Language-to-Code Translation with a Single Labeled Example

Kaj Bostrom[α*], **Harsh Jhamtani**[β], **Hao Fang**[β], **Sam Thomson**[β], **Richard Shin**[β],
Patrick Xia[β], **Benjamin Van Durme**[β], **Jason Eisner**[β], **Jacob Andreas**[β]
[α]University of Texas at Austin [β]Microsoft

Abstract

Tools for translating natural language into code promise natural, open-ended interaction with databases, web APIs, and other software systems. However, this promise is complicated by the diversity and continual development of these systems, each with its own interface and distinct set of features. Building a new language-to-code translator, even starting with a large language model (LM), typically requires annotating a large set of natural language commands with their associated programs. In this paper, we describe ICIP (In-Context Inverse Programming), a method for bootstrapping a language-to-code system using mostly (or entirely) *unlabeled* programs written using a potentially unfamiliar (but human-readable) library or API. ICIP uses a pre-trained LM to assign candidate natural language commands to these programs, then iteratively refines the commands to ensure global consistency. Across nine different application domains from the Overnight and Spider benchmarks and `text-davinci-003` and `CodeLlama-7b-Instruct` models, ICIP outperforms a number of prompting baselines. Indeed, in a "nearly unsupervised" setting with only a single annotated program and 100 unlabeled examples, it achieves up to 85% of the performance of a fully supervised system.

1 Introduction

Even with the growing capability and flexibility of language models (LMs), many tasks in NLP are best solved by translating language into a program or another formal representation. The program might query a database (Zelle and Mooney, 1996), carry out a request (Semantic Machines et al., 2020), call an external tool (Schick et al., 2023), or assist with a programming task (Maddison and Tarlow, 2014). Modern LMs can predict programs from natural language amazingly well

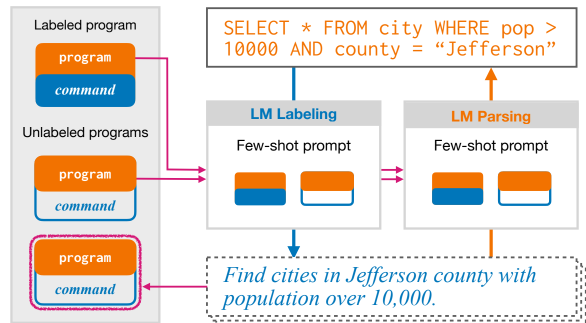

Figure 1: Overview of our approach. Starting with zero or more seed programs labeled with natural language commands, and a collection of unlabeled programs, we label programs with commands using a pre-trained language model, choosing commands that lead the LM to re-generate the target program with high probability. These automatically labeled programs are used as input to the next round of labeling, and finally used as in-context examples for a semantic parser.

when these programs look like ones that appeared in the training data (Chen et al., 2021; Chai et al., 2023). LMs can even learn new language-to-code mappings in context, given enough labeled data to prompt them with examples that demonstrate the relevant associations (Austin et al., 2021).

However, program generation is a moving target: APIs, databases, and even programming languages themselves are constantly changing, and we often find ourselves without convenient natural language descriptions or instructions for a given task, even if we have examples of working programs or an API reference. For example, we might wish to enable users to interact with new databases, or enable developers to equip LMs with new tools, or enable code autocomplete with a newly developed library, without requiring database-, tool-, or language-specific data annotation. How can we automate language-to-code translation for an unfamiliar type of code?

This paper introduces ICIP (In-Context Inverse

*Correspondence to `kaj@cs.utexas.edu`.

Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, pages 8101–8112
November 12-16, 2024 ©2024 Association for Computational Linguistics

Programming), a method for using LMs to perform "nearly unsupervised" language-to-code translation for new APIs and libraries. The key insight motivating ICIP is that most code is meant to be readable, and human programmers can often understand and describe an unfamiliar program even before they have the skills to write that program on their own. By using an LM to generate candidate natural language commands of a large collection of programs, then refining these commands toward global consistency, it is possible to bootstrap a large library of (language, program) pairs even when starting with mostly unlabeled programs as input (Fig. 1). Once labels are inferred, these labeled programs may be used to prompt or fine-tune an ordinary LM-based language-to-code model. More formally, ICIP treats natural language commands as latent variables, and performs local search to incrementally optimize the joint likelihood of a collection of programs under a prompted LLM.

ICIP builds on a large body of semi-supervised approaches to semantic parsing (Yin et al., 2018; Zhong et al., 2020). Like them, it uses cycle-consistency losses to assign consistent natural language labels to unlabeled programs. But large-scale LM pre-training enables qualitatively different learning procedures for new languages and libraries. First, the non-parametric nature of learning in prompted LMs means that improved program labels immediately translate into improved labeling *and* parsing of other examples; second, LMs' ability to generate plausible program descriptions in a zero-shot fashion means that we can start with considerably less labeled data than past work.

Results on two English language-to-code benchmarks spanning nine application domains (Overnight, Wang et al., 2015, and Spider, Yu et al., 2018) and two LMs (`text-davinci-003`, Brown et al., 2020 and `CodeLlama-7b-Instruct`, Roziere et al., 2023) show that ICIP improves over ordinary one- and many-shot prompting by as much as a factor of four, achieving as much as 85% of the accuracy attained by a *fully supervised* model when starting with only one labeled example and 100 unlabeled ones. ICIP is robust to several variations in the problem setting: it gives reasonable performance even with *no* unlabeled examples, and with drastic changes to program syntax.

In summary, this paper describes **(1)** a new method for "nearly unsupervised" text-to-code translation in LMs based on iterative refinement of generated commands, and **(2)** a set of experiments showing that this method is empirically effective, giving substantial improvements across two models and two semantic parsing benchmarks. Our results corroborate the effectiveness of pretrained LMs in bootstrapping their own supervision (Zheng et al., 2023; Li et al., 2023b; Zelikman et al., 2022).

2 Background

Translating natural language into code (or other formal representations) is a key sub-task in natural language processing, with applications spanning computational linguistics, natural language interfaces, and dialog systems. Historically, many language-to-code systems were based on explicitly structured, compositional models (Wong and Mooney, 2007; Zettlemoyer and Collins, 2005; Kwiatkowski et al., 2011). More recent systems have generally used a combination of neural sequence models, large-scale pre-training, and constrained decoding (Dong and Lapata, 2016; Roy et al., 2022).

Large datasets of natural language text and code are widely available, but paired language–program data is comparatively rare. As a result, there has been long-standing interest in building language-to-code systems with less data. One prominent line of work focuses on *weak supervision*—using natural language questions paired with answers to automatically infer programs that produce the right result when executed (Clarke et al., 2010; Liang et al., 2011; Artzi and Zettlemoyer, 2013; Grand et al., 2023).

Most relevant to the this paper, another line of research focuses on *semi-supervised* approaches, which learn from a mixture of paired language and program data as well as unpaired data in one or both modalities. These approaches typically combine supervised learning with a *cycle-consistency* objective, alternating between optimizing a language-to-code model on annotated programs, then assigning unlabeled programs natural language descriptions that produce the right code when translated (Yin et al., 2018; Zhong et al., 2020). ICIP extends these approaches along several axes—it generalizes semantic parsing with cycle consistency to the in-context learning setting, and our best-performing model variant (MaxRT below) employs a filtering criterion not used in previous work. Together, these enable ICIP to obtain significantly improved sample efficiency.

Outside the domain of language-driven program synthesis, semi-supervised learning procedures

based on enforcing cycle-consistency have been used (for pre-training, fine-tuning, and prompting) in tasks spanning machine translation, instruction following, and open-ended text generation (Li et al., 2011; Sennrich et al., 2016; Li et al., 2023b,a). We expect that future work might apply ICIP-type approaches to these problems as well. In a similar spirit to the present work, Keskar et al. (2019) also leverage LMs' ability to perform an "easy" recognition problem (labeling text with control codes) to bootstrap supervision for a hard inverse problem (controllable text generation).

3 Approach

3.1 Data

We wish to learn a mapping from natural language **commands** ℓ to **programs** π. To guide learning, we assume we have access to two sources of information: (1) a dataset $D_{L\Pi}$ of **labeled** command–program pairs (ℓ_i, π_i) (which might contain zero or one examples); and (2) a dataset D_Π of **unlabeled** programs π_i.

3.2 Base Models

In addition to these data sources, we assume access to a **language model** p_{LM}, which has been pre-trained on general language and program data, but not the specific language or library APIs used in π. Crucially, we assume that p_{LM} is capable of **in-context learning**; i.e., when conditioning on a sequence of labeled (command, program) pairs followed by an unlabeled program,

$$p_{\text{LM}}(\pi_n \mid \ell_1, \pi_1, \ldots, \ell_{n-1}, \pi_{n-1}, \ell_n)$$

assigns high probability to programs π_n likely to be associated with ℓ_n. Similarly,

$$p_{\text{LM}}(\ell_n \mid \pi_1, \ell_1, \ldots, \pi_{n-1}, \ell_{n-1}, \pi_n)$$

assigns high probability to plausible natural language labels ℓ_n for π_n.

In practice, LMs condition on sets of programs like $\{(\ell_i, \pi_i)\}$ above via a **prompt**, which concatenates them into a single string. When the number of labeled examples is large, prompt construction may additionally involve a **retrieval** step, in which initial examples (ℓ_i, π_i) are sub-selected from the full example set based on similarity between ℓ_i and ℓ. Our approach is generally compatible with many approaches to prompt construction, so in this section we will write $p_{\text{LM}}(\pi \mid D, \ell)$ to represent the distribution over programs given a prompt built from a set of labeled examples D and an input command ℓ. We will similarly write $p_{\text{LM}}(\ell \mid D, \pi)$ for a distribution over commands given examples and input programs. Prompt construction details for our experiments are discussed in Section 4.

3.3 Learning

Given M labeled programs $D_{L\Pi}$ and N unlabeled programs D_Π as defined above, our goal is to construct a set of **automatically labeled command-program pairs** $\hat{D}_{L\Pi}$, by inferring a command $\hat{\ell}_i$ for every $\pi_i \in D_\Pi$.[1] Once we have these programs, we can immediately use them to construct language-to-code model for our target domain by conditioning p_{LM} on a prompt derived from $D_{L\Pi} \cup \hat{D}_{L\Pi}$.

To do so, we initialize $\hat{D}_{L\Pi} = \emptyset$, then alternate between two steps:

Sampling. For *each* unlabeled program π_i in D_Π, sample a set of candidate natural language commands that should correspond to π_i:

$$L'_i = \{\hat{\ell}_i^1 \ldots \hat{\ell}_i^k\} \sim p_{\text{LM}}(\ell \mid D_{L\Pi} \cup \hat{D}_{L\Pi}, \pi_i)$$

We may then define a dataset of candidates:

$$D'_{L\Pi} = \bigcup_i \{(\hat{\ell}_i^j, \pi_i) : \hat{\ell}_i^j \in L'_i\}$$

Intuitively, the sampling step uses the labeled and (possibly empty) unlabeled programs to guess a plausible natural language command for π_i.

Filtering. For each L'_i, select a subset:

$$\hat{L}_i = \{\hat{\ell}_i^j \in L'_i : \texttt{filter}(\hat{\ell}_i^j, \pi_i, D_{L\Pi} \cup D'_{L\Pi})\}$$

where `filter` is a criterion for rejecting low-quality candidates (e.g. which fail to round-trip, or which are classified by another LM as incorrect, as in Li et al., 2023b), which will be described in more detail below.[2] Finally,

[1] Semi-supervised learning problems of this kind often arise in natural language processing tasks. For example, Merialdo (1994) performed semi-supervised learning of part-of-speech tagging HMMs using Expectation–Maximization.

[2] It may be surprising that this step uses the unfiltered dataset D' from the current iteration, rather than the filtered dataset from the previous iteration. This choice makes it possible to use more than just the seed set to guide filtering in the first iteration. Unfiltered examples improved performance in the first iteration, and did not meaningfully affect performance (relative to using filtered examples) in later iterations.

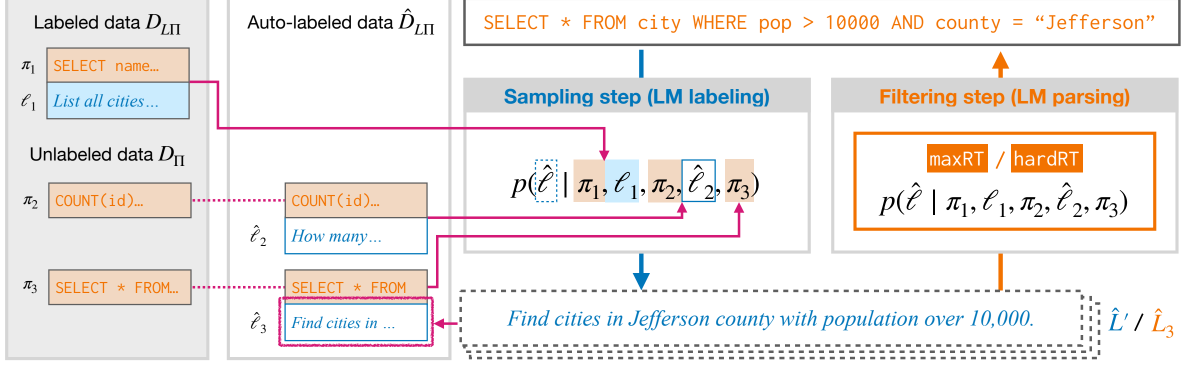

Figure 2: Details of the proposed learning algorithm. During the **sampling step**, the current set of seed labeled and automatically labeled examples is used to generate a set of candidate natural language commands associated with each unlabeled program. During this step, the LM is prompted with seed labels and automatically generated labels from the previous iteration. During the **filtering** step, this set is sub-selected to include only those that will be parsed into the original program with high probability, again using previously labeled examples in the prompt.

update:

$$\hat{D}_{L\Pi} \leftarrow \bigcup_i \{(\hat{\ell}_i^j, \pi_i) : \hat{\ell}_i^j \in \hat{L}_i\}$$

Intuitively, this step filters down the set of generated labels for correctness, then updates the auto-labeled set $\hat{D}_{L\Pi}$ to include labeled pairs that passed the filter.

Together, the two steps implement a form of rejection sampling, with p_{LM} as the proposal distribution and `filter` as (a discriminator for) the target distribution. The full ICIP procedure is described formally in Algorithm 1 and visualized in Fig. 2. The sampling and filtering steps may be iterated, in which case labels inferred at early timesteps can inform the choice of labels for other examples at later timesteps. This two-step procedure has an intuitive similarity to more famous alternating optimization algorithms like expectation–maximization (EM; Dempster et al., 1977) and mean-field variational Bayes (e.g., Jordan et al., 1999). The sampling and filtering steps *together* accomplish something roughly analogous to the E step in EM. The resulting filtered program-command pairs can be viewed as a particle-based approximate representation of the distribution of latent commands. While the M step of EM would ordinarily retrain explicit model parameters to maximize their expected likelihood given the distribution of latents produced in the E step, our model has no such tunable parameters. Instead, our equivalent to the M step is performed implicitly as the pretrained LM's behavior updates through in-context learning from the examples generated in the E step. While we do not optimize any explicit parameters, recent work suggests that in-context learning in sufficiently large transformers may, in fact, behave akin to optimization (Akyürek et al., 2023; Von Oswald et al., 2023).

The choice of an appropriate `filter` is crucial to the quality of the in-context demonstrations that act as supervision for this step. We consider two in our experiments:

Hard Round Trip (HardRT). This version of ICIP accepts any command that, when translated by the LM into a program given the current example set, yields the right program:

$$\texttt{filter}(\hat{\ell}, \pi, D) = 1[\pi = \operatorname*{argmax}_{\pi'} p_{\text{LM}}(\pi' \mid D, \hat{\ell})]$$

This filter may produce zero, one, or multiple commands for each program. "Hard" cycle consistency constraints of this kind were also used by Zhong et al. (2020) for learning semantic parsers. The idea of filtering out possibly incorrect imputed values has also appeared in confidence-thresholded variants of EM (e.g., Yarowsky, 1995).

Max Round Trip (MaxRT). This version of ICIP accepts the command that, when passed as input to the LM, causes the LM to assign the *highest possible probability* to the target program (even if it does not generate the target program as output):

$$\texttt{filter}(\hat{\ell}, \pi, D) = 1[\hat{\ell} = \operatorname*{argmax}_{\ell'} p_{\text{LM}}(\pi \mid D, \ell')]$$

This filter always produces exactly one command for each program.

What are the tradeoffs between HardRT and MaxRT? HardRT keeps more commands for pro-

Algorithm 1 In-Context Inverse Programming

1: $D_{L\Pi} \leftarrow$ labeled seed programs
2: $D_{\Pi} \leftarrow$ unlabeled programs
3: $\hat{D}_{L\Pi} \leftarrow \emptyset$ *// holds auto-labeled programs*
4: **for** $1 \leq t \leq N_{\text{iters}}$ **do**
5: $\quad T \leftarrow \emptyset$ *// holds in-progress labels*
6: $\quad$ **for** $\pi_i \in D_{\Pi}$ **do**
7: $\quad\quad L'_i \sim p_{\text{LM}}(\ell \mid D_{L\Pi} \cup \hat{D}_{L\Pi}, \pi_i)$
8: $\quad\quad$ **for** $\ell_i^j \in L'_i$ **do**
9: $\quad\quad\quad$ **if** $\texttt{filter}(\ell_i^j, \pi_i, D_{L\Pi} \cup \hat{D}_{L\Pi})$ **then**
10: $\quad\quad\quad\quad T \leftarrow T \cup \{(\ell_i^j, \pi_i)\}$
11: $\quad \hat{D}_{L\Pi} \leftarrow D_{L\Pi} \cup T$

grams π_i that are easier to translate, and drops programs π_i that are hard to translate. This shifts the distribution of $\hat{D}_{L\Pi}$ toward the former type of program, perhaps harming the system's ability to produce the latter. MaxRT ensures more uniform coverage of the space of programs, at the cost of potentially giving π_i a label for which π_i is *not* the correct semantic parse.

The argmax in both the HardRT and MaxRT filters involves an intractable search over all programs or labels. In HardRT, we approximate it by performing greedy decoding from p_{LM}. In MaxRT, we approximate it by choosing the highest-scoring element in the set of *candidate labels* L'_i.

We remark that other filters are possible in principle. For example, for each unlabeled program π_i, we could use normalized importance sampling to estimate the posterior distribution over its candidate labels $D'_{L\Pi}$: weight each $\hat{\ell}_i^j$ proportionately to $p_{\text{LM}}(\hat{\ell}_i^j)\, p_{\text{LM}}(\pi_i \mid \hat{\ell}_i^j) \,/\, p_{\text{LM}}(\hat{\ell}_i^j \mid \pi_i)$, then sample from this distribution, as in Monte Carlo EM.

4 Experimental Setup

Datasets. In our main experiments, we evaluate semantic parsing on two English-language benchmark datasets: Spider (Yu et al., 2018), a text-to-SQL task, and Overnight (Wang et al., 2015), a benchmark evaluating semantic parsing for question answering in eight application domains spanning calendaring, restaurant search, spatial reasoning, and more. . Both datasets contain several thousand labeled program-utterance pairs. The Spider and Overnight test splits contain 1,034 and 2,740 examples respectively. All of our test numbers report average accuracy over the entire test split.[3]

[3]While complete training sets for the GPT and Llama models used in this paper are not publicly available, a Github search for Overnight examples in the pre-processed form used for our main experiments returned no search results. We are thus reasonably confident that models were exposed to these datasets for the first time during evaluation. Section 5.3 shows that results are robust to dramatic obfuscation of the datasets.

To create the small training sets required to run our method and baselines, we sample $M + N$ examples from the training split, without replacement, using the first M and last N respectively for the labeled set $D_{L\Pi}$ and the unlabeled set D_{Π}. For each condition $(M + N)$ that we report on, there is some variance in the results due to the particular draw of $(D_{L\Pi}, D_{\Pi})$; thus Table 1 reports the mean over three draws, ± 1 standard deviation.

Metrics. We report two measures of program recovery performance: (1) program accuracy, the proportion of predicted programs that exactly match the reference programs associated with a set of test utterances, and (2) answer (or "denotation") accuracy, the proportion of predicted programs that return the same result as their utterance's reference program when executed. In Overnight, execution occurs against a fixed model; in Spider, each query is executed against a query-specific database.

Modeling details. We conduct experiments using two language models: `text-davinci-003`, a proprietary model developed by OpenAI and tailored for general text instruction-following, and `CodeLlama-7b-Instruct`, a public model with 7 billion parameters produced by continuing the pretraining of a Llama 2 model (Touvron et al., 2023) on the code subset of its training corpus, then finetuning it to add instruction-following capability.

We implement parsing and labeling with few-shot prompting. We construct a prompt by concatenating an instruction and a set of labeled or unlabeled exemplars (as described below), then sample continuations from the language model conditioned on this prompt. When labeling programs, we provide the instruction *Explain each of the following programs with a natural language string*. When parsing, we use the instruction *Let's translate what a human user says into what a computer might say*.

When generating multiple candidate labels, we draw samples from the model with softmax temperature 1.0 and output distribution truncated to probability mass 0.95 (Holtzman et al., 2020) in order to encourage diversity while maintaining coherence. We sample 8 candidate labels per program in ICIP +MaxRT; in preliminary experiments, sampling more candidates did not yield further im-

System	M+N	Overnight Blocks Prog	Overnight Blocks Ans
Few-shot	(1+0)	$6.5_{\pm 2.4}$	$16.6_{\pm 9.5}$
Few-shot	(8+0)	$19.5_{\pm 3.3}$	$35.0_{\pm 4.5}$
Few-shot + unl.	(1+100)	$18.9_{\pm 2.6}$	$31.2_{\pm 3.6}$
ICIP + MaxRT	(0+100)	$25.9_{\pm 1.6}$	$36.7_{\pm 2.1}$
ICIP + HardRT	(1+100)	$26.8_{\pm 0.8}$	$39.7_{\pm 1.4}$
ICIP + MaxRT	(1+100)	$29.1_{\pm 2.2}$	$42.7_{\pm 3.4}$
Fully labeled	(100+0)	$33.2_{\pm 1.9}$	$49.7_{\pm 2.3}$

(a) `text-davinci-003`

System	M+N	Overnight (All) Prog	Overnight (All) Ans	Spider Prog
Few-shot	(1+0)	4.3	7.1	$0.8_{\pm 2.2}$
Few-shot	(8+0)	23.6	36.2	$10.0_{\pm 3.1}$
Few-shot + unl.	(1+100)	19.7	30.1	$9.6_{\pm 2.8}$
ICIP + MaxRT	(0+100)	16.9	26.1	$6.8_{\pm 5.4}$
ICIP + HardRT	(1+100)	29.9	41.7	$11.6_{\pm 2.3}$
ICIP + MaxRT	(1+100)	25.4	37.6	$11.7_{\pm 3.2}$
Fully labeled	(100+0)	44.9	61.4	$13.0_{\pm 2.0}$

(b) `CodeLlama-7b-Instruct`

Table 1: Results on the Overnight and Spider datasets. Metrics are described at the start of Section 4. The M+N column indicates the number of labeled and unlabeled examples used to construct prompts. **Prog** indicates exact-match percentage of programs on held-out test examples. **Ans** indicates answer (or denotation) accuracy: the percentage of programs which yield the same result as the corresponding gold program when executed. Subscripts indicate standard error across seeds. These are not calculated for the Overnight (All) condition, which shows average performance across both seeds and datasets; see Appendix A for results and measures of variation in each domain.

provement. When generating parses, we use greedy decoding, at each step selecting the token with maximum probability. We additionally ensure syntactic correctness using speculative constrained decoding (Shin and Van Durme, 2022).

We limit all few-shot prompts to at most 8 exemplars. When more than 8 examples are available, the prompts uses the 8 that are most similar according to the number of unigrams and bigrams shared between the subword-tokenized input in question and examples in the retrieval pool. Exemplars for parsing are ranked for retrieval based on utterance overlap, and exemplars for labeling are ranked by program overlap. Exact matches to the input utterance or program are excluded during training.

We run experiments on both datasets with the `CodeLlama-7b-Instruct` model, and additionally run `text-davinci-003` on the Overnight Blocks sub-domain (due to computational constraints).

Baselines. We compare our proposed algorithms to a simple baseline of few-shot parsing using the above instruction followed by *labeled* exemplars from $D_{L\Pi}$ (limited to 8 as explained above). These are the 1+0, 8+0, and 100+0 conditions.

We also compare to an extended baseline which, right after the instruction, inserts the text *Here are some examples of programs we might want* followed by *unlabeled* exemplars from D_{Π} (again limited to 8), and then continues with the labeled exemplars as before. This 1+100 condition provides a fair comparison to ICIP's 1+100 condition.

5 Results

5.1 ICIP learns effectively from a single labeled example

One-shot learning results are shown in the ICIP (1+100) rows of Table 1. On both models and both datasets, ICIP outperforms ordinary one-shot prompting as well as prompting with one labeled and many unlabeled examples. With the `text-davinci-003` model and MaxRT filtering, it substantially outperforms in-context learning with eight labeled examples, and approaches the performance of a *fully supervised* system with 100 labeled examples. While gains are not as pronounced with the smaller `CodeLlama-7b-Instruct` model, ICIP still dramatically improves upon models that learn from the same number of labeled examples. (The difference between `text-davinci-003` and `CodeLlama-7b-Instruct` in the fully labeled condition indicates that differences are mainly attributable to the LMs themselves, and not to ICIP.) Table 2 shows example inferred commands.

The effectiveness of MaxRT highlights two key technical advantages of ICIP over previous semi-supervised semantic parsing methods such as Zhong et al. (2020). In particular, the use of a likelihood-based filtering criterion, rather than the correct-output criterion used in previous work, both improves accuracy and eliminates the need to execute LM predictions (or even sample from the LM). This, in turn, means that ICIP can be applied efficiently even with very large LMs and in settings where program execution is not possible.

ICIP (1+100)	
Program	`(call listValue (call filter (call filter (call getProperty (call singleton en.block) (string !type)) (string is_special)) (string height) (string =) (number 3 en.inch)))`
Round 1	*find me all blocks of special type that are 3 inches tall*
Round 2	*find me all special blocks that have a height of 3 inches*
Gold	*find me all 3 inch tall special blocks*
ICIP (1+100)	
Program	`(call listValue (call countSuperlative (call getProperty (call singleton en.block) (string !type)) (string max) (string above) (call getProperty (call singleton en.block) (string !type))))`
Round 1	*find me the maximum block that is above any other blocks*
Round 2	*find the block type with the maximum number of blocks above it*
Gold	*find the block located on top of the highest number of other blocks*
ICIP (0+100)	
Program	`(call listValue (call aggregate (string avg) (call getProperty (call getProperty (call singleton en.block) (string !type)) (string height))))`
Round 1	*This program calculates the average height of all blocks of the same type in the singleton block group.*
Round 2	*This program calculates the average height of the type property of the en.block singleton.*
Gold	*what is the average height of a block*

Table 2: Examples of inferred program annotations, all using `text-davinci-003` on the Overnight Blocks sub-task. Errors are shown in red. The first two groups are prompted with one labeled example, and annotations have consistent style; the third group shows a fully-unsupervised model, which produces very different (and perhaps lower-quality) annotations, which nevertheless yield a surprisingly effective text-to-code model (see Table 1).

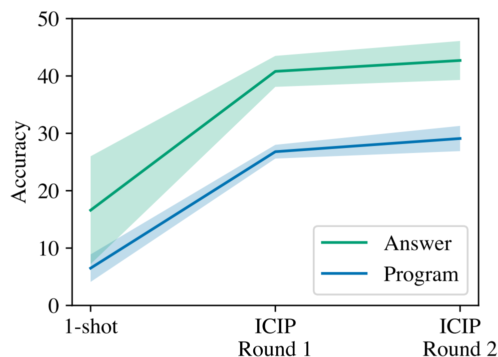

Figure 3: Parsing accuracy on Overnight Blocks using `text-davinci-003` and a single seed label increases with successive rounds of ICIP +MaxRT. Shaded areas indicate the sample standard deviation over 5 seeds.

5.2 ICIP learns effectively even with *no* labeled examples

While not the primary focus of our evaluation, an even stronger version of this result may be found in the ICIP (0+100) rows of Table 1. These experiments show that, starting only with unlabeled programs, ICIP infers commands that, when provided in a prompt, give significant improvements over even one-shot baselines. However, these improvements are smaller, and in some cases matched by ordinary few-shot prompting.

Examples of inferred commands are shown in Table 2. Notably, they diverge significantly from the style and formatting of the commands in the labeled training and test data, yet are close enough to provide an effective prompt for mapping test commands to programs.

5.3 ICIP is robust to variations in program syntax and style

The pre-trained LMs we use as a starting point have seen a significant amount of data in both Lisp (the language in which Overnight programs are expressed) and SQL (for Spider). Does ICIP continue to work when aspects of syntax, and not just program vocabulary, diverge from what was seen during pre-training?

To evaluate this, we conduct an additional set of experiments on the Blocks sub-task from Overnight in which we systematically transform the syntax of program expressions themselves: using C-style function calls (*C calls*), altering parentheses (*Var. braces*) or removing them (*No braces*), reversing the order of function and arguments (*Arg. reverse*) or anonymizing the names of functions themselves (*Anon. fns.*) Examples of these transformations are shown in Table 3, and results of ICIP and a fully-labeled oracle are shown in Table 4. These experiments show that ICIP is robust to large changes in syntax, achieving non-trivial scores, and outperforming non-ICIP baselines, across all transformation types.

Command	*What is the length of block 1?*
Original	`(call listValue (call getProperty en.block.block1 (string length)))`
C calls	`listValue(getProperty(en.block.block1, string(length)))`
Var. braces	`{call listValue {call getProperty en.block.block1 {string length}}}`
No braces	`call listValue call getProperty en.block.block1 string length`
Arg. reverse	`(call (call (string length) en.block.block1 getProperty) listValue)`
Anon. fns.	`(p0 p1 (p0 p2 en.block.block1 (p4 length)))`

Table 3: Examples of program transformations.

Transformation	Blocks Answer Acc. ICIP + MaxRT (1+100)	Fully labeled (100+0)
Original	42.7	49.7
C calls	37.8	50.8
Var. braces	40.6	50.1
No braces	35.8	48.6
Arg. reverse	34.6	45.8
Anon fns.	34.6	48.6

Table 4: Results of program transformation experiments. ICIP continues to perform nontrivially even under significant alterations to program syntax.

5.4 ICIP iteratively improves label quality

The examples in Table 2 suggest that imputed natural language commands not only reflect the semantics of their associated programs, but improve over multiple rounds of iteration, highlighting the role of the iterative nature of ICIP in generating high-quality labeled examples. The overall change in accuracy across rounds is measured in Fig. 3.

6 Analysis

We conclude with a qualitative analysis of the predictions and labels generated by ICIP. We focus on Overnight Blocks, as it contains a large number of programs in a single domain from which aggregate statistics can be computed, and the MaxRT filtering method, which achieves the best overall performance in this domain.

6.1 How diverse are generated labels?

One behavior commonly observed in latent-variable models like ICIP is *mode collapse*, wherein all latent variables ℓ are assigned the same value. This appears not to be a significant problem for ICIP: 96.9% of predicted labels were unique (compare to 99% for humans). When computing the all-pairs self-BLEU for both sets, ICIP labels have a score of 18.2 and human labels have a score of 7.6. Together, these results indicate that ICIP sometimes produces repetition at the token level, but still assigns unique labels to distinct programs.

6.2 What programs are hard to predict?

Are hard programs longer than short ones? Gold program length does not appear to meaningfully predict parsing difficulty: programs that ICIP predicts correctly over a majority of seeds are on average 21.4 ± 5.2 tokens in length, while programs ICIP predicts incorrectly are on average 23.5 $\pm$ 5.2 tokens long.

Are some program features harder to translate than others? We computed the program tokens that occur more frequently (per the normalized unigram distribution) in gold programs which ICIP fails to predict correctly compared to the cases ICIP predicts correctly. The program tokens that occur most frequently in incorrect cases are: `reverse`, `filter`, `string`, `right`, `above`, and `left`. The fully labeled (100+0) model struggles with a very similar list of constructs: `reverse`, `filter`, `block1`, `above`, `below`, and `right`. Thus, the difficulty here may actually stem from the basic difficulty of translating these types of relations into code, rather than a specific deficiency of ICIP. More generally, these results suggest that ICIP exhibits a similar error distribution to supervised methods.

6.3 How do imputed commands compare to ground-truth commands?

In ground-truth commands, the average length is 9.3 ± 3.3 words. After one iteration, ICIP's commands are 11.3 ± 3.3 words in length; after two iterations, this grows slightly to 12.0 ± 3.3 words. Thus, ICIP generates slightly longer annotations on average. We hypothesize that this is because more detailed, explicit commands are selected in the filtering step, as they are more likely to enable an accurate reconstruction of the input program.

7 Conclusion

We have presented In-Context Inverse Programming (ICIP), a "nearly unsupervised" procedure for building language-to-code models for new libraries, data sources, or languages starting with a collection

of unlabeled programs and as few as one labeled example. ICIP works by iteratively predicting natural language commands for unlabeled programs that enable reconstruction of the original program with high probability. Experiments on multiple datasets and models show that this procedure is effective, substantially outperforming one- and many-shot prompting methods in low-data regimes.

Our results highlight the fact that pre-trained LMs' ability to perform easy tasks, like labeling programs with descriptions, can be used to bootstrap models for harder tasks by encouraging both local consistency (via round-trip constraints) and global consistency (via in-context learning from inferred demonstrations).

Limitations

While ICIP is "one-shot" in the sense of leveraging a single labeled example in the target domain, it relies critically on very large-scale pretraining of the base LM—enough to support both in-context learning and zero-shot annotation of unfamiliar program constructs with at least plausible natural language commands. ICIP thus relies on some degree of similarity between new domains and existing ones, and may not generalize robustly to languages in which code is not at least partially comprehensible to non-expert humans.

Due to the closed nature of `text-davinci-003`, we cannot absolutely rule out the possibility that versions of held-out programs appeared in the training data. Our experiments in Section 5.3 are intended to address these concerns. As best as can be determined by searching public code-bases, neither the original or transformed overnight programs have never appeared in any form in a public LM training set.

After creating up to 100 synthetic labeled examples, we used only 8 of them at a time during semantic parsing. We did not investigate parameter-efficient fine-tuning of the LM on all of the synthetic examples, which can be even more effective. Performance might also be improved further by creating a much larger synthetic dataset.

Finally, we note that experiments in this paper use English-language data for all commands. The behavior of ICIP in other languages (and especially low-resource natural languages) has not been evaluated.

Ethical Considerations

We do not anticipate any ethical concerns associated with this work.

References

Ekin Akyürek, Dale Schuurmans, Jacob Andreas, Tengyu Ma, and Denny Zhou. 2023. What learning algorithm is in-context learning? investigations with linear models. In *The Eleventh International Conference on Learning Representations*.

Yoav Artzi and Luke Zettlemoyer. 2013. Weakly supervised learning of semantic parsers for mapping instructions to actions. *Transactions of the Association for Computational Linguistics*, 1:49–62.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program synthesis with large language models. *arXiv*, 2108.07732.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Yekun Chai, Shuohuan Wang, Chao Pang, Yu Sun, Hao Tian, and Hua Wu. 2023. ERNIE-code: Beyond English-centric cross-lingual pretraining for programming languages. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 10628–10650, Toronto, Canada. Association for Computational Linguistics.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *arXiv*, 2107.03374.

James Clarke, Dan Goldwasser, Ming-Wei Chang, and Dan Roth. 2010. Driving semantic parsing from

the world's response. In *Proceedings of the Fourteenth Conference on Computational Natural Language Learning*, pages 18–27, Uppsala, Sweden. Association for Computational Linguistics.

Arthur P Dempster, Nan M Laird, and Donald B Rubin. 1977. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society: Series B (Methodological)*, 39(1):1–22.

Li Dong and Mirella Lapata. 2016. Language to logical form with neural attention. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 33–43, Berlin, Germany. Association for Computational Linguistics.

Gabriel Grand, Lionel Wong, Matthew Bowers, Theo X Olausson, Muxin Liu, Joshua B Tenenbaum, and Jacob Andreas. 2023. LILO: Learning interpretable libraries by compressing and documenting code. *arXiv*, 2310.19791.

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2020. The curious case of neural text degeneration. In *Proceedings of the International Conference on Learning Representations*.

Michael I Jordan, Zoubin Ghahramani, Tommi S Jaakkola, and Lawrence K Saul. 1999. An introduction to variational methods for graphical models. *Machine learning*, 37:183–233.

Nitish Shirish Keskar, Bryan McCann, Lav Varshney, Caiming Xiong, and Richard Socher. 2019. CTRL: A conditional transformer language model for controllable generation. *arXiv*, 1909.05858.

Tom Kwiatkowski, Luke Zettlemoyer, Sharon Goldwater, and Mark Steedman. 2011. Lexical generalization in CCG grammar induction for semantic parsing. In *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing*, pages 1512–1523, Edinburgh, Scotland, UK. Association for Computational Linguistics.

Belinda Z. Li, Maxwell Nye, and Jacob Andreas. 2023a. Language modeling with latent situations. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 12556–12571, Toronto, Canada. Association for Computational Linguistics.

Xian Li, Ping Yu, Chunting Zhou, Timo Schick, Luke Zettlemoyer, Omer Levy, Jason Weston, and Mike Lewis. 2023b. Self-alignment with instruction backtranslation. *arXiv*, 2308.06259.

Zhifei Li, Ziyuan Wang, Jason Eisner, Sanjeev Khudanpur, and Brian Roark. 2011. Minimum imputed-risk: Unsupervised discriminative training for machine translation. In *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing*, pages 920–929, Edinburgh, Scotland, UK. Association for Computational Linguistics.

Percy Liang, Michael Jordan, and Dan Klein. 2011. Learning dependency-based compositional semantics. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 590–599, Portland, Oregon, USA. Association for Computational Linguistics.

Chris J. Maddison and Daniel Tarlow. 2014. Structured generative models of natural source code. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ICML'14, page II–649–II–657. JMLR.org.

Bernard Merialdo. 1994. Tagging English text with a probabilistic model. *Computational Linguistics*, 20(2):155–171.

Subhro Roy, Sam Thomson, Tongfei Chen, Richard Shin, Adam Pauls, Jason Eisner, and Benjamin Van Durme. 2022. BenchCLAMP: A benchmark for evaluating language models on semantic parsing. *arXiv*, 2206.10668.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems*.

Semantic Machines, Jacob Andreas, John Bufe, David Burkett, Charles Chen, Josh Clausman, Jean Crawford, Kate Crim, Jordan DeLoach, Leah Dorner, Jason Eisner, et al. 2020. Task-oriented dialogue as dataflow synthesis. *Transactions of the Association for Computational Linguistics*, 8:556–571.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Improving neural machine translation models with monolingual data. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 86–96, Berlin, Germany. Association for Computational Linguistics.

Richard Shin and Benjamin Van Durme. 2022. Few-shot semantic parsing with language models trained on code. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 5417–5425, Seattle, United States. Association for Computational Linguistics.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller,

Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv*, 2307.09288.

Johannes Von Oswald, Eyvind Niklasson, Ettore Randazzo, João Sacramento, Alexander Mordvintsev, Andrey Zhmoginov, and Max Vladymyrov. 2023. Transformers learn in-context by gradient descent. In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23. JMLR.org.

Yushi Wang, Jonathan Berant, and Percy Liang. 2015. Building a semantic parser overnight. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 1332–1342, Beijing, China. Association for Computational Linguistics.

Yuk Wah Wong and Raymond Mooney. 2007. Learning synchronous grammars for semantic parsing with lambda calculus. In *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics*, pages 960–967, Prague, Czech Republic. Association for Computational Linguistics.

David Yarowsky. 1995. Unsupervised word sense disambiguation rivaling supervised methods. In *33rd Annual Meeting of the Association for Computational Linguistics*, pages 189–196, Cambridge, Massachusetts, USA. Association for Computational Linguistics.

Pengcheng Yin, Chunting Zhou, Junxian He, and Graham Neubig. 2018. StructVAE: Tree-structured latent variable models for semi-supervised semantic parsing. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 754–765, Melbourne, Australia. Association for Computational Linguistics.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3911–3921, Brussels, Belgium. Association for Computational Linguistics.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah Goodman. 2022. STar: Bootstrapping reasoning with reasoning. In *Advances in Neural Information Processing Systems*.

John M. Zelle and Raymond J. Mooney. 1996. Learning to parse database queries using inductive logic programming. In *Proceedings of the National Conference on Artificial Intelligence*. AAAI Press.

Luke Zettlemoyer and Michael Collins. 2005. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence*, volume abs/1207.1420.

Huaixiu Steven Zheng, Swaroop Mishra, Xinyun Chen, Heng-Tze Cheng, Ed H. Chi, Quoc V Le, and Denny Zhou. 2023. Take a step back: Evoking reasoning via abstraction in large language models. *arXiv*, 2310.06117.

Victor Zhong, Mike Lewis, Sida I. Wang, and Luke Zettlemoyer. 2020. Grounded adaptation for zero-shot executable semantic parsing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6869–6882, Online. Association for Computational Linguistics.

A Overnight Results by Domain

	calendar	
	Prog	**Ans**
Few-shot (1+0)	4.6 ± 3.6	9.1 ± 2.4
Few-shot (8+0)	22.0 ± 8.4	34.1 ± 12.6
ICIP + MaxRT (0+100)	17.1 ± 3.8	29.2 ± 3.6
Few-shot + Unl. (1+100)	22.0 ± 3.1	32.9 ± 6.1
ICIP + HardRT (1+100)	29.6 ± 5.6	44.0 ± 5.1
ICIP + MaxRT (1+100)	30.8 ± 3.6	43.3 ± 3.6
Fully Labeled (100+0)	47.6 ± 3.3	65.9 ± 2.9

	blocks	
	Prog	**Ans**
Few-shot (1+0)	2.5 ± 0.9	3.0 ± 1.1
Few-shot (8+0)	12.5 ± 2.1	25.6 ± 2.8
ICIP + MaxRT (0+100)	8.7 ± 0.8	14.6 ± 1.0
Few-shot + Unl. (1+100)	13.8 ± 2.9	22.9 ± 7.7
ICIP + HardRT (1+100)	19.7 ± 1.3	27.3 ± 1.8
ICIP + MaxRT (1+100)	16.0 ± 2.9	25.2 ± 4.2
Fully Labeled (100+0)	26.4 ± 1.7	42.0 ± 2.8

	socialnetwork	
	Prog	**Ans**
Few-shot (1+0)	2.4 ± 1.0	3.4 ± 1.0
Few-shot (8+0)	12.9 ± 2.2	26.9 ± 3.2
ICIP + MaxRT (0+100)	6.4 ± 0.9	12.3 ± 3.6
Few-shot + Unl. (1+100)	7.9 ± 0.7	18.4 ± 0.2
ICIP + HardRT (1+100)	9.4 ± 3.3	15.3 ± 2.8
ICIP + MaxRT (1+100)	8.6 ± 1.3	17.4 ± 1.3
Fully Labeled (100+0)	35.9 ± 1.3	50.9 ± 0.8

	restaurants	
	Prog	**Ans**
Few-shot (1+0)	4.6 ± 3.4	8.8 ± 5.8
Few-shot (8+0)	24.2 ± 1.1	42.3 ± 2.0
ICIP + MaxRT (0+100)	21.3 ± 4.2	34.0 ± 7.1
Few-shot + Unl. (1+100)	25.3 ± 2.1	41.1 ± 3.0
ICIP + HardRT (1+100)	32.0 ± 1.0	50.9 ± 2.2
ICIP + MaxRT (1+100)	28.9 ± 2.6	45.8 ± 2.3
Fully Labeled (100+0)	44.1 ± 2.1	69.8 ± 1.3

	basketball	
	Prog	**Ans**
Few-shot (1+0)	9.6 ± 1.1	9.9 ± 1.4
Few-shot (8+0)	45.7 ± 9.5	52.8 ± 8.3
ICIP + MaxRT (0+100)	29.8 ± 6.9	33.2 ± 7.6
Few-shot + Unl. (1+100)	23.6 ± 2.6	26.6 ± 2.9
ICIP + HardRT (1+100)	55.8 ± 4.9	60.1 ± 5.2
ICIP + MaxRT (1+100)	39.0 ± 11.6	42.4 ± 14.0
Fully Labeled (100+0)	69.7 ± 1.7	74.0 ± 2.1

	housing	
	Prog	**Ans**
Few-shot (1+0)	4.8 ± 1.8	10.4 ± 3.8
Few-shot (8+0)	15.3 ± 1.9	31.6 ± 8.3
ICIP + MaxRT (0+100)	13.1 ± 1.2	28.4 ± 0.8
Few-shot + Unl. (1+100)	16.4 ± 3.7	29.5 ± 12.7
ICIP + HardRT (1+100)	22.9 ± 0.3	44.4 ± 4.8
ICIP + MaxRT (1+100)	23.1 ± 3.9	45.5 ± 3.7
Fully Labeled (100+0)	35.3 ± 2.4	61.4 ± 1.1

	publications	
	Prog	**Ans**
Few-shot (1+0)	1.9 ± 1.9	4.3 ± 1.9
Few-shot (8+0)	25.5 ± 5.6	38.9 ± 9.3
ICIP + MaxRT (0+100)	15.1 ± 2.9	25.9 ± 4.2
Few-shot + Unl. (1+100)	19.5 ± 2.9	30.4 ± 2.7
ICIP + HardRT (1+100)	26.5 ± 3.4	40.0 ± 2.0
ICIP + MaxRT (1+100)	20.7 ± 2.9	35.0 ± 3.9
Fully Labeled (100+0)	41.0 ± 2.2	56.9 ± 2.2

	recipes	
	Prog	**Ans**
Few-shot (1+0)	3.9 ± 1.4	7.4 ± 2.6
Few-shot (8+0)	30.4 ± 9.1	37.3 ± 8.3
ICIP + MaxRT (0+100)	23.9 ± 1.0	31.5 ± 1.2
Few-shot + Unl. (1+100)	29.3 ± 3.3	39.2 ± 4.6
ICIP + HardRT (1+100)	43.7 ± 6.0	51.5 ± 5.6
ICIP + MaxRT (1+100)	36.6 ± 7.2	46.3 ± 4.6
Fully Labeled (100+0)	59.1 ± 3.9	70.4 ± 4.0