Can Humans Teach Machines to Code?

Céline Hocquette[1], **Johannes Langer**[2], **Andrew Cropper**[3,4], **Ute Schmid**[2]
[1]University of Southampton
[2]University of Bamberg
[3]ELLIS Institute Finland
[4]University of Helsinki

Abstract

The goal of inductive program synthesis is for a machine to automatically generate a program from user-supplied examples. A key underlying assumption is that humans can provide sufficient examples to teach a concept to a machine. To evaluate the validity of this assumption, we conduct a study where human participants provide examples for six programming concepts, such as finding the maximum element of a list. We evaluate the generalisation performance of five program synthesis systems trained on input-output examples (i) from a human group, (ii) from a *gold standard* set, and (iii) randomly sampled. Our results suggest that human-provided examples are typically insufficient for a program synthesis system to learn an accurate program.

Supplementary material — https://github.com/johannes-langer/can-humans-teach-machines-to-code-aaai26

1 Introduction

Synthesising a computer program from an incomplete specification is known as *(inductive) program synthesis*[1] which is a grand challenge in artificial intelligence (Flener and Schmid 2008; Gulwani et al. 2015). In *programming by example* (PBE), the specification consists of input-output examples demonstrating the desired program's behaviour. Examples are typically provided by humans with little programming experience (Lieberman 2001).

Program synthesis can be viewed as a machine learning problem (Mitchell 1997). However, while standard machine learning approaches learn an attribute-value model (De Raedt 2008), program synthesis approaches learn a computer program, such as a LISP (Summers 1977), Prolog (Shapiro 1983), or Haskell (Katayama 2008) program.

As with all machine learning paradigms, the performance of a program synthesis system heavily depends on the quality of its training examples (Winston 1970; Flach 2012). While most standard machine learning approaches rely on large amounts of training data to ensure good performance, obtaining thousands of examples is often impractical in program synthesis applications where examples are user-provided. For instance, FlashFill (Gulwani 2011), a PBE system in Microsoft Excel, induces string transformation programs from user-provided examples, such as transforming *"Alan Mathison Turing"* to *"A.M. Turing"*. In such scenarios, expecting human users to provide thousands of examples is unrealistic.

A key assumption in user-driven program synthesis is that humans provide sufficient examples for a synthesis system to learn the desired concept (Lieberman 1986; Lau 2009). However, to the best of our knowledge, this assumption lacks both empirical evidence and theoretical support. In other words, it remains an open question whether humans provide sufficient examples for program synthesis systems to generalise effectively.

To address this gap, our goal is to answer the question *"Do humans provide sufficient examples to teach a programming concept to a program synthesis system?"*. By 'sufficient', we mean that a synthesis system learns a program that generalises with high accuracy to unseen examples. This question differs from prior studies, which focus on whether humans teach optimally in terms of the number of examples (Cakmak and Thomaz 2014). Instead, we investigate whether human-provided examples are sufficient for a synthesis system to learn the desired concepts. Additionally, we focus on teaching program synthesis systems rather than traditional machine learning classifiers.

To answer this question, we conduct an empirical study where we ask human participants to provide examples to teach programming concepts[2]. Unlike studies that evaluate the effects of teaching guidance (Cakmak and Thomaz 2014), we do not give participants teaching instructions. Our goal is to evaluate whether humans naturally provide sufficient examples, rather than whether a specific instructional setting could enable them to do so. This choice reflects real-world scenarios where users interact with synthesis systems in unconstrained environments without explicit instructions on how to construct examples. We specifically investigate a non-interactive setting, where synthesis systems learn only from examples without any feedback from the human to the system. This design isolates how humans naturally provide examples without iterative correction. We collect examples from partic-

Copyright © 2026, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

[1]We acknowledge there are other forms of program synthesis, such as deductive program synthesis (Manna and Waldinger 1971). We focus on *inductive* program synthesis.

[2]No ethics statement was required by the home university for this empirical study.

ipants with diverse backgrounds: (i) non-computer scientists, and (ii) computer scientists without expertise in machine learning or program synthesis. Additionally, we use a *gold standard* set of examples as a baseline. We evaluate multiple synthesis systems on these examples, including inductive logic programming (ILP) systems, one Bayesian inductive learning system, and one code-trained large language model (LLM). For each participant, we train each synthesis system on their examples and evaluate the predictive performance of the learned program.

Our study focuses on teaching recursive list concepts. Lists are a core data structure in computer science and have many applications, such as in computational biology, where proteins, genes, and DNA are represented as strings (De Raedt 2008). Moreover, list functions are a standard benchmark for human learning (Rule, Tenenbaum, and Piantadosi 2020; Rule et al. 2024) and program synthesis (Summers 1977; Shapiro 1983; Hofmann, Kitzelmann, and Schmid 2009; Gulwani 2011; Lin et al. 2014; Ellis et al. 2018; Cropper and Muggleton 2019; De Raedt, Schmid, and Langer 2023), including modern LLM-based code generators (Chen et al. 2021).

Novelty and Contributions Our main contribution is an empirical study investigating whether humans can teach machines to code. To our knowledge, our study is the first to evaluate whether human-provided examples are sufficient to train program synthesis systems. Our results show two insights:

1. A common assumption in program synthesis (and machine learning) is that human-provided examples are sufficient to teach a concept to a machine. Our empirical results challenge this assumption and show that, for most of the concepts evaluated, participants do not provide sufficient examples for any of the synthesis systems tested to learn an accurate program. This negative result has potential implications for end-user applications of program synthesis, such as teaching home robot assistants.
2. Our results show that synthesis systems perform better when trained on randomly generated examples compared to those provided by humans. Since many program synthesis (and machine learning) algorithms are optimised using synthetic datasets, this result suggests that synthesis systems might struggle when trained on human-provided examples, highlighting potential limitations for deploying synthesis systems in user-driven applications.

2 Related Work

Program synthesis. LLMs have been widely used to generate programs from a natural language description (Chen et al. 2021; Li et al. 2022; Le et al. 2022), or from a combination of a description and a few examples (Polosukhin and Skidanov 2018; Austin et al. 2021). By contrast, we learn programs using only examples. Recent studies suggest that pretrained LLMs struggle to synthesise programs from just a few examples (Li and Ellis 2024). Although fine-tuning can improve their performance, LLMs still have difficulties with out-of-distribution problems.

Machine teaching. Machine teaching (Zhu 2015; Telle, Hernández-Orallo, and Ferri 2019) is the problem of finding an optimal (usually minimal in size) training set that allows a learner to uniquely identify a target concept. While most research in machine teaching is theoretical, we present empirical results involving humans. Instead of searching for an optimal training set, we examine whether humans naturally provide sufficient examples to teach concepts. Machine teaching considers both humans and machines as teachers and learners (Zhu et al. 2018), while we focus on humans teaching machines.

Human teaching humans. Shafto, Goodman, and Griffiths [2014] show that human teachers select examples in a non-random manner, and prefer those that represent the mean and extent of the underlying distribution, which improves learning effectiveness. Our work considers machine learners instead of human learners.

Human teaching machines. Studies show that humans often teach machines suboptimally, providing more examples than necessary (Khan, Mutlu, and Zhu 2011; Cakmak and Thomaz 2014). Khan, Mutlu, and Zhu [2011] empirically show that human teachers often start with extreme examples, rather than those near the decision boundary, which aligns with the curriculum learning principle. While this study asks participants to select examples from a small predefined set, we ask participants to generate examples. In contrast to previous works on sequential decision-making (Cakmak and Lopes 2012) and binary classification tasks (Cakmak and Thomaz 2014), our study focuses on program synthesis concepts involving recursion. Furthermore, while previous studies evaluate the impact of teaching guidance (Cakmak and Thomaz 2014; Cakmak and Lopes 2012), we provide little teaching guidance to participants and evaluate whether humans naturally provide sufficient examples.

List functions. Lists are a simple yet rich domain. As Rule (2020) explains, lists use numbers in multiple roles (symbols, ordinals, and cardinals) and their recursive structure supports the expression of complex concepts. Lists naturally align with familiar psychological concepts and are commonly used in classic concept learning tasks. However, recursion is cognitively challenging compared to natural categories for which humans can be considered experts (Kahney 1983). Rule [2020] uses list transformation problems to compare the performance of humans with program synthesis approaches. However, their work does not address how effectively humans can teach a program synthesis system a concept.

3 Empirical Study

We conduct an empirical study to explore whether humans naturally provide sufficient examples for a program synthesis system to learn the desired concept. Our primary question is:

Q1. Do humans provide sufficient examples to teach a programming concept to a program synthesis system?

To answer **Q1**, we ask human participants to provide examples to teach programming concepts. We separately train various program synthesis systems on these examples and evaluate the predictive accuracy of the learned programs on

unseen test data. Unlike previous studies that evaluate the impact of teaching guidance on teaching performance (Cakmak and Thomaz 2014), we do not give any instructions on the type of examples to provide. Instead, we evaluate whether humans, without guidance, naturally provide examples from which a program synthesis system generalises the desired concept.

We hypothesise that familiarity with the problem domain might help humans choose better examples. For instance, individuals with programming experience might have a deeper understanding of edge cases, such as the empty list for list manipulation concepts. Therefore, our second question is:

Q2. Does having a background in computer science improve a human's ability to teach a programming concept to a program synthesis system?

To answer **Q2**, we compare the performance of program synthesis systems trained on examples provided by participants with (i) no computer science experience (NCS), and (ii) computer science education but without specific expertise in program synthesis or machine learning (CS).

As mentioned in the introduction, machine learning researchers often optimise their algorithms using synthetic datasets. We investigate whether human-provided examples differ from random/synthetic examples. Therefore, our third question is:

Q3. Do humans provide better examples than randomly sampled examples?

To answer **Q3**, we compare the learning performance of synthesis systems trained on human-provided examples versus randomly generated ones.

Method

List Concepts We use six list manipulation concepts shown in Table 1. These concepts are commonly taught in introductory computer science courses and are standard benchmarks for evaluating both inductive program synthesis systems (Hofmann, Kitzelmann, and Schmid 2009; Ellis et al. 2018; Cropper and Morel 2021; De Raedt, Schmid, and Langer 2023) and human learning (Rule 2020). We selected these concepts because they are accessible to non-experts with no programming background. For each of these concepts, recursion is necessary to write a program that generalises to lists of arbitrary length. Although a program implementing each of these concepts could theoretically operate on lists with elements of any type, we instruct participants to use natural numbers between 0 and 100.

Participants We recruited participants by e-mail. We invited (i) individuals without a computer science background, and (ii) computer science students (BSc, MSc, and PhD). A total of 39 participants completed the study: 14 declared having no computer science background (NCS group), and 25 declared having a computer science background (CS group). The NCS group spent an average of 36min $\pm$ 8min answering the survey and the CS group 21min $\pm$ 3min. We refer to the combined NCS and CS participant groups as the human group. Additionally, we include a *gold standard* set of examples produced by a postgraduate researcher with expertise in program synthesis. This individual was aware of the synthesis systems evaluated and familiar with their underlying algorithms. The gold standard examples serve as a baseline.

Data collection We collect data through a web-based survey that introduced participants to a study on teaching list manipulation concepts to computers using examples. We show participants the following instruction:

> *We will give you a verbal description of a concept. We will then ask you to provide examples of the concept that you think are necessary to teach the concept to a computer. An example has both an input and an output. Inputs and outputs can be lists, natural numbers, Booleans (TRUE/FALSE) or 'none' (no value). Other symbols (negative integers, fractions, ...) are disallowed. Use only natural numbers between 0 and 100. A concept might have one or two inputs.*

We show a worked-out example task (*'count the number of even numbers in a list'*). This example task includes a textual description of the concept (*'Given as input a list, find the number of even numbers in that list.'*) together with three input-output examples ($[0, 2, 4, 6, 8] \mapsto 5$; $[9, 7, 5, 3, 1] \mapsto 0$; and $[0, 5, 9, 4, 3, 1, 6, 7, 8, 10, 2] \mapsto 6$). We tell participants that they will see several verbal descriptions of concepts and give them the following instructions:

> *You can enter up to ten (10) examples for each concept. Try to explain the concept using as few examples as you think are necessary to teach the concept to a computer and give the examples in the most informative order.*

Next, we present six textual descriptions (Table 1) to participants in a random order. Finally, we ask participants to provide demographic information, including their computer science background. The appendix includes the full set of instructions. The instructions do not mention program synthesis, as participants have little to no experience with it. Participants do not interact with the learning systems nor receive feedback while providing examples.

Data Cleaning Some participants included *invalid symbols* in their examples. The number of examples with invalid symbols from the NCS and CS groups respectively is $N_{\text{NCS}} = 0$ and $N_{\text{CS}} = 5$. For instance, a participant provided the example $[a, b, c, d, e, f, g] \mapsto 7$ for *length*, which is invalid because only natural numbers are allowed as elements. We correct invalid symbols by replacing them with the nearest admissible symbol. However, we do not correct elements greater than 100, as the synthesis systems used in the study can handle them. Additionally, some participants provided examples in the *wrong format* ($N_{\text{NCS}} = 13$, $N_{\text{CS}} = 5$). For instance, a participant wrote a semicolon instead of a comma. We re-formatted these examples. We discuss other types of errors, left uncorrected, in Section 4.

Random examples To determine whether human-provided examples lead to better performance than randomly generated ones, we compare human-provided examples against examples drawn from two different random distributions:

RandomUniform. We sample the length of lists in examples from a uniform distribution between 0 and 100, and list elements from a uniform distribution between 0 and 100.

Concept	Description
last	Given as input a list, return the last element of that list.
length	Given as input a list, return the number of elements in that list.
append	Given as input a list and a natural number, return the list with the number inserted at the end of the list.
maxlist	Given as input a list, return the element with the highest value in that list.
dropk	Given as input a list and a natural number k, return the input list without its first k elements.
sorted	Given as input a list, return *true* if the list is sorted in ascending order, and *false* if it is not.

Table 1: Textual descriptions of the programming concepts evaluated.

RandomNormal. We hypothesise that longer examples might be more informative, leading to better performance. To test this hypothesis, we generate examples in which the length of lists follows a normal distribution that reproduces the length distribution of examples provided by participants. We truncate this distribution to ensure positive lengths. We determine its parameters (mean, standard deviation, minimum, and maximum) from the participants' examples for each concept.

Learning Systems We use a portfolio of program synthesis systems representing a variety of synthesis paradigms. We report the best-performing system for each set of examples to evaluate whether any approach can successfully generalise from the provided data. We evaluate the following systems.

POPPER. POPPER (Cropper and Morel 2021; Cropper and Hocquette 2023) is an ILP (Muggleton 1991) system based on constraint programming. POPPER tolerates noisy examples and can learn recursive programs from only positive examples. POPPER is guaranteed to learn a solution (a program that correctly generalises the training examples) if one exists in the search space. It has been used on list function tasks (Hocquette and Cropper 2024).

METAGOL. METAGOL (Muggleton, Lin, and Tamaddoni-Nezhad 2015) is an ILP system based on a Prolog meta-interpreter. It can also learn recursive programs from positive examples only and is guaranteed to learn a solution. It has also been used on list function tasks (Lin et al. 2014; Rule et al. 2024).

ALEPH. ALEPH (Srinivasan 2001) is a separate-and-conquer ILP system which tolerates noisy examples.

HL. Hacker-Like (HL) (Rule 2020; Rule et al. 2024) is a Bayesian inductive learning system designed to model human-like learning. It uses Monte Carlo tree search to search over metaprograms. It is specifically designed to learn list manipulation programs.

DEEPSEEK-CODER. DEEPSEEK-CODER-V2 (Zhu et al. 2024) is a Mixture-of-Experts code language model trained on a variety of programming languages. We use the small 16B parameter version hosted locally within the Ollama framework. We selected this model because it can be run entirely offline and is open-source. For consistency with the ILP systems used, we prompt the model to generate Prolog programs. However, since Prolog is not natively supported by the model, we also evaluate its performance when prompted to generate Python code. We independently evaluate the performance from both languages and report results for the better-performing one.

Learning Settings We use the built-in bias for HL. Following the literature (Cropper and Morel 2021), we allow the other systems to learn programs with the relations *max*, *head*, *tail*, *decrement*, *increment*, *geq*, *eq*, *empty*, *zero*, *one*, *even*, and *odd*. For the concept *append*, we additionally include the relation *comps*.

For DEEPSEEK-CODER, we use a two-stage prompting process. In the first prompt, we provide the training examples, together with the arity and types of the target concept. We instruct the system to explain the function that produces the given input-output pairs and to generate corresponding code (in either Prolog or Python). Since the system's responses do not follow a consistent format, we use a second prompt to extract and return only the code from the previous response. We found that this approach yields more reliable and higher-quality results than directly prompting for code alone. Once a program is generated, we test it against the training examples. If the program contains syntax errors or fails on any examples, we re-prompt the model with the errors and request a revised program. We allow up to three revisions, as we observed that the model rarely improves its program beyond three revisions. The appendix shows our prompts.

Experimental Procedure For each programming concept and each participant π, we train each synthesis system s separately using the first n examples provided by π. We evaluate the predictive accuracy (the proportion of correct predictions on unseen test data) of the program learned by s using 2000 test examples. The default accuracy is 50%. We aggregate the predictive accuracies across all synthesis systems and report the maximum. In other words, we report the highest accuracy achieved by any synthesis system to evaluate whether at least one synthesis system can effectively learn from the examples provided. We vary the number of training examples n from 1 to 10 and repeat training. We repeat these steps for each participant and programming concept. We compare the predictive accuracies of the NCS and CS groups against those of the *gold standard* and the random distributions (*RandomUniform* and *RandomNormal*). We calculate the mean and standard deviation. Error bars in the figures and tables denote standard deviation. We also evaluate the statistics of the examples provided by participants. Box plots display the minimum, first quartile, median, third quartile, and maximum values. We use an Intel compute node with dual 2.0 GHz Intel Xeon Gold 6138 processors, 40 CPU cores, and 192 GB of DDR4 memory.

Results

We present the results of our empirical study.

Q1. Do humans provide sufficient examples? Figures 1 and 2 show the average predictive accuracies of the best-performing synthesis system trained on the first n examples provided by each participant, aggregated across all programming concepts. The appendix shows the accuracies for each programming concept. These figures show that the *gold standard* examples consistently provide sufficient information for at least one synthesis system to achieve maximal predictive accuracy (100%) across all programming concepts. This result demonstrates the existence of a set of examples from which a synthesis system can learn each target concept. Because this finding supports an existential claim, a single *gold standard* set is sufficient.

By contrast, Figures 1 and 2 show that both the NCS and CS groups achieve, on average, lower accuracy than the *gold standard*. This result suggests that humans often struggle to provide examples from which a program synthesis system correctly identifies the desired concept. In other words, while a set of sufficient examples exists, as demonstrated by the *gold standard*, humans often fail to provide them. For instance, for the concept *sorted*, none of the NCS participants achieve maximal accuracy. Out of 14 NCS participants, 11 achieve at least 70% accuracy, and 7 at least 90%. Out of 25 CS participants, only 13 achieve at least 70% accuracy, 10 at least 90%, and 3 maximum (100%) accuracy.

Overall, these results suggest that the answer to **Q1** is **no**: humans typically struggle to provide sufficient examples to teach programming concepts to synthesis systems. We discuss potential reasons for this result in Section 4.

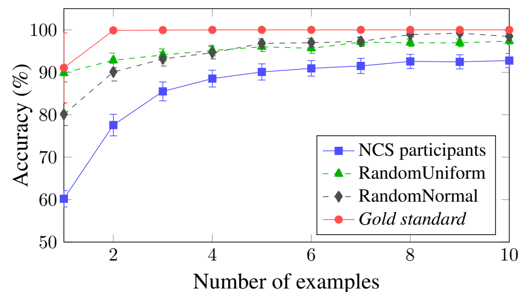

Figure 1: Predictive accuracies for the NCS group averaged over all concepts when trained on progressively larger example sets.

Q2. Does having a background in computer science improve a human's teaching ability? Table 2 shows the predictive accuracies for each concept achieved from all of the examples provided by an NCS or CS participant. A Mann-Whitney U-test (Mann and Whitney 1947) finds no statistically significant differences between the NCS and CS groups for any of the concepts. We use a non-parametric test because a Shapiro-Wilk test (Shapiro and Wilk 1965) indicates that the accuracies are not normally distributed (most example sets result in either maximum accuracy (100%) or default accuracy (50%) and only a few sets achieve intermediate values). This result suggests that, in this context and among non-experts, a computer science background does not significantly affect the ability to teach a programming concept to a

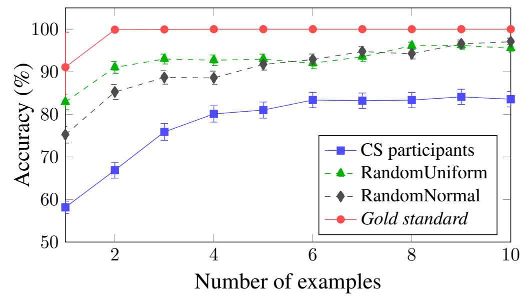

Figure 2: Predictive accuracies for the CS group averaged over all concepts when trained on progressively larger example sets.

synthesis system. In other words, domain-specific knowledge alone may not always enhance teaching effectiveness.

However, the *gold standard* consistently achieves maximal accuracy (100%), which may indicate that understanding the learner's algorithm plays an important role. This observation suggests that effective teaching might instead require both familiarity with the concepts and an understanding of how the learner's algorithm works. Further investigation is needed to confirm this hypothesis and clarify the role of algorithmic understanding in teaching effectiveness.

Overall, these results suggest that the answer to **Q2** is likely **no**: a participant's background does not significantly impact their ability to teach a programming concept to a synthesis system. However, understanding the learner's algorithm could play an important role in effective teaching.

	NCS	**CS**
last	89 ± 21	$\mathbf{93} \pm 17$
length	$\mathbf{100} \pm 0$	94 ± 16
append	$\mathbf{95} \pm 11$	$\mathbf{95} \pm 14$
maxlist	$\mathbf{96} \pm 13$	79 ± 24
dropk	$\mathbf{92} \pm 18$	66 ± 23
sorted	$\mathbf{85} \pm 17$	74 ± 19

Table 2: Mean and standard deviation of predictive accuracies for full example sets for the NCS and CS groups.

Q3. Do humans provide better examples than randomly sampled examples? Figures 1 and 2 show the average predictive accuracies of the best-performing synthesis systems trained with human examples and examples with length sampled from a uniform or normal distribution. For each participant group (CS or NCS), we conduct tests using the predictive accuracies from two, three, or all ten examples, resulting in a total of six analyses. We choose these specific points to test how informative the first examples are, and the full sets of examples. A Kruskal-Wallis H-test (Kruskal and Wallis 1952), adjusted with a Benjamini-Hochberg false discovery control method (Benjamini and Hochberg 1995) to account for family-wise error rate[3], shows a significant group effect

[3]When performing multiple statistical tests on the same data, the probability of not falsely rejecting a null hypothesis is $(1-\alpha)^n$ with n being the number of tests performed.

when using two examples for the NCS group and two or ten examples for the CS group. Post-hoc U-tests further identify that, for the NCS group, accuracies from human-provided examples are significantly lower than accuracies from the uniform distribution when using 2 examples ($p = .01$). For the CS group, accuracies from human-provided examples are significantly lower than those from both random distributions when using 2 examples ($p < .001$), and lower than accuracies from the normal distribution when using 10 examples. These results indicate that the NCS and CS groups perform worse than both random distributions. We discuss potential reasons for these findings in Section 4.

Overall, these results suggest that the answer to **Q3** is likely **no**: humans do not provide better examples than randomly sampled ones.

4 Discussion

The results from Section 3 suggest that non-expert humans, regardless of their background in computer science, struggle to provide sufficient examples to teach programming concepts to synthesis systems. In this section, we discuss potential explanations.

Erroneous Examples One potential explanation for the lower accuracy of the NCS and CS groups is the presence of errors in their examples. We identify two types of errors.

A *systematic error* occurs when all examples in an example set contain the same error ($N_{\text{NCS}} = 3$, $N_{\text{CS}} = 8$). This type of error suggests that the participant provided examples for a different concept. For instance, an NCS participant provided examples for the concept *"return the first k elements of a list"* for *dropk*, such as $[4, 8, 7, 6, 2, 1, 5, 4, 8, 3, 1], 2 \mapsto [4, 8]$, when the correct output is $[7, 6, 2, 1, 5, 4, 8, 3, 1]$. Similarly, a CS participant provided examples for the concept *"remove the element k in the input list"* for *dropk* and provided examples such as $[20, 13, 42, 53, 23, 21], 23 \mapsto [20, 13, 42, 53, 21]$.

A *non-systematic error* is an error on a single example of an example set ($N_{\text{NCS}} = 1$, $N_{\text{CS}} = 3$). For instance, a CS participant provided the example $[10, 0, 59, 68, 23, 42, 53] \mapsto 59$ for *maxlist*, and an NCS participant gave $[99999, 7676768] \mapsto false$ for *sorted*.

To evaluate the impact of these errors, we exclude examples with either a systematic or non-systematic error and retrain the synthesis systems. Table 3 shows the resulting predictive accuracies. A U-test reveals a significant difference between the NCS and CS groups for *dropk*, with the NCS group performing better. This result suggests that, in some cases, humans with domain-specific knowledge may provide less suitable examples than those without. However, examples provided by humans without such knowledge are more impacted by errors.

While excluding erroneous examples improves accuracy, they still are not maximal. This result suggests that, even without errors, the examples provided by humans are insufficient for the systems to correctly generalise. For instance, for *sorted*, a participant provided only the examples $[1, 2, 3, 4, 5] \mapsto true$; $[0, 0, 0, 0, 1] \mapsto true$; $[5, 4, 3, 2, 1] \mapsto false$; $[1, 0, 0, 0, 0] \mapsto false$, from which a system incorrectly learns that a list is sorted if its second element is greater than or equal to its first.

	NCS	CS
last	89 ± 21	$\mathbf{93} \pm 17$
length	$\mathbf{100} \pm 0$	94 ± 16
append	95 ± 11	$\mathbf{100} \pm 0$
maxlist	$\mathbf{96} \pm 13$	79 ± 25
*dropk**	$\mathbf{99} \pm 1$	69 ± 25
sorted	$\mathbf{84} \pm 17$	74 ± 19

Table 3: Mean and standard deviation of predictive accuracies for full example sets for the NCS and CS groups, after ignoring examples with systematic or non-systematic errors. $^* : p < .05$

In summary, this result suggests that, while human errors limit performance, participants also did not provide sufficient examples.

Simpler Examples Another potential factor that may contribute to lower accuracy is the simplicity of humans' examples. To examine this explanation, we analyse their number, length, and element values.

Number of examples. Figure 3 shows the distribution of the number of examples provided by participants. It shows that both the NCS and CS groups typically provided more examples than the *gold standard* for all concepts. For instance, the *gold standard* set contains only 3 examples for *maxlist*, while the NCS and CS groups provided 6.7 ± 0.7 and 6.6 ± 0.5 examples, respectively. An H-test reveals a significant group effect ($p < .05$). Post-hoc U-tests show that the *gold standard* set contains significantly fewer examples than those provided by both the NCS and CS groups across all concepts ($p < .05$). We use a non-parametric test since a Shapiro-Wilk test indicates that the number of examples is not normally distributed. This result might surprise the reader, as one might expect that more examples would improve teaching effectiveness. However, as Telle, Hernández-Orallo, and Ferri [2019] note, effective machine teaching depends not only on the number of examples but also on their length.

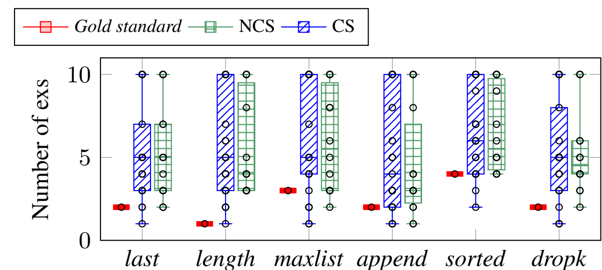

Figure 3: Number of examples provided by participants.

Length of lists. Figure 4 shows the distribution of list lengths of the examples provided by participants. The NCS and CS groups generally produced examples of similar lengths across programming concepts, whereas the *gold stan-*

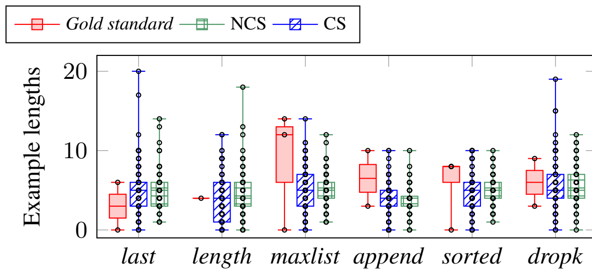

Figure 4: Length of lists provided by participants.

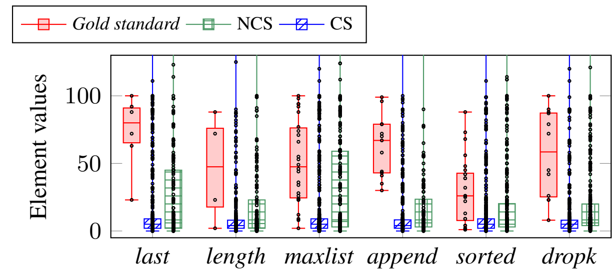

Figure 5: Values of elements provided by participants.

dard examples vary in length depending on the concept. For instance, for *last* and *length*, at least 75% of examples have fewer than 6 elements for both participant groups and the *gold standard* set. However, for *maxlist*, the *gold standard* examples have lengths 0, 12, and 14, whereas at least 75% from the NCS and CS groups contain fewer than seven elements. This difference is important because longer examples can contain more bits and hence carry more information. For instance, for *last*, a participant provided two examples of length 4, leading a synthesis system to learn the concept *return the fourth element of the input list*, which is simpler to express than the intended concept *last*. By contrast, the *gold standard* set includes a single example of length 6, from which a synthesis system correctly learns the concept *last*, as *return the sixth element of the list* is more complex to express than *last*. Participants may prefer shorter examples because they are easier to generate and cognitively simpler, whereas the *gold standard* examples include longer instances.

Element values. Figure 5 shows the distribution of element values in the lists provided by participants. It shows that elements in lists from both the NCS and CS groups tend to have smaller values and less variability compared to those in the *gold standard* examples. For instance, for *sorted*, 75% of elements are below 9 for the CS group and below 20 for the NCS group, whereas 50% of elements in the *gold standard* examples range between 8 and 43. This difference is important because lower variability in element values increases the likelihood of coincidental patterns. For instance, for *sorted*, an NCS participant provides the examples $[1, 3, 6, 9, 14, 18] \mapsto true$; $[1, 2, 3, 4, 5] \mapsto true$; $[0, 1] \mapsto true$; $[3, 1, 12, 2, 7] \mapsto false$. From these examples, a synthesis system incorrectly learns that a list is sorted if its first element is strictly less than 2. In contrast, the *gold standard* examples have greater diversity and include sorted lists with unique, non-repeating elements, reducing the chance of such spurious generalisations.

Overall, these results suggest that the quality of examples, rather than their quantity, plays a crucial role in effective teaching.

5 Conclusions and Limitations

Our empirical study explores whether humans can teach machines to code. Our results are negative and suggest that non-experts, regardless of their computer science background, struggle to provide sufficient examples to effectively teach list manipulation concepts to program synthesis systems. However, there exists a *gold standard* set of examples from which a synthesis system achieves perfect accuracy, demonstrating that sufficient examples exist for the programming concepts tested. Furthermore, our results suggest that synthesis systems perform better when trained on randomly generated examples than on those provided by humans.

Future Work

Eliciting teaching strategies. We show participants a general instruction which did not specify that examples are for a program synthesis system, which may have led them to give general-purpose examples. However, our results show that CS participants made use of their knowledge and, for instance, often included an example for the base case for recursive concepts. Future work should investigate how teaching instructions, as suggested by Khan, Mutlu, and Zhu (2011) and Cakmak and Thomaz (2014), could elicit more effective teaching strategies.

Better systems. We tested 5 synthesis systems, but none consistently learns the desired concepts from human examples. One system consistently learns perfectly accurate programs from the *gold standard* examples. However, an untested system might successfully learn from human data.

Outlook. In PBE (Lieberman 2001) and end-user programming (Gulwani, Polozov, and Singh 2017), it is assumed that humans can provide sufficient examples to teach a concept. Our findings suggest that this assumption does not hold in non-interactive settings. However, in practice, users may interact iteratively with synthesis systems in a feedback loop. Exploring such interactive scenarios represents an important direction for future work. Furthermore, most of the program synthesis research relies on randomly generated examples for training. Our results suggest that training using random examples yields substantially better performance compared to using human examples. If these systems are intended to be eventually trained by humans, this result should motivate researchers to incorporate more human-generated examples into training datasets. Consequently, our study raises two challenges for the field. First, our study emphasises the necessity to develop systems that better adapt to human examples, for instance with hybrid machine-human example selection. Second, our results highlight the need to build algorithms capable of generating training sets of random examples that more accurately reflect human examples.

Acknowledgments

For open access, the authors have applied a CC BY public copyright licence to any author-accepted manuscript version arising from this submission. The authors thank Lun Ai, Filipe Gouveia, and Minghao Liu for valuable feedback, and the participants for their contributions to the study. Andrew Cropper was supported by his EPSRC fellowship (EP/V040340/1).

References

Austin, J.; Odena, A.; Nye, M.; Bosma, M.; Michalewski, H.; Dohan, D.; Jiang, E.; Cai, C.; Terry, M.; Le, Q.; et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Benjamini, Y.; and Hochberg, Y. 1995. Controlling the false discovery rate: a practical and powerful approach to multiple testing. *Journal of the Royal statistical society: series B (Methodological)*, 57(1): 289–300.

Cakmak, M.; and Lopes, M. 2012. Algorithmic and Human Teaching of Sequential Decision Tasks. In Hoffmann, J.; and Selman, B., eds., *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, July 22-26, 2012, Toronto, Ontario, Canada*, 1536–1542. AAAI Press.

Cakmak, M.; and Thomaz, A. L. 2014. Eliciting good teaching from humans for machine learners. *Artif. Intell.*, 217: 198–215.

Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; de Oliveira Pinto, H. P.; et al. 2021. Evaluating Large Language Models Trained on Code. *CoRR*, abs/2107.03374.

Cropper, A.; and Hocquette, C. 2023. Learning Logic Programs by Combining Programs. In *ECAI 2023 - 26th European Conference on Artificial Intelligence*, volume 372, 501–508. IOS Press.

Cropper, A.; and Morel, R. 2021. Learning programs by learning from failures. *Mach. Learn.*, 110(4): 801–856.

Cropper, A.; and Muggleton, S. H. 2019. Learning efficient logic programs. *Mach. Learn.*, 108(7): 1063–1083.

De Raedt, L. 2008. *Logical and relational learning*. Cognitive Technologies. Springer. ISBN 978-3-540-20040-6.

De Raedt, L.; Schmid, U.; and Langer, J. 2023. Approaches and Applications of Inductive Programming (Dagstuhl Seminar 23442). *Dagstuhl Reports*, 13(10): 182–211.

Ellis, K.; Morales, L.; Sablé-Meyer, M.; Solar-Lezama, A.; and Tenenbaum, J. 2018. Learning Libraries of Subroutines for Neurally-Guided Bayesian Program Induction. In *NeurIPS 2018*, 7816–7826.

Flach, P. A. 2012. *Machine Learning - The Art and Science of Algorithms that Make Sense of Data*. Cambridge University Press. ISBN 978-1-10-742222-3.

Flener, P.; and Schmid, U. 2008. An introduction to inductive programming. *Artificial Intelligence Review*, 29(1): 45–62.

Gulwani, S. 2011. Automating string processing in spreadsheets using input-output examples. In *POPL*, 317–330.

Gulwani, S.; Hernández-Orallo, J.; Kitzelmann, E.; Muggleton, S. H.; Schmid, U.; and Zorn, B. G. 2015. Inductive programming meets the real world. *Commun. ACM*, 58(11): 90–99.

Gulwani, S.; Polozov, O.; and Singh, R. 2017. Program Synthesis. *Found. Trends Program. Lang.*, 4(1-2): 1–119.

Hocquette, C.; and Cropper, A. 2024. Relational decomposition for program synthesis. *arXiv preprint arXiv:2408.12212*.

Hofmann, M.; Kitzelmann, E.; and Schmid, U. 2009. A unifying framework for analysis and evaluation of inductive programming systems. In *2nd Conference on Artificiel General Intelligence (2009)*, 74–79. Atlantis Press.

Kahney, H. 1983. What do novice programmers know about recursion. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 235–239.

Katayama, S. 2008. Efficient Exhaustive Generation of Functional Programs Using Monte-Carlo Search with Iterative Deepening. In *PRICAI 2008: Trends in Artificial Intelligence*, 199–210.

Khan, F.; Mutlu, B.; and Zhu, J. 2011. How do humans teach: On curriculum learning and teaching dimension. In *Advances in Neural Information Processing Systems 24: 25th Annual Conference on Neural Information Processing Systems.*, 1449–1457.

Kruskal, W. H.; and Wallis, W. A. 1952. Use of ranks in one-criterion variance analysis. *Journal of the American statistical Association*, 47(260): 583–621.

Lau, T. 2009. Why programming-by-demonstration systems fail: Lessons learned for usable AI. *AI Magazine*, 30(4): 65–65.

Le, H.; Wang, Y.; Gotmare, A. D.; Savarese, S.; and Hoi, S. C. H. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35: 21314–21328.

Li, W.-D.; and Ellis, K. 2024. Is programming by example solved by LLMs? *Advances in Neural Information Processing Systems*, 37: 44761–44790.

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

Lieberman, H. 1986. An example based environment for beginning programmers. *Instructional Science*, 14(3): 277–292.

Lieberman, H. 2001. *Your wish is my command: Programming by example*. Morgan Kaufmann.

Lin, D.; Dechter, E.; Ellis, K.; Tenenbaum, J. B.; and Muggleton, S. 2014. Bias reformulation for one-shot function induction. In *ECAI*, 525–530.

Mann, H. B.; and Whitney, D. R. 1947. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, 50–60.

Manna, Z.; and Waldinger, R. J. 1971. Toward automatic program synthesis. *Communications of the ACM*, 14(3): 151–165.

Mitchell, T. M. 1997. *Machine Learning*. McGraw-Hill.

Muggleton, S. H. 1991. Inductive Logic Programming. *New Gener. Comput.*, 8(4): 295–318.

Muggleton, S. H.; Lin, D.; and Tamaddoni-Nezhad, A. 2015. Meta-interpretive learning of higher-order dyadic Datalog: predicate invention revisited. *Mach. Learn.*, 100(1): 49–73.

Polosukhin, I.; and Skidanov, A. 2018. Neural program search: Solving programming tasks from description and examples. *arXiv preprint arXiv:1802.04335*.

Rule, J. S. 2020. *The child as hacker: building more human-like models of learning*. Ph.D. thesis, Massachusetts Institute of Technology.

Rule, J. S.; Piantadosi, S. T.; Cropper, A.; Ellis, K.; Nye, M.; and Tenenbaum, J. B. 2024. Symbolic metaprogram search improves learning efficiency and explains rule learning in humans. *Nature Communications*, 15(1): 6847.

Rule, J. S.; Tenenbaum, J. B.; and Piantadosi, S. T. 2020. The Child as Hacker. *Trends in Cognitive Sciences*, 24(11): 900–915.

Shafto, P.; Goodman, N. D.; and Griffiths, T. L. 2014. A rational account of pedagogical reasoning: Teaching by, and learning from, examples. *Cognitive Psychology*, 71: 55–89.

Shapiro, E. Y. 1983. *Algorithmic Program DeBugging*. Cambridge, MA, USA: MIT Press. ISBN 0262192187.

Shapiro, S. S.; and Wilk, M. B. 1965. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3-4): 591–611.

Srinivasan, A. 2001. The ALEPH manual. *Machine Learning at the Computing Laboratory, Oxford University*.

Summers, P. D. 1977. A Methodology for LISP Program Construction from Examples. *J. ACM*, 24(1): 161–175.

Telle, J. A.; Hernández-Orallo, J.; and Ferri, C. 2019. The teaching size: computable teachers and learners for universal languages. *Machine Learning*, 108(8): 1653–1675.

Winston, P. H. 1970. Learning structural descriptions from examples. Technical Report MIT/LCS/TR-76, MIT.

Zhu, Q.; Guo, D.; Shao, Z.; Yang, D.; Wang, P.; Xu, R.; Wu, Y.; Li, Y.; Gao, H.; Ma, S.; et al. 2024. DeepSeek-Coder-V2: Breaking the Barrier of Closed-Source Models in Code Intelligence. *arXiv preprint arXiv:2406.11931*.

Zhu, X. 2015. Machine Teaching: An Inverse Problem to Machine Learning and an Approach Toward Optimal Education. In Bonet, B.; and Koenig, S., eds., *AAAI*, 4083–4087. AAAI Press.

Zhu, X.; Singla, A.; Zilles, S.; and Rafferty, A. N. 2018. An overview of machine teaching. *arXiv preprint arXiv:1801.05927*.