Can Humans Teach Machines to Code?

Céline Hocquette[a,*], **Johannes Langer**[b], **Andrew Cropper**[a] **and Ute Schmid**[b]

[a]University of Oxford
[b]University of Bamberg

Abstract. The goal of inductive program synthesis is for a machine to automatically generate a program from user-supplied examples of the desired behaviour of the program. A key underlying assumption is that humans can provide examples of sufficient quality to teach a concept to a machine. However, as far as we are aware, this assumption lacks both empirical and theoretical support. To address this limitation, we explore the question *"Can humans teach machines to code?"*. To answer this question, we conduct a study where we ask humans to generate examples for six programming tasks, such as finding the maximum element of a list. We compare the performance of a program synthesis system trained on (i) human-provided examples, (ii) randomly sampled examples, and (iii) expert-provided examples. Our results show that, on most of the tasks, non-expert participants did not provide sufficient examples for a program synthesis system to learn an accurate program. Our results also show that non-experts need to provide more examples than both randomly sampled and expert-provided examples.

1 Introduction

Synthesising a computer program from an incomplete specification is known as *(inductive) program synthesis*[1] and is a grand challenge in AI [14, 17]. In *programming by examples* (PBE), the specification consists of a set of input-output examples of the desired program's behaviour. These examples are typically provided by humans with little programming experience [27].

Program synthesis can be viewed as a machine learning problem [31]. However, whereas standard machine learning approaches learn a model represented as attribute-value pairs [10], program synthesis approaches learn a computer program, such as a LISP [41], Prolog [38], or Haskell [21] program.

As with all forms of machine learning, the performance of a program synthesis system depends on the quality of its training examples [43, 13]. To achieve good performance, training examples must be representative of the concept being taught. Most machine learning approaches rely on large amounts of training data to ensure good performance. However, in many program synthesis applications, it is impractical to obtain thousands of training examples, especially applications that obtain examples directly from a user. For instance, FlashFill [16] induces string transformation programs from user-provided examples in Microsoft Excel, such as transforming *"Alan Mathison Turing"* to *"A.M. Turing"*. In such scenarios, it is unrealistic to expect human users to provide thousands of training examples.

In these user-focused applications, it is widely assumed that humans can provide sufficient training examples for a synthesis system to learn the target concept [26, 24]. 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 to teach a program synthesis system a concept.

To overcome this knowledge gap, our goal is to answer the question *"Can humans teach machines to code?"*. Specifically, our research question is *"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 with high accuracy on unseen examples of the target concept. This question differs from existing work that explores whether humans can teach arbitrary concepts to arbitrary machine learners [5] since we focus on teaching program synthesis concepts. Moreover, we are interested in whether humans naturally supply examples from which a program synthesis system generalises the desired concept.

To answer this question, we focus on recursive list manipulation concepts. We focus on lists as they have been extensively investigated both in program synthesis [41, 38, 19, 16, 28, 11, 9, 35] and human learning [34, 36] contexts[2]. For instance, suppose you want to teach a machine the concept *"return the last element of a list"*. Then a human teacher might provide examples such as *[a,l,i,c,e]* $\mapsto e$, *[j,i,m]* $\mapsto m$, *[s,a,l,l,y]* $\mapsto y$. Given these examples, a program synthesis system should generate a program that correctly generalises the training examples and, crucially, generalises to unseen examples.

We conduct an empirical study where we ask human participants to provide minimal sets of examples to teach programming concepts. We do not provide teaching guidance to participants, as we are interested in whether humans naturally provide sufficient examples to teach a program synthesis systems, rather than exploring whether humans can, in principle, provide helpful examples. We solicit examples from three different groups: (i) non-computer scientists, (ii) computer scientists who do not necessarily work on machine learning or program synthesis, and (iii) expert computer scientists who know about program synthesis. We hypothesise experts know the best examples to provide. We evaluate the predictive performance of a program synthesis system trained on each of these example sets.

Novelty and Contributions

The main contribution of this paper is an empirical study that investigates whether humans can teach machines to code. As far as we are aware, this empirical study is the first that evaluates how well humans can train program synthesis systems. Our results show two

* Corresponding Author Email: celine.hocquette@cs.ox.ac.uk.

[1] We acknowledge there are other forms of program synthesis, such as deductive program synthesis [30]. We focus on *inductive* program synthesis.

[2] See Rule [35] (Chapter 3) for more background on list concepts.

key insights. First, many program synthesis (and machine learning) researchers work under the assumption that humans are capable of providing sufficient examples to teach a concept to a machine. However, our empirical results suggest that this assumption does not hold. On most of the tasks evaluated, non-expert participants did not provide sufficient examples for the program synthesis system to learn an accurate hypothesis. This result has potential implications for the applications of program synthesis. For instance, if using program synthesis to teach a home robot assistant to perform a task, our results suggest that a human could need to provide more examples than they would prefer to achieve the desired functionality. Second, our results show that synthesis systems perform better on randomly generated examples compared to examples provided by non-expert humans. Since many machine learning researchers optimise their algorithms on synthetic datasets, our results suggest that such systems might struggle when trained on human-provided examples.

2 Related Work

Machine teaching. Machine teaching [44, 42] is the problem of finding an optimal (usually minimal in size) training set for a teacher such that a learner can uniquely identify a target concept. Goldman and Kearns [15] show that the problem of finding an optimal teaching sequence for an arbitrary concept, i.e. of finding a minimal sequence of examples, is NP-hard. While most existing literature on machine teaching are theoretical results based on learnability, we present empirical results involving humans. Machine teaching encompasses different scenarios in which the teacher and the learner may be humans or machines [45]. In this paper, our focus is on the scenario of humans teaching machines. Our work also differs in that we do not search for an optimal training set, but instead investigate whether humans naturally provide sufficient training examples to effectively teach concepts.

Curriculum learning. In curriculum learning, training examples are ordered, typically by increasing complexity, to facilitate learning [12, 1]. Curriculum learning can accelerate convergence towards more accurate models [1, 28]. Lelis et al. [25] look at the problem of using neural-guided tree search algorithms to generate curricula for helping humans learn puzzle games. In our work, we evaluate whether humans provide helpful examples to teach a machine.

Pedagogical reasoning. In the context of human-to-human concept teaching, it is generally assumed that the human teacher has full knowledge of a concept. Their goal is to choose data to facilitate learning. Shafto et al. [37] show that examples selected by the teacher exhibit a distinct non-random pattern. For instance, when teaching prototype concepts, teachers purposefully select examples that represent the mean and extent of the true distribution, rather than selecting examples at random. Such purposeful selection improves the effectiveness of the learning process compared to random examples. Our work considers machine learners instead of human learners.

Human teaching humans. In the context of human-to-human teaching, empirical evidence suggests that human teachers select examples which they consider to be most helpful to the information needs of a human learner [37].

Human teaching machines. An empirical study by Khan et al. [22] explores the strategies employed by humans when teaching machines. Their study centres on the task of teaching a threshold value to a robot. Their results show that human teachers do not provide examples near the decision boundary but use extreme examples first which is consistent with the curriculum learning principle. Furthermore, their results show that human teachers did not provide minimal example sets. Cakmak and Thomaz [5] investigate humans teaching machine through three binary classification tasks: Chernoff faces, image classification, and animals. Their results also show that humans do not generate minimal example sets. These works focus on tasks involving sequential decision-making [4] and binary classification [5]. By contrast, our study focuses on program synthesis tasks, which use recursion. Cakmak and Thomaz [5] evaluate the impact of different teaching guidance on learning performance. By contrast, we compare the performance of machine learners trained on human-generated examples against those trained on randomly sampled examples. Butler et al. [3] show that generating a solution requires more cognitive effort than selecting it from a set of predefined sets. By contrast, we provide little teaching guidance to participants and evaluate whether humans naturally do provide sufficient examples.

List transformations. Rule [35] uses list transformation problems to explore how well humans can learn concepts and how well humans perform compared to program synthesis approaches. However, they do not consider the question of how well a human can teach a program synthesis system a concept. Recursion is crucial to generalise from a small number of examples to lists of arbitrary size. However, recursion is a cognitively challenging task in contrast to natural categories for which humans can be considered experts [20].

3 Teaching Problem

We describe our problem setting.

3.1 PBE problem

An example is a pair (i, o) formed of an input i and an output o. We denote as $\mathcal{X}$ an example space, i.e. a set of examples. We denote as $\mathcal{H}$ a hypothesis space, i.e. a set of programs. We define a PBE task:

Definition 1 (PBE task). A *PBE task* is a tuple $(E, \mathcal{H})$ where $E \subseteq \mathcal{X}$ is a set of examples.

Given a program p and an input i, we denote as $[\![p(i)]\!]$ the result of executing p on i. We define a PBE solution:

Definition 2 (PBE solution). For a PBE task $(E, \mathcal{H})$, a program $p \in \mathcal{H}$ is a *PBE solution* when p satisfies every example in E, i.e. $[\![p(i)]\!] = o$ for every $(i, o) \in E$.

We evaluate a PBE solution using *predictive accuracy*. We assume a probability distribution D over $\mathcal{X}$ and define the predictive accuracy:

Definition 3 (Predictive accuracy). The *predictive accuracy* of $p \in \mathcal{H}$ with respect to a target program $t \in \mathcal{H}$ and distribution D over $\mathcal{X}$ is the probability that p produces the correct output for an example $(i, o) \in \mathcal{X}$ drawn at random according to D:

$$acc(p, t) = P_{(i,o)\in\mathcal{X}\sim D}[\![p(i)]\!] = [\![t(i)]\!]$$

3.2 Teaching Problem

We define a teaching task:

Definition 4 (Teaching task). A *teaching task* is a tuple $(\mathcal{X}, t, \alpha)$ where $t \in \mathcal{H}$ is a *target* program and α is a real number verifying $0 \leq \alpha \leq 1$.

We define a teaching solution:

Definition 5 (**Teaching solution**). For a teaching task $(\mathcal{X}, t, \alpha)$ and a distribution D over $\mathcal{X}$, a set of examples $E \subseteq \mathcal{X}$ is a *teaching solution* when there exists a solution p to the PBE task $(E, \mathcal{H})$ which has predictive accuracy with respect to t and D at least α: $acc(p, t) \geq \alpha$.

We call $E \subseteq \mathcal{X}$ an *example set* of $\mathcal{X}$. Let $\mathcal{P}(\mathcal{X})$ be the powerset of $\mathcal{X}$, i.e. the set of all example sets of $\mathcal{X}$. In general, there might be multiple solutions to a teaching task. We associate a cost to each example set and prefer optimal solutions, which are solutions with minimal cost. Let $cost : \mathcal{P}(\mathcal{X}) \rightarrow \mathbb{N}$ be an arbitrary cost function that measures the cost of an example set. We define an optimal teaching solution:

Definition 6 (**Optimal teaching solution**). For a teaching task $(\mathcal{X}, t, \alpha)$, a set of examples $E \subseteq \mathcal{X}$ is an *optimal* teaching solution when (i) E is a teaching solution for $(\mathcal{X}, t, \alpha)$, and (ii) for all $E' \subseteq \mathcal{X}$ such that E' is a teaching solution for $(\mathcal{X}, t, \alpha)$, $cost(E) \leq cost(E')$.

In other words, the teaching problem is to find a set of examples from which a learner can identify the target program. In the following, we use the term *target concept* interchangeably with the term *target program*.

In this paper, our goal is to evaluate the performance of humans in solving teaching tasks. In other words, given a target program t, our goal is to evaluate how well humans find an example set E so that a learner can learn t from E.

4 Empirical Study

We conduct an empirical study to explore whether humans provide a teaching solution (Definition 5) to teaching tasks (Definition 4), i.e. whether humans provide an example set from which a program synthesis system learns 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 participants to provide examples to solve teaching tasks where the target is a programming concept. We then evaluate the predictive accuracies of a program synthesis system trained with human-provided examples. We evaluate whether human examples are sufficient to achieve high predictive accuracies. Unlike similar studies [5, 22], we do not give teaching guidance to participants, i.e. we do not give instructions on the kind of examples they should provide. In this open-ended context, rather than exploring whether humans can, in principle, provide helpful examples, we are interested in whether humans naturally supply examples from which a program synthesis system generalises the desired concept.

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

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

To answer **Q2**, we compare the learning performance of a program synthesis system trained with examples from participants with varying levels of computer science expertise. Specifically, participants belong to three groups (i) a group with no programming experience (NCS), (ii) a group with computer science education (CS), and (iii) an expert in program synthesis (Expert).

As mentioned in the introduction, machine learning researchers often optimise their algorithms using synthetic datasets. We investigate whether human-provided examples differ from such 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 a synthesis system when trained with human-provided examples versus randomly generated examples.

4.1 Material and Methods

4.1.1 Programming concepts

We use six list transformation concepts shown in Table 1. These concepts are commonly used in introductory computer science courses and are designed to be understandable even for non-experts without programming expertise. Moreover, these concepts are used as standard benchmarks for inductive program synthesis systems [19, 8]. Figure 1 shows an example program for the concept *dropk*. Recursion is necessary to write a program for each of these concepts. While a program for these concepts can theoretically operate on lists with elements of any type, we instruct participants to restrict their examples to lists of natural numbers between 0 and 100.

```
def dropk(l:list, k:int):
    if k == 0:
        return l
    return dropk(l[1:], k-1)
```

Figure 1: Example program for the concept *dropk*. This program iteratively removes the first element from the input list and decrements the integer k until $k = 0$, at which point it returns the input list. In other words, given an input list l and an integer k, this program returns a list which is equal to l without its first k elements.

4.1.2 Learning System

We use the state-of-the-art program synthesis system POPPER [8, 7], an inductive logic programming [32, 6] system. We use POPPER because it can learn recursive programs from only positive examples and tolerates noisy examples. Additionally, POPPER is guaranteed to find the smallest program that correctly generalises the given examples[3]. Following the literature [8], we allow POPPER to learn programs with the triadic relation *max*, the dyadic relations *head*, *tail*, *decrement*, *increment*, *geq*, *eq*, and the monadic relations *empty*, *zero*, *one*, *even*, and *odd*. We also use the triadic relation *comps* for the task *append*.

4.1.3 Interface

We use a survey web application for data collection. We show participants the following introduction to the study:

> *This study is on teaching concepts to computers by providing examples. The study focuses on list manipulation concepts.*

We then present our definition of a list:

[3] We also tried ALEPH [40] and METAGOL [33]. However, their performance was considerably worse than POPPER on all example sets. We have therefore excluded the results.

Name	Concept	Description
last	*Find the last element of a list*	Given as input a list, return the last element of that list.
length	*Find the length of a list*	Given as input a list, return the number of elements in that list.
append	*Append an element to a list*	Given as input a list and a natural number, return the list with the number inserted at the end of the list.
maxlist	*Find the maximum element of a list*	Given as input a list, return the element with the highest value in that list.
dropk	*Drop the first k elements of a list*	Given as input a list and a natural number k, return the input list without its first k elements.
sorted	*Identify a sorted list*	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 considered.

A list is a sequence of comma-separated values between square brackets, such as [1,2,3] or [7,4,15,8]. An empty list is written as [].

We ask participants to provide examples in the form of input-output pairs to explain the given concepts:

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 (the pairs $[0,2,4,6,8] \mapsto 5$; $[9,7,5,3,1] \mapsto 0$; 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 participants with six textual descriptions corresponding to each concept listed in Table 1. Concepts are presented in a randomised order. Finally, we ask participants to provide demographic information, which includes details about their background in computer science.

The instructions do not mention a program synthesis system, as participants have no experience with program synthesis. Throughout the study, participants do not interact with the learning system, nor do they receive any feedback while providing examples. The technical appendix includes the full set of instructions presented to participants, along with a screenshot of the interface.

4.1.4 Training examples

Participants. Participants of the study were recruited by e-mail. We invited (i) computer science students (bachelor, master, and PhD), (ii) students and other persons without a background in computer science, and (iii) one postgraduate researcher who is familiar with the underlying algorithm of the program synthesis system. The study was completed by 40 participants, with 14 declaring having no background in computer science (NCS group), 25 declaring having a background in computer science (CS group), and one expert in program synthesis (Expert). The mean age of participants was 25 ± 5. The NCS group spent 36min $\pm$ 8min answering the survey and the CS group 21min $\pm$ 3min.

Illegal symbols and formatting errors. Some participants provided examples including *illegal symbols* ($N_{NCS} = 0$, $N_{CS} = 5$). For example, one participant provided the example $[a,b,c,d,e,f,g] \mapsto 7$ for the task *length*, which is illegal because only natural numbers were allowed as list elements. We corrected those symbols to the closest admissible symbol. We did not correct elements greater than 100. Also, some participants provided examples in the *wrong format* ($N_{NCS} = 13$, $N_{CS} = 5$). For instance, one participant wrote a semicolon instead of a comma. Another participant provided the input integer with the output for the task *dropk*. We re-formatted these examples. Other errors, non-corrected, are discussed in Section 4.3.1.

Random examples. We compare human-provided examples with randomly sampled examples. We sample positive examples for each concept. We consider two random distributions. First, we sample the length of lists in examples from a uniform distribution bounded between 0 and 100 (*RandomUniform*). We hypothesise that longer examples might be more informative, leading to better performance. Therefore, we also consider a normal distribution which mimics the distribution of the example lengths provided by non-experts (*RandomNormal*). We sample example lengths from a truncated normal distribution, to ensure lengths are positive. The mean, standard deviation, minimum, and maximum are determined from the examples provided by the human participants (CS or NCS) for each task. For both the uniform and normal distributions, we sample elements within lists from a uniform distribution bounded between 0 and 100.

4.1.5 Evaluation

For each task and for each participant p, we train POPPER using the first n examples provided by p. We increment the number of training examples n and repeat training. We measure the predictive accuracy of the programs learned by POPPER using a set of $k = 2000$ test examples. The default predictive accuracy is 50%. We compare predictive accuracies across different participant groups (NCS, CS, and Expert) and against random distributions (*RandomUniform* and *RandomNormal*). We calculate the mean and standard error. Error bars in the figures and tables denote standard error. We also evaluate the statistics of the examples provided by participants. Box plots display the minimum, first quartile, median, third quartile, and maximum of the statistics evaluated. We use an 8-core 3.2 GHz Apple M1 and a single CPU to run the evaluation.

4.1.6 Ethics statement

No ethics statement was required by the home university for the conducted empirical study.

4.2 Results

We present the results of our empirical study.

Task	NCS	CS	Expert
last	89 ± 6	95 ± 3	**100**
length	$\mathbf{100 \pm 0}$	96 ± 3	**100**
append	95 ± 3	92 ± 3	**100**
maxlist	96 ± 4	82 ± 5	**100**
dropk	92 ± 5	80 ± 5	**100**
sorted	84 ± 5	77 ± 4	**100**

Table 2: Predictive accuracy for full example sets for the CS group, NCS group, and the expert. The error bars denote standard error.

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

Figures 2 and 3 show the mean predictive accuracies across all tasks of the programs learned by POPPER trained on the first n examples provided by a participant. The appendix shows the predictive accuracies for each task. We compare the examples from the NCS group with expert examples (Figure 2), and those from the CS group with expert examples (Figure 3). These results show that the expert consistently provides sufficient examples, leading POPPER to achieve maximal predictive accuracy (100%) for every task. Therefore, there exists an example set which perfectly solves each of the teaching tasks proposed. By contrast, both the NCS and CS groups provide examples resulting in lower accuracy for almost all tasks.

This result indicates that non-expert participants struggle to provide an example set from which the program synthesis system can identify the desired concept. In other words, this result shows that humans can provide sufficient examples but non-experts do not. For instance, for the task *sorted*, none of the example sets provided by the NCS group achieved maximal accuracy. In particular, no participant from the NCS group included the base case, the empty list, as an example. Out of 14 NCS participants, only 6 provided an example set resulting in at least 99% accuracy. Similarly, among 25 CS participants, 10 provided an example set resulting in at least 99% accuracy, and 3 provided an example set resulting in maximal (100%) accuracy.

Overall, these results suggest that the answer to **Q1** is no, i.e. non-expert humans struggle to provide sufficient examples to teach concepts to a program synthesis system. We discuss potential reasons for this result in Section 4.3.

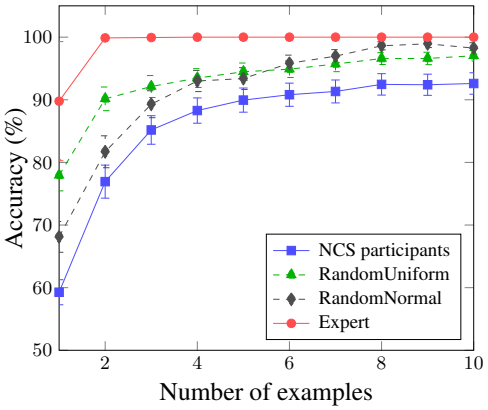

Figure 2: Predictive accuracies for the NCS group over all tasks when trained on progressively larger example sets. The error bars denote standard error.

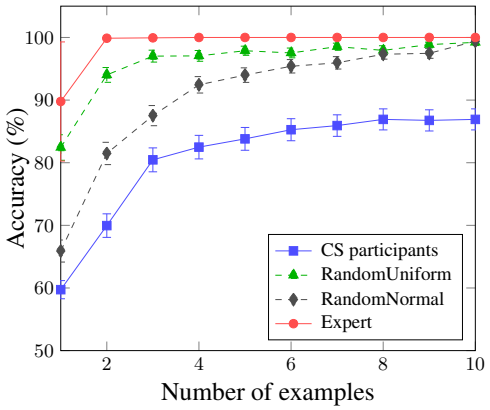

Figure 3: Predictive accuracies for the CS group over all tasks when trained on progressively larger example sets. The error bars denote standard error.

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

Table 2 shows the predictive accuracies achieved with the examples from the NCS and CS groups. A Shapiro-Wilk test [39] shows that these accuracies are not normally distributed. For many example sets, POPPER identifies the desired program, achieving maximum accuracy (100%). For many other example sets, POPPER identifies a program with default accuracy (50%), and for fewer sets, POPPER achieves an accuracy falling between these extremes. We exclude expert accuracies from our statistical analysis due to the inability to conduct meaningful statistical computations with a sample size of 1 and lacking a measure of dispersion. A single Mann-Whitney U-test[4] [29] indicates that the difference between the NCS and CS groups is not significant. The lack of a significant difference suggests that, in this context and among non-experts, the background of participants does not enhance their ability to teach a concept to a program synthesis system. In other words, domain-specific knowledge might not always improve teaching effectiveness. However, the performance of the expert suggests that understanding the underlying algorithm of the learner can play an important part. Therefore, these results indicate that effective teaching requires not only familiarity with the concepts being taught but also an understanding of the learner's learning process.

Overall, these results suggest that the answer to **Q2** is no, i.e. the background of a human does not significantly impact their ability to teach a programming concept to a synthesis system. While understanding of the domain might not be beneficial, comprehension of the learner's learning algorithm can be.

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

Figures 2 and 3 show the predictive accuracies of POPPER when trained using non-expert examples and examples with length sampled from a uniform or normal distribution. Due to the dependency of the parameters in the truncated normal distribution on population metrics, we perform multiple statistical tests. For each participant group (CS, NCS), we conduct three separate tests, using the predictive accuracies obtained from two, three, or all ten of the provided examples, resulting in a total of six analyses. We choose these specific points to test how

[4] A Mann-Whitney U-test is a non-parametric alternative for Student's t-test.

Task	NCS	CS	Expert
last	92 ± 5	95 ± 3	**100**
length	$\mathbf{100 \pm 0}$	96 ± 3	**100**
append	95 ± 3	98 ± 2	**100**
maxlist	96 ± 4	81 ± 5	**100**
dropk	99 ± 0	89 ± 5	**100**
sorted	83 ± 5	77 ± 4	**100**

Table 3: Predictive accuracy for full example sets for the CS group, NCS group, and the expert, after ignoring example sets with systematic and non-systematic errors. The error bars denote standard error.

informative are (i) the first examples, and (ii) the full set of examples provided by humans. Similar to the participant data, the accuracies of programs learned from randomly generated examples do not follow a normal distribution. A Kruskal-Wallis H-test[5] [23], followed by a Benjamini-Hochberg false discovery control method [2] to adjust p-values to account for family-wise error rate[6], shows a significant group effect when using 2 examples for the NCS group, and when using 2 or 10 examples for the CS group. A Post-hoc U-test [29] further identifies that, for the NCS group, accuracies from humans are significantly lower than accuracies from the uniform distribution when using 2 examples ($p = .01$). For the CS group, accuracies from the humans are significantly lower than accuracies from both random distributions when using 2 examples ($p < .001$), and lower than accuracies from the normal distribution when using 10 examples. These results show that, compared to both a uniform and parameter-matching normal distribution, the NCS and CS groups have worse performance. We discuss potential reasons for this result in Section 4.3.

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

4.3 Discussion

The results from Section 4.2 suggest that non-expert humans struggle to provide examples sufficient to teach a programming concept to a synthesis system. In this section, we discuss potential explanations.

4.3.1 Erroneous Examples

One explanation for why the NCS and CS groups did not achieve maximal accuracy is that some participants provided erroneous examples. We identify two different types of errors.

Some participants made *systematic* errors which are similar errors on all examples of their example set ($N_{NCS} = 3$, $N_{CS} = 8$). This kind of error suggests the participant provided examples for a different concept. For instance, a participant from the NCS group provided examples for the concept *"return the first k elements of a list"* for the task *dropk*, and provided examples such as $[4, 8, 7, 6, 2, 1, 5, 4, 8, 3, 1], 2 \mapsto [4, 8]$ when a correct output for this input should have been $[4, 8, 7, 6, 2, 1, 5, 4, 8, 3, 1], 2 \mapsto [7, 6, 2, 1, 5, 4, 8, 3, 1]$. A participant from the CS group provided examples for the concept *"remove the element k in the input list"* for the task *dropk* and provided examples such as $[20, 13, 42, 53, 23, 21], 23 \mapsto [20, 13, 42, 53, 21]$.

Some participants made *non-systematic* errors, which are errors on a single example of their training set ($N_{NCS} = 1$, $N_{CS} = 3$). For instance, a participant from the CS group provided the example $[10, 0, 59, 68, 23, 42, 53] \mapsto 59$ for the task *maxlist*. A participant from the NCS group provided the example $[99999, 7676768] \mapsto false$ for the task *sorted*.

To evaluate the impact of systematic and non-systematic errors, we exclude example sets containing any of these errors and retrain POPPER. Table 3 shows the predictive accuracy of POPPER on these revised example sets. It shows that the predictive accuracy is not maximal, suggesting that participants did not provide sufficient examples for the system to identify the desired concept. For instance, to teach the concept *last*, one participant provided only two examples of length 4, and POPPER learns the concept *return the fourth element of the input list*. Similarly, for the concept *sorted*, another participant provided the examples $[1, 2, 3, 4, 5] \mapsto true$; $[0, 0, 0, 0, 1] \mapsto true$; $[5, 4, 3, 2, 1] \mapsto false$; $[1, 0, 0, 0, 0] \mapsto false$, and POPPER learns that a list is sorted if its second element is greater or equal to its first element.

Overall, these results suggest that, while human errors limit performance, participants also did not provide sufficient examples.

4.3.2 Simpler Examples

Another factor contributing to low accuracy could be the simplicity of examples provided by non-expert participants. To support this explanation, we analyse statistics of the example sets provided by the NCS group, the CS group, and the expert. We evaluate the number of examples, the lengths of lists, and the values of elements within lists.

Number of examples. Figure 5 shows the distribution of the number of examples provided by participants. It shows that, for all tasks, both the NCS and CS groups typically provided a greater number of examples compared to the expert. For instance, while the expert provided only 3 examples for teaching the task *maxlist*, the NCS and CS groups provided 6.7 ± 0.7 and 6.6 ± 0.5 examples, respectively. A Shapiro-Wilk test [39] shows that the number of examples does not follow a normal distribution. A H-test [23] shows a significant effect for the group ($p < .05$), and post-hoc U-tests [29] show that the expert provided significantly fewer examples than both the NCS and CS groups over all tasks ($p < .05$). This result might surprise the reader, as one might assume that more examples would be more beneficial for teaching a concept. However, as Telle et al. [42] point out, in machine teaching, it is crucial to consider not only the number of examples but also the length of examples in example sets.

Length of lists. Figure 6 shows the distribution of the length of lists provided by participants. Both the NCS and CS groups provided examples of similar lengths across all tasks. However, the expert provided examples of varying lengths depending on the task. For the tasks *last* and *length*, both the NCS and CS groups provided examples with lengths similar to those provided by the expert. Examples lengths are relatively small. However, for the other tasks (*append*, *maxlist*, *dropk*, and *sorted*), the expert provided examples with longer lengths, while both the NCS and CS groups typically did not. As Figure 4 shows, these tasks are difficult for both the NCS and CS groups which did not reach maximal accuracies. For instance, for the task *maxlist*, while examples provided by the expert have a mean length of 8.7 ± 4.4, examples from the NCS and CS groups have a mean length of 5.3 $\pm$ 0.2 and 5.1 $\pm$ 0.2 respectively. This difference is important because examples with larger lengths can contain more bits and hence more information.

Longer examples, such as those provided by the expert, tend to rule out short programs. For instance, for the task *last*, one participant

[5] A Kruskal-Wallis H-test is a non-parametric variant of the independent one-way ANOVA.

[6] 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.

provided two examples of length 4. POPPER learns the concept *return the fourth element of the list*, which is simpler to express compared to the concept of *last*. By contrast, the expert provided a single example of length 6. POPPER learns the concept of *last* because the concept *return the sixth element of the list* is more complex to express compared to the concept of *last*.

Non-expert participants might have provided cognitively simpler examples because they are easier to generate. Moreover, human participants might assume that the learner processes information similarly to a human, leading them to avoid complex lists as they are more difficult to parse. Conversely, experts likely included more complex examples based on their deeper understanding of the synthesis system. Figures 2 and 3 show that randomly sampling lengths of lists can improve learning performance. Furthermore, they show using a parameter-matching normal distribution yields performance closer to that of humans for both the NCS and CS groups. However, despite this improvement, the learning performance of non-expert humans remains inferior to that achieved with normally distributed list lengths. These results suggest that the length of lists greatly influences the ability to teach a synthesis system a concept.

Elements values. Figure 7 shows the distribution of element values provided by participants. It indicates that elements in lists from both the NCS and CS groups generally have smaller values compared to those provided by the expert. In particular, elements provided by non-experts have less variability, resulting in a higher occurrence of coincidental patterns. For instance, for the task *sorted*, one participant provided examples such as $[1, 3, 6, 9, 14, 18] \mapsto true$; $[1, 2, 3, 4, 5] \mapsto true$; $[0, 1] \mapsto true$; $[3, 1, 12, 2, 7] \mapsto false$. From these examples, POPPER simply learns that a list is sorted if its first element is less than or equal to 1. By contrast, the sorted lists provided by the expert contain unique elements, with no element repeated in another sorted list.

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

5 Conclusions and Future Work

Our empirical study explores whether humans can teach machines to code. Our results suggest that non-expert humans struggle to provide sufficient examples to effectively teach list manipulation concepts to a program synthesis system. Regardless of their computer science background, in most cases, the examples provided by non-experts humans did not allow the system tested to induce the intended program. By contrast, an expert familiar with the system provided examples allowing the system to perfectly learn all the desired concepts.

This study is the first exploration of the question of whether humans can teach machines to code. There are, therefore, limitations.

We showed participants a general instruction which did not specifically demand that the examples need to be aimed at a program synthesis system. Therefore, it might be the case that participants provided general-purpose examples. However, our results show that participants with a background in computer science made use of their knowledge of recursive programs and, for instance, often included an example for the base case. Nevertheless, the effect of varying instructions to elicit specific teaching strategies and help humans provide higher-quality examples, similar to the methodologies proposed by Cakmak and Thomaz [5] and Khan et al. [22], needs to be explored in future work.

Although we tested several synthesis systems [40, 33, 8], none of them could learn the desired concepts from the non-expert examples. By contrast, a system could learn perfectly accurate programs from expert-provided examples. There might, however, be another untested

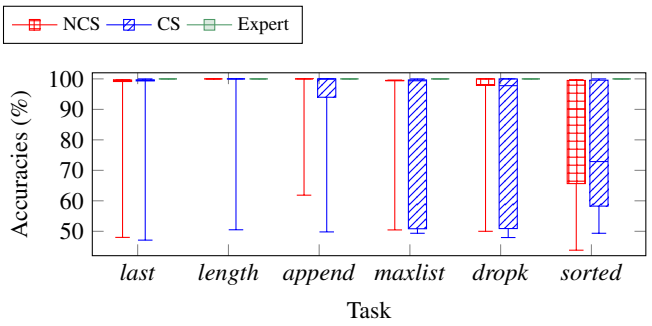

Figure 4: Predictive accuracies for 10 examples.

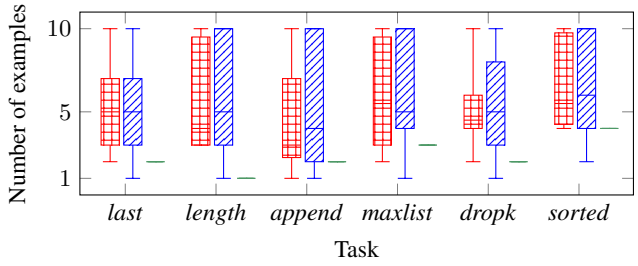

Figure 5: Number of examples provided by participants.

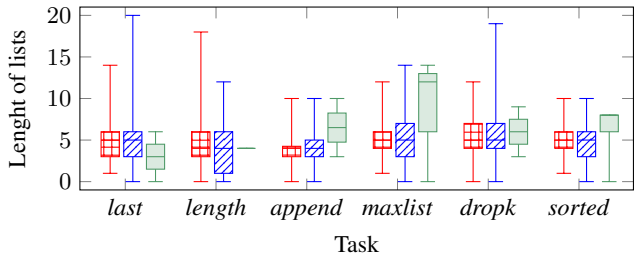

Figure 6: Length of lists provided by participants.

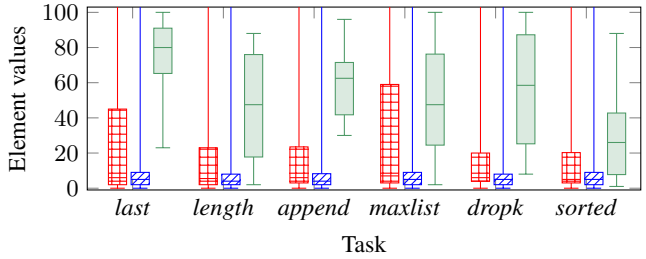

Figure 7: Values of elements provided by participants.

system that could be successful in learning from the non-expert examples. This study highlights the need to extend current systems to adapt to non-expert examples.

Our results have potential implications for the field of inductive program synthesis. In PBE [27] or end-user programming [18], it is assumed that humans can provide sufficient examples to teach a concept. Our results suggest that this assumption does not always hold. Furthermore, most of the existing work in program synthesis uses randomly generated examples for training. Our results suggest that training using randomly sampled examples yields substantially better performance compared to using examples provided by non-expert humans. If these systems are intended to be eventually trained by humans, this result should motivate researchers to incorporate more human-generated examples into the training dataset.

Consequently, our study raises two challenges for the field. Firstly, our study emphasises the necessity of developing learning systems that can better adapt to the teaching abilities of humans. Secondly, our

results highlight the need to build algorithms capable of generating training sets of random examples that more closely resemble those produced by humans.

References

[1] Y. Bengio, J. Louradour, R. Collobert, and J. Weston. Curriculum learning. In *ICML 2019*, volume 382, pages 41–48. ACM, 2009. doi: 10.1145/1553374.1553380. URL https://doi.org/10.1145/1553374.1553380.

[2] Y. Benjamini and Y. Hochberg. 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, 1995.

[3] A. B. Butler, L. L. Scherer, and R. Reiter-Palmon. Effects of solution elicitation aids and need for cognition on the generation of solutions to ill-structured problems. *Creativity Research Journal*, 15(2-3):235–244, 2003.

[4] M. Cakmak and M. Lopes. Algorithmic and human teaching of sequential decision tasks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 26, pages 1536–1542, 2012.

[5] M. Cakmak and A. L. Thomaz. Eliciting good teaching from humans for machine learners. *Artif. Intell.*, 217:198–215, 2014. doi: 10.1016/j.artint.2014.08.005. URL https://doi.org/10.1016/j.artint.2014.08.005.

[6] A. Cropper and S. Dumancic. Inductive logic programming at 30: A new introduction. *J. Artif. Intell. Res.*, 74:765–850, 2022. doi: 10.1613/JAIR.1.13507. URL https://doi.org/10.1613/jair.1.13507.

[7] A. Cropper and C. Hocquette. Learning logic programs by combining programs. In *ECAI 2023 - 26th European Conference on Artificial Intelligence*, volume 372, pages 501–508. IOS Press, 2023. doi: 10.3233/FAIA230309. URL https://doi.org/10.3233/FAIA230309.

[8] A. Cropper and R. Morel. Learning programs by learning from failures. *Mach. Learn.*, 110(4):801–856, 2021. doi: 10.1007/s10994-020-05934-z. URL https://doi.org/10.1007/s10994-020-05934-z.

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

[10] L. De Raedt. *Logical and relational learning*. Springer, 2008. ISBN 978-3-540-20040-6. doi: 10.1007/978-3-540-68856-3.

[11] K. Ellis, L. Morales, M. Sablé-Meyer, A. Solar-Lezama, and J. Tenenbaum. Learning libraries of subroutines for neurally-guided bayesian program induction. In *NeurIPS 2018*, pages 7816–7826, 2018.

[12] J. L. Elman. Learning and development in neural networks: The importance of starting small. *Cognition*, 48(1):71–99, 1993.

[13] P. A. Flach. *Machine Learning - The Art and Science of Algorithms that Make Sense of Data*. Cambridge University Press, 2012. ISBN 978-1-10-742222-3. URL http://www.cambridge.org/de/academic/subjects/computer-science/pattern-recognition-and-machine-learning/machine-learning-art-and-science-algorithms-make-sense-data.

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

[15] S. A. Goldman and M. J. Kearns. On the complexity of teaching. *Journal of Computer and System Sciences*, 50(1):20–31, 1995.

[16] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, pages 317–330, 2011. doi: 10.1145/1926385.1926423. URL http://doi.acm.org/10.1145/1926385.1926423.

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

[18] S. Gulwani, O. Polozov, and R. Singh. Program synthesis. *Found. Trends Program. Lang.*, 4(1-2):1–119, 2017. doi: 10.1561/2500000010. URL https://doi.org/10.1561/2500000010.

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

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

[21] S. Katayama. Efficient exhaustive generation of functional programs using monte-carlo search with iterative deepening. In *PRICAI 2008: Trends in Artificial Intelligence*, pages 199–210, 2008.

[22] F. Khan, B. Mutlu, and J. Zhu. How do humans teach: On curriculum learning and teaching dimension. In *Advances in Neural Information Processing Systems*, volume 24, 2011.

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

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

[25] L. H. S. Lelis, J. G. G. V. Nova, E. Chen, N. R. Sturtevant, C. D. Epp, and M. Bowling. Learning curricula for humans: An empirical study with puzzles from the witness. In L. D. Raedt, editor, *IJCAI*, pages 3877–3883, 2022. doi: 10.24963/ijcai.2022/538. URL https://doi.org/10.24963/ijcai.2022/538.

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

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

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

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

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

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

[32] S. Muggleton. Inductive logic programming. *New Generation Computing*, (4):295–318, 1991.

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

[34] J. Rule, E. Schulz, S. T. Piantadosi, and J. Tenenbaum. Learning list concepts through program induction. In C. Kalish, M. A. Rau, X. J. Zhu, and T. T. Rogers, editors, *CogSci*, 2018.

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

[36] J. S. Rule, J. B. Tenenbaum, and S. T. Piantadosi. The child as hacker. *Trends in Cognitive Sciences*, 24(11):900–915, 2020. ISSN 1364-6613. doi: https://doi.org/10.1016/j.tics.2020.07.005. URL https://www.sciencedirect.com/science/article/pii/S1364661320301741.

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

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

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

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

[41] P. D. Summers. A methodology for LISP program construction from examples. *J. ACM*, 24(1):161–175, 1977.

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

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

[44] X. Zhu. Machine teaching: An inverse problem to machine learning and an approach toward optimal education. In B. Bonet and S. Koenig, editors, *AAAI*, pages 4083–4087. AAAI Press, 2015. URL http://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/view/9487.

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

Appendices

A Interface

Figures 8 to 11 show our experimental interface. First, we showed participants an introductory text (Figure 8). Then, we showed an example of task (Figure 9). Next, we provided instructions on formatting answers (Figure 10). Finally, we presented six tasks, such as identifying a sorted list (Figure 11).

B Expert examples

Table 4 shows the examples provided by the expert.

Introduction

This study is on teaching concepts to computers by providing examples. The study focuses on list manipulation concepts. A list is a sequence of comma-separated values between square brackets, such as [1,2,3] or [7,4,15,8]. An empty list is written as []. One such list manipulation concept is "count the number of even numbers in a list". For instance, the list [1,2,3,4,2] has three even numbers.

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. For instance, consider the concept "count the number of even numbers in a list". The list [10,11,33,20] has two even numbers. Therefore, the pair formed of the list [10,11,33,20] as input and the number 2 as output is one possible example of this concept. Another example is the list [2,2,2,1] as input and the number 3 as 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. You can enter up to 50 characters in a text box, which will limit the number of elements your lists can have.

You will see several verbal descriptions of concepts. 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.

Figure 8: Introductory text presented to participants.

Example: Count the number of even numbers in a list

This is an example concept. Your tasks will be presented in this manner.

Concept: **Given as input a list, find the number of even numbers in that list**.

For this problem, an input is a list of natural numbers and an output is a natural number.

To explain the concept above, you could provide these examples:

[0,2,4,6,8] → 5

[9,7,5,3,1] → 0

[0,5,9,4,3,1,6,7,8,10,2] → 6

Figure 9: An example of task: count the number of even numbers in a list.

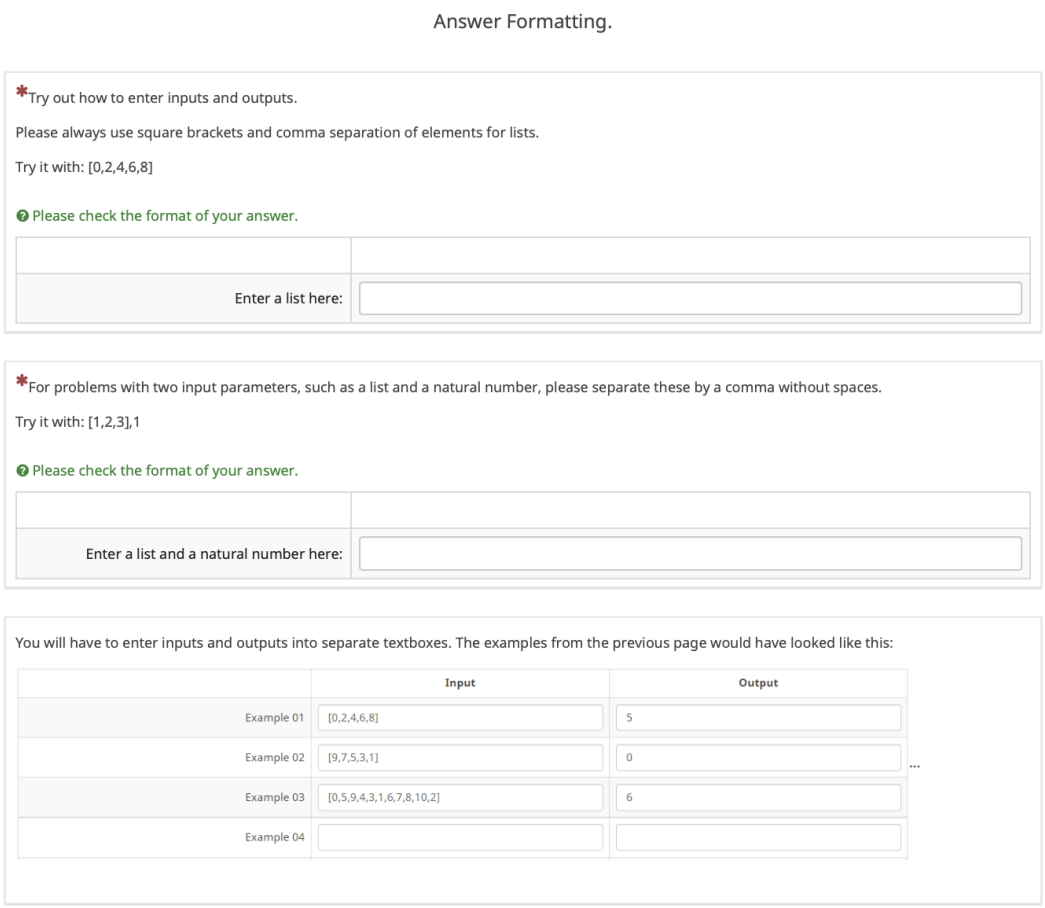

Figure 10: Description of our answer formatting.

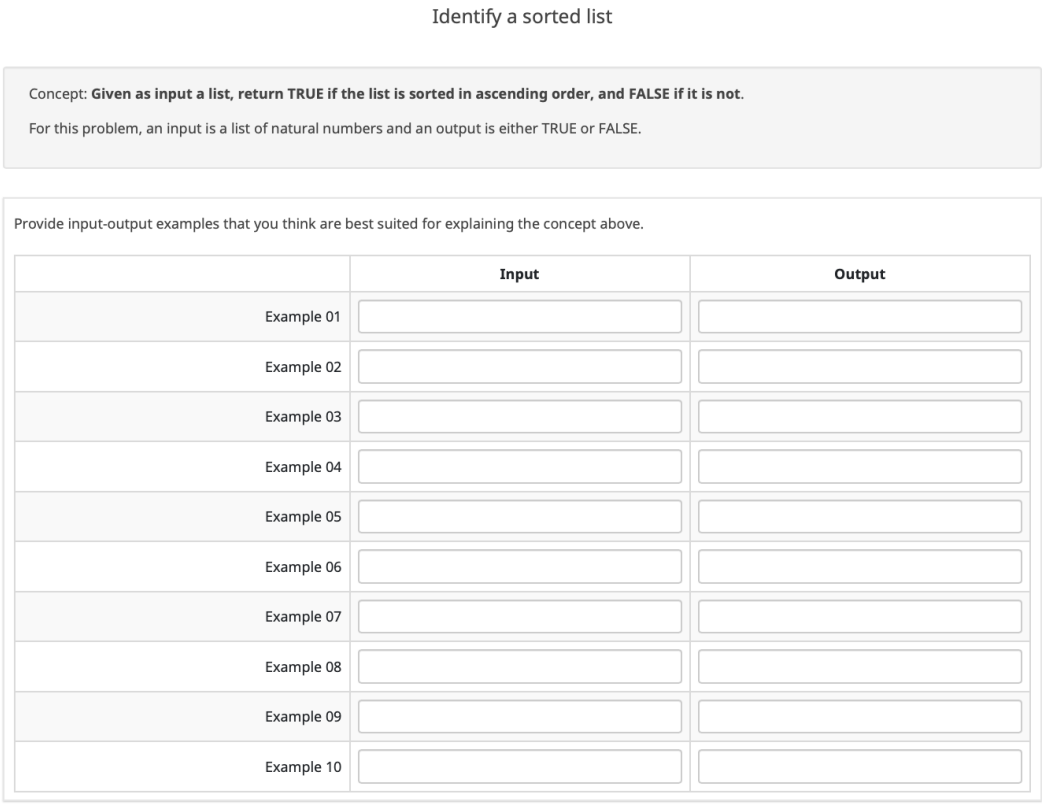

Figure 11: Task *sorted*.

Task	Examples
last	$[72, 88, 23, 92, 63, 100] \mapsto 100$ $[] \mapsto none$
length	$[72, 88, 23, 2] \mapsto 4$
append	$[43, 99, 79], 66 \mapsto [43, 99, 79, 66]$ $[70, 30, 72, 44, 35, 67, 58, 79, 96, 41], 56 \mapsto [70, 30, 72, 44, 35, 67, 58, 79, 96, 41, 56]$
maxlist	$[71, 88, 23, 24, 44, 46, 77, 92, 66, 100, 26, 94, 49, 53] \mapsto 100$ $[8, 74, 36, 28, 94, 55, 34, 98, 23, 12, 9, 2] \mapsto 98$ $[] \mapsto none$
dropk	$[72, 88, 23, 100, 26, 42, 8, 79, 90], 6 \mapsto [8, 79, 90]$ $[23, 45, 87], 0 \mapsto [23, 45, 87]$
sorted	$[1, 2, 4, 9, 13, 26, 39, 42] \mapsto true$ $[1, 2, 4, 9, 26, 25, 39, 42] \mapsto false$ $[22, 32, 45, 48, 56, 68, 73, 88] \mapsto true$ $[] \mapsto true$

Table 4: Examples provided by the expert

C Experimental results

Figures 12 to 17 show the detail of the results for each of the tasks for the NCS group. Figures 18 to 23 show the detail of the results for each of the tasks for the CS group.

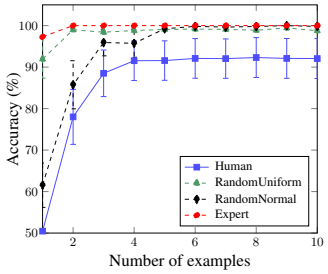

Figure 12: Predictive accuracies for the NCS group for the task *dropk*.

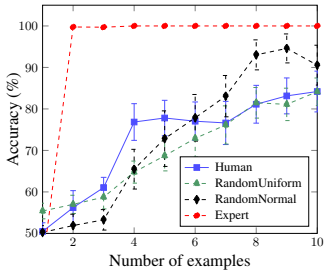

Figure 13: Predictive accuracies for the NCS group for the task *sorted*.

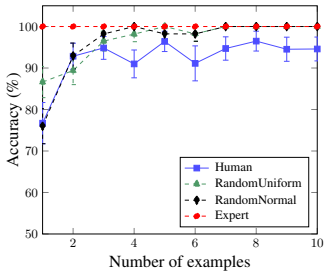

Figure 14: Predictive accuracies for the NCS group for the task *append*.

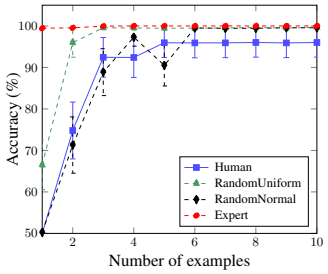

Figure 15: Predictive accuracies for the NCS group for the task *maxlist*.

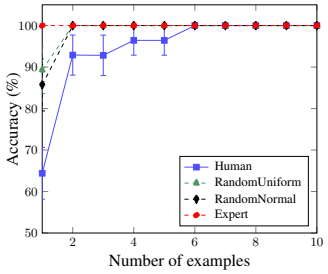

Figure 16: Predictive accuracies for the NCS group for the task *length*.

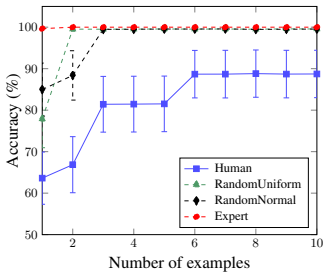

Figure 17: Predictive accuracies for the NCS group for the task *last*.

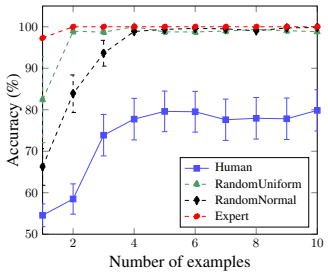

Figure 18: Predictive accuracies for the CS group for the task *dropk*.

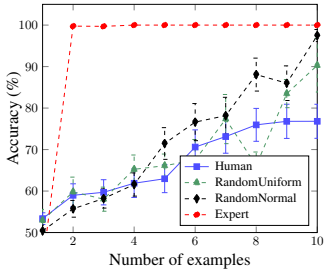

Figure 19: Predictive accuracies for the CS group for the task *sorted*.

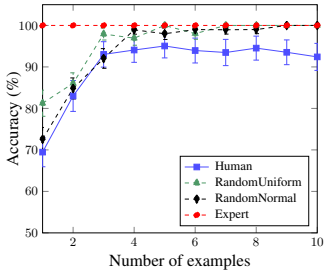

Figure 20: Predictive accuracies for the CS group for the task *append*.

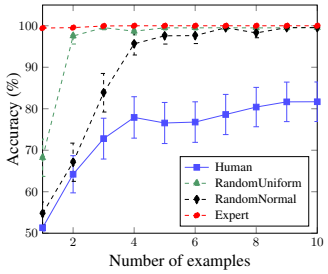

Figure 21: Predictive accuracies for the CS group for the task *maxlist*.

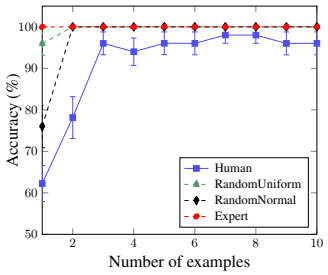

Figure 22: Predictive accuracies for the CS group for the task *length*.

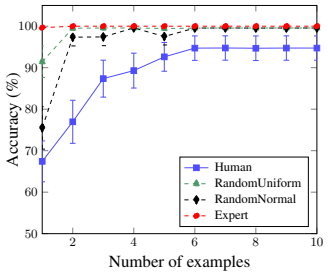

Figure 23: Predictive accuracies for the CS group for the task *last*.