When to Show a Suggestion? Integrating Human Feedback in AI-Assisted Programming

Hussein Mozannar[1], **Gagan Bansal**[2], **Adam Fourney**[2], **Eric Horvitz**[2]

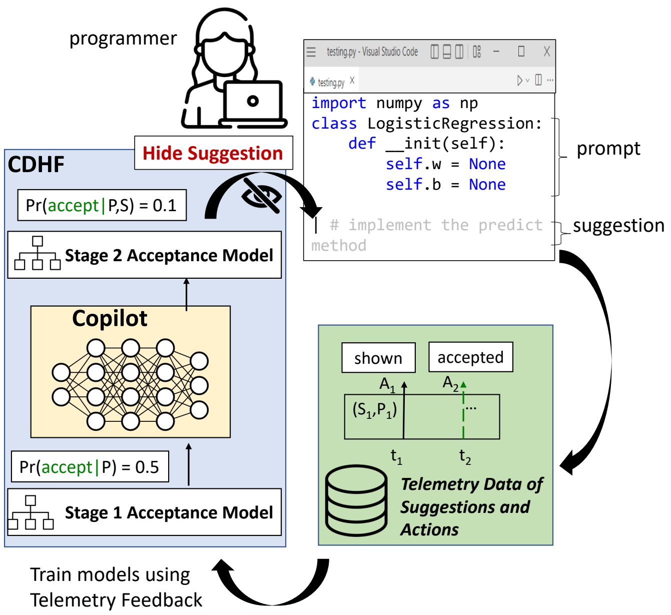
[1]Massachusetts Institute of Technology
[2]Microsoft Research
mozannar@mit.edu

Abstract

AI powered code-recommendation systems, such as Copilot and CodeWhisperer, provide code suggestions inside a programmer's environment (e.g., an IDE) with the aim of improving productivity. We pursue mechanisms for leveraging signals about programmers' acceptance and rejection of code suggestions to guide recommendations. We harness data drawn from interactions with GitHub Copilot, a system used by millions of programmers, to develop interventions that can save time for programmers. We introduce a utility-theoretic framework to drive decisions about suggestions to display versus withhold. The approach, conditional suggestion display from human feedback (CDHF), relies on a cascade of models that provide the likelihood that recommended code will be accepted. These likelihoods are used to selectively hide suggestions, reducing both latency and programmer verification time. Using data from 535 programmers, we perform a retrospective evaluation of CDHF and show that we can avoid displaying a significant fraction of suggestions that would have been rejected. We further demonstrate the importance of incorporating the programmer's latent unobserved state in decisions about when to display suggestions through an ablation study. Finally, we showcase how using suggestion acceptance as a reward signal for guiding the display of suggestions can lead to suggestions of reduced quality, indicating an unexpected pitfall.

Introduction

Code recommendation systems powered by large-scale neural language models, such as Github Copilot (Github 2022) and Amazon CodeWhisperer (Amazon 2022), are aimed at providing programmers with code suggestions to help improve their productivity. These systems usually operate by displaying the suggestion as *ghost text*—a grayed-out code suggestion inline inside the IDE. Programmers can accept the suggestion, browse through different suggestions, or reject the suggestion (see Figure 1). The code suggestions appear either at the explicit invocation of the programmer or when the programmer pauses their cursor when writing code. GitHub reported a recent randomized study with 95 participants who wrote a web server, where they found that Copilot could potentially reduce task completion time by a factor of two (Kalliamvakou 2022). These and other reports that the code-recommendation systems can improve programmer productivity motivate our research to pursue improvements to these systems.



```
import numpy as np
class LogisticRegression:
    def __init__(self):
        self.w = None
        self.b = None

# implement the predict method
```

Figure 1: Operating mode of Copilot inside Visual Studio Code showing how CDHF influences the interaction by selectively hiding certain suggestions. Data collected by the interaction is stored in telemetry and used to train CDHF to create a feedback loop.

Code-recommendation systems are powered by large language models (LLMs) such as GPT that are trained on standard language modeling objectives using the Common Crawl data (Radford et al. 2019), and then fine-tuned on public code repositories (Chen et al. 2021). The public roll-out of the code recommendation models has attracted millions of programmers, enabling a unique opportunity to leverage the data of programmers interacting with the models. In this work, we study GitHub's Copilot which is used by millions of programmers (Github 2022). For a set of programmers within our organization who consented to have their usage data collected, we collected telemetry data of Copilot suggestions, along with their associated prompts and the programmer's action to accept or reject the suggestions. We leverage this telemetry data to design mechanisms and interventions that can improve the interaction between programmers and Copilot.

Specifically, we seek to identify *when* to show a code suggestion. We first define the expected utility of a displaying a suggestion, a value that measures the impact of showing a suggestion on the overall time to write a specific piece of code. This value provides an optimal criterion for when to show a suggestion. However, computing the utility of suggestions is difficult and not currently feasible. Instead, we rely on the result that suggestion utility increases the more likely a suggestion is to be accepted and decreases with increasing latency to generate a suggestion—two quantities we can reliably estimate and control, respectively. We develop a procedure, named **c**onditional suggestion **d**isplay from **h**uman **f**eedback (CDHF) which guides whether to show or hide suggestions. At each pause in keystrokes, CDHF decides whether if it is worthwhile to generate a suggestion and if the programmer is likely to accept the generated suggestion. CDHF employs a cascade of models that predict acceptance of suggestions. The optimization procedure guarantees that any suggestion that was hidden (or not generated) would have been rejected if it was shown with a probability of at least p, where, e.g., p can be 0.99.

Using data from programming sessions of 535 programmers with feedback on 168k suggestions, we perform a retrospective evaluation of CDHF. We show that we can hide 25% of suggestions that were shown while guaranteeing that 95% of them would have been rejected. Further, we avoid generating 13% of these suggestions. The results show that CDHF would increase the acceptance rate by 7.2%. The procedure allows for controlling a trade-off that balances the number of suggestions that are displayed with increases in latency, controlled with a parameter that halts generations. We note that a minimal version of CDHF has been implemented in a newer version of GitHub Copilot (Zhao 2023) following the presentation of earlier versions of our work to GitHub. Our paper provides a roadmap for building and fielding better forms of suggestion display.

Beyond decisions about displaying recommendations, we examine the feasibility of using suggestion acceptance as a reward signal to select *which* suggestions to display and show how partial completions can be prioritized over the generations of complete code segments. While we investigate Copilot in this work, we believe our insights extend to other AI models and non-code-based tasks. Please refer to the arXiv version of this work for an appendix (Mozannar et al. 2023).

Related Work

The closest related work to ours is the procedure to selectively hide suggestions in (Sun et al. 2022) (quality estimation before completion, QEBC). In distinction to this work, QEBC (Sun et al. 2022) is not based on human feedback of accepting suggestions but rather is based on constructing a learned estimator of the quality of code completions from datasets of paired code segments and model completions. Our CDHF estimator uses real programmer behavior data and is based on data from a code-recommendation system in current use (Copilot) as opposed to custom-trained ones in (Sun et al. 2022). Different metrics and datasets have been proposed to evaluate the performance of code recommendation models, but these typically assess how well the model can complete code in an offline setting without developer input rather than evaluating how well it assists programmers in situ (Ziegler et al. 2022; Li et al. 2022; Evtikhiev et al. 2022; Dakhel et al. 2022). Integrating human preferences when training machine learning models has long been studied in the literature (Knox and Stone 2008; MacGlashan et al. 2017). In particular, reinforcement learning from human feedback (RLHF) has been used to improve LLMs used as conversational chatbots (Ziegler et al. 2019; Bai et al. 2022), notably ChatGPT (OpenAI 2022). In contrast, CDHF uses human feedback collected organically through telemetry. The objective is fast inference to reduce latency and hiding suggestions rather than updating the LLM. Further related work can be found in the appendix. Our theoretical formulations build on earlier work on harnessing machine learning and utility to guide AI versus human-powered contributions in human-AI interactive settings (Horvitz 1999), which we apply to our setting.

Problem Setting

Copilot. We consider Copilot, which is a commonly used and exemplary tool of AI-powered code recommendations used by millions of programmers (Github 2022). Copilot is powered by a large language model (LLM) to provide code suggestions to programmers within an IDE whenever the programmer pauses their typing. An illustration of Copilot suggesting code as an inline, single-colored snippet is displayed in Figure 1. The programmer can choose to accept this suggestion via a keyboard shortcut (e.g., tab).

AI-Assisted Programming. We attempt a mathematical formalization of programming with the help of a code recommendation model such as Copilot, which we dub *AI-Assisted Programming*. The programmer wishes to complete a certain task T, for example, to implement a logistic regression classifier. As the programmer writes code starting from time 0, Copilot attempts to provide code suggestions at different times. At a given time t, Copilot[1] uses a portion of the code X_t to generate a prompt P_t, which is passed to the underlying LLM. Copilot then generates a code suggestion S_t, which is shown to the user at time $t + \tau$ where τ accounts for the LLM latency. Once the suggestion is shown, the programmer must make an action at a certain time $t' > t + \tau$, the action is $A_{t'} \in \{ \text{accept, reject}\}$; the *reject* action is triggered implicitly by continuing to type.

Telemetry. Copilot logs aspects of the interactions via telemetry, which we leverage in our study. We refer to event positions drawn from a discretization of times spanning a session. Specifically, whenever a suggestion is shown, accepted or rejected, we record a tuple to the telemetry

[1]We discuss implementation details of Copilot at a high level; our work is based on the August 2022 version of Copilot.

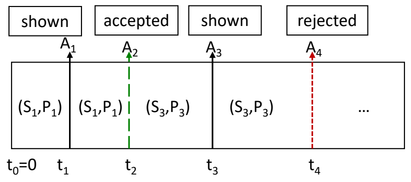


Figure 2: Schematic of telemetry with Copilot as a timeline. For a given coding session, the telemetry contains a sequence of timestamps and actions with associated prompts and suggestions.

database, (t_i, A_i, P_i, S_i), where t_i represents the within-session timestamp of the i^{th} event ($t_0 = 0$), A_i details the action taken (augmented to include 'shown'), and P_i and S_i capture the prompt and suggestion, respectively. Figure 2 displays a portion of timeline built from telemetry data drawn from a coding session. Telemetry data from each programmer is stored in a database $D = \{(t_i, A_i, P_i, S_i)\}_{i=1}^n$ and represents a discretized representation of the interaction and provides the human feedback data we leverage.

Programmer State. When faced with a suggestion, is a programmer looking and verifying it, or rather engaged in other activities such as thinking about their code or looking at documentation? The state of the programmer is important in the expected value of the recommendation. However, we cannot answer this question as the telemetry does not capture the programmer's activities and thinking between two consecutive time stamps t_i and t_{i+1}, i.e., the space in between the arrows in Figure 2 which we refer to as the *programmer's latent state*. In an earlier publication (Mozannar et al. 2022), we describe a study of 21 participants focused on gaining an understanding of sequences of states visited during the writing of code, including latent states. The work employed videos and interviews to acquire information about the latent states. We showed in the work that including information about latent states can significantly boost predictions about accepting recommendations, motivating the collection of data beyond that captured in telemetry.

In this work, we endeavor to understand the impact of the programmer latent state denoted as ϕ_t and its effect on our ability to leverage the telemetry (human feedback data) to improve AI-code recommendation systems.

Theoretical Formulation of Suggestion Utility

A critical design question in programmer-Copilot interaction is **when** should the model inject a suggestion into the IDE? The version of that we Copilot provides a suggestion when it detects a brief pause in the IDE. Alternative interaction designs would require the programmer to ask for suggestions using a keyboard shortcut or to enable a mix of human and machine initiatives. Requiring the programmer to ask may lead to sub-optimal interactions because its success would rely on programmers having an accurate mental model of Copilot abilities (Sarkar et al. 2022) which can require long-term interactions with the model (Bansal et al. 2019) or training (Mozannar, Satyanarayan, and Sontag 2022). Second, requiring an explicit invocation can disrupt the natural flow of programming, breaking a state of flow achieved during intensive focus (Csikszentmihalyi and Larson 2014). Designs requiring user initiative as well as those automatically displaying content can burden users with interruptions that decrease task performance (Bailey, Konstan, and Carlis 2001; Cutrell, Czerwinski, and Horvitz 2001). We note that such costs can be inferred and accounted for formally in utility-theoretic systems (Horvitz and Apacible 2003; Horvitz, Jacobs, and Hovel 1999).

Ideally, Copilot should display suggestions when the suggestions provide net value to programmers. For example, consider the task of completing a function and the time taken to complete it as a proxy for the total effort. If the expected time required to verify and edit Copilot's suggestion exceeds the time to write the code by themselves (counterfactual cost), then Copilot should not show its suggestions. Conversely, if the expected time to write exceeds the time to verify and edit, it may be useful to display the suggestion. We now formalize this intuition with a utility-theoretic formulation and, in the next section, discuss the methodology to make it practical.

Programmer Model. At a given time instance time during a session, Copilot extracts a prompt P from the code file X and generates a code suggestion S. If this suggestion is shown, we assume the programmer spends an expected time $\mathbb{E}[\text{verification}|X, S, \phi]$ to verify it and accepts the suggestion with probability $\mathbb{P}(A = \text{accept}|X, S, \phi)$. Once a suggestion is accepted, the programmer may further edit the suggestion with expected time $\mathbb{E}[\text{editing}|X, S, \phi, A = \text{accept}]$ to achieve their task. On the other hand, if the programmer rejects the suggestion, they would have to spend time writing code that achieves their task, denoted by $\mathbb{E}[\text{writing}|X, S, A = \text{reject}]$. Thus, the total time incurred with showing a suggestion, denoted as $\mathbb{E}[\text{S shown}|X, S, \phi]$, is:

$$\begin{aligned}\mathbb{E}[\text{S shown}|X, \phi] &= \mathbb{E}[\text{verification}|X, S, \phi] \qquad (1)\\ &+ \mathbb{P}(A = \text{accept}|X, S, \phi) \cdot \mathbb{E}[\text{editing}|X, S, \phi, A = \text{accept}]\\ &+ \mathbb{P}(A = \text{reject}|X, S, \phi) \cdot \mathbb{E}[\text{writing}|X, S, \phi, A = \text{reject}]\end{aligned}$$

While editing and writing, Copilot may further make more suggestions; thus, the editing time and writing time should include interactions with future suggestions. Now, on the other hand, if the suggestion is *not* shown, the programmer will spend time $\mathbb{E}[\text{writing}|X]$ writing code for their task. We also need to factor in latency, the time cost τ to compute a suggestion once we decide to create a suggestion. Latency is only experienced by the programmer if their latent state ϕ includes expecting a suggestion and waiting for it. If the programmer is expecting a suggestion, we should add τ to the total time when we show a suggestion; otherwise, the programmer continues to write code not expecting a suggestion.

We now define suggestion utility, a value that indicates the change in programmers' coding time due to showing the suggestion.

Definition 1 (Suggestion Utility)**.** *The time impact δ, denoted as the* suggestion utility*, of showing S versus not* *showing is defined as:*

$$\delta = \underbrace{\mathbb{E}[\text{writing}|X, \phi]}_{\text{S not shown}} - \underbrace{\mathbb{E}[\text{S shown}|X, \phi]}_{\text{S shown}} - \underbrace{\mathbb{E}[\tau|X, \phi]}_{\text{latency}} \quad (2)$$

From the above, a suggestion S at a given time should **only be shown** if $\delta > 0$ (Equation 2), where the programmer will spend less time to achieve their task if it is shown. An optimal scheme to know when to show suggestions would be to generate suggestions as frequently as possible, compute their *suggestion utility* δ, and display them if $\delta > 0$.

Feasibility of Estimating δ. Per Equation (1), computing *suggestion utility* requires the computation of four quantities: (1) the expected time spent verifying a suggestion, (2) the expected time editing a suggestion, (3) the expected time to write a segment of code and (4) the probability of accepting a suggestion. One can attempt to build an estimator for (1), by predicting from the prompt and suggestion the time spent verifying a suggestion i which would be $t_{i+1} - t_i$ using standard regression estimators. Unfortunately, using the same features and the dataset detailed in our experimental section, our best estimator is only able to achieve an $R^2 = 0.13$, which is not much better than a naive median time estimate. This may be due to the high variance and unobserved confounders governing verification time. Estimating editing and verification time (quantities 2 and 3 above) is only more complex and challenging. Thus, we restrict our methodology to seeing when we can evaluate δ using only our estimator for the probability of acceptance (4).

Learning Programmer's Acceptance Decisions. The full conditional for the probability that the programmer accepts a suggestion is $\mathbb{P}(A = \text{accept}|X, S, \phi)$. Given the telemetry, we can only compute $\mathbb{P}(A = \text{accept}|X, S)$ where the programmer's latent state cannot be observed. Using standard calibrated classification methods, we can estimate the probability $\mathbb{P}(A = \text{accept}|X, S)$ by using the actions A_i as the labels. We show that a simple mechanism of thresholding the estimated probability that the programmer accepts a suggestion is equivalent under certain assumptions to checking if $\delta < 0$:

Proposition 1. *Under assumptions that the programmer spends more time writing code when they reject a suggestion compared to when they accept a suggestion and edit it, given specific code, suggestion, and latent state* (X, S, ϕ)*, if the programmer's probability of accepting* $\mathbb{P}(A = \textit{accept}|X, S, \phi)$ *a suggestion is below* $\mathbb{P}^*$*, which is defined as:*

$$\mathbb{P}^* = \frac{\mathbb{E}[\text{verification}] + \mathbb{E}[\text{latency}]}{\mathbb{E}[\text{writing}|A = \text{reject}] - \mathbb{E}[\text{editing}|A = \text{accept}]} \quad (3)$$

then the suggestion should not be shown. Note that $\mathbb{P}^*$ *is defined as a function* $\mathbb{P}^*(X, S, \phi)$ *evaluated pointwise.*

The formal statement and proof are available in the appendix. The above proposition shows that comparing the probability of acceptance to $\mathbb{P}^*$ can guide when to show the suggestion. We provide a graphical view of the analysis in Figure 3, in the spirit of related analyses on utility-guided interactive interfaces (Horvitz 1999). Practically, if we compare the probability of acceptance to a constant lower bound of $\mathbb{P}^*$, we can guarantee that we hide suggestions only when $\delta < 0$.

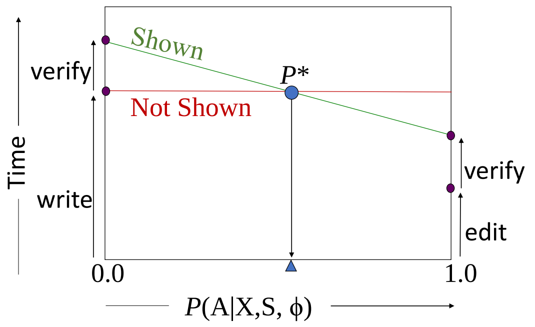


Figure 3: Graphical depiction of analysis of Proposition 1 when the latency is zero. The y-axis shows total time and the x-axis is the programmer's probability of accepting $\mathbb{P}(A = \text{accept}|X, S, \phi)$. At probability $\mathbb{P}^*$, showing and not showing the suggestion have equal time cost.

Effect of Programmer Latent State. As mentioned previously, the programmer's latent state is not available via telemetry. Thus, we can only provide predictions of $\mathbb{P}(A = \text{accept}|X, S)$ versus explicit consideration of the latent state, $\mathbb{P}(A = \text{accept}|X, S, \phi)$. In earlier work (Mozannar et al. 2022), we collected telemetry data of 21 programmers performing various tasks and had participants retrospectively label the telemetry with their latent state from a set of twelve unique states (1096 suggestions). We use this data to build predictive models with and without the latent state using the same methodology in the experiments section. Using a leave-one-out programmer evaluation strategy, the model without the latent state achieves accuracy 61.9 ± 1.9 while the model with the latent state achieves 83.6 ± 2.4, a statistically significant difference according to a paired t-test ($p = 6.9e-7, t = 7.11$); a similar result occurs when comparing areas under the receiver operating characteristic curve (AUC). These results highlights an opportunity to gather external data beyond telemetry to build such predictive models and indicates that acceptance may not simply be a property of suggestions and code context.

Conditional Suggestion Display From Human Feedback

In this section, we describe the CDHF method that can be implemented using telemetry data to identify when to show suggestions, as illustrated in Figure 1. We note from Equation (3) that the higher the probability of accepting the suggestion and the lower the latency to generate the suggestion, the more likely the suggestion is useful ($\delta > 0$). Our proposed approach is as follows: Each time the programmer pauses typing, we decide using a predictor whether to show a suggestion. Crucially, we do this using a two-stage scheme to avoid generating suggestions when we know the programmer would reject them.

Display Decision. Let $m(X, S)$ be a binary predictor that denotes whether, at a given moment in the code X, we should show the suggestion S; we call this the display decision. If $m(X, S) = 1$, we display the suggestion; otherwise, we do not. The most straightforward way to build such a function m is to estimate the programmer's probability of accepting the suggestion: $\mathbb{P}(A = \text{accept})|X, S)$ and then threshold the probability so that suggestions that fall below a probability t are hidden. However, this will lead us to generate suggestions including those that will never be shown, thus wasting computing resources. We propose to decompose the function m so that we first decide using only the code whether we can make the display decision without generating the suggestion S with a function $r(X)$. If $r(X) = 1$, we make the display decision using a stage 1 model $m_1(X)$ without generating the suggestion, otherwise if $r(X) = 0$ we generate the suggestion S and make the display decision with a stage 2 model $m_2(X, S)$ as follows:

$$m(X, S) = r(X) \cdot m_1(X) + (1 - r(X)) \cdot m_2(X, S) \quad (4)$$

This formulation allows us to avoid generating suggestions when we can make an accurate display decision in advance of knowing the suggestion. For example, in a setting where the programmer has rejected the last 30 suggestions, they are unlikely to accept the next suggestion.

Objective and Guarantees. Our objective in learning the functions r, m_1, m_2 is to (1) hide as many suggestions that would have been rejected and (2) maximize the number of display decisions made without generating the suggestion to reduce latency on the system. There is an inherent trade-off between these two objectives as making decisions with access to the suggestions would be more accurate. Moreover, we want to make sure we do not hide suggestions that would have been accepted, as this would limit the usefulness of the code assistant. Therefore, we impose a constraint that, whenever we hide a suggestion, there is at least a probability p it would have been rejected, a constraint on the true negative rate (TNR). We translate the objectives and the constraint into the following optimization problem:

$$\max_{r, m_1, m_2} \lambda \mathbb{E}[1 - m(X, S)] + (1 - \lambda)\mathbb{E}[r(x)] \quad (5)$$

$$s.t. \;\; \mathbb{P}(A = \text{reject}|m(X, S) = 0) \geq p \quad (6)$$

Parameterization. We can control the trade-off between the two objectives with a hyperparameter $\lambda \in [0, 1]$. Equivalently, instead of controlling the trade-off with λ, we can set a constraint on $\mathbb{E}[r(x)] := R$ and set $\lambda = 1$. We propose an intuitive post-hoc procedure to solve the optimization problem (5): We first learn calibrated estimators of the probability of accepting suggestions $\hat{\mathbb{P}}(A = \text{accept}|X, S)$ (with suggestion) and $\hat{\mathbb{P}}(A = \text{accept}|X)$ (without suggestion). We then parameterize:

$$m_1(X) = \mathbb{I}_{\hat{\mathbb{P}}(A=\text{accept}|X) \geq t_1}, m_2(X) = \mathbb{I}_{\hat{\mathbb{P}}(A=\text{accept}|X,S) \geq t_2}$$

and $r(X) = \mathbb{I}_{H(\hat{\mathbb{P}}(A=\text{accept}|X)) \leq t_r}$ ($H(.)$ is Shannon's Entropy), and optimize *jointly* over the tuple of thresholds t_1, t_2, t_r over $[0, 1]^3$. This is a fairly efficient procedure that can achieve good results. We note that this procedure saves latency indirectly by reducing the number of LLM calls across the session and across different users, and that we should still enable the user to see the suggestion with a special keyboard shortcut to override the display decisions. In the next section, we perform a retrospective evaluation of CDHF.

Experiments

Our main aim with experiments is to understand how well the CDHF procedure can make display decisions in a retrospective evaluation. Code is available[2] and additional details can be found in the appendix.

Dataset and Feature Engineering.

Dataset. To build and evaluate our methods, we extract a large number of telemetry logs from Copilot users (mostly software engineers and researchers) at Microsoft. Programmers provided consent for the use of their data, and its use was approved by Microsoft's ethics advisory board. Specifically, for a two-week time period, we extracted all the telemetry events for 535 users who coded in Python. This totals 4,749 coding sessions, where a session is defined as a continuous sequence of user actions with at most 30 minutes between consecutive events. These sessions are from real-world usage of Copilot for daily tasks of the software engineers and researchers, the data was collected prior to the inception of our work. On average, each user contributes nine sessions, with each session lasting 21 minutes (median, 12 minutes). Sessions contain an average of 97 events (show, accept, and reject). This totals to almost 1,675 hours of coding with 168,807 shown events and 33,523 accept events, yielding an acceptance rate of 21.4% (not meant to represent Copilot's average acceptance rate).

Model Features. The telemetry dataset D described above contains for each user a list of events in each of their coding sessions; we denote $D_{i,j}$ to be the list of events for the j'th session of the i'th user. The dataset $\mathbf{D} = \{D_{i,j}\}$ contains for each user i and session j, a list of events occurring in the corresponding coding session. We extract only the accept and reject events, as well as prompt and session features of the corresponding shown events. For each prompt and suggestion pair, we extract: the programmer id as one hot vector, the length of the document, the previous five actions, suggestion features (e.g., suggestion length), previous features of the last five suggestions shown, the confidence reported by Copilot, an embedding using codeBERTa (Feng et al. 2020) of prompt and suggestion, presence of Python keywords (e.g., `import`, `def try`, etc.), and the output of the Tree-sitter Parser (Kalliamvakou 2023). Finally, we extract features of the prompt, including its embedding, textual features, and parser outputs. Figure 4 summarizes the feature engineering. It is crucial to note that the features do not leak any information about future events and can be computed as soon as a suggestion is generated by Copilot. For the first stage model (m_1) in CDHF, suggestion features are omitted while we include all features for the second stage model (m_2). This feature engineering incorporates past actions and

[2]https://github.com/microsoft/coderec_programming_states

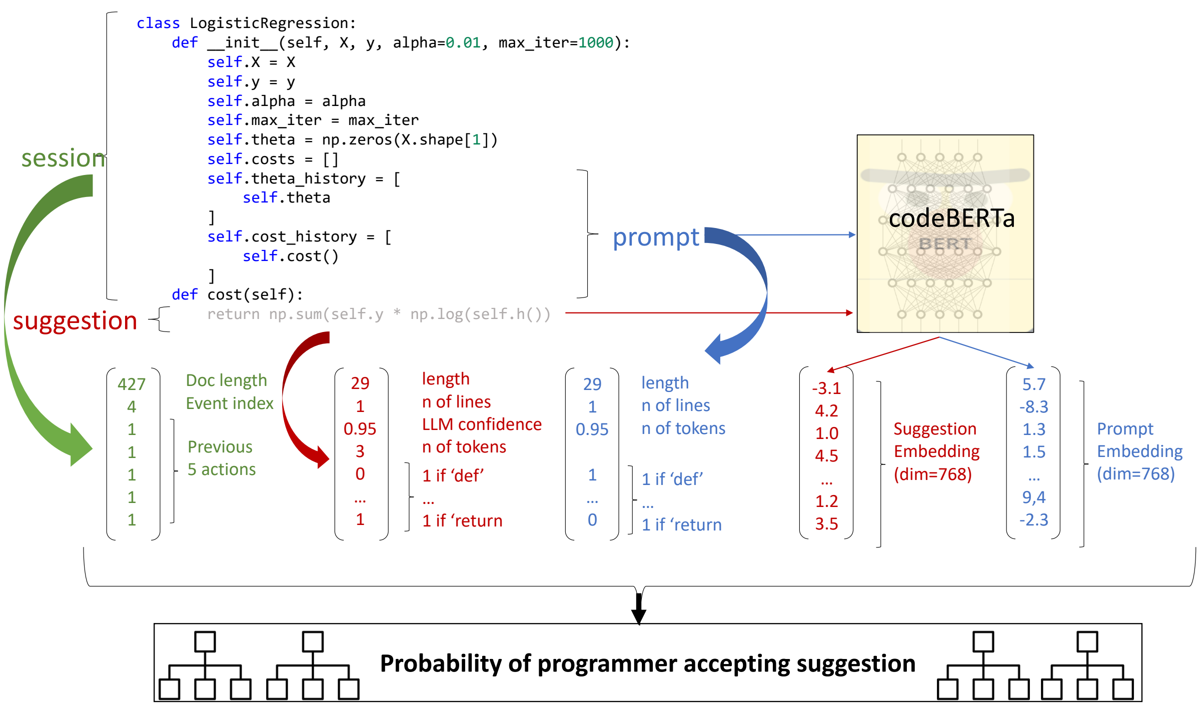

```
class LogisticRegression:
    def __init__(self, X, y, alpha=0.01, max_iter=1000):
        self.X = X
        self.y = y
        self.alpha = alpha
        self.max_iter = max_iter
        self.theta = np.zeros(X.shape[1])
        self.costs = []
        self.theta_history = [
            self.theta
        ]
        self.cost_history = [
            self.cost()
        ]
    def cost(self):
        return np.sum(self.y * np.log(self.h())
```

Figure 4: Features used to build action prediction model in Experiments , including from the suggestion, prompt, and session.

suggestions that the programmer has seen and allows us to use regular ML algorithms instead of time-series methods.

Model Evaluation

Before we evaluate CDHF, we perform an evaluation of the programmer acceptance model $m_2(X, S)$. We split the telemetry dataset in a 70:10:20 split for training, validation, and testing respectively. Importantly, we do this split in two ways: (1) by randomly splitting over programmers so that no single programmer is shared across the three splits and, (2) by randomly splitting over sessions so that users in training can also be seen in testing to allow for personalization.

Results. We evaluate different standard machine learning models on this task and find that the best-performing model is eXtreme Gradient Boosting (XGB) (Chen et al. 2015). When we split across users, XGB is able to achieve 81.1% (95% CI 80.7-81.6) accuracy and, more importantly, 0.780 (95% CI 0.775-0.786) AUC. In the appendix, we show metrics for different models evaluated, including deep networks (Mozannar et al. 2023). The results indicate that the model is able to distinguish between suggestions that are likely to be accepted versus those likely to be rejected. The model is also well calibrated: the expected calibration error is 0.10 (Naeini, Cooper, and Hauskrecht 2015).

We note a significant increase in AUC when we allow for personalization: including programmer ID as a feature and splitting across sessions, this leads to an AUC of 0.795 (95% CI 0.789-0.801), a significant increase (basis of m_2 model). When we remove suggestion features from the model, the resulting model (basis of m_1 model) achieves an AUC of 0.631 (95% CI 0.624-0.638). The time to compute the features needed for the models and performing inference on a single data point can take 10ms with a GPU and less than 1ms on a CPU when omitting embeddings, in addition to latency of sending and receiving information between server and client. In the appendix, we show results for different ablation of model features, sample complexity plots, and feature importance plots.

Retrospective Evaluation of CDHF

We train the models m_1 and m_2 using the training set per the previous subsection. We set the thresholds t_1, t_2, t_r on the validation set for CDHF and evaluate on the test set.

Results. In Figure 5, we vary the desired TNR rate (accuracy when a suggestion is hidden) and plot how many suggestions we can hide from those previously displayed while guaranteeing the desired TNR rate. We show the behavior of the CDHF method with different λ values, or, equivalently, with different constraints on how often the m_1 model (first stage) is used: $R := \mathbb{E}[r(x)]$. To illustrate what CDHF can accomplish, we can hide 25.3% of suggestions that were shown while guaranteeing that 94.7% of them would have been rejected and avoid generating 12.9% of the suggestions. If we have no concerns for latency, we can hide 52.9% of suggestions while guaranteeing that 91.3% of them would have been rejected. Figure 5 shows how we can achieve different trade-offs by selecting an operating point on any given curve. CDHF is able to satisfy the constraint of FNR on the test set with a violation of at most 0.3% i.e., a guarantee of 95% FNR on the validation set equates to 94.7-95.3% on the test set.

Counterfactual Increase in Acceptance Rate. On the

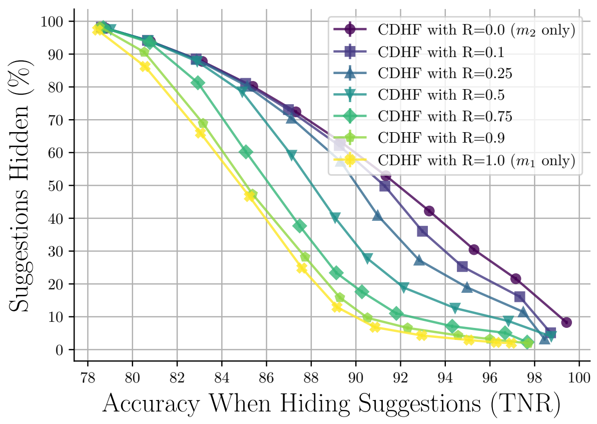


Figure 5: Evaluation of CDHF for selectively hiding suggestions. For a given constraint on FNR (accuracy when a suggestion is hidden) on the x-axis, we show on the y-axis the fraction of the total suggestions we can hide while guaranteeing the desired FNR. We plot these curves while varying how often the decision is made generating suggestions (R:=$\mathbb{E}[r(x)]$, when R=0, we generate the suggestion then decide to hide or not, when R=1, we decide to hide without knowing the suggestion).

test set, the acceptance rate of suggestions is 22.5%. Retrospectively, if we had used CDHF to hide 52.9% of suggestions, we could compute a counterfactual acceptance rate. The counterfactual acceptance rate can be computed as: $\frac{\text{S accepted}\cdot(1-\%\text{ hidden}\cdot(1-TNR))}{\text{S shown}\cdot\%\text{ not hidden}} = \frac{22.5(1-0.529\cdot 0.087)}{0.471} = 45.6\%$, which is a 23.1 point increase, a value we expect to be an overestimate.

Discussion and Limitations. The retrospective evaluation shows that CDHF has promise in reducing developer time spent verifying suggestions or waiting for suggestions. We note that our evaluation is retrospective. Although GitHub has shown that conditional suggestion filters similar to CDHF increase the acceptance rate of suggestions, a user study is required to verify whether the method makes programmers more productive. As Goodhart's law states, once a metric becomes a target, it ceases to become a good measure; acceptance rate is no exception. Moreover, if CDHF is not trained with sufficient data that captures the programmer's use cases, it can make the programming experience worse by hiding useful suggestions. Moreover, a rejected suggestion may still be useful, which we do not account for here. Finally, the optimization problem in (5) is amenable to procedures inspired by learning to defer (Mozannar and Sontag 2020) that can outperform the post-hoc procedure proposed.

Which Suggestion to Show?

We focus in this study on the problem of when to display suggestions. We did not tackle the question of which suggestions to display among a candidate set. Given access to telemetry data, which consists of contextualized suggestions with accept and reject signals, one can interpret an accept as the act of preferring a suggestion over no suggestion. It is reasonable to harness the telemetry data as a preference dataset and build a reward model of programmers' preferences, which would be equivalent to estimating the programmer's acceptance probability $\hat{\mathbb{P}}(A = \text{accept}|X, S)$. Thus, a reasonable procedure is to take a candidate set of suggestions $\mathcal{S}$ and display the suggestion that maximizes the probability of acceptance across the set; this is essentially the best-of-n baseline approach in RLHF (Rafailov et al. 2023).

Potential Bias Towards Short Suggestions. We hypothesize that such a ranking scheme would not be productive and can lead to poor suggestions of short length. Our rationale is the following: suppose the LLM is able to generate a multi-line suggestion S for a user query that approximately matches what the user desires. To maximize the probability that the user accepts the suggestion, it would be advantageous to split the suggestion S line-by-line and display it to the user step-by-step. The reasoning is that it is more likely for the first line of S to be correct rather than all of S being correct, hence being more likely to be accepted.

Experiment. To test this hypothesis, we perform the following experiment: We learn a model m of suggestion acceptance given only the prompt and suggestion embeddings with no session features on the telemetry data from the previous section. We then leverage the HumanEval dataset (Chen et al. 2021), which consists of 164 Python problems, each with an associated docstring and a ground truth function body solution. Solutions have at least two lines and seven median lines of code. Given the model m and each problem, we let the prompt be the concatenation of the docstring and the first k lines of the solution and let the candidate set of suggestions $\mathcal{S}$ be as follows: Given the solution S represented as an array of tokens of length N, we let $\mathcal{S} = \{S[:i]\}_{i=1}^{N}$. For example, if the solution S was "return np.mean(x)", then $\mathcal{S} = \{$"return", "return np.mean(x)"$\}$.

Results. We vary the parameter k in the set $\{0, 1, 2, 3\}$ so that the prompt goes from the docstring to include lines of the solution. When $k = 0$, the normalized length of the highest-rated suggestion, according to the model across the 164 problems, is almost uniform across $[0, 1]$, a Kolmogorov–Smirnov test compared to the uniform distribution has a p-value of 0.53 (KS=0.06). Optimally, we want the normalized length to cluster around 1 to include the full solution. However, when $k > 0$, meaning that the prompt includes lines of code, we find that for over 60 of the 164 problems, the highest-scored suggestion lies in $[0, 0.2]$, and, for at least 40 problems, it is the first token. This provides some evidence that optimizing for acceptance can be biased toward shorter suggestions since the highest-ranked suggestion is often in the first few lines of the solution.

Limitations. However, there are important limitations in our experiment. First, the model m is only trained on Copilot suggestions. Thus, the bias towards short suggestions can be due in part to Copilot potentially showing short suggestions. Maximizing acceptance would not alleviate such a bias. Second, while the use of embeddings of the suggestions for the reward model led to accurate predictions of accepts (AUC=0.701), they might be biased in some ways as compared to alliance on fine-tuning the language model.

Acknowledgments

Hussein Mozannar partly conducted this work during an internship at Microsoft Research (MSR). We acknowledge valuable feedback from colleagues across MSR and GitHub including Saleema Amershi, Victor Dibia, Forough Poursabzi, Andrew Rice, Eirini Kalliamvakou, and Edward Aftandilian.

References

Amazon. 2022. ML-powered coding companion – Amazon CodeWhisperer.

Bai, Y.; Jones, A.; Ndousse, K.; Askell, A.; Chen, A.; DasSarma, N.; Drain, D.; Fort, S.; Ganguli, D.; Henighan, T.; et al. 2022. Training a helpful and harmless assistant with reinforcement learning from human feedback. *arXiv preprint arXiv:2204.05862*.

Bailey, B. P.; Konstan, J. A.; and Carlis, J. V. 2001. The Effects of Interruptions on Task Performance, Annoyance, and Anxiety in the User Interface. In *Interact*, volume 1, 593–601.

Bansal, G.; Nushi, B.; Kamar, E.; Lasecki, W. S.; Weld, D. S.; and Horvitz, E. 2019. Beyond accuracy: The role of mental models in human-AI team performance. In *Proceedings of HCOMP*, volume 7, 2–11.

Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; Pinto, H. P. d. O.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Chen, T.; He, T.; Benesty, M.; Khotilovich, V.; Tang, Y.; Cho, H.; Chen, K.; et al. 2015. Xgboost: extreme gradient boosting. *R package version 0.4-2*, 1(4): 1–4.

Csikszentmihalyi, M.; and Larson, R. 2014. *Flow and the foundations of positive psychology*, volume 10. Springer.

Cutrell, E.; Czerwinski, M.; and Horvitz, E. 2001. Notification, Disruption, and Memory: Effects of Messaging Interruptions on Memory and Performance. In *IFIP TC13 International Conference on Human-Computer Interaction*.

Dakhel, A. M.; Majdinasab, V.; Nikanjam, A.; Khomh, F.; Desmarais, M. C.; Ming, Z.; et al. 2022. GitHub Copilot AI pair programmer: Asset or Liability? *arXiv preprint arXiv:2206.15331*.

Evtikhiev, M.; Bogomolov, E.; Sokolov, Y.; and Bryksin, T. 2022. Out of the BLEU: how should we assess quality of the Code Generation models? *arXiv preprint arXiv:2208.03133*.

Feng, Z.; Guo, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, T.; Jiang, D.; et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*.

Github. 2022. GitHub copilot - your AI pair programmer.

Horvitz, E. 1999. Principles of mixed-initiative user interfaces. In *Proceedings of CHI*, 159–166.

Horvitz, E.; and Apacible, J. 2003. Learning and Reasoning about Interruption. In *Proceedings of the 5th International Conference on Multimodal Interfaces*, ICMI '03, 20–27.

Horvitz, E.; Jacobs, A.; and Hovel, D. 1999. Attention-Sensitive Alerting. In *Proceedings of UAI*, 305–313.

Kalliamvakou, E. 2022. Research: Quantifying github copilot's impact on developer productivity and happiness.

Kalliamvakou, E. 2023. Tree-sitter parser generator tool.

Knox, W. B.; and Stone, P. 2008. Tamer: Training an agent manually via evaluative reinforcement. In *2008 7th IEEE international conference on development and learning*, 292–297. IEEE.

Li, Y.; Choi, D.; Chung, J.; Kushman, N.; Schrittwieser, J.; Leblond, R.; Eccles, T.; Keeling, J.; Gimeno, F.; Lago, A. D.; et al. 2022. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814*.

MacGlashan, J.; Ho, M. K.; Loftin, R.; Peng, B.; Wang, G.; Roberts, D. L.; Taylor, M. E.; and Littman, M. L. 2017. Interactive learning from policy-dependent human feedback. In *ICML*, 2285–2294. PMLR.

Mozannar, H.; Bansal, G.; Fourney, A.; and Horvitz, E. 2022. Reading Between the Lines: Modeling User Behavior and Costs in AI-Assisted Programming. *arXiv preprint arXiv:2210.14306*.

Mozannar, H.; Bansal, G.; Fourney, A.; and Horvitz, E. 2023. When to Show a Suggestion? Integrating Human Feedback in AI-Assisted Programming. *arXiv preprint arXiv:2306.04930*.

Mozannar, H.; Satyanarayan, A.; and Sontag, D. 2022. Teaching humans when to defer to a classifier via exemplars. In *AAAI*, volume 36, 5323–5331.

Mozannar, H.; and Sontag, D. 2020. Consistent estimators for learning to defer to an expert. In *ICML*, 7076–7087. PMLR.

Naeini, M. P.; Cooper, G.; and Hauskrecht, M. 2015. Obtaining well calibrated probabilities using bayesian binning. In *Proceedings of AAAI*.

OpenAI. 2022. ChatGPT: Optimizing Language Models for Dialogue.

Radford, A.; Wu, J.; Child, R.; Luan, D.; Amodei, D.; Sutskever, I.; et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8): 9.

Rafailov, R.; Sharma, A.; Mitchell, E.; Ermon, S.; Manning, C. D.; and Finn, C. 2023. Direct preference optimization: Your language model is secretly a reward model. *arXiv preprint arXiv:2305.18290*.

Sarkar, A.; Gordon, A. D.; Negreanu, C.; Poelitz, C.; Ragavan, S. S.; and Zorn, B. 2022. What is it like to program with artificial intelligence? *arXiv preprint arXiv:2208.06213*.

Sun, Z.; Du, X.; Song, F.; Wang, S.; Ni, M.; and Li, L. 2022. Learning to Prevent Profitless Neural Code Completion. *arXiv preprint arXiv:2209.05948*.

Zhao, S. 2023. GitHub Copilot now has a better AI model and new capabilities. https://github.blog/2023-02-14-github-copilot-now-has-a-better-ai-model-and-new-capabilities/.

Ziegler, A.; Kalliamvakou, E.; Li, X. A.; Rice, A.; Rifkin, D.; Simister, S.; Sittampalam, G.; and Aftandilian, E. 2022. Productivity assessment of neural code completion. In *Proceedings of SIGPLAN*, 21–29.

Ziegler, D. M.; Stiennon, N.; Wu, J.; Brown, T. B.; Radford, A.; Amodei, D.; Christiano, P.; and Irving, G. 2019. Fine-tuning language models from human preferences. *arXiv preprint arXiv:1909.08593*.