Pragmatic Code Autocomplete

Gabriel Poesia, Noah Goodman
Department of Computer Science, Stanford University, Stanford, CA 94305
{poesia,ndgoodman}@stanford.edu

Abstract

Human language is ambiguous, with intended meanings recovered via pragmatic reasoning in context. Such reliance on context is essential for the efficiency of human communication. Programming languages, in stark contrast, are defined by unambiguous grammars. In this work, we aim to make programming languages more concise by allowing programmers to utilize a controlled level of ambiguity. Specifically, we allow single-character abbreviations for common keywords and identifiers. Our system first proposes a set of strings that can be abbreviated by the user. Using only 100 abbreviations, we observe that a corpus of Python code can be compressed by 15%, a number that can be improved even further by specializing the abbreviations to a particular code base. We then use a contextualized sequence-to-sequence model to rank potential expansions of inputs that include abbreviations. In an offline reconstruction task our model achieves accuracies ranging from 93% to 99%, depending on the programming language and user settings. The model is small enough to run on a commodity CPU in real-time. We evaluate the usability of our system in a user study, integrating it in Microsoft VSCode, a popular code text editor. We observe that our system performs well and is complementary to traditional autocomplete features.

1 Introduction

Human languages have evolved over time to be highly effective and efficient. Though a reliable means of communication, our languages are strikingly ambiguous at every linguistic level. Morphemes, words and even entire sentences might have very different meanings in different contexts. Yet this ambiguity does not hinder the communicative efficacy of natural languages. This can be attributed to *pragmatic reasoning* used by listeners to infer the meaning of speakers' utterances: context usually provides the missing information listeners need to disambiguate meaning (Piantadosi, Tily, and Gibson 2012).

Programming languages differ radically: To make compilers and interpreters practical, a programming language's syntax is always unambiguous, defining a single way to parse each program. While the absence of ambiguity facilitates the creation of development tools, it also forces programs to be more verbose. This drawback becomes clear when we look at programming languages as a communication channel between programmers and computers. Using tools from information theory, researchers have provided several arguments for why any efficient communication system must allow for ambiguity when context is informative about meaning (Piantadosi, Tily, and Gibson 2012; Fortuny and Corominas-Murtra 2013; Solé and Seoane 2015).

```
import sys                                import sys
class ExperimentDriver:                   class ExperimentDriver:
    d _(s):  ---------------------------->    def __init__(self):

...                                       ...

# CLI                                     # CLI
i _ == '_':  ---------------------------> if __name__ == '__main__':
```

Figure 1: Illustration of the system we propose, utilizing the Python programming language. Users can type their code using one-character abbreviations, as on the left. Given a whole abbreviated line, our model expands it using context with near-perfect accuracy.

While programming languages eliminate ambiguity by design, it is still true that context carries significant predictive information about source code. This fact is extensively exploited by autocomplete systems, which are pervasive in modern programming environments. Traditionally, these systems predict completions of the code the user intends to type. Because there is often high uncertainty about which of the plausible completions was intended, autocomplete functionality usually offers users a list of their top predictions. However, HCI researchers have observed that the cost of picking between predictions can eliminate efficiency gains users would gain from a predictive typing system (Quinn and Zhai 2016; Palin et al. 2019). This issue seems to pose a dilemma. One the one hand, the redundancy of programming languages motivates predictive features to reduce users' typing effort. But on the other hand, asking users to decide between multiple possible predictions brings its own cognitive and efficiency cost.

In this work, we provide a way around this apparent conflict. To mitigate the verbosity of programming languages, we propose a strategy for using ambiguity in a controlled fashion to significantly reduce users' typing effort. First, we observe that keywords and identifiers constitute 75% of the

Copyright © 2021, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

characters in Python code in a sample of 1000 repositories from Github[1]. Furthermore, the distribution of uses of such keywords and identifiers is highly skewed. We find (Section 4) a set of 100 keywords such that 75% of all lines of Python code from our corpus contain at least one of them. Moreover, simply abbreviating these keywords to their first character compresses the entire corpus by 15%. Very much like homonyms in natural languages, these abbreviations introduce ambiguity: for instance, both `return` and `range` are abbreviated as `r`. However, by using the context in which the abbreviations occur, our model finds the correct expansion of an abbreviated line of code with near-perfect accuracy. Intuitively, `return` and `range` are used in contexts that are different enough for a model to be accurate in determining which occurrences of `r` should expand to each, as well as which occurrences should not be expanded at all. Our model's offline performance demonstrates that this intuition holds for a significant number of abbreviations.

These results motivate a system in which users type a whole line of code using these ambiguous short-hands, and have the system then expand it back to valid code. Figure 1 illustrates how this feature, which we call *pragmatic autocomplete*, works during user interaction. Users type code using abbreviations, like on the left. After typing an entire line of code, the user presses a button that triggers our model's expansion prediction. This produces the line of code shown on the right side, moving the cursor to the next line. Importantly, because of the model's high accuracy, no user confirmation is necessary: occasional errors can be manually fixed by the user, an event that becomes rarer the more the user adapts to the system. We validate that this proposed feature is usable and reliable in a user study, integrated with Microsoft VSCode's text editor. In summary, this paper makes the following contributions:

- We formulate the problem of *Pragmatic Code Inference*. This simple formulation generalizes the problem that traditional autocomplete systems solve, while accommodating other solutions that simplify the process of writing code.
- We describe an algorithm that proposes an optimal set of one-character identifier abbreviations for a corpus of code, and a model that can expand these abbreviations with near-perfect accuracy.
- We evaluate the proposed system both in offline tasks in 3 programming languages, and also in a user study. While the offline experiments demonstrate that the proposed system is reliable and that context is an important feature, the user study shows that it is easy to use and complementary to the autocomplete feature present in popular text editors.

2 Related Work

Since typing on digital devices became a widespread activity, autocomplete is now a ubiquitous feature of text input systems (Palin et al. 2019). These systems usually predict the next characters or words that the user is about to type, allowing users to confirm the prediction instead of typing it. Having high prediction accuracy is necessary in order to effectively improve users' typing rate (Trnka et al. 2009). However, even with highly accurate systems, the cognitive effort of verifying and choosing between suggestions oftentimes overwhelms users and ends up reducing their effective speed. This effect has been repeatedly observed in the Human-Computer Interaction (HCI) community (Koester and Levine 1996; Trnka et al. 2009; Quinn and Zhai 2016; Palin et al. 2019). To avoid this problem, our system finds abbreviation that it is able to invert with close to 100% accuracy, avoiding the need to consult the user on every prediction. Our interaction flow is most similar in spirit to the *keyword-based* autocomplete approach in (Lee, Hashimoto, and Liang 2019). In the system they propose, users can type out only the most informative words from the intended sentence. Then, their model inserts at once all predictable words that were left out. For example, the user might type "10 late" expecting the system to complete the message "I will be 10 minutes late".

For programming languages, models for code completion have been widely studied in the recent past (Raychev, Vechev, and Yahav 2014; Asaduzzaman et al. 2014; Bielik, Raychev, and Vechev 2016; Hashimoto et al. 2018; Svyatkovskiy et al. 2019; Svyatkovskoy et al. 2020). However, despite the lessons from previous work in HCI, these code completion systems are typically not evaluated with users. To the best of our knowledge, only Pythia (Svyatkovskiy et al. 2019) was integrated into a real code editor (Microsoft Visual Studio). The authors describe several practical techniques that they needed to make the feature fast enough to be used during user interaction. However, they do not report any experiments related to user efficiency. Because the interaction model is known to be crucial to improve user's efficiency, we chose to not adapt these offline models ourselves to attempt a comparison in our user study. Instead, we compare it with the autocomplete feature from the popular Microsoft VSCode editor.

3 Problem

While reducing typing effort for users is a common goal for a large body of work, the specific formulation of the problem that proposed systems attack varies widely. In this section, we propose a simple formulation that unifies different approaches to the more general problem of helping users type more efficiently, and then instantiate it to our particular solution.

Suppose the user wants to type an intended line of code l, from a universe L (e.g. lines of code in Python), in a context $c \in C$ (e.g. the previous lines of code in the file). In a common typing environment, one option is always to type l using $|l|$ keystrokes, without any assistance. However, predictive features can be used to require less input. We formalize this process as follows. A *predictive typing system* can be broken down into 3 components: (i) a set of signals S, which are strings that the user might type, (ii) a predictive function $p_\theta(l|s, c)$, that takes a user-provided signal s and its context c, and outputs a probability distribution over L, and (iii) a user interaction mechanism used to resolve ambiguity, when the prediction generated by p is uncertain.

[1] https://github.com

Within this formulation, two common desiderata become clear: we want a system to allow users to provide short signals S, while still having high predictive accuracy. Precisely, suppose the user input is generated by a probability distribution with density $\mathbb{P}$ over triples (s, c, l), where $s \in S$ is the user-provided signal, $c \in C$ is the context in which the signal was provided, and $l \in L$ is the intended line of code. Then, a predictive typing system will optimize the trade-off between two goals. First, we want to maximize the *compression* the user gains when providing short signals instead of directly providing the intended lines of code. We can define the system's *compression* $C_{\mathbb{P}}$ as:

$$C_{\mathbb{P}} = 1 - \mathbb{E}_{(s,c,l)\sim\mathbb{P}}\left[\frac{|s|}{|l|}\right] \ .$$

For example, a compression $C_{\mathbb{P}} = 0.10$ would mean that, on average, users save typing 10% of the characters by using the system's predictive feature. Moreover, we want the predictions to have high *accuracy*. If the user provides signal s under context c, the system's best guess about the intended line is the line l that maximizes p_θ. Formally, we can denote this best-guess prediction by a prediction function $\hat{l}_\theta$, defined as:

$$\hat{l}_\theta(s, c) = \arg\max_{l} p_\theta(l|s, c) \ .$$

Then, the system's accuracy A_θ is the probability of the prediction $\hat{l}(s, c)$ being correct. Precisely,

$$A_\theta = \mathbb{P}\left(l = \hat{l}_\theta(s, c)\right) \ .$$

Therefore, we define the problem of *Pragmatic Code Inference* as (i) designing the set of possible user signals S, which determines $C_{\mathbb{P}}$, and then (ii) optimizing the system's predictive accuracy A_θ. From this formulation, we can obtain traditional autocomplete systems by only allowing signals S to be well-formed prefixes of lines of code in L. Then, p_θ assigns probabilities to possible continuations of the provided prefix. However, predicting the full completion of a line of code given a small prefix is in general not feasible. Therefore, to be practical, such systems typically make predictions one or a few tokens at a time, with repeated interactions with the user for each line of code. Consulting the user every time the system performs a prediction seems inevitable, since the high uncertainty in determining what continuation the user wants is inherent to the problem. However, this problem formulation makes evident that this uncertainty stems from the choice of S. By choosing a different set of signals S, one might hope to still achieve significant compression while making prediction easier. This is the goal of the approach we lay out next.

4 Approach

In this section, we provide our approach to building a *predictive typing system* in light of the formulation from Section 3, including a solution to the *Pragmatic Code Inference* problem. This involves designing a set of short user signals S that are amenable to accurate inference given context. We start by discussing higher-level design goals, and then describe the algorithms in each component of the system.

Design Goal: Avoid Disambiguation Interactions

As we observed before, the typical choice of autocomplete systems of taking signals S that are prefixes of L limits them to having low accuracy. This happens because there often are multiple plausible completions of a prefix of a line of code, depending on the user's intent. Under this constraint, a final user prompt to confirm every prediction is required for the system to be usable. However, it is known from experimental observations in HCI that these interactions with predictive systems might outweigh benefits in input compression. Therefore, we instead aim at obtaining a model with enough accuracy such that we can avoid as much as possible to interact with users for resolving ambiguities. To that end, we need to carefully design the set of signals S that our system accepts. Our goal is to constrain S in a way that allows a simple model to have high accuracy while still obtaining a significant *compression*.

Abbreviations

Our approach takes inspiration from the efficiency of human communication. Over time, long words tend to be shortened, often collapsing into homonyms. But while most English words have multiple senses, context usually contains enough information to disambiguate each word (Piantadosi, Tily, and Gibson 2012). Since relevant context might come after the word itself in a sentence, humans use the entire utterance to make their final interpretation of individual words (Lieberman 1963). These observations lead us to choose for signals S, whole lines of code in which some short tokens have multiple interpretations, analogous to words with multiple senses in natural languages. Specifically, we simply allow tokens belonging to a defined set to be shortened to their initial character.

To choose the tokens that can be abbreviated in a principled fashion, we directly maximize *compression* as defined in Section 3. For such, assume we have a representative dataset $\mathcal{D}$ of code in the target programming language. Given the number of desired abbreviations n, Algorithm 1 finds the set of n tokens that yield optimal compression under the abbreviation scheme we described. First, the algorithm splits all lines of code from $\mathcal{D}$ into tokens, building a token frequency table $\mathcal{F}$. For each distinct token t that occurs $\mathcal{F}[t]$ times in $\mathcal{D}$, the number of characters we would erase from $\mathcal{D}$ if occurrences of t were replaced by t_1 (the first character of t) is exactly $(|t| - 1) \times \mathcal{F}[t]$. Since we're only considering whole tokens, deciding to abbreviate one token t (e.g. the keyword `return`) does not influence other tokens that have t as a substring (e.g. a variable called `returnValue`). Therefore, the decision for each token is independent, and the optimal choice for n tokens is to greedily take those that maximize $(|t| - 1) \times \mathcal{F}[t]$. $FindAbbreviatableSet$ returns a list of such n tokens, ordered by their contributions to compression.

While this set optimizes compression, in practice we would also like to have a method for tuning the system's accuracy. Algorithm 2 provides such a mechanism.

Given an ordered list obtained by $FindAbbreviatableSet$, $CapCollisions$ modifies it so that for each character, there are at most c possible expansions for it. Naturally, in case the number of expansions surpasses the limit, it keeps the c alternatives that contribute the most to compression. This parameter c of $CapCollisions$ provides a knob for trading compression for better performance. A smaller c increases potential accuracy, since there are less options to choose from. In addition, it also bounds the number of options that need to be evaluated at runtime, giving users a simple way to calibrate the system's responsiveness depending on the available hardware.

```
Function FindAbbreviatableSet(𝒟, n)
    ℱ[·] ← 0
    foreach l ∈ 𝒟 do
        foreach t ∈ tokenize(l) do
            ℱ[t] ← ℱ[t] + 1
        end
    end
    𝒯 ← ℱ.keys().sort(λt . (|t| − 1) × −ℱ[t])
    return 𝒯[1...n]
```

Algorithm 1: Finds a list of n tokens that maximize compression on a dataset $\mathcal{D}$ of lines of code, when those tokens are abbreviated to a single character.

```
Function CapCollisions(𝒯, c)
    𝒯′ ← [ ]
    for i ← 1 to |𝒯| do
        if 𝒯′.count(λt . t_1 = 𝒯[i]_1) < c then
            𝒯′.append(𝒯[i])
    end
    return 𝒯′
```

Algorithm 2: Removes elements from a list of tokens $\mathcal{T}$ so that the number of potential expansions for any abbreviation is limited by c. This allows users to smoothly trade-off compression for higher accuracy.

Expanding Abbreviations

With a fixed *abbreviatable set* $\mathcal{S}$ obtained by combining Algorithms 1 and 2, we can now train a neural network to invert these abbreviations. During training, we assume a *maximally terse* user, that uses all abbreviations whenever possible. This process is illustrated in Figure 2. We start with a dataset $\mathcal{D}$ of code (a), containing lines of code paired with a fixed number of K lines that preceded it in its original file[2]. We also have the set $\mathcal{S}$, depicted in (b). After sampling one line from $\mathcal{D}$ (e.g. `return self.request.size()`), we abbreviate all tokens in it that belong to $\mathcal{S}$, obtaining an abbreviated line (e.g. `r s.r.size()`, shown in (c)). We then feed this shortened line into a standard character-based sequence-to-sequence model with attention (d), trained with teacher forcing to recover the original line (e). Context is embedded using an one-dimensional convolutional layer followed by max-pooling, and fed into the network using ConcatCell (Jaech and Ostendorf 2018). The same character embeddings are used for the signal and the context, and are learned end-to-end during training. With this procedure, we obtain a model for $p_\theta(l|s, c)$.

[2] The context will contain less than K previous lines for samples taken from the top $K-1$ lines of a file.

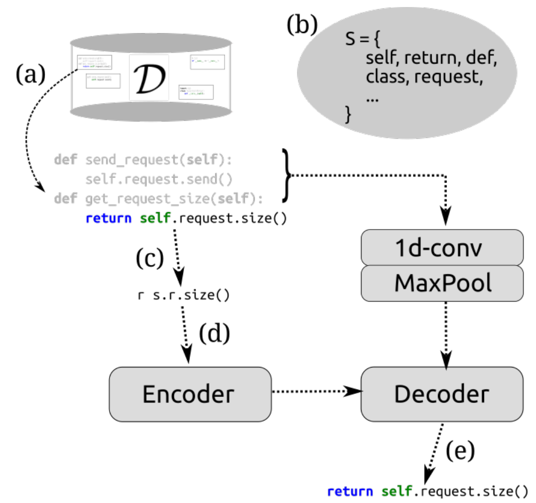

Figure 2: Architecture and training of the conditional language model that expands abbreviations. We start with a dataset (a) and a set S of abbreviatable tokens (b). During training, we get a sample line of code, apply all possible abbreviations (c), and feed it into a contextual sequence-to-sequence model (d) that is trained using teacher forcing to recover the original line (e).

During test time, the set of possible expansions for any signal s can be directly enumerated by using the set $\mathcal{S}$. Because of the abbreviation scheme we use, only single-character tokens in the input are candidates for expansion. Each of those can either remain as they are or be expanded to one of the strings in $\mathcal{S}$ that start with that character. Instead of having the neural network output l, which sometimes produces errors unrelated to the abbreviations, we instead enumerate the potential expansions with beam search, using $p_\theta(l|s, c)$ to rank them. We normalize the log-probability of each candidate by its length, otherwise expansions would be naturally discouraged for making the output longer. If the set $\mathcal{S}$ was post-processed by `CapCollisions`, the branching factor during beam search is bounded by c. As we show in Section 5, this system is already useful when c is set to numbers ranging from 2 to 4, which guarantees good responsiveness during use.

Personalization

The described training procedure assumes a dataset of representative source code in a given programming language. However, one specific user or project might use libraries, names or coding patterns that are not globally common. The generic aspect of the set $\mathcal{S}$ is apparent when we inspect the abbreviatable tokens we obtain for our Python, Java and JavaScript datasets. Besides the basic keywords of each lan-

guage, we find class and function names mostly belonging to standard libraries. Thus, in order to maximize the compression our system yields to one particular user, we want to be able to personalize it.

Assuming the user has a small corpus of existing code, this personalization of the model can be done by starting with a pre-trained model, and then simply (i) using Algorithm 1 to find the list of tokens $\mathcal{T}_{user}$ that maximizes compression for the user's code base, and then (ii) fine-tuning the neural model using the user's code, exactly as done in training. As we report in Section 5, applying this straightforward adaptation step to a single code repository significantly boosts the compression the system yields, while maintaining high accuracy. Moreover, this procedure is efficient enough to run on a commodity CPU for a medium-sized repository. To give a concrete idea, fine-tuning to a repository with 10k lines of Python takes less than 5 hours on a modern laptop. This is important since individual users may not have GPUs, and sending code to an external server brings potential privacy concerns.

User Interaction

Initially, the user does not know which keywords can be abbreviated. To allow for a smooth learning curve, our prototype highlights tokens in the set $\mathcal{S}$ when the user types them in their full form. This communicates, without interruption, that the user might choose to abbreviate that token the next time it is needed. Despite this cuing, there may be a substantial learning curve while users internalized which tokens can be shortened. To expand abbreviations, the user presses a shortcut once after an entire line is typed – that line is expanded and the cursor moved to the next.

5 Experimental Results

In this section, we assess how well the *pragmatic autocomplete* system we described meets the goals of allowing concise inputs, having high accuracy and being usable to humans. We implemented[3] our algorithms and models using the PyTorch 1.6.0 library (Paszke et al. 2019). To train and evaluate our models, we collected a dataset of code from open source repositories in Python, Java and JavaScript, the 3 programming languages that were ranked as the most popular in the Stack Overflow Developer Survey 2020[4]. We ran all experiments on a server with an NVIDIA™ TITAN™ Xp GPU with 12GB of memory. Architecture details and parameters can be found in the Appendix. We use this prototype to answer the following research questions:

1. How accurate is the model in expanding abbreviated lines from a maximally terse user? Is context, both in the form of previous lines and words surrounding an abbreviation within-line, important to obtaining high accuracy?
2. Does the model allow for a smooth trade-off between compression and accuracy, as we change parameters?
3. What is the impact of fine-tuning the model on a particular repository in terms of accuracy and compression?
4. Can human users benefit from using our *pragmatic autocomplete* feature to type code?

Dataset

We used the Github API to collect a list of 7 million public repositories. From these repositories, we sampled 10^5 files of source code written in each of our target programming languages. We split these files into training/validation/test sets using a standard 80%/10%/10% split. From each file, we extract 20 examples of lines of code along with the 10 previous lines as its context. These limits are not reached in small files or in the first lines of a file. This gives us a dataset of roughly 1.3 million lines of code with context in each language.

Accuracy and Context

First, we evaluate how accurate our model is in expanding abbreviations produced by a maximally terse user. In this experiment, for each programming language, we use `FindAbbreviatableSet` on the training set to compute a set of 100 abbreviatable strings. Here, we use the entire obtained set, without capping the number of collisions (i.e. without `CapCollisions`). We then abbreviate all examples in the test set, and evaluate the model's accuracy in reconstructing the original line. We vary the number of previous lines given as context to our model. As a fully non-contextual baseline, we use an LSTM-based language model for each language to rank potential expansions. The results we obtained are in Table 1. We observe that the two forms of context that we consider are both important to obtain high accuracy results. This observation is consistent across programming languages. For all of them, a Conditional Language Model (CLM) that takes no previous lines performs significantly better than a Language Model (LM) that does not consider the user signal, but instead ranks alternatives in an isolated manner (i.e. asking how likely a given expansion is regardless of the user signal).

Adding previous lines as context improves further. The best context size depends on the programming language. After a certain point, adding more context will have minimal or even negative impact in accuracy. Indeed, given our architecture, providing distant context could dilute the useful information in nearby context: since context is embedded using convolutions and max-pooling, each dimension of its representation intuitively tells whether a given pattern (matched by a convolution kernel) occurs at least once in the context. That signal is informative when it refers to lines of code that are nearby, but less so when the occurrence of the pattern is distant from the current line. Overall, given the gains we observed when using small contexts, we can give a positive answer to our first research question: using context to interpret user signals is useful, and it is possible to obtain high top-1 accuracy with a simple contextual model.

Ambiguity and Compression

We now evaluate the trade-off between compression and accuracy that *CapCollisions* provides, again considering a maximally terse user. A degenerate model that allows no

[3] Code is available at https://github.com/gpoesia/magicomplete

[4] https://insights.stackoverflow.com/survey/2020

Model	Python	Java	JavaScript
LM	0.746	0.817	0.736
CLM, ctx size 0	0.908	0.942	0.880
CLM, ctx size 1	0.924	0.957	0.907
CLM, ctx size 3	0.938	**0.963**	0.925
CLM, ctx size 5	0.942	**0.963**	**0.931**
CLM, ctx size 7	**0.945**	**0.963**	0.926

Table 1: Top-1 accuracy of our Conditional Language Model (CLM) given a context of varying size, and a non-contextual Language Model (LM), in a reconstruction task that considers a maximally terse user.

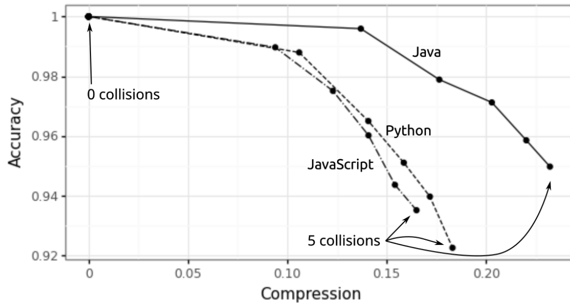

Figure 3: Accuracy/compression trade-off for our Conditional Language Model, varying the maximum number of elements we allow in the abbreviatable set that start with the same character (parameter c of `CapCollisions`).

collision in any of the characters, obtained by setting $c = 0$ (i.e. no abbreviations), necessarily obtains 100% accuracy, since there is no abbreviation to expand. For the same reason, it also yields 0% compression. When we increase c, we would like to see only small losses in accuracy with significant improvements in compression. This is indeed what we find (Figure 3). The gains in using even small values of c are significant: with $c = 3$, we get roughly 15% compression in Python and JavaScript, and above 20% compression in Java. Meanwhile, accuracy remains high – ranging from 93% to 99% depending on c and the programming language. These observations allow a positive answer to our second research question: the contextual model provides a smooth trade-off between compression and accuracy.

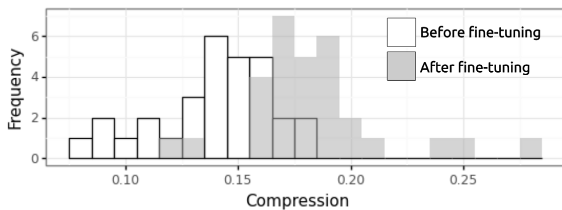

Figure 4: Compression obtained in 30 repositories before and after fine-tuning the system.

Adaptation and Fine-tuning

Particular repositories may have very different statistics than the language as a whole (for instance, due to heavy use of libraries and repository-specific helper functions). This suggests that adapting our system to a given repository may be useful. To evaluate this idea we uniformly sampled 10000 Python repositories from our corpus. Most repositories on Github are rather small, with a median of only 4 files. Fine-tuning the system for a project is more reliable when we have enough data to both train and evaluate the updated model. Therefore, we took 30 random repositories among the 100 largest in our sample. We did not attempt to find optimal configurations for any particular repository; this experiment was only run once, splitting the files in each repository into 80% for training and 20% for evaluation. We start with our generic Python model with 100 abbreviations. Then, for each repository, we recompute the optimal abbreviatable set, and then fine-tune the model for 10 epochs on the specific repository data.

When applying all abbreviations in each repository before and after the abbreviatable set is adapted, the benefits of fine-tuning become clear. The distribution shifts towards higher compressions, as shown in Figure 4. Indeed, the median compression rises from 13.8% without adaptation to 17.7% with.

This gain is only useful if accuracy is maintained. Accuracies of the generic model gets in each repository are fairly high, with a median of 93.5%. After we adapt the abbreviations, but before the model is fine-tuned, these accuracies drop significantly, to a median accuracy of 75.7%. Thus, the model does not adapt to new abbreviations in a zero-shot manner. After fine-tuning with the new abbreviations, however, we again observe high accuracies in all repositories, with a similar distribution as before. The median accuracy drops slightly, to 91%, which could potentially be improved by using $CapCollisions$ and tuning hyperparameters. These results allow us to give a positive answer to our third research question: fine-tuning to a specific repository further improves performance.

User Study

We integrated our prototype of *pragmatic autocomplete* in Microsoft VSCode, the most popular code text editor in the Stack Overflow Developer Survey 2019[5]. VSCode's editor component can be executed in a Web environment. It has all editing features available in the offline version, including it's default autocomplete that predicts the next token while the user types. We recruited 10 participants, who were students at `redacted`. Participants were shown 5 Java code snippets, on the left side of the screen, one at a time, and had to type it on the right side as fast as possible using four different autocomplete conditions: "No autocomplete", "Pragmatic", "VSCode default", or "VSCode+Pragmatic". The snippets were up to 10 lines of code, randomly sampled from our dataset before the experiment. Each participant saw each snippet in each condition, with order of snippets and conditions randomized. Before each condition, they watched a

[5]The 2020 survey did not ask about development environments.

short video that showed how the editor works with the autocomplete feature that they were about to use. In total, the experiment took from 25 to 50 minutes. Participants accessed the experiment interface from their own computers. (Our model was running on a shared server with no GPU, and a Intel™Xeon™2.40GHz CPU.) We measured users' effective typing speeds and number of keystrokes needed to type each snippet in each setting. After the experiment, participants filled a survey asking about the frequency with which they use VSCode, which of the 4 settings they would most like to use, and an optional free-form comment about their experience during the task.

Typing speeds in the three autocomplete settings are shown in Figure 5; here we used the "No autocomplete" setting to normalize our measures in other settings, since typing speeds varied widely between participants (ranging from 127 to 421 characters per minute in a snippet). We notice that there is a learning curve for our pragmatic autocomplete, with users increasing typing speed over the course of usage. By the last snippet, using both autocomplete features users were on average 9.3% faster than not using any autocomplete (7.4% with just *Pragmatic Autocomplete*). This is still lower than what users obtained by using VSCode's autocomplete alone: in that setting, in the last snippet, users were 13% faster than not using any autocomplete. Typing speed doesn't improve across trials using only VSCode's features. This is expected since VSCode's autocomplete is very similar to other editors, furthermore most users were already familiar with VSCode: in a survey, only 3 users reported never or rarely using VSCode, while the other 7 use it occasionally or as their primary editor.

The picture is different when we look at the number of keystrokes participants needed. A visualization of this result is shown in Figure 6. We find *Pragmatic Autocomplete* reduces the number of keystrokes when added to either the "No autocomplete" or to the "VSCode default" setting. In the last two snippets in the setting "VSCode+Pragmatic" participants used only 83.8% of the keystrokes they needed without autocomplete, compared to 88.3% with just VSCode's autocomplete. These systems are thus complementary. (Indeed, our feature focuses on reducing the effort to type very frequent tokens. Users typically ignore traditional autocomplete for those, as they would need to shift attention to the prompt. On the other hand, our autocomplete does not yet expand local identifiers, which VSCode does.) The observed keystroke savings indicate that, with enough familiarity with our system and a lesser need to stop and confirm that it did what was expected, typing speeds should improve. To observe significant typing speed gains over traditional autocomplete, we would likely need to observe users for a longer period. Furthermore, we would ideally observe programmers in a more natural setting, since the task used here–copying code–is significantly different than programming.

When asked to rank which setting they would most likely use, all 10 participants reported they would enable both autocomplete systems. If they had to choose just one, 9 participants would prefer the more familiar default autocomplete, and no participant would prefer having no autocomplete. Overall, these results provide positive evidence for our fourth research question: *Pragmatic Autocomplete* is usable and complements traditional systems.

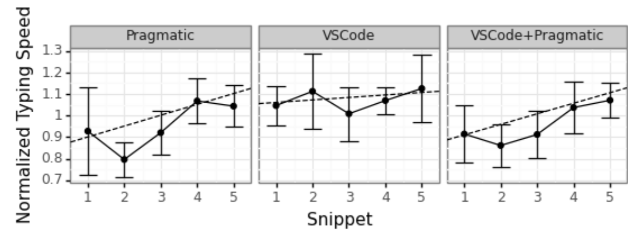

Figure 5: Average user typing speed using each autocomplete setting in each snippet that each participant typed (normalized by speed not using autocomplete for the same snippet). Error bars are bootstrapped 95% confidence intervals, and we show the best regression line. Typing speed started below 1 for Pragmatic and VSCode+Pragmatic, but improved significantly during the experiment ($R^2 = 0.53$ and 0.74, respectively). Typing speed in VSCode was higher, but improved less throughout the task ($R^2 = 0.15$).

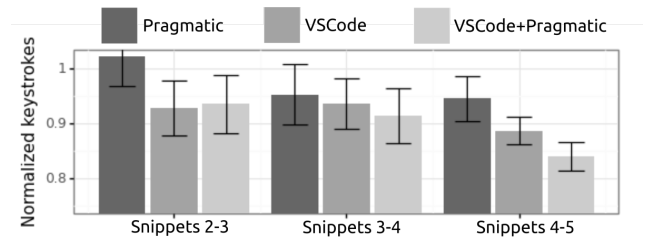

Figure 6: Average keystrokes users needed to type two consecutive snippets, as the experiment progresses. Counts are normalized by "No autocomplete", and shown with 95% bootstrapped confidence intervals. *Pragmatic Autocomplete* increasingly adds savings on top of *No autocomplete* and *VSCode*.

6 Conclusion

The goal of autocomplete systems in programming is to provide users with a more efficient interface for typing code. This has traditionally been accomplished by predicting continuations of prefixes of the input, but that does not need to be the case. We thus proposed a *Pragmatic Autocomplete* system, that allows for ambiguity at the level of keywords and identifiers. Our approach was inspired by the way natural language embraces ambiguity, which can then be resolved in context. Our model achieved high accuracy disambiguating abbreviations in context, in a way that complements traditional autocomplete systems. We hope that this new formulation of predictive typing systems can open up other avenues of research in designing intuitive and compact user signals that a model can interpret, yielding more efficient forms of communication between users and machines.

Acknowledgements

We thank the student volunteers that participated in the user study, especially after all data from the first run of the study was stolen for ransom and we needed other volunteers in short notice. This work was supported by a NSF Expeditions Grant, Award Number (FAIN) 1918771.

A Architecture Details

The neural model we described in Section 4 uses an LSTM with 512 hidden units for both the bidirectional and the decoder, and 50-dimensional character embeddings. We embed context into a 128-dimensional vector. All models were trained for 10 epochs, using the Adam optimizer with a learning rate of 0.0005 and $\beta = (0.9, 0.999)$, and a batch size of 512. The learning rate was set by doing simple search over 10^{-i} and 5×10^{-i} for $i \in \{1, 2, 3, 4, 5\}$, using validation accuracy to pick the value we used in later experiments. The size of our character and context embeddings were not tuned, and batch size was set to the maximum power of two that fit in our GPUs. One epoch on roughly 1 million training examples takes about one hour. On our CPU, processing one example typically takes between 80 and 150 milliseconds, depending on how many expansions have to be evaluated. The model can potentially be made smaller with similar performance by tuning all parameters altogether, but we did not attempt to do so.

Similarly, the Language Model baseline is a vanilla LSTM with a hidden layer size of 512. We tested the same set of learning rates, and found the same value of 0.0005 to work best.

References

Asaduzzaman, M.; Roy, C. K.; Schneider, K. A.; and Hou, D. 2014. Cscc: Simple, efficient, context sensitive code completion. In *2014 IEEE International Conference on Software Maintenance and Evolution*, 71–80. IEEE.

Bielik, P.; Raychev, V.; and Vechev, M. 2016. PHOG: probabilistic model for code. In *International Conference on Machine Learning*, 2933–2942.

Fortuny, J.; and Corominas-Murtra, B. 2013. On the origin of ambiguity in efficient communication. *Journal of Logic, language and Information* 22(3): 249–267.

Hashimoto, T. B.; Guu, K.; Oren, Y.; and Liang, P. S. 2018. A retrieve-and-edit framework for predicting structured outputs. In *Advances in Neural Information Processing Systems*, 10052–10062.

Jaech, A.; and Ostendorf, M. 2018. Low-rank RNN adaptation for context-aware language modeling. *Transactions of the Association for Computational Linguistics* 6: 497–510.

Koester, H. H.; and Levine, S. 1996. Effect of a word prediction feature on user performance. *Augmentative and alternative communication* 12(3): 155–168.

Lee, M.; Hashimoto, T. B.; and Liang, P. 2019. Learning Autocomplete Systems as a Communication Game. In *3rd NeurIPS Workshop on Emergent Communication (EmeCom@ NeurIPS 2019).*

Lieberman, P. 1963. Some effects of semantic and grammatical context on the production and perception of speech. *Language and speech* 6(3): 172–187.

Palin, K.; Feit, A. M.; Kim, S.; Kristensson, P. O.; and Oulasvirta, A. 2019. How Do People Type on Mobile Devices? Observations from a Study with 37,000 Volunteers. In *Proceedings of the 21st International Conference on Human-Computer Interaction with Mobile Devices and Services*, MobileHCI '19. New York, NY, USA: Association for Computing Machinery. ISBN 9781450368254. doi:10.1145/3338286.3340120. URL https://doi.org/10.1145/3338286.3340120.

Paszke, A.; Gross, S.; Massa, F.; Lerer, A.; Bradbury, J.; Chanan, G.; Killeen, T.; Lin, Z.; Gimelshein, N.; Antiga, L.; et al. 2019. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, 8026–8037.

Piantadosi, S. T.; Tily, H.; and Gibson, E. 2012. The communicative function of ambiguity in language. *Cognition* 122(3): 280–291.

Quinn, P.; and Zhai, S. 2016. A cost-benefit study of text entry suggestion interaction. In *Proceedings of the 2016 CHI conference on human factors in computing systems*, 83–88.

Raychev, V.; Vechev, M.; and Yahav, E. 2014. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 419–428.

Solé, R. V.; and Seoane, L. F. 2015. Ambiguity in language networks. *The Linguistic Review* 32(1): 5–35.

Svyatkovskiy, A.; Zhao, Y.; Fu, S.; and Sundaresan, N. 2019. Pythia: ai-assisted code completion system. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2727–2735.

Svyatkovskoy, A.; Lee, S.; Hadjitofi, A.; Riechert, M.; Franco, J.; and Allamanis, M. 2020. Fast and Memory-Efficient Neural Code Completion. *arXiv preprint arXiv:2004.13651* .

Trnka, K.; McCaw, J.; Yarrington, D.; McCoy, K. F.; and Pennington, C. 2009. User Interaction with Word Prediction: The Effects of Prediction Quality. *ACM Trans. Access. Comput.* 1(3). ISSN 1936-7228. doi:10.1145/1497302.1497307. URL https://doi.org/10.1145/1497302.1497307.