% of the time and core library/third-party types ca. 25% each. In other words, compared to the types they use in a file, developers do not disproportionally ask for completion of publicly declared vs. project-internal types. It is especially surprising that intra-project types account for more than half of all completions; in fact, the most prominent sub-category in both datasets is method invocations within the same project, containing over a quarter of all real-world completions. API recommendation tools that only recommend public APIs (as is common, including BMN#) would thus fail to address over half of the real-world queries. Overall, no assumption commonly made when synthesizing benchmarks (e.g., include all tokens, or only identifiers, or only public APIs) approximates this data.

> *Finding #2:* Typical simulations do not match real completions, overlooking e.g. intra-project API completions.

3) Typing Effort: As an example of a more simple factor, if the purpose is to reduce key-strokes, then the number of characters in an identifier may play a role in whether programmers choose to use the completion engine. Thus, punctuation and numeric tokens are unlikely to be requested,

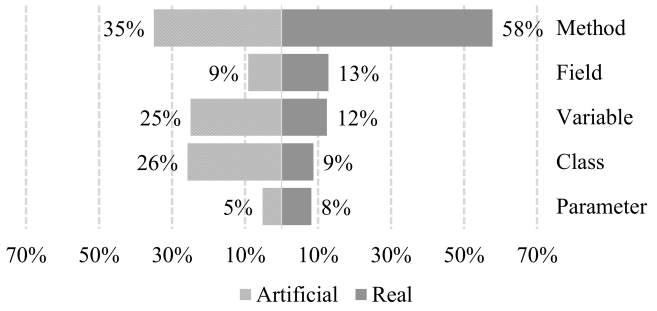

Fig. 1. Skew of completion type prevalence between artificial and real data.

since just typing them would be equally fast (we conjectured a similar motivation for identifiers with primitive types). If reducing key-strokes is a motivation, one might expect identifiers involved in real-world completions to be longer than ones in artificial completions. We do indeed observe this effect in our dataset: artificial completions are significantly shorter (avg. 9.06 characters) than real ones (9.87 characters), with a sizable effect (t-test: $p \ll 2\text{e}{-16}$, Cohen's D.: 1.33).[6]

In the absence of controlled trials, we cannot draw conclusions about the relation between this result and effort. It is possible that developers use the completion engine more actively if they anticipate having to type longer identifiers (thus purely saving effort), but the presence of a completion engine may also cause them to use longer identifier names (which can be more informative) for identifiers they intend to use frequently, because they know that the completion engine will help out. A mixture of these effects is probably at play. For one, the correlation between prefix length and identifier length in the real completion data is very weak ($R^2 \approx 0.1$), so developers do not write substantially longer prefixes for longer identifiers, suggesting that they may at least anticipate the potential savings in typing effort. Furthermore, both the longest identifiers and the biggest character savings (identifier length minus prefix length) were found in project-internal variables, which are all within the developer's control; the smallest were among core library references.

> *Finding #3:* Real-world completions are also influenced by simple motivations such as typing effort.

B. RQ2: Do these different characteristics affect code completion models?

We next study the performance of our various models on the real code completions in our dataset. We analyze and contrast their performance with that on artificial completions in greater detail in Section III-C, and focus first on characterizing the performance of these models on real-world completions.

1) n-gram models: The baseline n-gram model performs at only 14.9% MRR. This performance goes up substantially when incorporating local information in the dynamic model.

[6]Both statistics are on the log-length, as these were approximately normally distributed.

TABLE III
PERFORMANCE OF OUR MODELS ON REAL COMPLETIONS. BMN# PERFORMANCE IS MEASURED ACROSS THE CASES IT COULD ADDRESS.

			Top-k Accuracy		
Model		**MRR**	1	5	10
n-gram	– Static	14.9%	11.4%	19.0%	21.8%
	– Dynamic	43.5%	32.7%	56.5%	63.6%
RNN	– Static	21.5%	17.5%	26.0%	28.8%
	– Dynamic	22.2%	18.2%	26.7%	29.6%
BMN#*		40.7%	30.7%	54.6%	58.8%

(*) Results on ca. 14% cases that BMN# is designed to complete

At the same time, although the static n-gram here performs similar compared to the artificial benchmark (where it reached 15.8% MRR on identifiers, see Table III), the dynamic model performs quite a bit worse (43% vs. 67%). This means that many completions could not be adequately answered by the n-gram model. Even then, it actually comes close to Visual Studio's own ranking performance despite the dataset's strong bias in its favor.[7] This is not surprising: cache-based n-grams routinely outperform IDE's inbuilt code completion [8], [9].

The dynamic n-gram model effectively combines many sources of "local" information. The impact of these various sources can shed some light on where developers' needed recommendations come from. Figure 2 shows an ablation analysis of the n-gram model's performance when focusing on specific types of locality. All components can greatly help performance by themselves, which is similar to Robbes & Lanza's findings [5]. The file-cache proves to be the strongest influence here, more than doubling the performance of the static model. Storing previous completions proves valuable as well, followed by including tokens from the surrounding files.[8] These components appear to use overlapping information (e.g., common API usage in a project), as the overall model improves only 4.3% over the file-cache-only model.

2) Deep Learners: On a positive note, the static RNN outperforms its n-gram equivalent by a substantial margin with the same amount of training data, which reflects its superior capacity. At the same time, its dynamic updating ability – limited to integrating patterns from new sequences, but not new words – barely helps the model, as nearly half of completions are tokens not seen at training time. Deep learning models that can learn new words (such as proper nouns) have been proposed for natural languages [26]–[28] and these results make it all the more pressing that such innovations are translated to the source code domain.

> *Finding #4:* Local context plays a large role in code completion accuracy, but still leaves many queries unanswered.

3) BMN#: This tool is different from the above models in that it uses structured information (specifically, types in context). It is exclusively geared towards completing method

[7]which is why it is not included; see Section IV-A

[8]Likely due to an imperfect view of project data, see Section II-A.

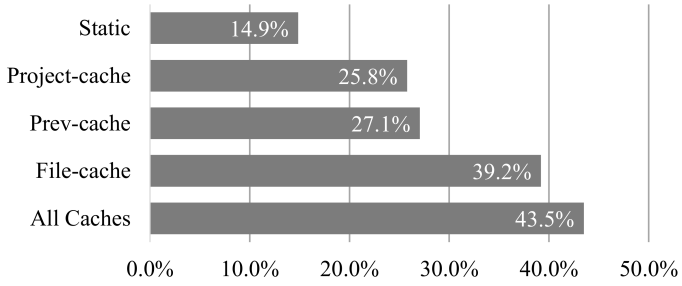

Fig. 2. Characterization of n-gram model's performance with increasing amounts of "local" information. "File" refers to a cache on other tokens in the file, "Prev" refers to a cache on previous completions used by this developer and "Project" refers to approximation of the project's code (Section II-A).

invocations on a known set of public APIs and, although our implementation of BMN# was modified to address as many of C# language features as possible, it only provides completions for 2,126 such invocations (out of a total 8,752). More specifically, it completes roughly a third of third-party API calls (with MRR of 35.1%), more than half of core library calls (with MRR of 42.1%) and ca. 2.6% of project-internal calls (all incorrect). Core libraries are most represented and intra-project calls least, despite being most common in real-world data, because the tool needs to know type signatures to be able to complete them and is not able to learn new type signatures (e.g., from within a project) at test time.

If we compare the n-gram and deep learner's performance on the subset of completions that BMN# does address, it outperforms both models slightly (40.7% vs. 38.0% and 37.0% MRR respectively). The deep learner is better at core method invocations but loses on third-party library calls; the n-gram model naturally outperforms it on internal API calls but loses out on the other categories.[9] Considering that BMN# does not have a dynamic variant, it performs surprisingly well, even eking out a win over the RNNs on third-party library references. Its success is largely due to its use of structural information, especially type information: being able to restrict results to type-compatible completions can be a substantial benefit. Adopting caching strategies similar to the n-gram model could help it overcome its limitation on project-internal method calls (although learning new vocabulary may be non-trivial) and boost its predictive power as a whole.

> *Finding #5:* Employing structural information can benefit accuracy, but risks being inflexible w.r.t. local patterns.

4) Confidence: We discussed the notion of a confidence threshold (based on the model's top recommendation's probability) in Section II-D. The n-gram model proves quite amenable to that option: setting the threshold to $\geq 0.1\%$ already yields a precision/recall of 67%/44%, boosting performance by more than twenty percentage points while providing completions for slightly less than half the events. At a higher threshold of 10%, these numbers become 94%/14% and at 50% onwards the model achieves perfect precision, albeit on only 3.2% of the remaining data. This suggests a kind of self-awareness of the n-gram model; when it expresses confidence, it is highly accurate. Unfortunately, it also loses a great amount of recall in the process, which makes such a model not quite applicable to real-world completions, though it may be amenable to combination with other models [14] (although attempting to combine it with BMN# was not successful).

We also investigated setting a confidence threshold for the deep learner, but this proved largely ineffective. At best, we were able to improve precision to ca. 23% at the cost of substantial loss in recall. This is atypical for deep learners, which often express a high degree of confidence in familiar contexts and have done so in code as well [14]. However, in our experiments, they seem to do so mainly for trivial tokens (e.g., punctuation and simple identifiers), which span the vast majority of tokens, but not those involved in completions. This behavior is not useful to developers in our dataset and suggests that deep learning based completion engines may need to be optimized specifically for code completion on identifiers, in the ways that they appear in real-world completions.

> *Finding #6:* Models struggle to attain high precision in familiar contexts without sacrificing substantial recall.

C. RQ3: What characteristics of real completions are overlooked in code completion models?

In this analysis, we combine our insights from RQ1 and RQ2 to analyze which discrepancies between artificial and real-world data meaningfully impact our code completion models. Unless otherwise noted, we focus mainly on the dynamic n-gram model's performance when referring to modeling accuracy here, since it emerged as the strongest model from our previous analysis.

1) Revisiting Completion Types: We revisit the different completion types after comparing the characteristics of artificial and real completions in RQ1.

a) Method invocations: these were found to be overwhelmingly more common in real-world completions. Although we do not know of work that has quantified this before, it is noteworthy that this type of completion has attracted particular interest in code completion research (e.g., [4], [6], [11]), suggesting that this pattern has not gone unnoticed in SE research. Method invocations are also the hardest category to our strongest model, followed by field references (which were harder for the RNNs, but also quite rare). They also come with the shortest average prefix (empty more than half the time, compared to just 6% of the time for local variables), suggesting that developers are often unsure what they are looking for. There is an especially large rift here with artificial data, as method completions were much easier in that dataset (more than twice as easy to our best model).

The hardest category of method invocations, however, *were those declared in the same project* (31% MRR), not third-party invocations (34% MRR). Perhaps the ongoing development

[9] A hybrid was attempted in which the most confident model in every context is allowed to predict, but this did not yield a performance gain, suggesting that the models are similarly confident in similar contexts.

nature of the project causes these invocations to be less repetitive and obvious. This is also the most frequently used type of completion, so it is pressing that completion engines take this task seriously; API recommendation tools often focus exclusively on popular public APIs (such as core libraries). It is worth noting that Visual Studio actually had more difficulty ranking external and core library method invocations. Since it has access to type resolution, it may be easier for it to match method invocations to the surrounding context. BMN# also has access to type information but is unable to learn new methods at test time, so it does not produce any useful completions on intra-project data. A hybrid approach between type-safe models and linguistic models may provide better performance, though a naïve combination that we attempted did not yield meaningful improvements.

The second aspect to completion types is where the identifier of interest was declared: inside the project, in the C# core library or in a third-party library. The difference in performance here is substantial. The static n-gram model reaches only 4.5% MRR on project-internal identifiers; the static RNN reaches 7.3%. Adding the dynamic component boosts the n-gram's performance, but barely the deep learner's (40.5% vs. 8.3%), because the current architecture it is unable to learn new identifiers. In contrast, these models scored 53% and 44% respectively on core library types (and 41% and 27% on third-party types).

> *Finding #7:* Artificial data can substantially mischaracterize prevalence and difficulty, especially of hard completions.

2) Deep Learning for Code Completion: The static RNN outperforms the static n-gram model, which is in line with previous work studying the relative performance of these two types of models [14]. The RNN especially wins out on variable, parameter and field recommendations, likely helped by its superior memory.[10] At the same time, the dynamic n-gram model is substantially better, as in the aforementioned study; in fact, the discrepancy is even larger on real completions. However, there is hope for the dynamic setting: the RNNs outperformed the dynamic n-gram (and BMN#) on all core API recommendation (methods, class names and fields). This makes sense: these are arguably the most stable part of a language's vocabulary between projects.

These observations reinforce the challenges laid out in prior work on deep neural networks for source code, but also offer hope for a future place for deep learners in code completion: if they can successfully learn to integrate new vocabulary, they may be able to outperform more traditional models with their superior memory capacity. Although such a mechanism may be based on recent advances in NLP [26]–[28], it will likely require substantial innovations as well, because source code's vocabulary (innovation) characteristics are very different from natural languages.

[10]An n-gram model can only see n - 1 tokens in the past (here $n = 6$)

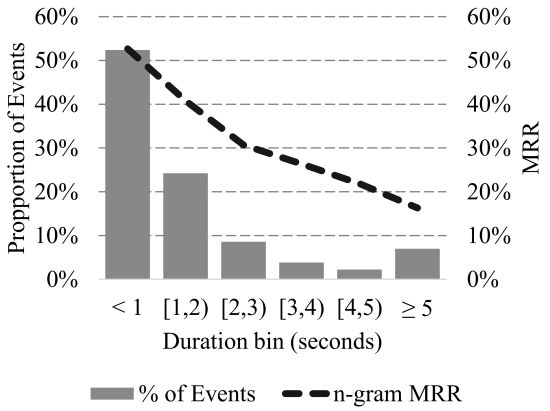

Fig. 3. The relation between the duration (binned by seconds) of a completion and (bars) the prevalence of this duration, and (dashed line) the average dynamic n-gram MRR for events in each duration range.

> *Finding #8:* Deep Learners can learn superior models from training data, but are severely limited on new data.

3) The Role of Duration: Finally, we observe a wide range of *durations* (the time taken before applying the completion) among the completions in our dataset. The majority take less than one second ($median = 927ms$), but their distribution is skewed ($mean = 2,096ms$) so that some take far longer. In fact, while 90% of completions take less than 3.76 seconds, the remaining 10% account for more than half of the time spent using the completion engine! This gives rise to an important observation: code completion accuracy is almost always measured by the fraction of cases that were correctly completed, not the actual savings in time or effort that the completion could bring. However, our data shows that there are many fast (and presumably easy) completions and few very slow, possibly much less obvious ones. These 10% slow completions may be woefully undervalued.

Looking deeper, local variables and parameters are the fastest completions (averaging ca. one second each), whereas method invocations and field accesses take longest to complete on average ($2.6s$ and $3.1s$ respectively); intra-project and third-party method completions alone accounted for 60% of time spent using code completion (while accounting for 46% of completions). Crucially, this is not just due to Visual Studio's ranking: although there is a moderate (negative) correlation between its MRR and completion duration (Pearson's: -0.59 across completion types),[11] several completion types defy this relation. However, our n-gram model's MRR shows an almost perfect inverse correlation with duration (Pearson's: -0.92). As Figure 3 shows, performance on slower completions drops by more than half after three seconds (the same holds for the RNNs), which means that most of the developer's time would

[11]Xianhao & Francisco also observed on this dataset that longer completion lists tend to increase selection time [29]; our analysis quantifies this effect.

TABLE IV
EXPLANATORY POWER, AS % OF DEVIANCE EXPLAINED, OF AN INCREMENTAL MODEL THAT FITS THE (LOG) DURATION OF CODE COMPLETIONS. WE INCLUDE INDIVIDUAL FACTORS (ALL SIGNIFICANT), GROUPS OF THESE FACTORS, AND THE OVERALL MODEL (TWO VARIANTS)

Factor(s)	**Duration Explained**
Process Factors	*13.4%*
Developer	12.8%
#Options	0.5%
Identifier Factors	*10.9%*
Completion Type	5.2%
Prefix Length	1.7%
Completion Length	5.2%
Context Factors	*9.7%*
Log(Count in corpus)	2.8%
In Project	0.8%
Previously Used	1.2%
In File	5.9%
All Factors	26.3%
Developer as random effect	30.5%

be spent with just 19% MRR (11% for RNNs and 24% for BMN# on their limited subset).

To better understand the factors that affect the duration of a completion, we built a linear regression model that incorporates all readily identifiable factors about the completion, separated into Process factors, Identifier factors and Context factors, in Table IV.[12] The final rows show the explanatory power of the overall model, both for a simple linear model and for a variant that treats the developer as a random effect (a mixed effect model). Many factors influence duration in our model. Substantial differences in completion speed between developers are the strongest individual factor, followed by both completion type and length; longer identifiers, method invocations, and field accesses imply longer durations, whereas the other types produce shorter durations. Finally, having used the identifier in the same file (and to a lesser extent, in surrounding files or in a previous completion) are correlated with reduced duration, which is reminiscent of our n-gram ablation analysis.

> *Finding #9:* Rare, difficult completions are vital to real-world efficacy – much more so than synthetic data suggests.

IV. DISCUSSION

We studied the limitations of using synthetic benchmarks of code completion and reported three findings regarding the characteristics of such benchmarks, three findings pertaining to challenges that models face on real completions, and three findings concerning factors that are overlooked in completion model design because of synthetic benchmarks. We discuss the implications of these findings for future code completion models and benchmarks.

[12]Specifically, we fit log-duration, as a Q-Q plot showed that durations are log-normally distributed (albeit with slightly fat tails) in our dataset.

A. Improving Benchmarks

Findings #1 – #3 concerned differences in characteristics between real-world and artificial completions. Although one might hope that the real-world data follows clear patterns that are easy to simulate (if not match the synthetic data entirely), this is decidedly not the case. Real completions were not clearly focused on any one group of tokens, neither in terms of frequency, nor in terms of effort. Even relatively simple characteristics, such as longer tokens occurring more frequently, do not hold universally and are guidelines at best. Neither is it realistic to synthesize completion benchmarks by sampling based on the real-world token distributions that we reported in our paper, as we found (among other issues) that method calls were relatively easy to complete in artificial data, but far harder in real-world data. The impact of duration further complicates this process. Evidently, real-world benchmarks of code completions are necessary.

Although we release the processed benchmark used in our work for further research, [15] this benchmark data is primarily suited for a case study such as ours; it has limited use to quantifying the absolute performance of our tested models, and has several more limitations (partially because it was not specifically collected to study code completions). For one, it is restricted to the completions that VS was able to provide; we did not have answer-data for completions that developers canceled. Neither does this data contain important human data, such as the reason for these cancellations, nor the motivations and expectations for any other completion.

There is, thus, a dire need for more real-world benchmarks to evaluate code completion engines, underscored by our findings and by the newfound need to understand slow and unsuccessful completions. Like our data, such benchmarks should aim to collect reproducible completions across several representation types and rich meta-information regarding the completions. We also recommend collecting qualitative information about completions related to the developer's objectives and experience. This should include insights into the causes for failed completions; e.g., was the true completion not listed, or the query incorrect; did the developer solve it by restructuring their code, or switch to searching online? Such information is necessary to better understand the least successful completions, which our analysis identified were most crucial to completion tool efficacy (both for improving accuracy and for reducing effort). Finally, some models for code completion can provide completions well beyond those seen in this dataset, such as entire syntactic constructs, or comments. Developer experiments are necessary to analyze the impact and efficacy of such innovations, and the resulting data can be used as benchmarks for improvements on such tools in turn.

B. Improving Tools

Our next set of findings (#4 – #6) captured factors that allow models to succeed on real-world completions, as well as their remaining struggle to achieve good prediction accuracy. Based on our findings, we make several recommendations here.

Future code completion engines should include information from the developer's local working context, since many completions concern this context and these completions are remarkably difficult to provide. Even models with substantial ability to integrate local information were only able to answer ca. half of these queries successfully and other models failed to learn from local information (almost) entirely. This should especially focus on intra-project API completions, which is ostensibly a harder problem than core/third-party API completions since much more usage data exists for the latter set.

This need for flexibility also ties into our findings on using confidence thresholds to improve precision or create hybrid models. Only the n-gram model was somewhat amenable to this strategy in our analysis, but generally, the models failed to show awareness of their own precision. Having such awareness could be very useful both for a better developer experience and to create tools that can employ several different types of models depending on the context. Such hybrid tools could also harvest the power of structured models, which in our analysis could outperform statistical models, but were only applicable on a small subset of completions; flexibility thus need not come in the form of a single model. Deep learners could begin to excel as well, if taught these properties successfully.

C. Improving Efficacy

Finally, findings #7 – #9 demonstrated that real-world performance of code completion models could be substantially distorted *relative* to artificial data, because the latter's characteristics mask areas of importance.

For one, the tokens that actually interest developers are much harder to complete than even surrounding tokens in the same file.[13] Although developers do request some relatively easy completions (apparently mainly to preserve typing effort), a large proportion of their requests is relatively hard. We stress that this *cannot* be seen in simulations: even accounting for similar syntactic categories and types, real completion queries were much harder to our models than artificial ones from the same context. Thus, code completion engines should focus precisely on the most difficult cases as these are likely to be underrepresented in artificial data.

Most importantly, artificial simulations do not account for an integral human factor: effort associated with completions. Duration of real-world completions, which cannot be simulated at all, has a large impact on the effort associated precisely (and reasonably) with the hardest completions. Whereas typing effort is fairly well addressed by the models in our study, at least in terms of short-duration completions, API discovery effort is much less so. Thus, work on code recommenders should include developers in their evaluation to increase their real-world impact.

D. Limitations

Our case study exclusively studies completions that Visual Studio was able to resolve with an applied completion. This both means that we cannot study completions that Visual Studio did not offer, nor can we quantify our tools' performance on any canceled completions. The data thus provides a limited window into the minds of developers, showing us only events with a positive outcome (although duration adds helpful, and indeed actionable, insights into the actual process). However, this dataset does allow us to study the contrast with artificial completions, as these are simulated on the same data, and the pitfalls and challenges for these models that emerge by contrast. Hence, this has been the focus of our study. We also note that all our models correlated well with Visual Studio in terms of ranking performance (MRR). As such, if canceled events were indeed relatively hard for VS, it is unlikely that their absence would bias our results by these being much easier for our models.

Within these constraints, we were able to establish reasonable *relative* differences between artificial and real-world completions. This serves the goal of our case study, and we were able to report a range of actionable findings based on this data. However, these constraints do limit the usefulness of this dataset: the performance of the tested models here should not be taken as their absolute performance on real-world data; we cannot measure that due to the stated limitations. This further underscores the need for more benchmarks to support code completion research and tool design.

[13] In fact, we found in a separate analysis that even identifiers adjacent to the true completion token were easier to predict on average.

V. Summary

We conducted a case study comparing code completions used by real developers with those in artificially synthesized benchmarks. We simulated artificial completions on a dataset of over 15,000 real-world completions invoked by 66 developers. Based on our analysis, real-world code completions have non-trivial distributions that do not match any typical synthetic benchmark. These differences also make them much harder to complete for state-of-the-art models, reducing their accuracy by half in some cases. Studying the causes for this allowed us to identify factors that are often overlooked in completion tool design, such as the importance of integrating local information, such as intra-project APIs. Furthermore, real completions have hidden characteristics that impact the actual efficacy of code completion tools in real-world use: they disproportionally concern less predictable tokens so that most of the developer's time is spent on completions on which our tools were less than 20% accurate. We formulate several recommendations for future code completion benchmarks, tools, and research, mainly that new code completion tools should be evaluated on benchmarks of real-world data and/or developer trials (which can become benchmarks) to ensure their real-world efficacy.

VI. Acknowledgment

This work has received funding from CHOOSE, under the Student Mobility scheme. Alberto Bacchelli gratefully acknowledges the support of the Swiss National Science Foundation, Project No. PP00P2_170529. Vincent Hellendoorn was partially supported by the National Science Foundation, award number 1414172, and by a Microsoft PhD Fellowship.

References

[1] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu, "On the "naturalness" of buggy code," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 428–439. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884848

[2] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *Ieee transactions on software engineering*, vol. 38, no. 1, pp. 54–72, 2012.

[3] Z. Xu, X. Zhang, L. Chen, K. Pei, and B. Xu, "Python probabilistic type inference with natural language support," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 607–618.

[4] A. T. Nguyen and T. N. Nguyen, "Graph-based statistical language model for code," in *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, vol. 1. IEEE, 2015, pp. 858–868.

[5] R. Robbes and M. Lanza, "How program history can improve code completion," in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2008, pp. 317–326.

[6] M. Bruch, M. Monperrus, and M. Mezini, "Learning from examples to improve code completion systems," in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 2009, pp. 213–222.

[7] S. Proksch, J. Lerch, and M. Mezini, "Intelligent code completion with bayesian networks," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 25, no. 1, p. 3, 2015.

[8] C. Franks, Z. Tu, P. Devanbu, and V. Hellendoorn, "Cacheca: A cache language model based code suggestion tool," in *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 705–708. [Online]. Available: http://dl.acm.org/citation.cfm?id=2819009.2819143

[9] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 2012, pp. 837–847.

[10] Z. Tu, Z. Su, and P. Devanbu, "On the localness of software," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 269–280. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635875

[11] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *Acm Sigplan Notices*, vol. 49, no. 6. ACM, 2014, pp. 419–428.

[12] S. Proksch, S. Amann, S. Nadi, and M. Mezini, "Evaluating the evaluations of code recommender systems: A reality check," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 111–121.

[13] S. Proksch, S. Amann, and M. Mezini, "Towards standardized evaluation of developer-assistance tools," in *Proceedings of the 4th International Workshop on Recommendation Systems for Software Engineering*. ACM, 2014, pp. 14–18.

[19] S. Proksch, S. Amann, and S. Nadi, "Enriched event streams: A general dataset for empirical studies on in-ide activities of software developers," in *Proceedings of the 15th Working Conference on Mining Software Repositories*, 2018.

[14] V. J. Hellendoorn and P. Devanbu, "Are deep neural networks the best choice for modeling source code?" in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 763–773.

[15] "When Code Completion Fails: a Case Study on Real-World Completions - Data and Material." https://doi.org/10.5281/zenodo.2562249, 2019.

[16] W. Snipes, A. R. Nair, and E. Murphy-Hill, "Experiences Gamifying Developer Adoption of Practices and Tools," in *International Conference on Software Engineering*, 2014.

[17] R. Minelli, A. Mocci, R. Robbes, and M. Lanza, "Taming the IDE with Fine-grained Interaction Data," in *International Conference on Program Comprehension*, 2016.

[18] S. Proksch, S. Nadi, S. Amann, and M. Mezini, "Enriching In-IDE Process Information with Fine-grained Source Code History," in *International Conference on Software Analysis, Evolution, and Reengineering*, 2017.

[20] S. Proksch, S. Amann, S. Nadi, and M. Mezini, "A dataset of simplified syntax trees for c#," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: ACM, 2016, pp. 476–479. [Online]. Available: http://doi.acm.org/10.1145/2901739.2903507

[21] P. Bielik, V. Raychev, and M. Vechev, "Phog: probabilistic model for code," in *International Conference on Machine Learning*, 2016, pp. 2933–2942.

[22] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "A statistical semantic language model for source code," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 532–542.

[23] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk, "Toward deep learning software repositories," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, ser. MSR '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 334–345. [Online]. Available: http://dl.acm.org/citation.cfm?id=2820518.2820559

[24] F. J. Pineda, "Generalization of back-propagation to recurrent neural networks," *Physical review letters*, vol. 59, no. 19, p. 2229, 1987.

[25] J. Siegmund, N. Peitek, C. Parnin, S. Apel, J. Hofmeister, C. Kästner, A. Begel, A. Bethmann, and A. Brechmann, "Measuring neural efficiency of program comprehension," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: ACM, 2017, pp. 140–150. [Online]. Available: http://doi.acm.org/10.1145/3106237.3106268

[26] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," *arXiv preprint arXiv:1508.07909*, 2015.

[27] M.-T. Luong, I. Sutskever, Q. V. Le, O. Vinyals, and W. Zaremba, "Addressing the rare word problem in neural machine translation," *arXiv preprint arXiv:1410.8206*, 2014.

[28] M.-T. Luong and C. D. Manning, "Achieving open vocabulary neural machine translation with hybrid word-character models," *arXiv preprint arXiv:1604.00788*, 2016.

[29] X. Jin and F. Servant, "The hidden cost of code completion: Understanding the impact of the recommendation-list length on its efficiency," in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR '18. New York, NY, USA: ACM, 2018, pp. 70–73. [Online]. Available: http://doi.acm.org/10.1145/3196398.3196474