R-U-SURE? Uncertainty-Aware Code Suggestions By Maximizing Utility Across Random User Intents

Daniel D. Johnson [1 2] **Daniel Tarlow** [1] **Christian Walder** [1]

Abstract

Large language models show impressive results at predicting structured text such as code, but also commonly introduce errors and hallucinations in their output. When used to assist software developers, these models may make mistakes that users must go back and fix, or worse, introduce subtle bugs that users may miss entirely. We propose *Randomized Utility-driven Synthesis of Uncertain REgions (R-U-SURE)*, an approach for building uncertainty-aware suggestions based on a decision-theoretic model of goal-conditioned utility, using random samples from a generative model as a proxy for the unobserved possible intents of the end user. Our technique combines minimum-Bayes-risk decoding, dual decomposition, and decision diagrams in order to efficiently produce structured uncertainty summaries, given only sample access to an arbitrary generative model of code and an optional AST parser. We demonstrate R-U-SURE on three developer-assistance tasks, and show that it can be applied different user interaction patterns without retraining the model and leads to more accurate uncertainty estimates than token-probability baselines. We also release our implementation as an open-source library at https://github.com/google-research/r_u_sure.

```
import collections
import json
from typing import List

TokenWithConfidence = collections.namedtuple('TokenWithConfidence',
    ('token', 'confidence'))

def render_suggestion(suggestion: List[TokenWithConfidence]) -> str:
  # (... definition omitted ...)

def suggestion_to_json(suggestion: List[TokenWithConfidence]) -> str:
  """Converts a suggestion to JSON.

  Args:
    suggestion: List of tokens and their confidence.

  Returns:
    JSON representation of the suggestion.
  """
  return json.dumps(suggestion)
```

```
import collections
import json
from typing import List

TokenWithConfidence = collections.namedtuple(
    'TokenWithConfidence', ('token', 'confidence'))

def render_suggestion(suggestion:
    List[TokenWithConfidence]) -> str:
  # (... definition omitted ...)

def suggestion_to_json(suggestion:
    List[TokenWithConfidence]) -> str:
  """Converts a suggestion to JSON.

  Args:
    suggestion: List of tokens and their
      confidence.

  Returns:
    JSON representation of the suggestion.
  """
  return json.dumps(<STOP>
```

```
"""Visualizing trends using SQLite and Matplotlib
"""
import sqlite3
from matplotlib import pyplot as plt

# Open the database
conn = sqlite3.connect('data/budget.db')
c = conn.cursor()

# Get the data
c.execute('''SELECT date, SUM(budget)
             FROM transactions
             GROUP BY date''')
data = c.fetchall()

# Close the database
conn.close()

# Create the plot
fig, ax = plt.subplots()
ax.plot(data)

# Show it
plt.show()
```

Figure 1. Given a partial file as context (bolded black code) and outputs from a fixed language model (blue code), our approach can be used to predict parts of generated programs that may need editing (top, orange background), adjust completion length to avoid uncertain parts (left, red text), or identify the most relevant statements from a larger prediction (right, green background), by searching over a space of uncertainty-augmented suggestions $\mathcal{S}$. (Examples lightly reformatted to fit this figure; see Appendix A.)

1. Introduction

Large language models have demonstrated remarkable abilities for generating both natural language (Brown et al., 2020; Chowdhery et al., 2022) and source code (Svyatkovskiy et al., 2020; Feng et al., 2020; Chen et al., 2021b; Li et al., 2022; Nijkamp et al., 2022). These abilities have led them to be incorporated into a number of developer assistance tools and services, such as GitHub Copilot and Tabnine.

Unfortunately, when faced with novel or unpredictable situations, large language models have a tendency to guess or "hallucinate" unwanted outputs (Maynez et al., 2020; Liu et al., 2021). In the context of software development, these guesses can slow development by requiring developers to spend time verifying the suggestion and deleting any incorrect parts (Mozannar et al., 2022; Barke et al., 2022; Upadhyaya et al., 2022), or worse, lead to undetected problems and less secure code (Pearce et al., 2021). Compounding this issue is the presence of *automation bias*, an effect where

[1]Google Research, Brain Team [2]University of Toronto, Department of Computer Science. Correspondence to: Daniel D. Johnson <ddjohnson@google.com>.

Proceedings of the 40th International Conference on Machine Learning, Honolulu, Hawaii, USA. PMLR 202, 2023. Copyright 2023 by the author(s).

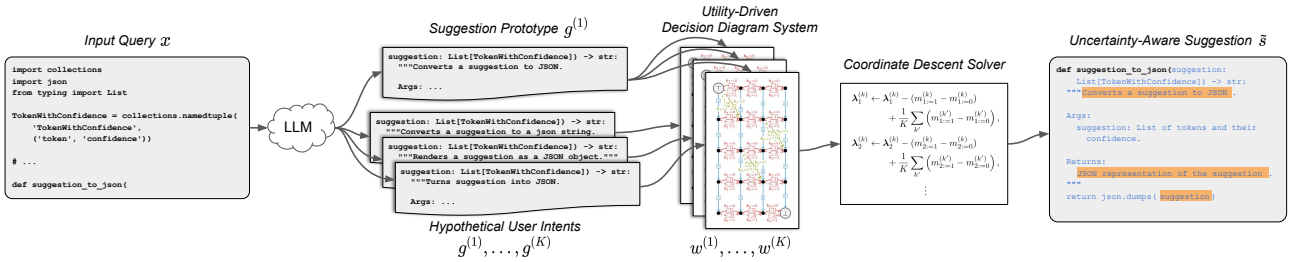


Figure 2. Overview of the R-U-SURE system. Given an input context x, we generate a set of samples $g^{(1)}, \ldots, g^{(K)}$ from a pretrained generative model. We select one sample $g^{(1)}$ as a *suggestion prototype* and interpret the full set of samples as hypothetical user intents (e.g. examples of code the user may want to write), as described in Section 3.1. For each hypothetical intent $g^{(k)}$, we can estimate the utility $u(g^{(k)}, s)$ of showing some concrete suggestion s to a user who intended to write $g^{(k)}$ (discussed in Section 2). We represent this efficiently using a system of decision diagrams $w^{(1)}, \ldots, w^{(k)}$ (Section 3.3), and use coordinate descent (Section 3.2) to identify a set of annotations to insert into the suggestion prototype $g^{(1)}$ to maximize expected utility over these samples, obtaining an uncertainty-aware suggestion $\tilde{s}$ that is likely to be more useful for the user's unobserved true intent.

users fail to notice issues in outputs of automated systems (Madi, 2022; Cummings, 2004; Lyell & Coiera, 2017).

An interesting property of generated code suggestions is that some parts of the user's intent (e.g. control flow and API boilerplate) can be predicted more easily than others (e.g. edge-case behavior or the signatures of novel functions). User interaction studies of ML coding assistants have revealed that software engineers would benefit if suggestions included indicators of model uncertainty (Mozannar et al., 2022; Weisz et al., 2021) or user-fillable "holes" (Barke et al., 2022). However, Vasconcelos et al. (2022) have found that per-token conditional probability estimates are insufficient to provide good predictions of necessary edits. Guo et al. (2021) propose a top-down generative model of code that uses a programming language's grammar to generate completions containing holes, but these holes must align with grammar nonterminals and cannot identify uncertain subregions within lists of statements or expressions.

In this work, we show that there is a way to harness the remarkable capabilities of pretrained language models to both generate high-quality code suggestions and also produce concise and semantically-meaningful representations of their own uncertainty, without requiring any fine-tuning. Our key insight is that, since language models of code are trained to predict file contents from context, we can reinterpret the samples from a well-trained language model as *plausible goal states* for the user. We can then use these samples to estimate how useful our suggestions would be for a user whose intent we do not know, and to *modify* those suggestions to make them useful across a diverse set of possible user intents.

As a concrete motivating example, consider the task of code completion under uncertainty, and suppose we wish to highlight specific regions of a completion suggestion to help end-users identify the parts of the suggestion they need to change, as shown at the top of Figure 1. To do this, we can define a space $\mathcal{S}$ of annotated suggestions, where some tokens are highlighted as UNSURE. We can then approximate how helpful a suggestion $s \in \mathcal{S}$ would be to a user who actually wants to write code g by computing a confidence-adjusted edit distance between s and g, assuming that UNSURE tokens will be double-checked by the user and thus be easier to edit if wrong but also save less time (and thus be less useful) than non-highlighted SURE tokens if they turn out to be correct. If we can find a set of annotations that has high utility across many hypothetical goals $g^{(k)}$ drawn from a language model, then as long as the language model is well calibrated, the UNSURE annotations should provide a summary of the model's uncertainty that is similarly useful for accomplishing the user's unknown goal g.

Our contributions are as follows:

- We describe a utility-driven framework (summarized in Figure 2) for producing uncertainty-aware suggestions given only sample-access to an arbitrary language model, by interpreting its samples as plausible user intents and using combinatorial optimization to identify the highest-utility suggestion, extending sample-based minimum Bayes risk decoding (Eikema & Aziz, 2020).
- We show how to apply dual decomposition to a novel decision diagram representation of edit-distance-based utility functions, yielding an efficient coordinate-descent optimizer and building a bridge between recent advances in language model decoding and combinatorial optimization.
- We construct a number of variants for our utility functions, enabling them to incorporate tree structure from an error-tolerant syntax-tree parser, account for both deletions and insertions, and respond to uncertainty by either annotating or removing the uncertain parts.

- We demonstrate our approach across three developer-assistance-inspired tasks (visualized in Figure 1), and show that our approach yields a better tradeoff between correct and incorrect predictions than heuristics based on cumulative or per-token probability.

2. Problem Statement

We tackle the problem of providing *contextual, uncertainty-aware suggestions* to assist users of ML-integrated tools with unobserved goals, with a particular focus on assisting software development. As we discuss in Section 1, there may not be enough information to fully determine the user's intent given the context. Our strategy is thus to augment the space of possible suggestions to account for the uncertainty in the user's intent in an explicit way. For instance, we can insert visual markers into a code-completion suggestion to draw attention to the parts of the suggestion that the user may wish to change. By doing so, we can avoid silently introducing incorrect behavior, and produce a suggestion that is useful regardless of what intent the user actually has.

We formalize this intuition using a decision theoretic framework. We let $\mathcal{X}$ be a set of contexts (e.g. the current partially-written code file and any other relevant IDE state) and $\mathcal{G}$ be a set of goals (e.g. the desired final state of the code file), with the specific context $x \in \mathcal{X}$ and goal $g \in \mathcal{G}$ of each user being distributed according to some unknown distribution $p(X = x, G = g)$. We further specify a set $\mathcal{S}$ of possible system suggestions along with a *utility metric* $u : \mathcal{G} \times \mathcal{S} \to \mathbb{R}$, where $u(g, s)$ measures (or approximates) *how useful* suggestion s is toward accomplishing some specific goal g that the user might have.

Consider again the motivating example introduced in Section 1. Letting Σ be a set of tokens, we can define $\mathcal{G} = \Sigma^*$ as the set of possible token sequences $g = [g_1, g_2, \ldots, g_M] \in \mathcal{G}$ the user may wish to write, with each $g_i \in \Sigma$. We can then define $\mathcal{S} = (\Sigma \times \mathcal{C})^*$ as a set containing uncertainty-annotated suggestions $s = [(s_1, c_1), (s_2, c_2), \ldots, (s_N, c_N)] \in \mathcal{S}$, where each suggestion is a sequence of pairs of tokens $s_i \in \Sigma$ and confidence indicators $c_i \in \mathcal{C} = \{\text{SURE}, \text{UNSURE}\}$. Finally, we can define $u(g, s)$ based on the edit distance from s to g, with a smaller penalty for deleting incorrect UNSURE tokens but a smaller reward for keeping correct ones. An example of how such a u might be implemented using dynamic programming is shown in Algorithm 1 of Figure 3, where α is the reward for keeping correct UNSURE tokens and β is the penalty for deleting incorrect UNSURE tokens. In Section 3.4 we discuss how we extend this idea to account for program syntax trees and inserted code.

More generally, we can think about each $s \in S$ as a possible suggestion our system could show, and use u to estimate the usefulness of that suggestion for a particular goal. For a given context $x \sim p(X)$, we wish to find a concrete suggestion s^* which is as useful as possible, e.g. that maximizes $u(g, s^*)$, in the presence of uncertainty about g. If we knew the true distribution $p(G|X)$, we might seek the suggestion that is most useful on average over the user's likely intents:

$$s^* = \arg\max_{s \in \mathcal{S}} \mathbb{E}_{g \sim p(G|X=x)}[u(g, s)] \tag{1}$$

This choice is also known as the *minimum Bayes risk* suggestion, as it minimizes the expected risk (negative utility) of the action under the conditional distribution $p(G|X)$.

3. Approach

Unfortunately, we do not have access to the distribution in Equation (1). We now present Randomized Utility-driven Synthesis of Uncertain REgions (R-U-SURE), a tractable procedure for approximating s^* by combining samples from a model using combinatorial optimization.

3.1. Approximating True Intents With Model Samples

We start by assuming that we have access to a well-calibrated generative model $\tilde{p}_\theta(G|X)$ that predicts a distribution of plausible goals in a given context. For instance, $\tilde{p}_\theta(G|X)$ could be a language model trained to produce completions of a partial file. Previous work has shown that samples from such a model can give a good proxy for true uncertainty in a generative model as long as the model is well calibrated (Eikema & Aziz, 2020; Ott et al., 2018).

As such, we can treat the model $\tilde{p}_\theta(G|X)$ as a proxy for the true conditional distribution $p(G|X)$, and try to find

$$\tilde{s}^* = \arg\max_{s \in \mathcal{S}} \mathbb{E}_{g \sim \tilde{p}_\theta(G|X=x)}[u(g, s)]. \tag{2}$$

Intuitively, if we find a suggestion $\tilde{s}^*$ that is reliably useful across the high-likelihood goals under $\tilde{p}_\theta(G|X = x)$, and any sample from p should also have high likelihood under $\tilde{p}_\theta$ (e.g. due to training with the cross-entropy objective), we can hope that such a suggestion is also useful for the true user intent (a sample from $p(G|X = x)$).

It is still intractable to exactly find $\tilde{s}^*$, due to the exponentially large set of possible user intents and possible suggestions. We thus search over a restricted space $\mathcal{S}(g^{(1)}) \subset \mathcal{S}$ derived from one of the model outputs $g^{(1)}$ (which we call the *suggestion prototype*), and use a Monte-Carlo estimate over K independent samples $g^{(k)} \sim \tilde{p}_\theta(G|X = x)$ from the model to estimate utility, similar to the minimum-Bayes-risk decoding strategy proposed by Eikema & Aziz (2020):

$$\tilde{s} = \arg\max_{s \in \mathcal{S}(g^{(1)})} \frac{1}{K} \sum_{k=1}^{K} u(g^{(k)}, s), \tag{3}$$

Algorithm 1 Sequence edit-distance utility $u(g, s)$

Input: $g = [g_1, \ldots, g_M], s = [(s_1, c_1), \ldots, (s_N, c_N)], \alpha, \beta$
Initialize T_j^i to $-\infty$ for $0 \leq i \leq M, 0 \leq j \leq N$
$T_0^0 \leftarrow 0.0$ *# Base case for dynamic programming*
for $i = 0$ **to** M, $j = 0$ **to** N **do**
  if $i > 0$ **and** $j > 0$ **and** $g_i = s_j$ **then** *# Match s_j and g_i*
    if $c_j =$ SURE **then** $T_j^i \leftarrow \max(T_j^i, T_{j-1}^{i-1} + 1)$
    if $c_j =$ UNSURE **then** $T_j^i \leftarrow \max(T_j^i, T_{j-1}^{i-1} + \alpha)$
  end if
  if $i > 0$ **then** *# Insert g_i*
    $T_j^i \leftarrow \max(T_j^i, T_j^{i-1} + 0)$
  end if
  if $j > 0$ **then** *# Delete s_j*
    if $c_j =$ SURE **then** $T_j^i \leftarrow \max(T_j^i, T_{j-1}^i - 1)$
    if $c_j =$ UNSURE **then** $T_j^i \leftarrow \max(T_j^i, T_{j-1}^i - \beta)$
  end if
end for
return T_N^M

Algorithm 2 Decision diagram for $w^{(k)}(\boldsymbol{b})$

Input: $g^{(k)} = [g_1, g_2, \ldots, g_M], g^{(1)} = [s_1, s_2, \ldots, s_N], \alpha, \beta$
Initialize an empty decision diagram
Label node $(0, 0)$ as $\top$
for $i = 0$ **to** M, $j = 0$ **to** N **do**
  if $i > 0$ **and** $j > 0$ **and** $g_i = s_j$ **then** *# Match s_j and g_i*
    Add edge $(i-1, j-1) \to (i, j)$, weight 1, label $\boldsymbol{b}_j := 0$
    Add edge $(i-1, j-1) \to (i, j)$, weight α, label $\boldsymbol{b}_j := 1$
  end if
  if $i > 0$ **then** *# Insert g_i*
    Add edge $(i-1, j) \to (i, j)$, weight 0
  end if
  if $j > 0$ **then** *# Delete s_j*
    Add edge $(i, j-1) \to (i, j)$, weight -1, label $\boldsymbol{b}_j := 0$
    Add edge $(i, j-1) \to (i, j)$, weight $-\beta$, label $\boldsymbol{b}_j := 1$
  end if
end for
Label node (M, N) as $\perp$
return the diagram

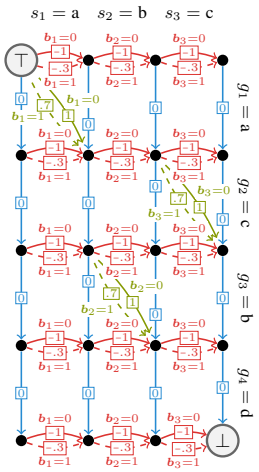


Figure 3. An example of an uncertainty-aware edit-distance-based utility function, and of the correspondence between dynamic programming and decision diagram construction. Algorithm 1 shows a dynamic programming implementation of the utility function $u(g, s)$ described in Section 2, taking as input a target goal state g, a fixed suggestion s, and weights α and β for correct and deleted UNSURE tokens, respectively. We assign positive utility to correctly predicted tokens instead of penalizing inserts so that an empty suggestion has zero utility. Setting $\alpha < 1$ and $\beta < 1$ allows us to approximate the effect of users double-checking UNSURE regions: UNSURE tokens provide less utility than SURE tokens if kept but are easier to delete if wrong. Algorithm 2 is an expanded version of this utility function that enables simultaneously searching over the c_j by building a decision diagram, as described in Section 3.3. On the right, we show an example decision diagram obtained by running Algorithm 2 with $g^{(1)} = [a, b, c]$, $g^{(k)} = [a, c, b, d]$, $\alpha = 0.7$ and $\beta = 0.3$, colored based on the algorithm steps. Note that our experiments use a more complex utility function, described in Section 3.4 and Appendix C.

Here K is a hyperparameter that determines how many samples we optimize over. For the space of confidence-aware suggestions described in Section 2, we can set

$$\mathcal{S}(g^{(1)}) = \{[(g_1^{(1)}, c_1), \ldots, (g_N^{(1)}, c_N)] : c_i \in \mathcal{C}\},$$

which corresponds to taking the suggestion tokens s_i from $g^{(1)}$ and just searching over the confidence markers c_i.

3.2. Decomposing Into Independent Subproblems

A standard technique for optimizing sums over combinatorial discrete spaces such as $\mathcal{S}(g^{(1)})$ is *dual decomposition*. We give a brief overview of dual decomposition as it applies to our problem; see Sontag et al. (2011) and Rush & Collins (2012) for an in-depth introduction.

We start by choosing an embedding ϕ of the search space $\mathcal{S}(g^{(1)})$ into a space of d-dimensional binary vectors, such that each possible suggestion $s \in \mathcal{S}(g^{(1)})$ maps to a unique vector $\boldsymbol{b} = \phi(s) \in \{0, 1\}^d$; here d may depend on $g^{(1)}$ and the chosen embedding ϕ. We then rewrite our utility function $u(g^{(k)}, s)$ as a function $w^{(k)}(\phi(s))$ of those binary vectors $\boldsymbol{b} = \phi(s)$, obtaining the equivalent optimization problem

$$\tilde{\boldsymbol{b}} = \operatorname*{arg\,max}_{\boldsymbol{b} \in \{0,1\}^d} \sum_{k=1}^{K} w^{(k)}(\boldsymbol{b}),$$

from which we can recover the solution to Equation (3) by inverting the embedding using information from $g^{(1)}$, i.e. by setting $\tilde{s} = \phi^{-1}(\tilde{\boldsymbol{b}})$.

Note that the embedding function ϕ will usually depend on the set of suggestions being considered. Taking the set $\mathcal{S}(g^{(1)})$ from the previous section as an example, if $g^{(1)}$ has N tokens we might choose $d = N$ and define $\boldsymbol{b} = \phi(s)$ by setting $\boldsymbol{b}_i = 1$ whenever $c_i =$ UNSURE in the suggestion $s = [(g_1^{(1)}, c_1), \ldots, (g_N^{(1)}, c_N)] \in \mathcal{S}(g^{(1)})$. (Our experiments use a somewhat more complex embedding function, described in Appendix C.)

Next, we reinterpret constrained optimization problem over copies of $\boldsymbol{b}$, i.e. as

$$U = \max_{\boldsymbol{b}^{(1)}, \ldots, \boldsymbol{b}^{(K)} \in \{0,1\}^d} \sum_{k=1}^{K} w^{(k)}(\boldsymbol{b}^{(k)}), \tag{4}$$

subject to $\boldsymbol{b}^{(1)} = \boldsymbol{b}^{(2)} = \cdots = \boldsymbol{b}^{(K)}$. Finally, we introduce a set of Lagrange multipliers $\boldsymbol{\lambda}^{(k)} \in \mathbb{R}^d$ with $\sum_k \boldsymbol{\lambda}^{(k)} = 0$ to relax these equality constraints:

$$W^{(k)}(\boldsymbol{b}^{(k)}, \boldsymbol{\lambda}^{(k)}) = w^{(k)}(\boldsymbol{b}^k) + \boldsymbol{\lambda}^{(k)} \cdot \boldsymbol{b}^{(k)}, \tag{5}$$

$$L(\boldsymbol{\lambda}^{(1:K)}) = \sum_{k=1}^{K} \max_{\boldsymbol{b}^{(k)}} W^{(k)}(\boldsymbol{b}^{(k)}, \boldsymbol{\lambda}^{(k)}) \geq U. \tag{6}$$

This is known as the Lagrangian dual problem: $L(\boldsymbol{\lambda}^{(1:K)})$ is a convex function of $\boldsymbol{\lambda}$ and an upper bound on U, and our goal is to find $\arg\min_{\boldsymbol{\lambda}} L(\boldsymbol{\lambda}^{(1:K)})$. If we can find $\boldsymbol{\lambda}^{(1:K)}$ such that $\boldsymbol{b}^{(1)} = \boldsymbol{b}^{(2)} = \cdots = \boldsymbol{b}^{(K)}$ in Equation (6), the bound is tight and we recover the solution to Equation (4).

(Note that this bound is not necessarily tight: we may have a nonzero duality gap $\min_{\boldsymbol{\lambda}} L(\boldsymbol{\lambda}) - U > 0$.) The key advantage of this formulation is that the only interaction between terms in the sum is the constraint $\sum_k \boldsymbol{\lambda}^{(k)} = 0$, which we can interpret as "messages" between subproblems. We can thus alternate between independently optimizing over the $\boldsymbol{b}^{(k)}$ for each $W^{(k)}$ term, and adjusting the $\boldsymbol{\lambda}^{(k)}$ ("message-passing") to tighten the dual bound by increasing agreement of the $\boldsymbol{b}^{(k)}$.

One efficient optimization algorithm of this form is "max-marginal-averaging", a version of coordinate descent described by Lange & Swoboda (2021). It works by iterating through variable indices i, computing the *max-marginals*

$$m_{i:=\beta}^{(k)} = \max_{\boldsymbol{b}^{(k)} \text{ s.t. } \boldsymbol{b}_i^{(k)}=\beta} W^{(k)}(\boldsymbol{b}^{(k)}, \boldsymbol{\lambda}^{(k)}), \quad \beta \in \{0,1\}$$

(which measure the utility of fixing $\boldsymbol{b}_i^{(k)}$ to $\beta = 0$ or $\beta = 1$), setting $\delta_i^{(k)} = m_{i:=1}^{(k)} - m_{i:=0}^{(k)}$, and then updating

$$\boldsymbol{\lambda}_i^{(k)} \leftarrow \boldsymbol{\lambda}_i^{(k)} - \delta_i^{(k)} + \frac{1}{K} \sum_{k'} \delta_i^{(k')} \tag{7}$$

This update ensures $\delta_i^{(k)} = \delta_i^{(k')}$ for all k, k', which implies that the same choice ($\boldsymbol{b}_i^{(k)} := 0$ or $\boldsymbol{b}_i^{(k)} := 1$) is optimal for every k and by the same amount. This is a coordinate descent update for $L(\boldsymbol{\lambda}^{(1:K)})$ with respect to the $\boldsymbol{\lambda}_i^{(1:K)}$ (Lange & Swoboda, 2021; Werner et al., 2020), and applying it produces a monotonically decreasing upper bound on U.

3.3. Expanding Utility Functions To Decision Diagrams

It remains to show how to efficiently compute the updates in Equation (7) corresponding to our objective in Equation (3). Our key idea is to focus on a family of utility functions that can be computed using an edit-distance-like dynamic program, and rewrite them in a form that enables us to simultaneously search over edit sequences, which are different for each subproblem, and confidence annotations, which must be chosen consistently across all subproblems.

Figure 3 gives an example of this transformation for the utility function introduced in Section 2. We start with Algorithm 1, which computes an edit-distance-based utility $u(g, s)$ for a specific suggestion s and a specific vector of confidence annotations c by searching over possible alignments of s and g. We then extend this implementation to Algorithm 2, which additionally searches over confidence annotations by embedding both the sequence of edits and the sequence of confidence annotations into a single *binary decision diagram* (BDD). Finding the maximum-utility path in this diagram simultaneously computes both the optimal alignment between s and g and the optimal confidence annotations c_i, and we can reconstruct the confidence annotations by following the path and setting $c_i = \text{UNSURE}$ whenever we encounter an edge labeled $\boldsymbol{b}_i := 1$ (inverting the original embedding ϕ).

We can now build a system of decision diagrams by constructing a separate BDD for each model sample $g^{(k)}$, and use this system to solve the optimization problem in Equation (6). Specifically, we can compute the max-marginals $m_{i:=\beta}^{(k)}$ for a given variable $\boldsymbol{b}_i$ and subproblem k by traversing the diagram for subproblem k and separately considering paths that assign $\boldsymbol{b}_i := 0$ and $\boldsymbol{b}_i := 1$. The messages $\boldsymbol{\lambda}_i^{(k)}$ can then be incorporated by modifying the costs of all edges in subproblem k that assign $\boldsymbol{b}_i := 1$, which biases that subproblem's search to prefer confidence annotations that are consistent with the choices of other subproblems.

In more detail, we follow Lange & Swoboda (2021) and use the BDD representation to run a sequence of max-marginal-averaging updates until $L(\boldsymbol{\lambda}^{(1:K)})$ stops improving. If all subproblems agree on the optimal assignment $\boldsymbol{b}$ (e.g. Equation (6) is tight), the maximum-utility paths for each subproblem now correspond to edit sequences for the same suggestion $\tilde{s} = \phi^{-1}(\boldsymbol{b}) \in \mathcal{S}(g^{(1)})$; we can thus reconstruct the solution to Equation (3) by combining information from $\boldsymbol{b}$ and the suggestion prototype $g^{(1)}$. If the subproblems do not agree, we instead decode an approximate solution to Equation (4) by greedily committing to the most promising assignment for each element of $\boldsymbol{b}$ and updating max-marginals to be consistent with the fixed assignments, similar to the heuristic described by Kolmogorov (2014), and then reconstruct a (possibly suboptimal) suggestion from this guess. We note that additional subtleties are needed to make the BDD representation efficient and to avoid unnecessary recomputation in the dynamic programs; see Appendix B.

3.4. Extending the Utility Function

Our method can be applied to any space of suggestions $\mathcal{S}$ and utility function $u(g, s)$ that can be efficiently represented as decision diagrams. In this section we briefly summarize a number of extensions to the basic utility function presented in Algorithm 1. These extensions, which we use for our experiments, enable us to adapt the R-U-SURE system to a variety of tasks without modifying the pretrained language model. (See Appendix C for details.)

Incorporating tree structure with hierarchical edits. When data has a natural tree structure (e.g. an abstract syntax tree for a program), we can incorporate this structure into $u(g, s)$ by requiring that edits respect the tree hierarchy. In particular, we implement a recursive utility function under which entire subtrees are either deleted, inserted, or recursively matched with other subtrees at the same depth.

Constraining locations of UNSURE *regions.* Similarly, we may have prior knowledge about which tokens are appropriate to mark as UNSURE; for instance, we may want to ensure that UNSURE tokens align with parsed expressions in the syntax tree. We can enforce this by introducing new binary

decision variables that track where UNSURE regions start and stop, and including a "constraint BDD" which ensures they start and stop in semantically-meaningful positions.

Adding localization and insertion penalties. Identifying which location to edit may be more difficult than actually performing the edit, and it may be useful to identify locations at which more code must be inserted even if all of the tokens in the suggestion are likely to be kept. To account for this, we can introduce an additional "localization penalty" each time an edit starts, independent of the size of the edit. This encourages our method to group edits into semantically meaningful chunks and to identify locations where missing code may need to be inserted, as long as we allow small UNSURE regions to be added in spaces between tokens.

Searching for prefixes. We may want to extract only a small portion of the model's initial suggestion, stopping once the uncertainty becomes too high. We can account for this by introducing new decision variables that determine whether or not to truncate the suggestion at various points, and modifying u to stop penalizing edits after the truncation point.

4. Related Work

Decoding by maximizing utility. A variety of sampling-based decoding strategies aiming to minimize Bayes risk have been proposed, with many works applying it to neural machine translation (Eikema & Aziz, 2020; Bhattacharyya et al., 2021; Kumar & Byrne, 2004; Ehling et al., 2007; Müller & Sennrich, 2021; Eikema & Aziz, 2021; Freitag et al., 2022) and to code generation (Li et al., 2022; Shi et al., 2022). These approaches generally use a utility function to select one sample from a larger generated set. González-Rubio & Casacuberta (2015) also explore combining parts of multiple samples to construct a single combined sample, and Lin & Chen (2010) propose using Bayes risk for extractive summarization. Although not framed as utility maximization, self-consistency decoding (Wang et al., 2022; Huang et al., 2022) also uses samples to identify the most likely correct answer under a model's distribution, and has been shown to improve reasoning ability.

Reinforcement-learning and sequential-decision-making techniques can also be used to maximize conditional expected utility, by interpreting tokens as actions and the utility as a reward (Lampouras & Vlachos, 2016; Sun et al., 2017; Chen et al., 2018; Prabhavalkar et al., 2018; Keneshloo et al., 2019; Leblond et al., 2021). Many works maximize quality metrics such as BLEU or error-rate, although others have used measures of program correctness (Le et al., 2022) or learned reward models (Ziegler et al., 2019; Ouyang et al., 2022; Bai et al., 2022). Others have trained models to imitate a more expensive reward-driven search process (Kuncoro et al., 2016; Liu et al., 2018; Sabour et al., 2019).

Selective and multi-choice prediction. One approach to avoid incorrect predictions under uncertainty is *selective classification*, i.e. abstaining from some predictions to minimize overall risk (Chow, 1957; El-Yaniv et al., 2010; Geifman & El-Yaniv, 2017; Dong et al., 2018; Ziyin et al., 2019). Another approach is to output multiple predictions, e.g. all classifications with confidence above a threshold (Vovk et al., 2005; Angelopoulos & Bates, 2021), or an ensemble of structured outputs which approximately covers the true output (Guzman-Rivera et al., 2012; 2014; Prasad et al., 2014; Lee et al., 2016; Bhattacharyya et al., 2018; Firman et al., 2018; Premachandran et al., 2014). When the space of possible outputs is very large, uncertain predictions can be compressed by representing multiple sequences as a lattice (Su et al., 2017; Sperber et al., 2017); lattice representations have also been used within a Bayes risk framework (Tromble et al., 2008; Xu et al., 2010).

Generating and identifying partial programs. A number of works have considered identifying common patterns in source code (Lozano et al., 2010; Allamanis & Sutton, 2014; Shin et al., 2019; Sivaraman et al., 2021), as well as generating programs with holes to aid in program synthesis (Nye et al., 2019; Ellis et al., 2020). Most relevant to our current work, Guo et al. (2021) propose GRAMMFORMER, a generative model for code that produces holes in parts of the syntax tree that are difficult to predict. GRAMMFORMER generates code top-down by iteratively expanding nonterminal nodes of a syntax tree, and is trained via a combination of random-order pretraining and RL finetuning, using a regular-expression-based objective that trades off between size and accuracy. In contrast, our approach requires only sample access to a pretrained generative model, can adapt to different utility functions and suggestion types without retraining the model, and can identify regions of uncertainty in both syntax trees and unstructured sequences (e.g. docstrings) without aligning them to a syntax tree derivation.

Uncertainty quantification and summarization. Past works have compared model-generated sequences to ground truth (Ott et al., 2018; Holtzman et al., 2019), studied the calibration of deep models in general (Carrell et al., 2022), and proposed new mechanisms for training better-calibrated models (Tran et al., 2022; Xiao et al., 2022). Kadavath et al. (2022) find that some large language models can sometimes answer natural-language questions about the accuracy of their own generated outputs, improving when multiple sampled outputs are included in the prompt, and Lin et al. (2022) show that language models can be fine-tuned to output calibrated uncertainty estimates. Our work relies on the calibration and sample-quality of the base intent model, but focuses on exposing this uncertainty to end-users. Also related are works which use attention and saliency maps to inform users about model behavior (Stevens & Su, 2020; Tenney et al., 2020), as well as works that visualize per-

token probabilities to summarize model uncertainty (Strobelt et al., 2021; Weisz et al., 2021; Sun et al., 2022). Most relevant to our work, Vasconcelos et al. (2022) found that visualizations of predicted locations of edits are strongly preferred by users over individual token probabilities and also significantly reduce editing time. They used a separate model trained to predict edits for a specific coding problem based on user editing traces; in contrast, our technique can be used to produce annotations without requiring additional model training or edit supervision.

Combinatorial optimization. Dual decomposition and block coordinate descent/ascent solvers have been applied to a variety of optimization problems, including combinatorial search (Swoboda et al., 2016), MAP inference (Sontag et al., 2011), and NLP tasks (Rush et al., 2010). Relevant to our work, Paul & Eisner (2012) and Peng et al. (2015) use dual decomposition with n-gram features to combine WFSAs, with applications to minimizing Bayes risk. There has also been recent interest in using binary decision diagrams as representations for combinatorial optimization (Castro et al., 2022). Our work expands on that of Lange & Swoboda (2021) by applying dual decomposition to a larger class of decision diagrams; see Appendix B.

5. Experiments

We evaluate our approach by applying it to three developer assistance tasks, each of which is visualized in Figure 1. For all tasks, we generate suggestion prototypes and hypothetical intents using a 5B-parameter decoder-only LM trained on 105B tokens of permissively-licensed open-source code from GitHub, and parse them into trees using an error-tolerant bracket-matching pseudo-parser (described in Appendix D). We compare our approach to a number of task-specific baselines, all of which build suggestions $s \in \mathcal{S}(g^{(1)})$ from the same suggestion prototype, and evaluate how well each method can predict the changes necessary to obtain the final code state from the dataset, measured by our utility function as well as token accuracy.

5.1. Localizing Edits In Code Suggestions

Our first task uses R-U-SURE to insert confidence annotations around parts of code completion suggestions that users are likely to edit. As discussed in Section 2, we configure our utility function so that UNSURE tokens have lower utility if matched but lower penalties if deleted. We additionally enforce hierarchical edits and syntactically-valid UNSURE regions and add extra localization penalties using the extensions described in Section 3.4; see Appendix E.2 for the specific configuration we use.

To evaluate our approach, we assemble a collection of 5000 held-out code files for each of the languages JAVA, JAVASCRIPT, C++ and PYTHON, and split them into (synthetic) completion contexts c and ground truth intents g using three strategies. One such scheme is PYTHON specific, so we obtain 45000 examples in total (see Appendix E for details). For each example, we sample 31 completions from the language model, then select the sample with the highest likelihood as the suggestion prototype, and use R-U-SURE to mark parts of the parsed tree as UNSURE. We compare our approach to heuristics based on token probabilities, which insert UNSURE regions around tokens whose conditional probability (TOKEN PROB) or total prefix probability (PREFIX PROB) is below a threshold; we also try marking everything SURE, and marking the maximum amount as UNSURE in our syntax-constrained space $\mathcal{S}(g^{(1)})$. (We find that annotating based on token probability can miss high-likelihood tokens that depend on earlier low-likelihood tokens, as shown in Figures 12 and 14 of Appendix A.)

	Utility (relative)	Est. Util. (relative)	LOO Util. (relative)	F_1 score (for UNSURE)
ALL SURE	$\equiv 0$	38.00	30.35	-
MAX UNSURE	81.83	106.08	101.90	12.74
TOKEN PR. 0.5	50.83	82.63	77.10	63.03
TOKEN PR. 0.7	58.42	88.89	83.68	64.38
TOKEN PR. 0.9	66.99	95.64	90.79	61.44
PREFIX PR. 0.5	83.33	108.52	104.29	41.92
PREFIX PR. 0.7	83.45	108.27	104.05	37.26
PREFIX PR. 0.9	83.08	107.61	103.41	29.53
OURS (Region)	**84.42**	**113.82**	**109.12**	**72.14**

Table 1. Breakdown of edit-localization performance. Methods that perform well on model samples (Est. and LOO Utility) also perform well on the ground truth user intent (Utility); we measure utility relative to labeling the entire suggestion SURE. Our approach achieves higher utility and also stronger token level F_1 score when interpreting UNSURE tokens as predicted edits.

We first investigate how well our optimizer of the sample-based approximate objective in Equation (3) succeeds at producing a good suggestion for the true unobserved intent g, as measured by our utility metric u. To this end, in Table 1 we report the utility $u(g, s)$ of each method's annotated suggestion s based on the true file contents g (Utility), and also our estimate of utility $\frac{1}{K}\sum_{k=1}^{K} u(g^{(k)}, s)$ computed across $K = 31$ samples from the model (Est. Util.). We additionally report the estimated utility $u(g^{(K+1)}, s)$ for a held-out model sample $g^{(K+1)}$ which was not used for optimization, denoting this "Leave-One-Out Utility" (LOO Util.). We find that methods with high average utility on model samples also achieve high average utility against the true file contents, with our method successfully obtaining high utility in both settings. A more detailed comparison in Figure 4 reveals that utility on the model samples $g^{(k)}$ and utility on the unobserved intent g are highly correlated for our suggestions, and Figure 5b shows that utility also

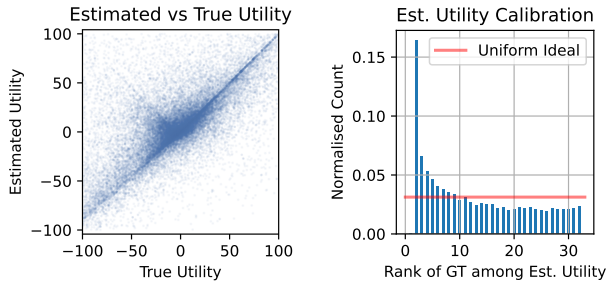
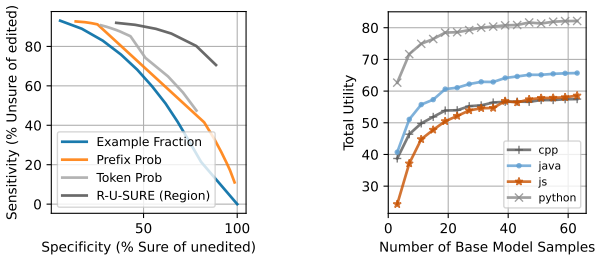


Figure 4. Comparison of true and estimated utility for edit localization task, as a scatterplot and as a rank-histogram (Candille & Talagrand, 2005). The true utility of our approach's suggestions is correlated with the estimate from the model samples, and appears fairly often at any rank within those samples, indicating that the samples are often a good proxy for the true uncertainty in the user's intent. However, some examples have a lower ground-truth utility than estimated, potentially indicative of model miscalibration.



(a) Sensitivity *vs* Specificity. (b) Utility *vs.* sample size K.

Figure 5. Analysis of results for edit localization. *(a)* Token level sensitivity / specificity trade-off across methods; our approach Pareto-dominates the others. (Here "Example Fraction" marks a fixed fraction as UNSURE.) *(b)* Mean utility for our approach w.r.t the ground-truth user intent as a function of the number of samples used in Equation (3), split by programming language.

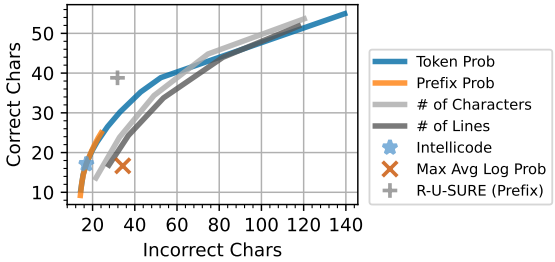


Figure 6. Character-level accuracy tradeoff for the suggestion-length task. R-U-SURE obtains a favourable trade-off. (Note that for simplicity we did not vary the hyperparameters of R-U-SURE for this task.)

improves as we optimize over more samples.

Note that our utility metric is only an approximation of the quality of an uncertainty-annotated suggestion, so having higher utility does not necessarily imply that our method produces more useful suggestions. To give more insight and evaluate how well maximizing our utility function truly summarizes the uncertainty of the model, we reinterpret uncertainty region annotations as a binary classification problem, with UNSURE tokens being predictions of where users will edit. We then compute the sensitivity (fraction of edited code correctly marked UNSURE) and specificity (fraction of unedited code marked SURE) for all methods with respect to the ground truth g. We visualize how these metrics vary as we sweep over the per-token utilities and costs of UNSURE tokens, obtaining the Pareto curve in Figure 5a; we also summarize the accuracy with F_1 scores in Table 1. We find that our approach is better at identifying locations of edits than the baselines, indicating that maximizing our utility metric does produce meaningful uncertainty estimates.

5.2. Selecting Suggestion Lengths

One common use of ML model suggestions for both code and natural language applications is to show inline grey "ghost text" suggestions as users type in the editor, and allow users to quickly accept the suggestion by pressing a key (often `tab`) (Svyatkovskiy et al., 2020; Barke et al., 2022). In this case, showing longer correct suggestions can accelerate developer productivity, but long incorrect suggestions can slow developers down (Barke et al., 2022).

To apply our approach to this setting, we disable insertion of UNSURE annotations, and instead search over prefixes of the prototype suggestion using the truncation variables described in Section 3.4; we continue to enforce hierarchical edits as well (see Appendix E.2). We compare our approach to a number of heuristics: predicting a fixed number of lines, predicting until we reach a low-probability token (Token Prob.) or until the total probability is below a threshold (Prefix Prob.), using the heuristic described by Svyatkovskiy et al. (2020) (IntelliCode Compose), and choosing the prefix that maximizes the ratio between the log-probability of the sequence and its length (Max Avg. Log Prob) inspired by the similar heuristics described by Chen et al. (2021a) and Zhang et al. (2022).

Figure 6 visualizes the tradeoff between correct and incorrect characters predicted by each method, and shows that our approach achieves a better tradeoff than other prefix-selection baselines. Table 4 in Appendix F shows that our approach also achieves strong results on our utility metric, similar to the edit localization task.

5.3. API Discovery

Even if there is not enough information to provide a useful completion suggestion at a specific location, it may still be possible to extract useful information from a genera-

	Utility	Correct tokens	Incorrect tokens
NOVEL CALLS	-1.39	1.18	3.04
ALL CALLS	-1.39	1.18	3.04
ALL CALLS + LHS + ARGS	-8.75	2.02	9.64
OURS (USEFUL CALLS)	**5.10**	3.56	2.10

Table 2. Comparison of utility and token-level accuracy statistics for API call sequence task. Our approach achieves higher utility by selecting a subset of method calls that is likely to appear, and including the LHS and arguments only when they are predictable.

tive model's suggestions. As an example of this, we use our method to identify *sequences of function calls* that the user is likely to write, even if the control flow structures around these calls are not predictable; this could be used to preemptively show documentation or type signatures.

We adapt our approach to this setting by extracting a sequence of function and method calls from the model's output, choosing $\mathcal{S}(g_1)$ so that it selects a subset of these calls as SURE, and defining $u(g, s)$ to find the longest common subsequence between the desired calls in g and the selected calls. Since we expect such suggestions to be used as an auxiliary aid rather than an inline suggestion, we set the utility of correct predictions to be higher than the penalty for incorrect ones (e.g. selected calls in s that were not used in g) and give a bonus for predicting tokens not seen before; we also assign zero utility to unselected calls. We implement this by applying the hierarchical edits and region location constraints described in Section 3.4 to a serialized representation of the extracted calls; see Appendix E.2 for the specific configuration we use.

We compare our approach against baselines which use all calls in the file or which only predict calls that use identifiers that are not in the context. Results are shown in Table 2; we again find that our approach both achieves strong performance on our utility metric and gives a favorable tradeoff between correct and incorrect predictions.

6. Discussion

We have demonstrated that R-U-SURE can flexibly incorporate uncertainty annotations into model suggestions across a variety of developer-assistance tasks, and that these annotations lead to both improved performance on our estimates of utility and also accurate predictions of the locations of edits. Importantly, our approach does not require retraining or fine-tuning the base generative language model, since it decouples the action (showing a suggestion) from the generative prediction task (predicting the user's intent).

A limitation of our approach is that it is restricted to utility functions that can be efficiently decomposed into decision diagrams. This is a good fit for edit-distance-based utility functions, and we believe the same principles could be extended to support multiple confidence levels or suggested alternatives. However, more general types of utility function (e.g. behavioral equivalence) may be difficult to approximate with our technique. We also assume that the base generative model is well-calibrated, and that a modest number of samples from it can summarize the possible edits required. It would be interesting to study how our system behaves with less-calibrated models, and how this changes as the capacity of the base model grows.

Our current implementation of R-U-SURE runs on the CPU using Numba (Lam et al., 2015), and takes between 1 second and 1 minute depending on the number, length, and complexity of the sampled programs (see Appendix A.1 for details). Notably, this runtime is dominated by the time to build the decision diagrams rather than the time to run the coordinate-ascent optimizer, likely because our utility function is very general and was designed for flexibility rather than decision-diagram construction efficiency. Although out of scope of this paper, we have explored distilling the outputs of R-U-SURE into a learned model similar to Kuncoro et al. (2016) and Kadavath et al. (2022), which can then be queried in real time with comparable accuracy to the original R-U-SURE system. Runtime could also be improved by rewriting in a lower-level langauge, specializing the utility function to a specific task, or using GPU acceleration (Abbas & Swoboda, 2021).

More broadly, we are excited by the potential to incorporate user interaction into minimum-Bayes-risk objectives to mitigate harms of model hallucinations. We see our work as a step toward ML-powered assistants that empower users and give appropriately conservative predictions in the presence of uncertainty about user intent and the world at large.

Acknowledgements

We would like to thank Jacob Hegna, Hassan Abolhassani, Jacob Austin, and Marc Rasi for contributing ideas toward early designs of the R-U-SURE system, Maxim Tabachnyk, Chris Gorgolewski, Vladimir Pchelin, Yurun Chen, Ilia Krets, Savinee Dancs, Alberto Elizondo, Iris Chu, Ambar Murillo, Ryan McGarry, Paige Bailey, and Kathy Nix for useful discussions and for collaborating on code completion applications of R-U-SURE, and Miltiadis Allamanis for providing valuable feedback on the paper draft. We would also like to thank Abhishek Rao, Alex Polozov, Joshua Howland, Kefan Xiao, and Vedant Misra for providing the language models and evaluation data used for our experimental results, and the members of Google Brain's Machine Learning for Code team for useful feedback throughout the project.

References

Abbas, A. and Swoboda, P. FastDOG: Fast discrete optimization on GPU. *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 439–449, 2021.

Allamanis, M. and Sutton, C. Mining idioms from source code. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014.

Angelopoulos, A. N. and Bates, S. A gentle introduction to conformal prediction and distribution-free uncertainty quantification. *arXiv preprint arXiv:2107.07511*, 2021.

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C. J., Terry, M., Le, Q. V., and Sutton, C. Program synthesis with large language models. *ArXiv*, abs/2108.07732, 2021.

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

Barke, S., James, M. B., and Polikarpova, N. Grounded copilot: How programmers interact with code-generating models. *ArXiv*, abs/2206.15000, 2022.

Bhattacharyya, A., Schiele, B., and Fritz, M. Accurate and diverse sampling of sequences based on a "best of many" sample objective. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 8485–8493, 2018.

Bhattacharyya, S., Rooshenas, A., Naskar, S., Sun, S., Iyyer, M., and McCallum, A. Energy-based reranking: Improving neural machine translation using energy-based models. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 4528–4537, 2021.

Bollig, B. and Buttkus, M. On the relative succinctness of sentential decision diagrams. *Theory of Computing Systems*, pp. 1–28, 2018.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T. J., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language models are few-shot learners. *ArXiv*, abs/2005.14165, 2020.

Candille, G. and Talagrand, O. Evaluation of probabilistic prediction systems for a scalar variable. *Quarterly Journal of the Royal Meteorological Society*, 131, 2005.

Carrell, A., Mallinar, N. R., Lucas, J., and Nakkiran, P. The calibration generalization gap. *ArXiv*, abs/2210.01964, 2022.

Castro, M. P., Ciré, A. A., and Beck, J. C. Decision diagrams for discrete optimization: A survey of recent advances. *INFORMS J. Comput.*, 34:2271–2295, 2022.

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. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021a.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Ponde, H., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D. W., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Babuschkin, I., Balaji, S. A., Jain, S., Carr, A., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M. M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W. Evaluating large language models trained on code. *ArXiv*, abs/2107.03374, 2021b.

Chen, Y., Li, V. O., Cho, K., and Bowman, S. R. A stable and effective learning strategy for trainable greedy decoding. *arXiv preprint arXiv:1804.07915*, 2018.

Chow, C.-K. An optimum character recognition system using decision functions. *IRE Transactions on Electronic Computers*, (4):247–254, 1957.

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., Schuh, P., Shi, K., Tsvyashchenko, S., Maynez, J., Rao, A., Barnes, P., Tay, Y., Shazeer, N. M., Prabhakaran, V., Reif, E., Du, N., Hutchinson, B. C., Pope, R., Bradbury, J., Austin, J., Isard, M., Gur-Ari, G., Yin, P., Duke, T., Levskaya, A., Ghemawat, S., Dev, S., Michalewski, H., García, X., Misra, V., Robinson, K., Fedus, L., Zhou, D., Ippolito, D., Luan, D., Lim, H., Zoph, B., Spiridonov, A., Sepassi, R., Dohan, D., Agrawal, S., Omernick, M., Dai, A. M., Pillai, T. S., Pellat, M., Lewkowycz, A., Moreira, E., Child, R., Polozov, O., Lee, K., Zhou, Z., Wang, X., Saeta, B., Díaz, M., Firat, O., Catasta, M., Wei, J., Meier-Hellstern, K. S., Eck, D., Dean, J., Petrov, S., and Fiedel, N. PaLM: Scaling language modeling with pathways. *ArXiv*, abs/2204.02311, 2022.

Cummings, M. L. Automation bias in intelligent time critical decision support systems. 2004.

Dong, L., Quirk, C., and Lapata, M. Confidence modeling for neural semantic parsing. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 743–753, 2018.

Ehling, N., Zens, R., and Ney, H. Minimum Bayes risk decoding for bleu. In *Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics Companion Volume Proceedings of the Demo and Poster Sessions*, pp. 101–104, 2007.

Eikema, B. and Aziz, W. Is MAP decoding all you need? the inadequacy of the mode in neural machine translation. In *Proceedings of the 28th International Conference on Computational Linguistics*, pp. 4506–4520, 2020.

Eikema, B. and Aziz, W. Sampling-based minimum Bayes risk decoding for neural machine translation. *arXiv preprint arXiv:2108.04718*, 2021.

El-Yaniv, R. et al. On the foundations of noise-free selective classification. *Journal of Machine Learning Research*, 11 (5), 2010.

Ellis, K., Wong, C., Nye, M., Sablé-Meyer, M., Cary, L., Morales, L., Hewitt, L., Solar-Lezama, A., and Tenenbaum, J. B. DreamCoder: Growing generalizable, interpretable knowledge with wake-sleep Bayesian program learning. *ArXiv*, abs/2006.08381, 2020.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., and Zhou, M. CodeBERT: A pre-trained model for programming and natural languages. *ArXiv*, abs/2002.08155, 2020.

Firman, M., Campbell, N. D., Agapito, L., and Brostow, G. J. Diversenet: When one right answer is not enough. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 5598–5607, 2018.

Freitag, M., Grangier, D., Tan, Q., and Liang, B. High quality rather than high model probability: Minimum Bayes risk decoding with neural metrics. *Transactions of the Association for Computational Linguistics*, 10:811–825, 2022.

Geifman, Y. and El-Yaniv, R. Selective classification for deep neural networks. *Advances in neural information processing systems*, 30, 2017.

González-Rubio, J. and Casacuberta, F. Minimum Bayes' risk subsequence combination for machine translation. *Pattern Analysis and Applications*, 18(3):523–533, 2015.

Guo, D., Svyatkovskiy, A., Yin, J., Duan, N., Brockschmidt, M., and Allamanis, M. Learning to complete code with sketches. In *International Conference on Learning Representations*, 2021.

Guzman-Rivera, A., Batra, D., and Kohli, P. Multiple choice learning: Learning to produce multiple structured outputs. *Advances in neural information processing systems*, 25, 2012.

Guzman-Rivera, A., Kohli, P., Batra, D., and Rutenbar, R. Efficiently enforcing diversity in multi-output structured prediction. In *Artificial Intelligence and Statistics*, pp. 284–292. PMLR, 2014.

Holtzman, A., Buys, J., Forbes, M., and Choi, Y. The curious case of neural text degeneration. *ArXiv*, abs/1904.09751, 2019.

Hooker, J. N. Decision diagrams and dynamic programming. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pp. 94–110. Springer, 2013.

Huang, J., Gu, S. S., Hou, L., Wu, Y., Wang, X., Yu, H., and Han, J. Large language models can self-improve. *ArXiv*, abs/2210.11610, 2022.

Kadavath, S., Conerly, T., Askell, A., Henighan, T. J., Drain, D., Perez, E., Schiefer, N., Dodds, Z., DasSarma, N., Tran-Johnson, E., Johnston, S., El-Showk, S., Jones, A., Elhage, N., Hume, T., Chen, A., Bai, Y., Bowman, S., Fort, S., Ganguli, D., Hernandez, D., Jacobson, J., Kernion, J., Kravec, S., Lovitt, L., Ndousse, K., Olsson, C., Ringer, S., Amodei, D., Brown, T. B., Clark, J., Joseph, N., Mann, B., McCandlish, S., Olah, C., and Kaplan, J. Language models (mostly) know what they know. *ArXiv*, abs/2207.05221, 2022.

Kanade, A., Maniatis, P., Balakrishnan, G., and Shi, K. Learning and evaluating contextual embedding of source code. In *International Conference on Machine Learning*, 2019.

Keneshloo, Y., Shi, T., Ramakrishnan, N., and Reddy, C. K. Deep reinforcement learning for sequence-to-sequence models. *IEEE transactions on neural networks and learning systems*, 31(7):2469–2489, 2019.

Kolmogorov, V. A new look at reweighted message passing. *IEEE transactions on pattern analysis and machine intelligence*, 37(5):919–930, 2014.

Kumar, S. and Byrne, W. Minimum Bayes-risk decoding for statistical machine translation. In *Proceedings of the Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics: HLT-NAACL 2004*, pp. 169–176, 2004.

Kuncoro, A., Ballesteros, M., Kong, L., Dyer, C., and Smith, N. A. Distilling an ensemble of greedy dependency parsers into one mst parser. In *Conference on Empirical Methods in Natural Language Processing*, 2016.

Lam, S. K., Pitrou, A., and Seibert, S. Numba: a llvm-based python jit compiler. In *LLVM '15*, 2015.

Lampouras, G. and Vlachos, A. Imitation learning for language generation from unaligned data. In *Proceedings of COLING 2016, the 26th International Conference on Computational Linguistics: Technical Papers*, pp. 1101–1112. The COLING 2016 Organizing Committee, 2016.

Lange, J.-H. and Swoboda, P. Efficient message passing for 0–1 ilps with binary decision diagrams. In *International Conference on Machine Learning*, pp. 6000–6010. PMLR, 2021.

Le, H., Wang, Y., Gotmare, A. D., Savarese, S., and Hoi, S. C. CodeRL: Mastering code generation through pretrained models and deep reinforcement learning. *arXiv preprint arXiv:2207.01780*, 2022.

Leblond, R., Alayrac, J.-B., Sifre, L., Pislar, M., Lespiau, J.-B., Antonoglou, I., Simonyan, K., and Vinyals, O. Machine translation decoding beyond beam search. *arXiv preprint arXiv:2104.05336*, 2021.

Lee, S., Purushwalkam Shiva Prakash, S., Cogswell, M., Ranjan, V., Crandall, D., and Batra, D. Stochastic multiple choice learning for training diverse deep ensembles. *Advances in Neural Information Processing Systems*, 29, 2016.

Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Dal Lago, A., et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

Lin, S. C., Hilton, J., and Evans, O. Teaching models to express their uncertainty in words. *ArXiv*, abs/2205.14334, 2022.

Lin, S.-H. and Chen, B. A risk minimization framework for extractive speech summarization. In *Annual Meeting of the Association for Computational Linguistics*, 2010.

Liu, T., Zhang, Y., Brockett, C. J., Mao, Y., Sui, Z., Chen, W., and Dolan, W. B. A token-level reference-free hallucination detection benchmark for free-form text generation. In *Annual Meeting of the Association for Computational Linguistics*, 2021.

Liu, Y., Che, W., Zhao, H., Qin, B., and Liu, T. Distilling knowledge for search-based structured prediction. In *Annual Meeting of the Association for Computational Linguistics*, 2018.

Lozano, A., Kellens, A., Mens, K., and Arévalo, G. B. Mining source code for structural regularities. *2010 17th Working Conference on Reverse Engineering*, pp. 22–31, 2010.

Lozano, L., Bergman, D., and Smith, J. C. On the consistent path problem. *Operations Research*, 68(6):1913–1931, 2020.

Lyell, D. and Coiera, E. W. Automation bias and verification complexity: a systematic review. *Journal of the American Medical Informatics Association*, 24:423–431, 2017.

Madi, N. S. A. How readable is model-generated code? examining readability and visual inspection of github copilot. In *International Conference on Automated Software Engineering*, 2022.

Maynez, J., Narayan, S., Bohnet, B., and McDonald, R. T. On faithfulness and factuality in abstractive summarization. *ArXiv*, abs/2005.00661, 2020.

Mozannar, H., Bansal, G., Fourney, A., and Horvitz, E. Reading between the lines: Modeling user behavior and costs in ai-assisted programming. *ArXiv*, abs/2210.14306, 2022.

Müller, M. and Sennrich, R. Understanding the properties of minimum Bayes risk decoding in neural machine translation. In *The Joint Conference of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing*, pp. 259–272. Association for Computational Linguistics, 2021.

Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., and Xiong, C. Codegen: An open large language model for code with multi-turn program synthesis. 2022.

Nye, M., Hewitt, L. B., Tenenbaum, J. B., and Solar-Lezama, A. Learning to infer program sketches. *ArXiv*, abs/1902.06349, 2019.

Ott, M., Auli, M., Grangier, D., and Ranzato, M. Analyzing uncertainty in neural machine translation. *ArXiv*, abs/1803.00047, 2018.

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C. L., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., et al. Training language models to follow instructions with human feedback. *arXiv preprint arXiv:2203.02155*, 2022.

Paul, M. J. and Eisner, J. Implicitly intersecting weighted automata using dual decomposition. In *North American Chapter of the Association for Computational Linguistics*, 2012.

Pearce, H. A., Ahmad, B., Tan, B., Dolan-Gavitt, B., and Karri, R. An empirical cybersecurity evaluation of GitHub copilot's code contributions. *ArXiv*, abs/2108.09293, 2021.

Peng, N., Cotterell, R., and Eisner, J. Dual decomposition inference for graphical models over strings. In *Conference on Empirical Methods in Natural Language Processing*, 2015.

Prabhavalkar, R., Sainath, T. N., Wu, Y., Nguyen, P., Chen, Z., Chiu, C.-C., and Kannan, A. Minimum word error rate training for attention-based sequence-to-sequence models. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 4839–4843. IEEE, 2018.

Prasad, A., Jegelka, S., and Batra, D. Submodular meets structured: Finding diverse subsets in exponentially-large structured item sets. *Advances in Neural Information Processing Systems*, 27, 2014.

Premachandran, V., Tarlow, D., and Batra, D. Empirical minimum Bayes risk prediction: How to extract an extra few % performance from vision models with just three more parameters. *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1043–1050, 2014.

Rush, A. M. and Collins, M. A tutorial on dual decomposition and Lagrangian relaxation for inference in natural language processing. *Journal of Artificial Intelligence Research*, 45:305–362, 2012.

Rush, A. M., Sontag, D. A., Collins, M., and Jaakkola, T. On dual decomposition and linear programming relaxations for natural language processing. In *Conference on Empirical Methods in Natural Language Processing*, 2010.

Sabour, S., Chan, W., and Norouzi, M. Optimal completion distillation for sequence learning. In *International Conference on Learning Representations*, 2019.

Shi, F., Fried, D., Ghazvininejad, M., Zettlemoyer, L., and Wang, S. I. Natural language to code translation with execution. *arXiv preprint arXiv:2204.11454*, 2022.

Shin, R., Allamanis, M., Brockschmidt, M., and Polozov, O. Program synthesis and semantic parsing with learned code idioms. In *Neural Information Processing Systems*, 2019.

Sivaraman, A., Abreu, R., Scott, A. C., Akomolede, T., and Chandra, S. Mining idioms in the wild. *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pp. 187–196, 2021.

Sontag, D., Globerson, A., and Jaakkola, T. Introduction to dual composition for inference. In *Optimization for Machine Learning*. MIT Press, 2011.

Sperber, M., Neubig, G., Niehues, J., and Waibel, A. H. Neural lattice-to-sequence models for uncertain inputs. In *Conference on Empirical Methods in Natural Language Processing*, 2017.

Stevens, S. and Su, Y. An investigation of language model interpretability via sentence editing. In *BlackboxNLP Workshop on Analyzing and Interpreting Neural Networks for NLP*, 2020.

Strobelt, H., Hoover, B., Satyanarayan, A., and Gehrmann, S. LMdiff: A visual diff tool to compare language models. *ArXiv*, abs/2111.01582, 2021.

Su, J., Tan, Z., Xiong, D., Ji, R., Shi, X., and Liu, Y. Lattice-based recurrent neural network encoders for neural machine translation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31, 2017.

Sun, J., Liao, Q. V., Muller, M. J., Agarwal, M., Houde, S., Talamadupula, K., and Weisz, J. D. Investigating explainability of generative ai for code through scenario-based design. *27th International Conference on Intelligent User Interfaces*, 2022.

Sun, W., Venkatraman, A., Gordon, G. J., Boots, B., and Bagnell, J. A. Deeply AggreVaTeD: Differentiable imitation learning for sequential prediction. In *International conference on machine learning*, pp. 3309–3318. PMLR, 2017.

Svyatkovskiy, A., Deng, S. K., Fu, S., and Sundaresan, N. IntelliCode Compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1433–1443, 2020.

Swoboda, P., Kuske, J., and Savchynskyy, B. A dual ascent framework for Lagrangean decomposition of combinatorial problems. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 4950–4960, 2016.

Tenney, I., Wexler, J., Bastings, J., Bolukbasi, T., Coenen, A., Gehrmann, S., Jiang, E., Pushkarna, M., Radebaugh, C., Reif, E., and Yuan, A. The language interpretability tool: Extensible, interactive visualizations and analysis for nlp models. In *Conference on Empirical Methods in Natural Language Processing*, 2020.

Tran, D., Liu, J., Dusenberry, M. W., Phan, D., Collier, M., Ren, J. J., Han, K., Wang, Z., Mariet, Z. E., Hu, H., Band, N., Rudner, T. G. J., Singhal, K., Nado, Z., van

Amersfoort, J. R., Kirsch, A., Jenatton, R., Thain, N., Yuan, H., Buchanan, E. K., Murphy, K., Sculley, D., Gal, Y., Ghahramani, Z., Snoek, J., and Lakshminarayanan, B. Plex: Towards reliability using pretrained large model extensions. *ArXiv*, abs/2207.07411, 2022.

Tromble, R. W., Kumar, S., Och, F. J., and Macherey, W. Lattice minimum Bayes-risk decoding for statistical machine translation. In *Conference on Empirical Methods in Natural Language Processing*, 2008.

Upadhyaya, G., Reinhardt, A., Rajan, H., Kim, M., Glassman, E. L., Hartmann, B., and Pinedo, J. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. *CHI Conference on Human Factors in Computing Systems Extended Abstracts*, 2022.

Vasconcelos, H., Bansal, G., Fourney, A., Liao, Q. V., and Wortman Vaughan, J. Generation probabilities are not enough: Improving error highlighting for ai code suggestions. In *NeurIPS Workshop on Human-Centered AI*, October 2022.

Vovk, V., Gammerman, A., and Shafer, G. *Algorithmic learning in a random world*. Springer Science & Business Media, 2005.

Wang, X., Wei, J., Schuurmans, D., Le, Q., hsin Chi, E. H., and Zhou, D. Self-consistency improves chain of thought reasoning in language models. *ArXiv*, abs/2203.11171, 2022.

Weisz, J. D., Muller, M. J., Houde, S., Richards, J. T., Ross, S. I., Martinez, F., Agarwal, M., and Talamadupula, K. Perfection not required? human-AI partnerships in code translation. *26th International Conference on Intelligent User Interfaces*, 2021.

Werner, T., Prusa, D., and Dlask, T. Relative interior rule in block-coordinate descent. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 7559–7567, 2020.

Xiao, Y., Liang, P. P., Bhatt, U., Neiswanger, W., Salakhutdinov, R., and Morency, L.-P. Uncertainty quantification with pre-trained language models: A large-scale empirical analysis. *ArXiv*, abs/2210.04714, 2022.

Xu, H., Povey, D., Mangu, L., and Zhu, J. An improved consensus-like method for minimum Bayes risk decoding and lattice combination. *2010 IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 4938–4941, 2010.

Zhang, T., Yu, T., Hashimoto, T. B., Lewis, M., Yih, W.-t., Fried, D., and Wang, S. I. Coder reviewer reranking for code generation. *arXiv preprint arXiv:2211.16490*, 2022.

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

Ziyin, L., Wang, Z. T., Liang, P. P., Salakhutdinov, R., Morency, L.-P., and Ueda, M. Deep gamblers: learning to abstain with portfolio theory. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, pp. 10623–10633, 2019.

A. Example Outputs of R-U-SURE

In this section we give examples of the outputs produced by R-U-SURE.

(For these examples, we use an 8B-parameter decoder-only LM, a slightly larger model than used for the main set of experiments, also trained on permissively-licensed open-source code from GitHub. We emphasize that our approach is model-agnostic and can be combined with any generative model.)

```
import collections
import functools
import operator
import json
from typing import List

TokenWithConfidence = collections.namedtuple('TokenWithConfidence', ('token', 'confidence'))

def render_suggestion(suggestion: List[TokenWithConfidence]) -> str:
  """Renders confidence annotations for a suggestion.

  Args:
    suggestion: List of tokens and their confidence.

  Returns:
    Rendering of tokens with confidence.
  """
  result = []
  for token_with_conf in suggestion:
    if token_with_conf.confidence == "sure":
      result.append(token_with_conf.token)
    else:
      result.append("[" + token_with_conf.token + "]")
  return "".join(result)

def suggestion_to_json(suggestion: List[TokenWithConfidence]) -> str:
  """Converts a suggestion to JSON.

  Args:
    suggestion: List of tokens and their confidence.

  Returns:
    JSON representation of the suggestion.
  """
  ♦
  return json.dumps(suggestion)

def token_with_conf_to_json(token_with_conf: TokenWithConfidence) -> str:
  """Converts a token and its confidence to JSON.

  Args:
    token_with_conf: Token and its confidence.

  Returns:
    JSON representation of the suggestion.
  """
  return json.dumps(token_with_conf)

def render_correction(correction: str) -> str:
  """Renders correction annotations for a suggestion.

  Args:
```

Figure 7. Full prompt and output for the example at the top of Figure 1. Note that the model has identified the docstring style from the context, and our system can identify which of the words in the docstring are boilerplate. Docstrings are represented as sequences of words and combined using our edit distance utility function. The '♦' character denotes a location where additional statements might be inserted.

```
def suggestion_to_json(suggestion: List[TokenWithConfidence]) -> str:
  """Converts a suggestion to JSON.

  Args:
    suggestion: List of tokens and their confidence.

  Returns:
    JSON representation of the suggestion.
  """
  return json.dumps(suggestion)

def token_with_conf_to_json(token_with_conf: TokenWithConfidence) ->
str:
  """Converts a token and its confidence to JSON.

  Args:
    token_with_conf: Token and its confidence.

  Returns:
    JSON representation of the suggestion.
  """
  return json.dumps(token_with_conf)

def render_correction(correction: str) -> str:
  """Renders correction annotations for a suggestion.

  Args:
```

```
def suggestion_to_json(suggestion: List[TokenWithConfidence]) -> str:
  """Converts a suggestion to a json string.

  Args:
    suggestion: List of tokens and their confidence.

  Returns:
    Rendering of tokens with confidence.
  """
  return json.dumps(suggestion)

def suggestion_from_json(json_str: str) -> List[TokenWithConfidence]:
  """Creates a suggestion from a json string.

  Args:
    json_str: A json string.

  Returns:
    List of tokens and their confidence.
  """
  return json.loads(json_str)

def suggestion_to_dict(suggestion: List[TokenWithConfidence]) -> dict:
  """Converts a suggestion to a dict.

  Args:
```

```
def suggestion_to_json(suggestion: List[List[TokenWithConfidence]]) ->
str:
  """Renders a list of suggestions in JSON format.

  Args:
    suggestion: A list of lists of tokens and their confidences.

  Returns:
    Rendering of tokens with confidence.
  """
  result = []
  for tokens_with_conf in suggestion:
    token_with_conf_json = {
        'token': [],
        'confidence': [],
    }
    for token_with_conf in tokens_with_conf:
      token_with_conf_json['token'].append(token_with_conf.token)
      token_with_conf_json['confidence'].append(
          token_with_conf.confidence)
    result.append(token_with_conf_json)
  return
```

```
def suggestion_to_json(suggestion: List[TokenWithConfidence]) -> str:
  """Renders a suggestion as a JSON object."""
  return json.dumps([token.token for token in suggestion])

def get_top_n(suggestions: List[List[TokenWithConfidence]], n: int) ->
List[List[TokenWithConfidence]]:
  """Returns the top n suggestions of a list of suggestions.

  Args:
    suggestions: List of suggestions for a single token.
    n: Number of suggestions to return.

  Returns:
    A list of suggestions of size n, with the highest confidence
first.
  """
  return sorted(suggestions, key=lambda s: len(s), reverse=True)[:n]

def join_suggestion(suggestion: List[List[TokenWithConfidence]],
```

```
def suggestion_to_json(suggestion: List[TokenWithConfidence]) -> str:
  """Converts a list of tokens and their confidence to JSON.

  Args:
    suggestion: List of tokens and their confidence.

  Returns:
    JSON representation of the list.
  """
  return json.dumps([dict(token=token_with_conf.token,
                          confidence=token_with_conf.confidence)
                     for token_with_conf in suggestion])

def json_to_suggestion(json: str) -> List[TokenWithConfidence]:
  """Parses a suggestion from a JSON string.

  Args:
    json: JSON representation of the list.

  Returns:
    List of tokens and their confidence.
  """
  return [TokenWithConfidence(token=t["token"],
```

```
def suggestion_to_json(suggestion_list: List[List[TokenWithConfidence]])
-> str:
  """Renders a list of suggestions in json format.

  Args:
    suggestions_list: A list of suggestions to render.

  Returns:
    The list of suggestions as json string.
  """
  return json.dumps(suggestion_list, default=lambda o: o.__dict__,
sort_keys=True, indent=2)


def _is_suggestion_list_equal(a: List[List[TokenWithConfidence]], b:
List[List[TokenWithConfidence]]) -> bool:
  """Checks if two lists of suggestions are equal.

  Args:
    a: First list of suggestions.
    b: Second list of suggestions.

  Returns:
    True if the lists are equal, otherwise
```

Figure 8. Six of the hypothetical user intents $g^{(1)}, g^{(2)}, \ldots, g^{(6)}$ for the example at the top of Figure 1, generated by sampling from the pretrained model. Full context omitted; see Figure 7.

Utility: 455.15 (vs. 590.0 without UNSURE annotations)

```
def suggestion_to_json(suggestion: List[TokenWithConfidence]) -> str:
  """Converts a suggestion to JSON.

  Args:
    suggestion: List of tokens and their confidence.

  Returns:
    JSON representation of the suggestion.
  """
  return json.dumps(suggestion)

def token_with_conf_to_json(token_with_conf: TokenWithConfidence) -> str:
  """Converts a token and its confidence to JSON.

  Args:
    token_with_conf: Token and its confidence.

  Returns:
    JSON representation of the suggestion.
  """
  return json.dumps(token_with_conf)

def render_correction(correction: str) -> str:
  """Renders correction annotations for a suggestion.

  Args:
```

Utility: 124.65 (vs. 71.0 without UNSURE annotations)

```
def suggestion_to_json(suggestion: List[TokenWithConfidence]) -> str:
  """Converts a suggestion to JSONa json string.

  Args:
    suggestion: List of tokens and their confidence.

  Returns:
    JSON representationRendering of the suggestionokens with confidence.
  """
  return json.dumps(suggestion)

def token_with_conf_to_jsonsuggestion_from_json(json_str: str) -> List[token_with_conf:
TokenWithConfidence] -> str:
  """ConvertsCreates a suggestion from a token and its confidence to JSONjson_string.

  Args:
    token_with_conf: Token and its confidencejson_str: A json string.

  Returns:
    JSON representationList of the suggestiontokens and their confidence.
  """
  return json.dumps(token_with_conf)loads(json_str)

def render_correction(correction: str) -> str:suggestion_to_dict(suggestion:
List[TokenWithConfidence]) -> dict:
  """Renders correction annotations forConverts a suggestion to a dict.

  Args:
```

Utility: -105.35 (vs. -413.0 without UNSURE annotations)

```
def suggestion_to_json(suggestion: List[TokenWithConfidenceList[TokenWithConfidence]]) -> str:
  """ConvertsRenders a suggestion tolist of suggestions in JSON format.

  Args:
    suggestion: ListA list of lists of tokens and their confidence.
confidences.

  Returns:
    JSON representationRendering of the suggestionokens with confidence.
  """
  result = []
  for tokens_with_conf in suggestion:
    token_with_conf_json = {
        'token': [],
        'confidence': [],
    }
    for token_with_conf in tokens_with_conf:
      token_with_conf_json['token'].append(token_with_conf.token)
      token_with_conf_json['confidence'].append(token_with_conf.confidence)
    result.append(token_with_conf_json)
  return json.dumps(suggestion)

def token_with_conf_to_json(token_with_conf: TokenWithConfidence) -> str:
  """Converts a token and its confidence to JSON.

  Args:
    token_with_conf: Token and its confidence.

  Returns:
    JSON representation of the suggestion.
  """
  return json.dumps(token_with_conf)

def render_correction(correction: str) -> str:
  """Renders correction annotations for a suggestion.

  Args:
```

Utility: -102.84 (vs. -356.0 without UNSURE annotations)

```
def suggestion_to_json(suggestion: List[TokenWithConfidence]) -> str:
  """ConvertsRenders a suggestion to JSON.

  Args:
    suggestion: List of tokens and their confidence.

  Returns:
    as a JSON representation of the suggestion.
  """object """
  return json.dumps(suggestion[token.token for token in suggestion])

def token_with_conf_to_json(token_with_conf: TokenWithConfidence) -> str_get_top_n(suggestions:
List[List[TokenWithConfidence]], n: int) -> List[List[TokenWithConfidence]]:
  """ConvertsReturns the top n suggestions of a token and its confidence to JSON.
list of suggestions.

  Args:
    suggestions: List of suggestions for a single tokenwith_conf: Token and its confidence.
    n: Number of suggestions to return.

  Returns:
    JSON representationA list of suggestions of size n, with the suggestionhighest confidence
first.
  """
  return json.dumps(token_with_conf)sorted(suggestions, key=lambda s: len(s), reverse=True)[:n]

def render_correction(correction: str) -> str:
  """Renders correction annotations for a suggestion.

  Args:
def join_suggestion(suggestion: List[List[TokenWithConfidence]],
```

Utility: 115.64 (vs. -89.0 without UNSURE annotations)

```
def suggestion_to_json(suggestion: List[TokenWithConfidence]) -> str:
  """Converts a suggestionlist of tokens and their confidence to JSON.

  Args:
    suggestion: List of tokens and their confidence.

  Returns:
    JSON representation of the suggestionlist.
  """
  return json.dumps(suggestion[dict(token=token_with_conf.token,
                                    confidence=token_with_conf.confidence)
                               for token_with_conf in suggestion])

def token_with_conf_to_jsonjson_to_suggestion(json: str) -> List[[token_with_conf:
TokenWithConfidence]] -> str:
  """ConvertsParses a token and its confidence tosuggestion from a JSON string.

  Args:
    token_with_conf: Token and its confidence.

  Returns:
    json: JSON representation of the suggestionlist.

  Returns:
    List of tokens and their confidence.
  """
  return json.dumps(token_with_conf)
[TokenWithConfidence(token=["token"],

def render_correction(correction: str) -> str:
  """Renders correction annotations for a suggestion.

  Args:
```

Utility: -166.35 (vs. -476.0 without UNSURE annotations)

```
def suggestion_to_json(suggestionsuggestion_list:
List[TokenWithConfidenceList[TokenWithConfidence]]) -> str:
  """ConvertsRenders a suggestion to JSONlist of suggestions in json format.

  Args:
    suggestion: Listsuggestions_list: A list of tokens and their confidence.
suggestions to render.

  Returns:
    JSON representationThe list of the suggestionsuggestions as json string.
  """
  return json.dumps(suggestionsuggestion_list, default=lambda o: o.__dict__, sort_keys=True,
indent=2)


def token_with_conf_to_json(token_with_conf: TokenWithConfidence) -> str:
  """Converts a token and its confidence to JSON.

  Args:
    token_with_conf: Token and its confidence.

  Returns:
    JSON representation of the suggestion.
  """
  return json.dumps(token_with_conf)

def render_correction(correction: str) -> str:_is_suggestion_list_equal(a:
List[List[TokenWithConfidence]], b: List[List[TokenWithConfidence]]) -> bool:
  """Renders correction annotations for a suggestionChecks if two lists of suggestions are equal.

  Args:
  Args:
    a: First list of suggestions.
    b: Second list of suggestions.

  Returns:
    True if the lists are equal, otherwise
```

Figure 9. Inferred edits from the output suggestion in Figure 7 to each of the hypothetical user intents in Figure 8, along with the utility estimates for each when we either insert UNSURE regions as shown or require all tokens to be marked SURE. Constant utility shifts do not affect relative utility of different suggestions, so for our results in Table 1 and Figure 5b, we report utility relative to marking all tokens as SURE (i.e. the difference between the two values shown here). Note that utility improves when adding UNSURE regions for all samples except the first, which was the sample used as the suggestion prototype.

```
"""Visualizing trends using SQLite and Matplotlib"""
import sqlite3
from matplotlib import pyplot as plt

# Open the database
conn = sqlite3.connect('data/budget.db')
c = conn.cursor()

# Get the data
c.execute('''SELECT date, SUM(budget)
             FROM transactions
             GROUP BY date''')
data = c.fetchall()

# Close the database
conn.close()

# Create the plot
fig, ax = plt.subplots()
ax.plot(data)

# Show it
plt.show()
```

```
conn = sqlite3.connect('data/budget.db')
c = conn.cursor()
c.execute('''SELECT date, SUM(budget)
             FROM transactions
             GROUP BY date''')
data = c.fetchall()
conn.close()
fig, ax = plt.subplots()
ax.plot(data)
plt.show()
```

Figure 10. Full prompt and output for the example at the right of Figure 1. Above, the full generated output of the model. Below, the possible calls we extracted by postprocessing the raw output, with highlighting denoting the calls selected by R-U-SURE. (Note that for this task R-U-SURE operates on this reduced set of calls only.)

R-U-SURE (Region)

```
# Write a function to convert the given binary number to its decimal equivalent.

def binary_to_decimal(binary):
    binary1 = binary
    decimal, i, n = 0, 0, 0
    while(binary != 0):
        dec = (binary % 10) + (binary % 10) * 10 * i
        decimal = decimal + dec * (2 ** n)
        binary /= 10
        i += 1
        n += 1
    return decimal

# Driver code
binary = "1101110011111"
print(binary_to_decimal(binary))
```

Ground Truth

```
# Write a function to convert the given binary number to its decimal equivalent.

def binary_to_decimal(binary):
    binary1 = binary
    decimal, i, n = 0, 0, 0
    while(binary != 0):
        dec = binary % 10
        decimal = decimal + dec * pow(2, i)
        binary = binary//10
        i += 1
    return (decimal)
```

Test cases for intended behavior

```
assert binary_to_decimal(100) == 4
assert binary_to_decimal(1011) == 11
assert binary_to_decimal(1101101) == 109
```

Figure 11. Output of R-U-SURE compared to the ground truth for an example in the Mostly Basic Python Problems benchmark dataset (Austin et al., 2021). We manually selected a location in the MBPP reference solution, then fed the prefix to the model. The model's implementation does not exactly match the intended behavior, but all incorrect parts are highlighted. (Note: MBPP examples were not used in our main experimental results.)

Token Probability Heatmap

```
# Write a function to convert the given binary number to its decimal equivalent.

def binary_to_decimal(binary):
    binary1 = binary
    decimal, i, n = 0, 0, 0
    while(binary != 0):
        dec = (binary % 10) + (binary % 10) * 10 * i
        decimal = decimal + dec * (2 ** n)
        binary /= 10
        i += 1
        n += 1
    return decimal

# Driver code
binary = "1101110011111"
print(binary_to_decimal(binary))
```

Baseline: Token Prob 0.7

```
# Write a function to convert the given binary number to its decimal equivalent.

def binary_to_decimal(binary):
    binary1 = binary
    decimal, i, n = 0, 0, 0
    while(binary != 0):
        dec = (binary % 10) + (binary % 10) * 10 * i
        decimal = decimal + dec * (2 ** n)
        binary /= 10
        i += 1
        n += 1
    return decimal

# Driver code
binary = "1101110011111"
print(binary_to_decimal(binary))
```

Baseline: Token Prob 0.9

```
# Write a function to convert the given binary number to its decimal equivalent.

def binary_to_decimal(binary):
    binary1 = binary
    decimal, i, n = 0, 0, 0
    while(binary != 0):
        dec = (binary % 10) + (binary % 10) * 10 * i
        decimal = decimal + dec * (2 ** n)
        binary /= 10
        i += 1
        n += 1
    return decimal

# Driver code
binary = "1101110011111"
print(binary_to_decimal(binary))
```

Figure 12. Per-token conditional probability heatmap and output of token-probability-based baselines for the MBPP example in Figure 11. Note that low-conditional-prob. tokens (such as the '`/=`' after '`binary`') are frequently followed by high-conditional-prob. tokens that only make sense in context of the earlier tokens (such as '`10`').

R-U-SURE (Region)

```
# Write a python function to count number of substrings with the sum of digits equal to their length.

from collections import defaultdict
def count_Substrings(s,n):
    count,sum = 0,0
    mp = defaultdict(lambda : 0)
    mp[0] += 1
    ♦
    for i in range(0,n):
        sum += s[i]
        count += mp[sum]
        mp[sum] += 1
        ♦
    return count

print(count_Substrings("abc",4))
```

Ground Truth

```
# Write a python function to count number of substrings with the sum of digits equal to their length.

from collections import defaultdict
def count_Substrings(s,n):
    count,sum = 0,0
    mp = defaultdict(lambda : 0)
    mp[0] += 1
    for i in range(n):
        sum += ord(s[i]) - ord('0')
        count += mp[sum - (i + 1)]
        mp[sum - (i + 1)] += 1
    return count
```

Test cases for intended behavior

```
assert count_Substrings('112112',6) == 6
assert count_Substrings('111',3) == 6
assert count_Substrings('1101112',7) == 12
```

Figure 13. Output of R-U-SURE compared to the ground truth for another example in the Mostly Basic Python Problems benchmark dataset (Austin et al., 2021). We manually selected a location in the MBPP reference solution, then fed the prefix to the model. Again, the model's implementation does not exactly match the intended behavior. In this case, most incorrect parts are highlighted, but there are some changes that must also be made outside of highlighted regions. The '♦' character denotes a location where additional statements might be inserted.

Token Probability Heatmap

```
# Write a python function to count number of substrings with the sum of digits equal to their length.

from collections import defaultdict
def count_Substrings(s,n):
    count,sum = 0,0
    mp = defaultdict(lambda : 0)
    mp[0] += 1
    for i in range(1,n):
        sum += s[i]
        count += mp[sum]
        mp[sum] += 1
    return count

print(count_Substrings("abc",1))
```

Baseline: Token Prob 0.7

```
# Write a python function to count number of substrings with the sum of digits equal to their length.

from collections import defaultdict
def count_Substrings(s,n):
    count,sum = 0,0
    mp = defaultdict(lambda : 0)
    mp[0] += 1
    for i in range(0,n):
        sum += s[i]
        count += mp[sum]
        mp[sum] += 1
    return count

print(count_Substrings("abc",4))
```

Baseline: Token Prob 0.9

```
# Write a function to convert the given binary number to its decimal equivalent.

from collections import defaultdict
def count_Substrings(s,n):
    count,sum = 0,0
    mp = defaultdict(lambda : 0)
    mp[0] += 1
    for i in range(0,n):
        sum += s[i]
        count += mp[sum]
        mp[sum] += 1
    return count

print(count_Substrings("abc",4))
```

Figure 14. Per-token conditional probability heatmap and output of token-probability-based baselines for the MBPP example in Figure 13.

```
import jax.numpy as jnp
import jax
import optax
import functools

import foo.models

# Set up for training
initial_params = foo.models.initial_params()

# Configure optimizer
optimizer = optax.adamw(1e-3, b1=0.9, b2=0.999)

# Define loss function
def loss_fn(params, batch):
  logits = foo.models.apply_fn(*batch, params=params)
  loss = jnp.mean(jax.nn.sigmoid_cross_entropy_with_logits(logits=logits, labels=batch[2]))
  return loss, logits

# Create initial model
def init_model(rng_key):
  _, params = initial_params(rng_key, (-1,))
  return params

# Define update rule
@functools.partial(jax.jit, static_argnums=(0,))
def update(params, grads, state, batch):
  state, params = state
```

```
optimizer = optax.adamw(1e-3, b1=0.9, b2=0.999)
logits = foo.models.apply_fn(*batch, params=params)
loss = jnp.mean(jax.nn.sigmoid_cross_entropy_with_logits(logits=logits, labels=batch[2]))
_, params = initial_params(rng_key, (-1,))
@functools.partial(jax.jit, static_argnums=(0,))
```

Figure 15. Output of R-U-SURE (Useful Calls) for a handwritten prompt involving usage of `optax`. After postprocessing, the only calls that appear often enough in the model samples to be extracted are calls to `optax.adamw` and `jnp.mean`; these would be good candidates for preemptively showing documentation.

A.1. Runtime of R-U-SURE

The wall-clock runtime of our implementation of the R-U-SURE system depends on the number of samples as well as the complexity of the programs. We demonstrate this by measuring the runtime for the two prompts shown in Figure 7 and Figure 11, across varying number of model samples and sample lengths. On a GCP n1-standard-8 virtual machine, we obtain the following results:

- Combining eight model samples, each restricted to eight lines, takes about 20 to 60 milliseconds for the dual decomposition solver and about 1 to 1.5 seconds for the parsing and diagram construction logic.
- Combining 32 eight-line samples takes between 0.1 and 1.5 seconds for the solver and about 3 to 5 seconds for parsing/diagram construction.
- Combining 32 longer model samples (with 256 vocabulary tokens, or about 23 lines) can take between 0.5 and 6 seconds for the solver and between 8 and 40 seconds for parsing/diagram construction depending on complexity (with the example in Figure 7 taking the longest).

We note that, in our current implementation, the parsing/diagram construction logic is designed to be flexible and makes heavy use of Python dictionaries. This could likely be sped up considerably for a specialized application. The solver can also be interrupted if necessary to obtain a possibly-suboptimal solution in a fixed amount of time.

In terms of asymptotic complexity, the time and space required to build the system and each iteration of coordinate ascent scales as $O(\ell^2 K)$, where ℓ is the length of the model suggestions and K is the number of samples.

B. Decision Diagrams: Definitions and Algorithms

In this section, we discuss our definition of decision diagrams and describe how we use them to enable efficient algorithms.

B.1. Our Definitions

Definition B.1. A **(nondeterministic, weighted) binary decision diagram** (BDD) D over binary vectors $\boldsymbol{b} \in \{0,1\}^d$ is a directed acyclic graph consisting of

- a node set N,
- an arc set A,
- mappings $h : A \to N$ and $t : A \to N$ such that each arc a is directed from node $h(a)$ to node $t(a)$,
- a mapping $w : A \to \mathbb{R}$ such that $w(a)$ is the weight of arc a,
- a mapping $\alpha : A \to (\{1,\dots,d\} \times \{0,1\}) \sqcup \{\text{NONE}\}$ such that, if $\alpha(a) = (i,\beta)$, then this edge can only be used when $\boldsymbol{b}_i = \beta$, and if $\alpha(a) = \text{NONE}$, this edge can always be used.
- a source node $\top \in N$, which is not the tail of any arc,
- a sink node $\bot \in N$, which is not the head of any arc.

Definition B.2. A **computation path** for a binary vector $\boldsymbol{b} \in \{0,1\}^d$ is a sequence of arcs $P = (a_1, a_2, \dots, a_n)$ from $\top$ to $\bot$ that are consistent with $\boldsymbol{b}$, e.g. such that $h(a_1) = \top$, $h(a_{i+1}) = t(a_i)$ for $1 \le i < n$, $t(a_n) = \bot$, and if $\alpha(a_i) = (j,\beta)$ for any i then $\boldsymbol{b}_j = \beta$. The weight of this path is the sum of arc weights $\sum_i w(a_i)$, which by abuse of notation we will denote $w(P)$. We denote the set of all computation paths for a particular vector $\boldsymbol{b}$ as $\mathcal{P}(D, \boldsymbol{b})$.

Definition B.3. A BDD D **represents** a binary function $w : \{0,1\}^d \to \mathbb{R} \cup \{-\infty\}$ (under $\max$-aggregation) if, for all $\boldsymbol{b} \in \{0,1\}^d$, we have

$$w(\boldsymbol{b}) = \max_{P \in \mathcal{P}(D,\boldsymbol{b})} w(P),$$

e.g. this is the weight of the maximum-weight path from $\top$ to $\bot$ consistent with $\mathcal{P}$, or $-\infty$ if there are no such paths.

Definition B.4. A BDD D is **ordered** if its nodes can be partitioned into layers according to some partition function $\ell : N \to \{0,1,\dots,d\}$ such that $\ell(\top) = 0$, $\ell(\bot) = d$, and for each arc $a \in A$:

- if $\alpha(a) = \text{NONE}$, then $\ell(h(a)) = \ell(t(a))$,
- if $\alpha(a) = (i,\beta)$, then $\ell(h(a)) = i - 1$ and $\ell(t(a)) = i$.

Intuitively, an ordered BDD is a BDD such that any path from $\top$ to $\bot$ assigns every index of $\boldsymbol{b}$ exactly once, in order of increasing index.

Definition B.5. A **system of BDDs** is a collection of BDDs D_i over the same set of binary vectors $\boldsymbol{b} \in \{0,1\}^d$. We say that a system of BDDs represents a binary function $w : \{0,1\}^d \to \mathbb{R} \cup \{-\infty\}$ if w can be written as a sum

$$w(\boldsymbol{b}) = \sum_i w^{(i)}(\boldsymbol{b})$$

and D_i represents $w^{(i)}$ for each i.

Our approach described in Section 3.3 can now be described more specifically as rewriting our original objective using a set of binary functions

$$w^{(k)}(\boldsymbol{b}) = \begin{cases} \frac{1}{K} u(g^{(k)}, f(\boldsymbol{b})) & \boldsymbol{b} \in \mathcal{B}, \\ -\infty & \text{otherwise.} \end{cases}$$

and then representing each such function with an ordered BDD. More generally, we allow representing $w^{(k)}(\boldsymbol{b})$ as a system of BDDs $(D_1^{(k)}, D_2^{(k)}, \dots, D_m^{(k)})$, and take advantage of this flexibility to efficiently separate the computation of the utility

function from constraints, which we describe in more detail in Appendix C. Combining all of the BDDs or BDD systems for each value of k then yields a system $(D_1, \ldots, D_J)$ that represents the total utility

$$U(\boldsymbol{b}) = \sum_{k=1}^{K} w^{(k)}(\boldsymbol{b})$$

as estimated across the samples $g^{(1)}, \ldots, g^{(k)}$. (Note that the number K of samples may or may not match the number of decision diagrams J in general, depending on whether any of the $w^{(k)}$ were represented as more than one diagram).

We note that any function $w^{(k)} : \mathcal{B} \to \mathbb{R}$ can be expressed as a weighted binary decision diagram, but the size of the diagram may grow exponentially with the number of binary choices d (Hooker, 2013). However, our edit-distance-based utility functions $u(g, s)$ can be represented as decision diagram whose size grows only quadratically with the number of tokens in s and g, due to the similarity between decision diagrams and dynamic programming algorithms.

B.2. A comparison to other definitions of decision diagrams

Decision diagrams have seen a number of uses for a variety of combinatorial optimization and search problems; Castro et al. (2022) gives an overview of many such uses. Here we briefly summarize some of the differences between our definition and others in the literature.

Determinism Many definitions of decision diagrams (e.g. Lozano et al. (2020); Lange & Swoboda (2021)) focus on *deterministic* decision diagrams, which have the additional properties that

- every node n other than $\perp$ is associated with a particular decision variable $\boldsymbol{b}_{\text{var}(n)}$ with $\text{var}(n) \in \{1, \ldots, d\}$,
- there are at most two edges from any given node n (i.e. with $h(a) = b$): one which assigns $\alpha(a) = (\text{var}(n), 0)$ and one which assigns $\alpha(a) = (\text{var}(n), 0)$,
- every arc assigns some variable, e.g. there is no edge with $\alpha(a) = \textsc{None}$.

Nondeterministic decision diagrams are related to deterministic ones in the same way that nondeterministic finite automata relate to deterministic finite automata: for a deterministic decision diagram, you can read off a single computation path P for a given vector $\boldsymbol{b}$ if it exists by following the sequence of branches, whereas for a nondeterministic decision diagram, you may need to search over many consistent sub-paths to identify one or more computation paths for a specific vector.

Some definitions of nondeterministic decision diagrams define them by introducing two types of node: ordinary nodes, which are associated with variables have two outgoing arcs tagged 0 and 1, and nondeterministic nodes, which have no variable and any number of outgoing arcs with $\alpha(a) = \textsc{None}$ (Bollig & Buttkus, 2018). For simplicity, our definition does not directly constrain edges based on any assignment of nodes to decision variables, but the two formulations are equivalently expressive, especially for ordered nondeterministic BDDs (for which ℓ approximately corresponds to a node-variable association).

Ambiguity Most definitions of nondeterministic BDDs focus on *unambiguous* nondeterministic BDDs, for which there is at most one computation path for any binary vector $\boldsymbol{b}$ (Bollig & Buttkus, 2018); these can also be referred to as *exactly representing* specific binary functions (Castro et al., 2022). In contrast, we explicitly allow BDDs to be ambiguous, and resolve conflicts by taking the max over edges. This makes it significantly easier to express our utility functions as decision diagrams, by essentially interleaving the edit-distance search algorithm with the decision diagram as part of a single optimization problem.

It turns out to be very straightforward to extend the min-(or max-)marginal averaging technique of Lange & Swoboda (2021) to work for ambiguous decision diagrams with only minimal changes, as we describe in the next section.

Reduction A common method for obtaining more efficient representations of decision diagrams is to reduce them to a particular canonical form, collapsing nodes that serve identical roles (Hooker, 2013; Castro et al., 2022). While it may be possible to reduce our decision diagrams to a more efficient form, we do not attempt to produce reduced decision diagrams in our implementation.

Binary v.s. multivalued Some definitions of decision diagrams allow variables to be assigned to values in a larger finite set $\mathcal{V}$; these are known as multivalued decision diagrams (Hooker, 2013; Castro et al., 2022). In practice, we implement our utility functions as multivalued decision diagrams; however, to make derivations simpler for the Lagrangian relaxation, we encode these multivalued choices as one-hot-encoded binary vectors before running our max-marginal optimization process.

B.3. Efficient algorithms for max-marginal message passing on BDDs

We now describe how to efficiently optimize a Lagrangian relaxation of a BDD system, as described in Sections 3.2 and 3.3. We consider the objective

$$U = \max_{\boldsymbol{b} \in \{0,1\}^d} \sum_{j=1}^{J} w^{(j)}(\boldsymbol{b}), \tag{8}$$

where we have changed the indexing to account for situations where the number of decision diagrams J does not equal the number of model samples K. We then construct the Lagrangian relaxation

$$W^{(j)}(\boldsymbol{b}^{(j)}, \boldsymbol{\lambda}) = w^{(j)}(\boldsymbol{b}^{(j)}) + \boldsymbol{\lambda}^{(j)} \cdot \boldsymbol{b}^{(j)}, \tag{9}$$

$$L(\boldsymbol{\lambda}^{(1:J)}) = \sum_{j=1}^{J} \max_{\boldsymbol{b}^{(j)}} W^{(j)}(\boldsymbol{b}^{(j)}, \boldsymbol{\lambda}) \tag{10}$$

where we require that $\sum_m \boldsymbol{\lambda}^(j) = 0$. Intuitively, if we have $\boldsymbol{b}_i^{(j)} \neq \boldsymbol{b}_i^{(j')}$, we can adjust the $\boldsymbol{\lambda}_i^{(j)}$ and $\boldsymbol{\lambda}_i^{(j')}$ in opposite directions to remove any utility benefits of violating the equality constraint. (However, this penalty acts independently on each variable, and may not be able to simultaneously enforce agreement for joint configurations of variables; this is what causes a nonzero duality gap.) We note that this is the same relaxation described by Lange & Swoboda (2021, Section 3.1), except for the more general form of $w^{(j)}$ (not just for linear programs) and the use of $\max$ rather than $\min$.

The max-marginal coordinate ascent update with respect to the variable block $(\lambda_i^{(1)}, \lambda_i^{(2)}, \ldots, \lambda_i^{(K)})$, as derived by Lange & Swoboda (2021), is then given by

$$m_{i:=\beta}^{(j)} = \max_{\boldsymbol{b}^{(j)} \text{ s.t. } \boldsymbol{b}_i^{(j)}=\beta} W^{(j)}(\boldsymbol{b}^{(j)}, \boldsymbol{\lambda}), \qquad \beta \in \{0,1\} \tag{11}$$

$$\boldsymbol{\lambda}_i^{(j)} \leftarrow \boldsymbol{\lambda}_i^{(j)} - (m_{i:=1}^{(j)} - m_{i:=0}^{(j)}) + \frac{1}{J} \sum_{j'} \left(m_{i:=1}^{(j')} - m_{i:=0}^{(j')} \right). \tag{12}$$

Our main requirement for computing this update is that we can efficiently compute the $m_{i:=\beta}^{(j)}$ for our current values of $\boldsymbol{\lambda}$. Fortunately, this can be done for nondeterministic weighted BDDs using a straightforward dynamic programming algorithm. This algorithm maintains two cached dynamic programming tables (PREFIX and SUFFIX) in order to make updates efficient: PREFIX stores the maximum weight from $\top$ to a given node (sorted by level ℓ), and SUFFIX stores the maximum weight from each node to $\bot$. We initialize these tables using Algorithm 3, then run Algorithm 5 to compute desired max marginals. Then, each time we update values for $\boldsymbol{\lambda}_i^{(1:J)}$, we must invalidate the caches for index i by running Algorithm 4.

A key property of this algorithm is that modifying the dual variables for a particular decision variable $\boldsymbol{b}_i^{(j)}$ only affects prefixes and suffixes that include assignments to $\boldsymbol{b}_i^{(j)}$. Thus, if we wish to compute max-marginals for $\boldsymbol{b}_{i-1}^{(j)}$ or $\boldsymbol{b}_{i+1}^{(j)}$ next, we can reuse almost all of the values from the cache, and only update the prefixes that changed due to modifications to $\boldsymbol{\lambda}_i^{(j)}$.

We take advantage of this property by running a series of alternating forward and backward sweeps, updating $\boldsymbol{\lambda}_1^{(1:J)}, \boldsymbol{\lambda}_2^{(1:J)}, \ldots, \boldsymbol{\lambda}_d^{(1:J)}$ during a forward sweep and then $\boldsymbol{\lambda}_d^{(1:J)}, \boldsymbol{\lambda}_{d-1}^{(1:J)}, \ldots, \boldsymbol{\lambda}_1^{(1:J)}$ in a backward sweep. Each of these sweeps visits every arc twice (once to compute max marginals and once to update the modified prefix or suffix), enabling us to run an entire min-marginal-averaging cycle with time complexity proportional to the size of the decision diagram.

Note that this algorithm is not guaranteed to find a primal solution if there is a nonzero dual gap, and may get stuck in certain fixed points even if the dual gap is zero (Werner et al., 2020). In our experiments, however, we find that the bound is tight (to within machine precision) over 85% of the time.

Algorithm 3 Stateful dynamic programming algorithm initialization step

Input: BDD $D_j = (N, A, h, t, w, \alpha, \top, \bot)$ with order ℓ, caches PREFIX and SUFFIX
 # Compute initial prefixes
Initialize PREFIX$[0, \top] = 0.0$
for each arc $a \in A$ with $\ell(h(a)) = \ell(t(a)) = 0$), in topologically-sorted order **do**
  Set PREFIX$[k, t(a)] = \max($PREFIX$[k, t(a)]$, PREFIX$[k, h(a)] + w(a))$
end for
 # Compute initial suffixes
Initialize SUFFIX$[d, \top] = 0.0$
for each arc $a \in A$ with $\ell(h(a)) = \ell(t(a)) = d$), in reverse topologically-sorted order **do**
  Set SUFFIX$[k, h(a)] = \max($SUFFIX$[k, h(a)]$, SUFFIX$[k, t(a)] + w(a))$
end for

Algorithm 4 Stateful dynamic programming cache invalidation step

Input: BDD $D_j = (N, A, h, t, w, \alpha, \top, \bot)$ with order ℓ, updated index i, caches PREFIX and SUFFIX
for k **in** $[i, i+1, \ldots, d]$ **do**
  Delete all entries of PREFIX$[k, :]$
end for
for k **in** $[0, 1, \ldots, i-1]$ **do**
  Delete all entries of SUFFIX$[k, :]$
end for

B.4. Extension to multivalued decision diagrams

In practice, although we analyze and implement our algorithms as if we are optimizing over binary variables, it is more convenient for our utility functions to be written in terms of assignments to an arbitrary finite set of values $\mathcal{V}$; this is sometimes known as a "multivalued" decision diagram (Hooker, 2013). We do this by enumerating the values of $\mathcal{V}$, and treating a particular choice $x_i = v \in \mathcal{V}$ as a collection of "indicator" assignments $\boldsymbol{b}_{(i,v)} := 1$, $\boldsymbol{b}_{(i,v')} := 0$ for $v' \neq v$.

We take advantage of our knowledge of this indicator structure when running our max-marginal step, to simplify the implementation. In particular, we perform simultaneous block updates over all indicator variables, computing

$$m^{(j)}_{(i,v):=1} = \max_{\boldsymbol{b}^{(j)} \text{ s.t. } \boldsymbol{b}^{(j)}_{(i,v)}=1,} W^{(j)}(\boldsymbol{b}^{(j)}, \boldsymbol{\lambda}) \tag{13}$$

$$\boldsymbol{\lambda}^{(j)}_{(i,v)} \leftarrow \boldsymbol{\lambda}^{(j)}_{(i,v)} - m^{(j)}_{(i,v):=1} + \frac{1}{J} \sum_{j'} m^{(j')}_{(i,v):=1}. \tag{14}$$

which is the update from Equation (12) but dropping the $m^{(j)}_{(i,v):=0}$ terms. This works because we know that, for any valid assignment to the indicator variables, exactly one such indicator will be active. Thus, each of the $m^{(j)}_{(i,v):=0}$ terms is equal to $m^{(j)}_{(i,v'):=1}$ for some alternative assignment v', which means making the $m^{(j)}_{(i,v'):=1}$ agree is sufficient to make the differences $m^{(j)}_{(i,v):=1} - m^{(j)}_{(i,v):=0}$ agree as well. (Indeed, in our actual implementation of Algorithm 5, we do not bother computing entries for the $m^{(j)}_{(i,v):=0}$ at all, since they are unused in the update Equation (14).)

This indicator representation also allows us to reuse parts of our implementation when decoding a heuristic primal solution, in the situations where our solver fails to find a setting for the dual variables that makes the dual bound tight. Specifically, we iterate through all of the variables, and greedily select the best assignment

$$v^*_i = \arg\max m^{(j)}_{(i,v):=1}$$

then set

$$\boldsymbol{\lambda}^{(j)}_{(i,v')} \leftarrow -\infty$$

for each $v' \neq v^*_i$. This effectively prunes any arc that assigns a different value from the graph, ensuring we decode a single consistent assignment.

Algorithm 5 Stateful dynamic programming algorithm for $m^{(j)}_{i:=\beta}$

Input: BDD $D_j = (N, A, h, t, w, \alpha, \top, \bot)$ with order ℓ, desired variable index i, $\boldsymbol{\lambda}^{(j)}$, caches PREFIX and SUFFIX
 # Compute necessary prefixes
for k **in** $[1, 2, \ldots, i-1]$ **do**
  if PREFIX does not have values for level k **then**
    for each node $n \in N$ with $\ell(n) = k$ **do**
      Initialize PREFIX$[k, n] = -\infty$
    end for
    # Process edges that assign $\mathbf{b}_k$
    for each arc $a \in A$ with $\ell(h(a)) = k-1$ and $\ell(t(a)) = k$, in topologically-sorted order **do**
      Let $(v, \beta) = \alpha(a)$, assert $v = k$ *# a must assign to* $\mathbf{b}_k$ *by Definition B.4*
      if $\beta = 1$ **then** *# Need to perturb by* $\boldsymbol{\lambda}^{(j)}_k$
        Set PREFIX$[k, t(a)] = \max($PREFIX$[k, t(a)]$, PREFIX$[k-1, h(a)] + w(a) + \boldsymbol{\lambda}^{(j)}_k)$
      else
        Set PREFIX$[k, t(a)] = \max($PREFIX$[k, t(a)]$, PREFIX$[k-1, h(a)] + w(a))$
      end if
    end for
    # Process edges in level k
    for each arc $a \in A$ with $\ell(h(a)) = k$ and $\ell(t(a)) = k$, in topologically-sorted order **do**
      Set PREFIX$[k, t(a)] = \max($PREFIX$[k, t(a)]$, PREFIX$[k, h(a)] + w(a))$
    end for
  end if
end for
 # Compute necessary suffixes
for k **in** $[d-1, d-2, \ldots, i]$ **do**
  if SUFFIX does not have values for level k **then**
    for each node $n \in N$ with $\ell(n) = k$ **do**
      Initialize SUFFIX$[k, n] = -\infty$
    end for
    # Process edges that assign $\mathbf{b}_{k+1}$
    for each arc $a \in A$ with $\ell(h(a)) = k$ and $\ell(t(a)) = k+1$, in reverse topologically-sorted order **do**
      Let $(v, \beta) = \alpha(a)$, assert $v = k+1$ *# a must assign to* $\mathbf{b}_{k+1}$ *by Definition B.4*
      if $\beta = 1$ **then** *# Need to perturb by* $\boldsymbol{\lambda}^{(j)}_{k+1}$
        Set SUFFIX$[k, h(a)] = \max($SUFFIX$[k, h(a)]$, SUFFIX$[k+1, t(a)] + w(a) + \boldsymbol{\lambda}^{(j)}_{k+1})$
      else
        Set SUFFIX$[k, h(a)] = \max($SUFFIX$[k, h(a)]$, SUFFIX$[k+1, t(a)] + w(a))$
      end if
    end for
    # Process edges in level k
    for each arc $a \in A$ with $\ell(h(a)) = k$ and $\ell(t(a)) = k$, in reverse topologically-sorted order **do**
      Set SUFFIX$[k, t(a)] = \max($SUFFIX$[k, t(a)]$, SUFFIX$[k, h(a)] + w(a))$
    end for
  end if
end for
 # Compute max marginals
Initialize $m^{(j)}_{i:=0}$ and $m^{(j)}_{i:=1}$ to $-\infty$
for each arc $a \in A$ with $\ell(h(a)) = i-1$ and $\ell(t(a)) = i$ **do**
  Let $(v, \beta) = \alpha(a)$, assert $v = i$ *# a must assign to* $\mathbf{b}_i$ *by Definition B.4*
  Set $m^{(j)}_{i:=\beta} = \max(m^{(j)}_{i:=\beta}$, PREFIX$[i-1, h(a)] + w(a) +$ SUFFIX$[i, t(a)] + \boldsymbol{\lambda}^{(j)}_i)$
end for
return $m^{(j)}_{i:=0}, m^{(j)}_{i:=1}$

C. Utility functions and tree representation

In this section, we give a high level description of our utility function implementation and of the tree representation we use for combining suggestions. We will also include the code for our utility functions in a later open-source release.

C.1. Tree representation

We represent the model samples and user intents as possibly-nested sequences of nodes of the following types:

- **Token nodes** represent programming language tokens, which we should try to match between the suggestion and the target intent. Token nodes contain a source string and optionally a type, and any two nodes with the same string and the same type will match. We typically use the type to encode information about the AST nodes.
- **Decoration nodes** denote locations of whitespace or other aspects of the suggestion that do not need to be considered as part of the edit distance calculation. These are not used during optimization.
- **Group nodes** contain an arbitrary number of child nodes, which may be token nodes, decoration nodes, or other group nodes. Each group node has an optional type, and any two group nodes of the same type can be matched together; matching two group nodes involves running an edit distance calculation on their children subsequences.

The suggestion prototype, usually the model sample with the highest probability, is augmented with a few additional nodes:

- **Region start nodes** represent locations where we may start a confidence region. Depending on the configuration, such regions may represent pockets of UNSURE within a default of SURE (e.g. for detecting edit locations), or pockets of SURE within a default of UNSURE (e.g. for extracting a subsequence of API calls).
- **Region end nodes** represent locations where we may end a confidence region that we started earlier in the (sub)sequence. Note that every confidence region that starts inside a group node is required to end within that same group node.
- **Truncation nodes** represent locations where we may decide to truncate the suggestion.

These nodes are inserted in various locations into the parse tree with a preprocessing step, which gives us a large amount of control over the space of augmented suggestions $\mathcal{S}$. For instance, for the edit localization task, we do not allow UNSURE regions to include single parentheses or brackets by placing matched brackets into a group and not allowing regions to start or end at the boundary of those groups. For the API call task, we use the region start/end nodes to identify SURE calls, but only allow calls to be selected one at a time by only inserting them inside the relevant call groups.

C.2. Utility function

We now describe our base utility function at a high level; the specific applications are determined by configuring this utility function with different costs and constraints.

C.2.1. UTILITY CONFIGURATION

Our utility function implementation is configured by a set of edit penalties:

- For each confidence level:
  - A per-character or per-token utility for matching tokens in the suggestion with those in the ground truth,
  - A per-character or per-token cost for deleting tokens in the suggestion
  - A penalty for starting to edit (either inserting or deleting)
- A penalty for changing confidence levels (e.g. to encourage fewer blocks of UNSURE).

C.2.2. EDIT-BASED DECISION DIAGRAM

Nodes in our decision diagram (which we call "states" to distinguish them from tree nodes, by analogy to finite state machines) are associated with a tuple of positions, one in the prototype and one in the hypothetical target intent (usually generated from the model), in a similar way as in Algorithm 2.

We further group our states into a number of types, used to track the progress of edits. The list of state types are:

- **PROCESS-PROTOTYPE (ADVANCE)**: We are advancing past region start/end nodes or truncation nodes in the prototype. We can either stay in PROCESS-PROTOTYPE and move past one of those nodes in the prototype, or transition to MATCH to match tokens or groups, or transition to MAY-DELETE if we need to edit at this location, which incurs an additional penalty.
- **MAY-DELETE**: We have decided we need to make an edit at this location. We are allowed to delete an arbitrary number of the prototype; we may also process any region start/end nodes or truncation nodes we see. We then transition into MAY-INSERT.
- **MAY-INSERT**: We are allowed to insert an arbitrary number of nodes in the target. We always insert after deleting, to reduce the number of redundant paths in the graph. Once we have inserted all that we need to, we can transition to MATCH.
- **MATCH**: We are prepared to match nodes in the prototype and target, after which we return to PROCESS-PROTOTYPE; we can also end the subproblem if we are at the end of two group nodes.
- **RECURSIVELY-DELETING (FORCED)**: We have committed to deleting an entire subtree, and are now deleting each of the nodes in it. We cannot stop until we exit the subtree.
- **RECURSIVELY-INSERTING (FORCED)**: We have committed to inserting an entire subtree, and are now inserting each of the nodes in it. We cannot stop until we exit the subtree.

Additionally, each node is associated with a confidence level (SURE or UNSURE); the active confidence level determines the utility associated with each of the state transitions described above.

Token nodes are handled depending on the state; in MATCH we must align two identical tokens to proceed, whereas in MAY-DELETE or MAY-INSERT we are allowed to delete or insert tokens individually.

Group nodes are handled using a recursive call. If we are processing two group nodes and we are in the MATCH state, we recursively build a decision diagram for the subsequences of the two nodes. If we delete a group node in the MAY-DELETE state, we call a recursive helper function that builds a small decision diagram that only deletes nodes and stays in the RECURSIVELY-DELETING state. Inserts are handled in an analogous way.

We implicitly embed the space of suggestions $\mathcal{S}(g^{(1)})$ into a space of binary vectors by introducing decisions for each of the control nodes. Here we focus on the version of our task that introduces UNSURE regions into a suggestion.

- At a Region Start node, if we are currently in SURE, we can transition to UNSURE. We track this choice with a decision variable assignment.
- At a Region End node, if we are currently in UNSURE, we can transition to SURE. We track this choice with a decision variable assignment.
- At a truncation node, we can choose to immediately jump from our current state to the final state, paying no more penalties but receiving no additional reward. We track this choice with a decision variable assignment.

We additionally include decision variables that track whether each token was inside a annotated region when we processed it; this information is redundant with the start/end nodes, but can improve the optimization by providing additional information in the message passing iterations. We then order these decision variables by their order of appearance in the graph, and interpret the values of each decision as the embedding $\phi(s)$ of each possible suggestion.

Figure 16 shows a rendering of the decision diagram we construct when combining two simple sequences. Note that the diagrams we use to combine actual model samples are much larger, since every token of the suggestion is represented by multiple states in the diagram. Also, this diagram is written in terms of negative utility (e.g. as a collection of costs).

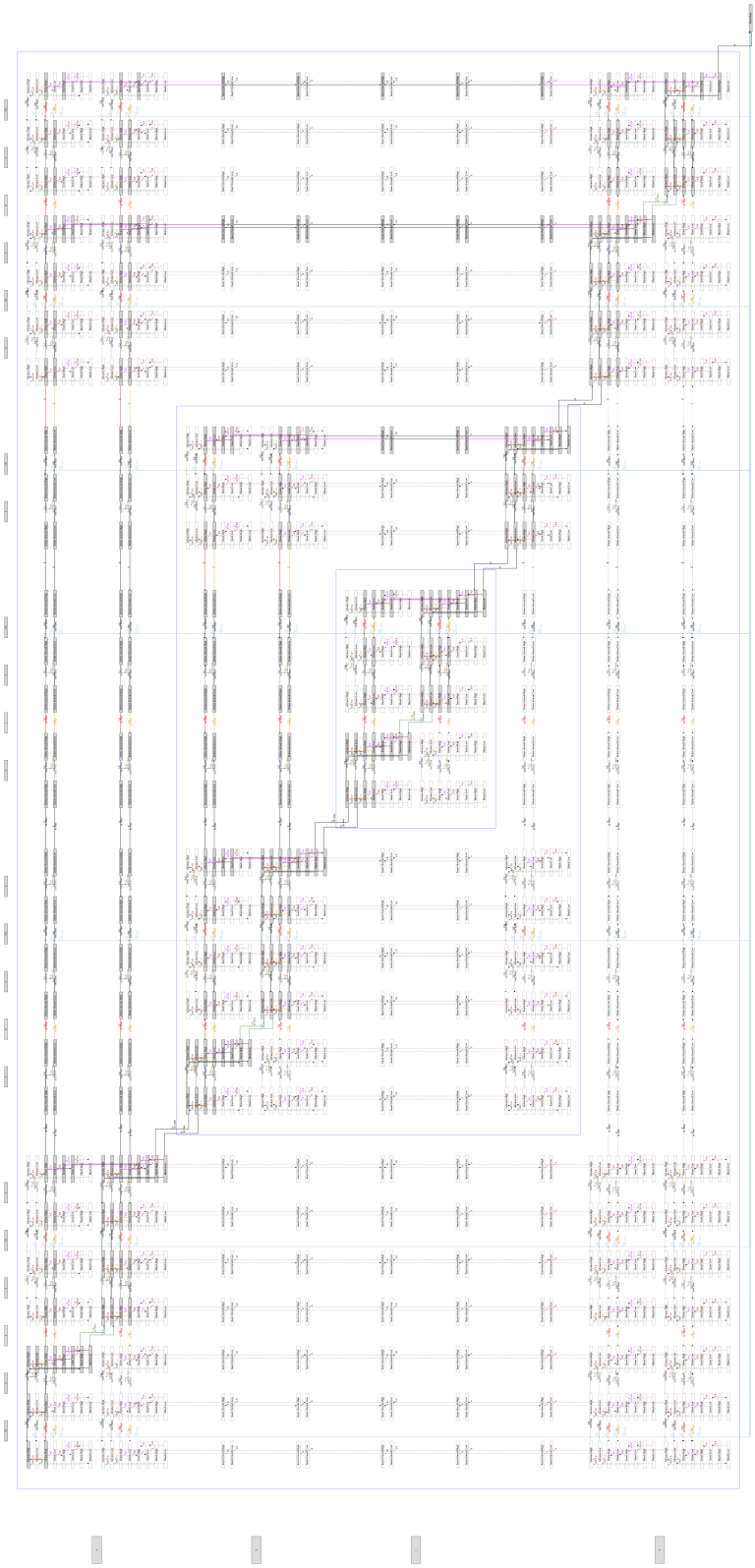


Figure 16. Rendering of the edit decision diagram with all features enabled, for aligning the sequence “a [b [c]] d” with itself.

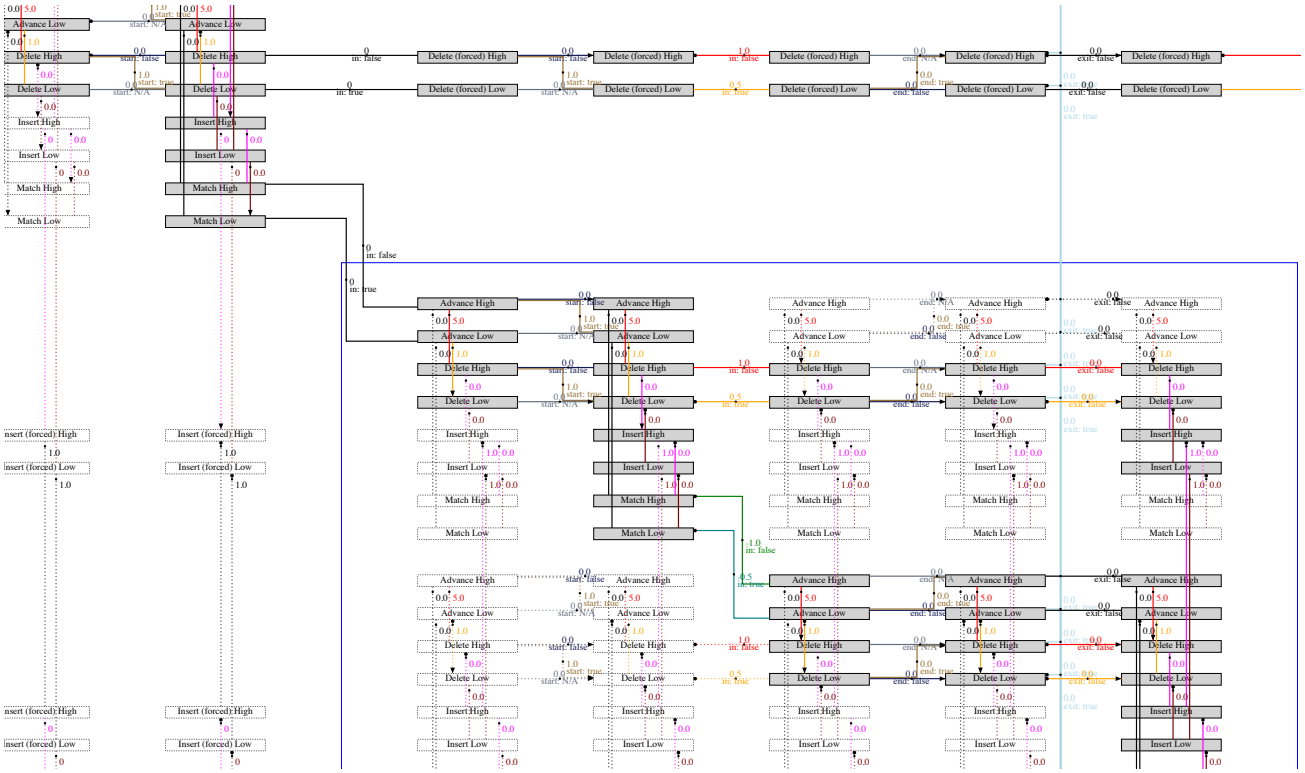


Figure 17. Zoomed-in view of a portion of the edit decision diagram in Figure 16

C.2.3. Constraint Decision Diagram

The above decision diagram ensures that edits respect the tree structure, but does not by itself ensure that annotated regions are aligned with that tree structure. We address this by building a second decision diagram, which depends only on the prototype sequence and which enforces the constraints on the annotated regions.

The second DAG tracks a more fine-grained set of confidence types:

- **OUTSIDE-REGION**: We are outside of any annotated region.
- **IN-REGION-TEMPORARY**: We are inside a annotated region that we started at the current nesting level.
- **IN-REGION-FORCED**: We are inside a annotated region that we started at a previous nesting level (e.g. we started it and then entered a group node subproblem).

Instead of a tuple of positions in the prototype and in target, we track a tuple of a position in the prototype and a "confidence nesting level", which represents how many ancestors of this node are in high-confidence regions rather than low-confidence regions. This allows us to keep track of how many group nodes we must exit before we are allowed to stop a low-confidence region.

Figure 18 shows a rendering of the decision diagram we construct when combining two simple sequences. Note that the utility of this diagram is zero along any path; the purpose of this diagram is to forbid certain subsets of variable assignments (e.g assign them negative utility, or infinite cost).

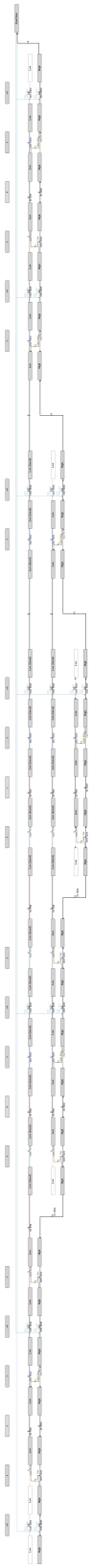


Figure 18. Rendering of the constraint decision diagram with all features enabled, for enforcing constraints in the sequence “a [b [c]] d”.

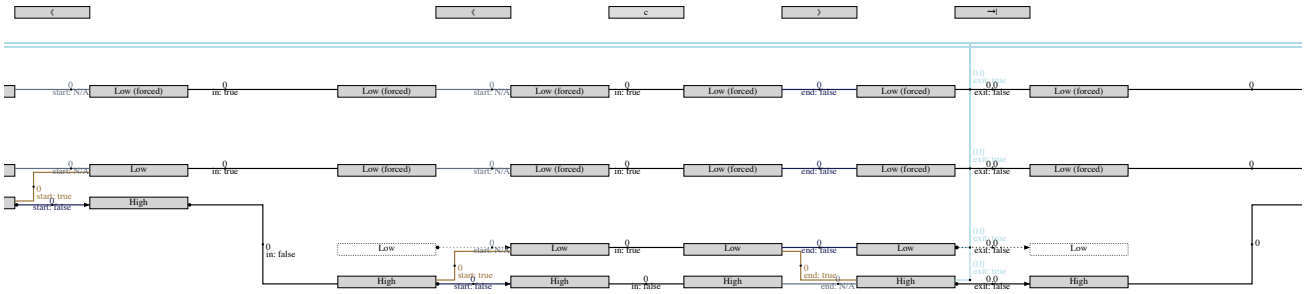


Figure 19. Zoomed-in view of a portion of the constraint decision diagram in Figure 18. “Low” refers to being in an annotated region, which corresponds to low-confidence UNSURE annotations for the edit localization task.

D. Overview of our Pseudo-Parser

We now provide a high-level overview of our pseudo-parser, which converts code fragments into abstract syntax tree (AST) like structures. Pairs of these pseudo-parse trees are used by R-U-SURE to construct matching graphs that parameterize the abstract space over which R-U-SURE searches for minimum Bayes risk solutions. By representing the source code by a syntactically meaningful tree structure, it is possible for R-U-SURE to produce completion results that respect the nature of source code and are especially syntactically meaningful.

Although a complete and precise specification of our pseudo-parsing algorithm is beyond the present scope, the full details will be available soon in our upcoming code release. In anticipation of that release, we now provide a high level description of our pseudo-parser and give some illustrative examples.

D.1. High Level Desiderata

We developed our bespoke pseudo-parser with two main goals in mind:

- Language independence: our system handles JAVA, JAVASCRIPT and C++ using a unified approach that requires only a handful of language specific parameters. PYTHON is handled similarly, but with some additional complexity due to nature of indents and dedents in that language.
- Error tolerance: we cannot assume syntactically valid code.

D.2. Algorithms

Tokenization Before pseudo-parsing, we convert source code text into a sequence of tokens that we identify by regular expression matching. For example, the following code fragment:

```
y = func(x)
```

is split into the following (token type, token) pairs:

```
ID           "y"
WHITE_SPACE  " "
PUNC         "="
WHITE_SPACE  " "
ID           "func"
BRACE        "("
ID           "x"
BRACE        ")"
NEWLINE      "\n"
```

Basic Bracket Matching Following tokenization comes the core part of our pseudo-parser, a *bracket-matching* algorithm that produces a nested structure. For the example above, this may be rendered as below. While the full details of the following rendering are not important, the nesting denoted by indentation clearly reveals relevant structure:

```
GROUP(ROOT): "y = func(x)\n"
  GROUP(SPLIT_GROUP): "y = func(x)\n"
    TOK(CONTENT_LEAF): "y"
    DEC: " "
    TOK(CONTENT_LEAF): "="
    DEC: " "
    TOK(CONTENT_LEAF): "func"
    GROUP(MATCH): "(x)"
      TOK(MATCH_LEFT): "("
      GROUP(MATCH_INNER): "x"
        TOK(CONTENT_LEAF): "x"
      TOK(MATCH_RIGHT): ")"
    DEC: "\n"
```

Error Correction How to handle sequences with unmatched brackets? In simple cases, missing closing brackets can be added to restore balance and recover relevant structure, for example this code:

```
(x
```

results in the following tree structure:

```
GROUP(ROOT): "(x\n)\n"
  GROUP(SPLIT_GROUP): "(x\n)\n"
    GROUP(MATCH): "(x\n)"
      TOK(MATCH_LEFT): "("
      GROUP(MATCH_INNER): "x\n"
        TOK(CONTENT_LEAF): "x"
        DEC: "\n"
      TOK(MATCH_RIGHT): ")"
    DEC: "\n"
```

which includes an additional closing brace.

Error Tolerance In some cases, such as the following:

```
(x])
```

no correction is made, and the erroneous brace (in this case the right square brace) is treated as a regular token, yielding

```
GROUP(ROOT): "(x])\n"
  GROUP(SPLIT_GROUP): "(x])\n"
    GROUP(MATCH): "(x])"
      TOK(MATCH_LEFT): "("
      GROUP(MATCH_INNER): "x]"
        TOK(CONTENT_LEAF): "x"
        TOK(CONTENT_LEAF): "]"
      TOK(MATCH_RIGHT): ")"
    DEC: "\n"
```

Handling PYTHON indents and dedents Unlike C++, JAVA and JAVASCRIPT, which use curly brackets, the PYTHON language uses white-space to denote code blocks. To handle this, we apply our pseudo parser twice. In the first step, we match standard brackets, so that

```
def f(
  x, y):
  return x
```

is parsed as

```
GROUP(ROOT): "def f(\n  x, y):\n  return x"
  TOK(CONTENT_LEAF): "def"
  DEC: " "
  TOK(CONTENT_LEAF): "f"
  GROUP(MATCH): "(\n  x, y)"
    TOK(MATCH_LEFT): "("
    GROUP(MATCH_INNER): "\n  x, y"
      DEC: "\n"
      DEC: "  "
      TOK(CONTENT_LEAF): "x"
      TOK(CONTENT_LEAF): ","
```

```
      DEC: " "
      TOK(CONTENT_LEAF): "y"
    TOK(MATCH_RIGHT): ")"
  TOK(CONTENT_LEAF): ":"
  DEC: "\n"
  DEC: "  "
  TOK(CONTENT_LEAF): "return"
  DEC: " "
  TOK(CONTENT_LEAF): "x"
```

from which we determine (using a specific algorithm that works with the above tree representation), that since the newline and subsequent white-space following the opening bracket is contained within a matched bracket pair, it is not to be treated as a python code block indent. Once we have detected what do appear to be valid python code block indents and dedents, we handle them with a second pass of our error tolerant bracket matching pseudo-parser, which in this case gives the result:

```
GROUP(ROOT): "def f(\n  x, y):\n  return x\n"
  GROUP(SPLIT_GROUP): "def f(\n  x, y):\n  return x\n"
    GROUP(SPLIT_GROUP): "def f(\n  x, y):\n"
      TOK(CONTENT_LEAF): "def"
      DEC: " "
      TOK(CONTENT_LEAF): "f"
      GROUP(MATCH): "(\n  x, y)"
        TOK(MATCH_LEFT): "("
        GROUP(MATCH_INNER): "\n  x, y"
          DEC: "\n"
          DEC: "  "
          TOK(CONTENT_LEAF): "x"
          TOK(CONTENT_LEAF): ","
          DEC: " "
          TOK(CONTENT_LEAF): "y"
        TOK(MATCH_RIGHT): ")"
      TOK(CONTENT_LEAF): ":"
      DEC: "\n"
    GROUP(SPLIT_GROUP): "  return x\n"
      GROUP(MATCH): "  return x"
        TOK(MATCH_LEFT): ""
        GROUP(MATCH_INNER): "  return x"
          DEC: "  "
          TOK(CONTENT_LEAF): "return"
          DEC: " "
          TOK(CONTENT_LEAF): "x"
        TOK(MATCH_RIGHT): ""
      DEC: "\n"
```

in which the matching python indents and dedents are denoted by empty strings.

Subtokenization of string literals To allow fine-grained edits within strings (such as docstrings), we further subtokenize tokens identified as string literals. This subtokenization process uses a generic lossless tokenizer originally designed by Kanade et al. (2019) and made available at https://github.com/google-research/google-research/tree/master/cubert/unified_tokenizer.py.

E. Additional Details on the Experimental Methodology

E.1. Example Generation

In this section we provide further details on the example generation methodology introduced in Section 5.1. An example is defined by

1. The choice of source code file from which to derive the example. We use permissively licensed code from scientific computing repositories hosted on GITHUB[1].
2. The starting (or cursor) location at which the hypothetical completion should begin, which is an index into the characters of the raw source code file.
3. The truncation point, which is the assumed ending location of both the ground truth target (taken from the original source file, and used for evaluation but not seen by R-U-SURE), and each of the $K = 31$ continuations that are samples drawn from the language model (and are used to form the minimum Bayes risk objective of R-U-SURE defined in Equation (3)).

The first two example types, applicable to all four programming languages that we consider, choose the starting location uniformly at random from the source code file, and only differ by their choice of truncation point as follows.

1. For the **untruncated target** setting, we simply let the truncation point be the end of the source code file (or the maximum number of tokens allowed in the model's completion).
2. For the **pseudo-parser heuristic target** method, we attempt to construct an evaluation target that is more tailored to practical settings, by truncating at a heuristically defined point beyond which further continuation may be overly ambiguous. To this end, we first pseudo parse the example code without truncation, and then find the nearest location following the starting (or cursor) position which either i) corresponds to the end of the nested sub-tree which contains the starting location (roughly speaking, the end of the current curly-braced block in JAVA, say), or if this does not exist because the cursor was not within a nested part of the source code file, ii) terminates at the end of the current statement (roughly speaking, the next semi-colon in JAVA, say).

The final **pydocstring target** example type is PYTHON specific and designed to yield a different distribution of examples that include a significant natural language component. To achieve this we let the starting location of the example be the beginning of a DOCSTRING comment in the source code file (immediately after the triple quotes) and the truncation point be immediately after the corresponding closing triple quotes. To identify the DOCSTRING, we again lean on our pseudo-parser: we search for triple quotes that occur at the beginning of indented code blocks. For the model samples, there is no guarantee that the DOCSTRING will be correctly closed; in such cases we simply fall back to the untruncated target approach.

Removing the context Finally, we note that due to our dataset construction strategy, and inspired by real-world code completion systems, our suggestions may begin partway in the middle of an expression. We address this by concatenating the context (the prefix of the file) and the model suggestions, pseudo-parsing the result, and then removing any node that is entirely contained in the context after parsing. This enables us to build a tree representation of only the part of the code that we would actually be suggesting, while still having its tree structure match the parse tree of the final code state.

E.2. Utility function configuration

We configure our base utility function (described in Appendix C) in different ways for each task.

Edit localization task. For this task, we configure the utility function with a per-character utility of 1 per matched SURE token and $\alpha = 0.7$ per matched UNSURE, and a per-character cost of 1 per deleted SURE and $\beta = 0.3$ per deleted UNSURE; this setting is such that tokens with a lower-than-70% chance of being kept are optimal to mark as UNSURE. (We vary these thresholds for the Pareto plot, by setting the UNSURE match utility to $\alpha = c$ and deletion cost to $\beta = 1 - c$ for varying c.) We also include a localization penalty of 5 per edit inside SURE regions, a penalty of 0.25 in UNSURE regions, and a penalty of 0.75 for starting a new UNSURE. These costs are also tuned so that, if there is a 30%-or-greater chance of starting to edit at a given location, it is better to insert a UNSURE region that includes the edit.

[1] https://github.com

Prefix task. We use the same configuration as the edit localization task, but additionally insert truncation nodes into the prototype suggestion, which enables us to search over points to stop the suggestion early. For the R-U-SURE (Prefix) variant, we do not insert any Region Start / Region End nodes, which forces the solver to label everything as SURE and only search for prefixes. For the R-U-SURE (Prefix + Region) variant, we include both Region Start/End nodes and truncation nodes.

API call task. For this task, we restrict our attention to Python files, and do additional postprocessing on both the model samples and the ground truth target in order to compute an estimated utility. We first search through the parsed file in order to identify statements that look like function calls; in particular, any statement that contains tokens immediately followed by an open parenthesis, and which does not start with `def` or `class`. We then extract a list of such calls and rearrange them into a shallow tree structure: the top level sequence is a sequence of group nodes, and each group node contains exactly one call. We further insert region start/end nodes into each call, before and after the parenthesis, respectively; these allow our method to decide how many attribute accesses to include in the call (e.g. ‘foo.bar.baz(‘ or just ‘bar.baz‘) and whether or not to include arguments or a left-hand-side assignment. For this task, we reinterpet the regions as being SURE rather than UNSURE. Since we only care about extracting a useful subsequence, we forbid any token matches outside of extracted regions, but set the costs of deletion and insertion to zero. We also forbid any deletions or insertions in an extracted region to ensure that the call matches exactly (instead of just having high token overlap). We implement this by building a simplified version of our edit distance graph that only includes nodes for the allowed types of edit.

Within an extracted region, we compute per-token weights, which are 1 for tokens we have already seen in the file and 10 for novel tokens (those not yet seen in the file); we also give 1 bonus point for correctly predicting the entire argument list. We then scale this base weight by 0.7 to get the utility of correct predictions, and scale it by 0.3 to get the penalty for incorrect ones. Note that this is the same set of rewards and penalties as in the edit localization task, however, the break-even point is lower for this version because deleting tokens has a penalty of 0 instead of -1.

F. Detailed Experimental Results

The additional detailed results provided in the appendix include:

- Figure 20: A detailed breakdown by target type (UNTRUNCATED, HEURISTIC and PYDOCSTRING) for both leave one out and ground truth targets, of the performance of the R-U-SURE (Region) method.
- Figure 21a: An analysis of the duality gap achieved by our dual decomposition solver, that shows that the optimal solution is found in the majority of cases.
- Figure 22: A plot of model performance by size of sample K which includes both leave one out and ground truth utilities.
- Table 3, Table 4 and Table 5: Detailed versions of the tables in the main paper.

	Utility (relative)	Est. Utility (relative)	LOO Utility (relative)	Sensitivity % UNSURE of edited	Specificity % SURE of unedited	F_1 score
ALL SURE	$\equiv 0$	38.00	30.35	0.00	100.00	-
MAXIMAL UNSURE	81.83	106.08	101.90	90.79	6.85	12.74
TOKEN PROB. 0.5	50.83	82.63	77.10	55.63	72.69	63.03
TOKEN PROB. 0.7	58.42	88.89	83.68	63.81	64.97	64.38
TOKEN PROB. 0.9	66.99	95.64	90.79	73.21	52.93	61.44
PREFIX PROB. 0.5	83.33	108.52	104.29	89.31	27.39	41.92
PREFIX PROB. 0.7	83.45	108.27	104.05	89.89	23.50	37.26
PREFIX PROB. 0.9	83.08	107.61	103.41	90.35	17.65	29.53
OURS (Region)	**84.42**	**113.82**	**109.12**	85.78	62.24	**72.14**

Table 3. Detailed results for our R-U-SURE (Region) method along with a selection of baselines, on the edit-localization task.

	GT Utility (mean)	Est. Utility (mean)	LOO Utility (mean)	Correct Chars	Incorrect Chars
20 CHARACTERS	-7.99	5.15	4.11	13.61	21.60
50 CHARACTERS	-8.92	7.12	5.55	23.83	32.75
100 CHARACTERS	-14.75	4.96	2.47	34.35	49.10
200 CHARACTERS	-29.88	-5.04	-9.20	44.81	74.69
500 CHARACTERS	-66.59	-33.18	-40.16	53.66	120.25
1 LINE	-10.91	4.00	2.45	16.85	27.76
2 LINES	-12.68	4.85	2.88	24.30	36.98
4 LINES	-19.85	1.76	-1.04	33.75	53.59
8 LINES	-37.75	-9.77	-14.31	43.97	81.72
16 LINES	-65.76	-31.32	-38.14	51.82	117.58
TOKEN PROB. 0.00	-84.46	-47.36	-55.52	54.94	139.40
TOKEN PROB. 0.01	-13.46	91.15	6.31	38.85	52.32
TOKEN PROB. 0.02	-7.56	78.02	10.78	35.24	42.80
TOKEN PROB. 0.05	-2.80	63.21	13.56	30.21	33.01
TOKEN PROB. 0.10	-0.58	53.57	14.44	26.50	27.08
TOKEN PROB. 0.20	0.52	44.77	13.09	22.65	22.13
TOKEN PROB. 0.30	0.69	14.35	13.43	20.45	19.76
TOKEN PROB. 0.50	0.19	12.82	12.07	17.30	17.12
TOKEN PROB. 0.70	-1.10	10.72	10.01	14.42	15.52
TOKEN PROB. 0.90	-3.80	7.21	6.54	10.62	14.42
PREFIX PROB.0.01	0.88	48.99	15.50	24.94	24.06
PREFIX PROB.0.02	1.04	46.30	15.25	23.68	22.64
PREFIX PROB.0.05	1.03	42.62	14.64	21.83	20.80
PREFIX PROB.0.10	0.83	39.57	13.99	20.20	19.37
PREFIX PROB.0.20	0.43	36.28	14.36	18.36	17.93
PREFIX PROB.0.30	0.04	34.01	12.26	17.03	16.99
PREFIX PROB.0.50	-1.00	30.50	10.63	14.76	15.75
PREFIX PROB.0.70	-2.40	27.39	8.58	12.50	14.90
PREFIX PROB.0.90	-5.01	23.43	5.52	9.21	14.22
MAX AVG. LOG PROB	-17.64	50.96	-4.08	16.66	34.31
INTELLICODE COMPOSE	0.04	12.71	11.90	17.10	17.06
OURS (PREFIX)	**7.00**	**30.49**	**28.03**	38.81	31.81
OURS (PREFIX+REGION)	**12.26**	**37.79**	**35.18**	36.40	22.31

Table 4. Comparison of utility and character-level accuracy statistics for the suggestion-length task; R-U-SURE (Prefix) achieves higher average utility than the comparable baselines. As an additional comparison, we include results for R-U-SURE (Prefix+Region), a variant that is also allowed to mark some tokens UNSURE, which improves our utility metric and decreases the number of incorrectly-predicted characters (where we only count SURE tokens as correct/incorrect).

	GT Utility (mean)	Est. Utility (mean)	LOO Utility (mean)	Corr. (total)	Corr. (novel)	Corr. (not novel)	Incorr. (total)	Incorr. (novel)	Incorr. (not novel)
ALL FULL	-8.75	-6.09	-7.15	2.02	0.33	1.68	9.64	3.28	6.36
ALL SHORT	-0.58	0.70	0.10	3.00	0.68	2.32	4.60	2.07	2.53
NOVEL SHORT	-1.39	-0.36	-0.94	1.18	0.68	0.50	3.04	2.07	0.96
OURS (API)	**5.10**	**6.74**	**6.53**	3.56	0.68	2.88	2.10	0.50	1.60

Table 5. Detailed results for our R-U-SURE (API) method along with a set of baselines.

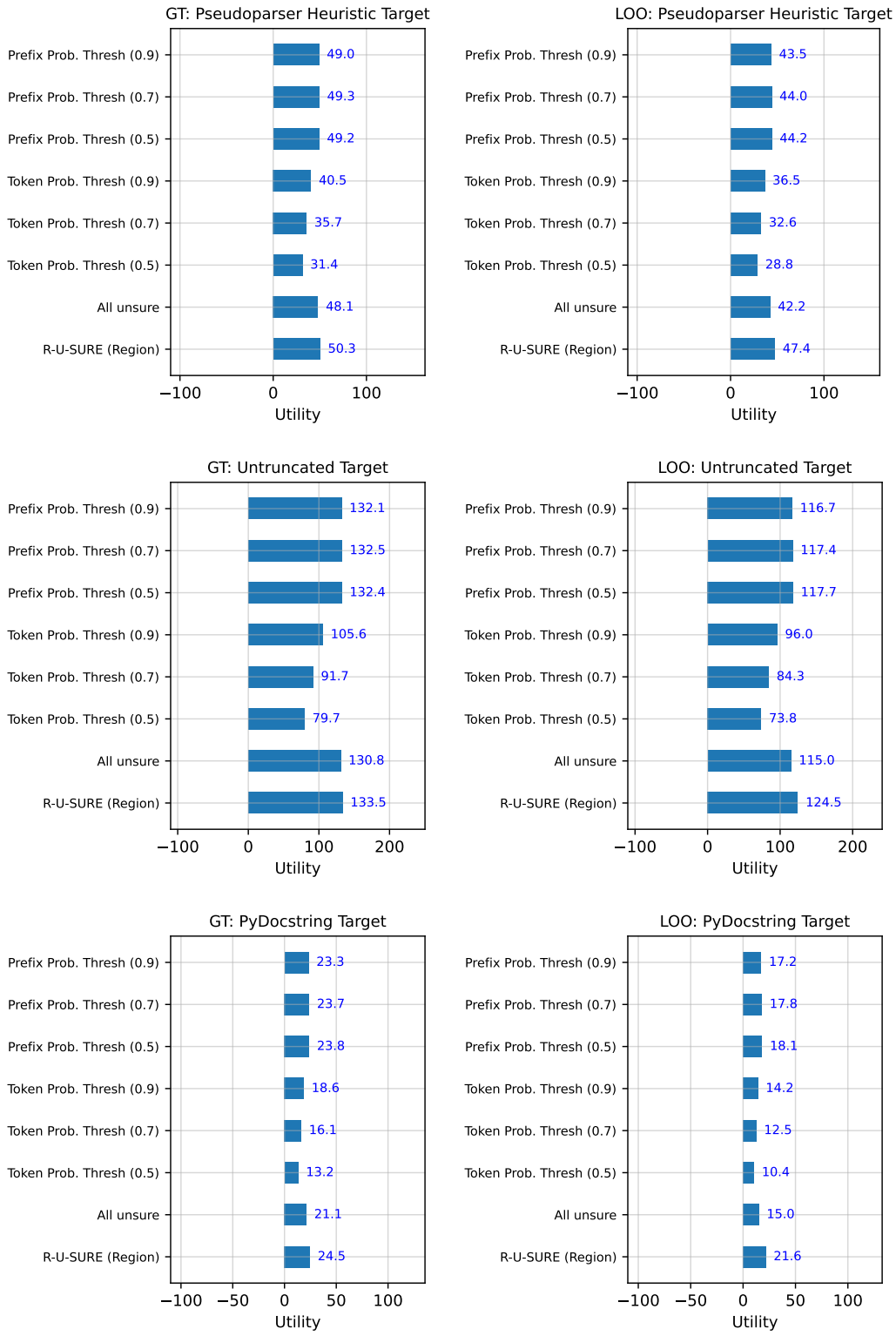
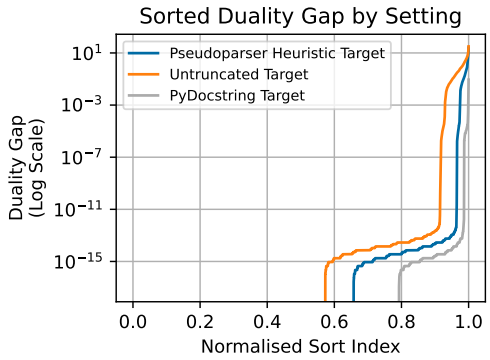
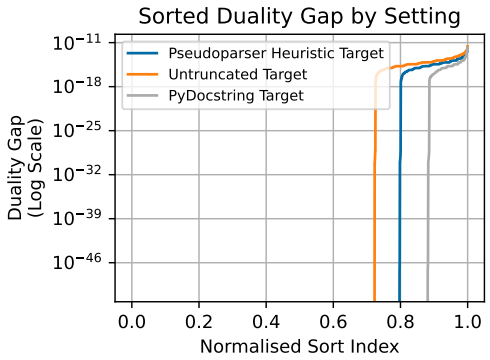


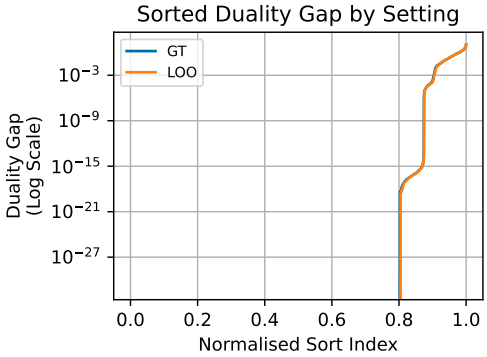
Figure 20. Average utility (higher is better) for our R-U-SURE (Region) and a variety of baseline methods for the uncertainty-regions task, evaluated on the ground truth (GT, left) user intent as well as a leave-one-out (LOO, right) sample from the model, and the three target settings (corresponding to the three rows of plots) noted in the figure titles. Note that methods that perform well in the leave-one-out setting also tend to perform well on the ground truth, but averages are slightly better across the board for leave-one-out.



(a) R-U-SURE (Region).

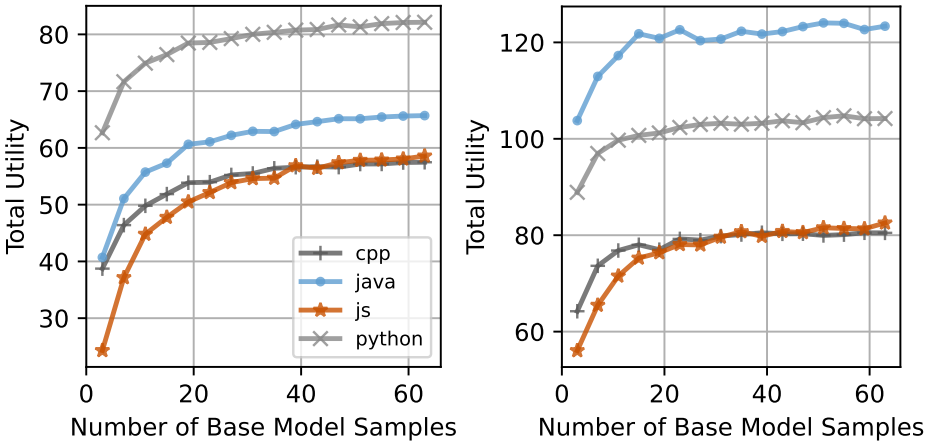


(b) R-U-SURE (Prefix).



(c) R-U-SURE (API).

Figure 21. The distribution of duality gaps presented as a log plot of the sorted values, broken down by utility function for each figure (a)-(c), and with a separate line for each type of prediction target (for (a) and (b)) or prediction target type (i.e. ground-truth or leave one out, for (c)). We observe that R-U-SURE (Prefix) always obtains practically zero gap (and hence primal optimality), while *e.g.* R-U-SURE (Region) does so on around 90-98% of cases, depending on the type of prediction target.



(a) Ground Truth. (b) Leave one out.

Figure 22. The dependence of model performance on the number of base model samples combined by R-U-SURE, evaluated with respect to the ground truth user intent (left) and a leave one out sample from the base model (right). The four lines represent the four programming languages we considered. We observe that the performance increases dramatically on the left, but that this increase is relatively flat around our maximum considered 31 samples.