ExPairT-LLM: Exact Learning for LLM Code Selection by Pairwise Queries

Tom Yuviler, Dana Drachsler-Cohen

Technion, Haifa, Israel
{tom.yuviler@campus, ddana@ee}.technion.ac.il

Abstract

Despite recent advances in LLMs, the task of code generation is still challenging. To cope, code selection algorithms select the best program from multiple programs generated by an LLM. However, existing algorithms can fail to identify the correct program, either because they fail to distinguish nonequivalent programs or because they rely on an LLM and assume it always correctly determines the output for every input. We present ExPairT-LLM, an exact learning algorithm for code selection that selects a program by posing two new types of queries to an LLM oracle: pairwise membership and pairwise equivalence. These queries are simpler for LLMs and enable ExPairT-LLM to identify the correct program through a tournament, which is robust to some LLM mistakes. We evaluate ExPairT-LLM on four popular code datasets. Its pass@1 (success rate) outperforms the state-of-the-art code selection algorithm on average by $+13.0\%$ and up to $+27.1\%$. It also improves the pass@1 of LLMs performing complex reasoning by $+24.0\%$.

Code — https://github.com/TomYuviler/ExPairT-LLM
Extended version — https://arxiv.org/abs/2511.10855

Introduction

Large language models (LLMs) show remarkable performance in code generation (Rozière et al. 2023; Chen et al. 2021; Nijkamp et al. 2023; Li et al. 2023; Guo et al. 2024; Luo et al. 2024; Feng et al. 2020; Hui et al. 2024). However, they may return code solutions that do not meet the task description, do not align with the input-output examples, or do not even compile (Liu et al. 2023; Dou et al. 2024). Code selection algorithms let the LLM generate multiple programs and automatically identify the correct program among them. Some of these approaches rely on input-output examples generated by the LLM (Chen et al. 2023a, 2024a; To, Nguyen, and Bui 2024); however, the outputs may be incorrect. Other works generate inputs, group programs based on their outputs, and select the cluster containing the most programs (Li et al. 2022; Shi et al. 2022). Another approach is to use neural networks to estimate the correctness of a program without relying on test inputs (Inala et al. 2022; Zhang et al. 2023b). However, these approaches can fail to identify the correct program if the input-output examples are incorrect or if the examples do not differentiate between nonequivalent programs, which can lead to returning an incorrect program. An exception is a method that looks for differentiating inputs through fuzzing and symbolic execution (Fan et al. 2024). However, it is not robust to LLM errors, and fuzzing and symbolic execution tend to fail on tasks with complex input constraints (Fan et al. 2024).

We extend exact learning (Angluin 1987a,b; Bshouty 2013; Angluin 2004) to the problem of LLM code selection. We rely on an LLM as the oracle, similarly to prior works for automatically generating input-output examples (Chen et al. 2023a, 2024a; To, Nguyen, and Bui 2024), evaluating generated text (Gao et al. 2023; Zhang et al. 2023c; Chen et al. 2023b), and annotating data (Tan et al. 2024; He et al. 2024; Ding et al. 2023). The challenge is that the traditional queries of exact learning are practically infeasible: (1) posing membership queries to the LLM can lead to a very large number of queries only to identify the correct output for a *single* input and (2) equivalence queries are still too challenging for LLMs. However, LLMs are very successful in pairwise comparisons. Previous works use LLMs as judges to decide between two generated texts (Liusie, Manakul, and Gales 2024; Liusie et al. 2024; Liu et al. 2024; Qin et al. 2024) or for chatbot evaluation (Zheng et al. 2023). Recent advancements in LLMs' complex reasoning (Wei et al. 2022; Huang and Chang 2023; Wang et al. 2023) provide the opportunity to use LLMs as judges for the complex task of code selection. However, naively selecting a program for a given task by a series of pairwise comparisons over the candidate programs is still too challenging for LLMs.

We introduce new kinds of queries: *pairwise equivalence* and *pairwise membership*. Our queries are simpler for LLMs and enable a learner to be robust to some oracle mistakes via a tournament (Trawinski and David 1963; Huber 1963). A pairwise equivalence query asks whether two programs are equivalent, and if not, the oracle returns a differentiating input. A pairwise membership query asks the oracle to select the better of two sets of outputs for given inputs. These queries enable identifying correct outputs and constructing a sufficient set of input examples to select a correct program. Further, they enable robustness to some oracle mistakes: differentiating inputs can be validated by running the programs, and the best output from a set of outputs is robustly deter-

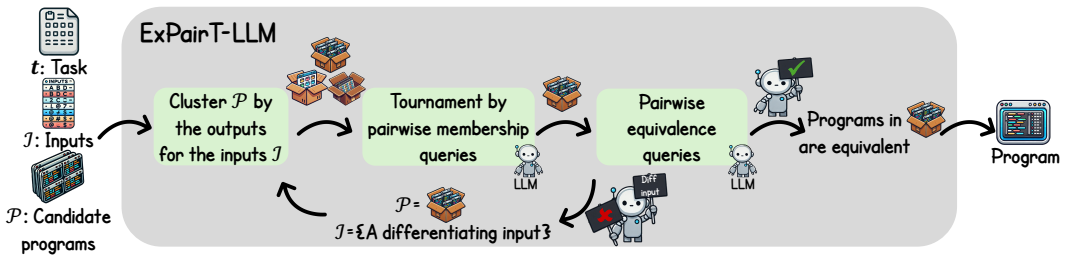

Figure 1: ExPairT-LLM: A code selection algorithm by pairwise queries.

mined by tournaments, which rely on pairwise comparisons.

We present ExPairT-LLM (**Ex**act learning by **Pair**wise **T**ournament), shown in Figure 1. Given a coding task, initial inputs, and candidate programs, it first clusters the programs based on their outputs for the inputs. It then identifies the correct cluster through a tournament between the clusters, which poses pairwise membership queries to an LLM. Next, it checks whether the selected cluster contains only equivalent programs by posing pairwise equivalence queries to the LLM. If so, it returns one of the programs. Otherwise, it refines the cluster based on the differentiating input and repeats this process. If the LLM is always correct, we show that ExPairT-LLM is an exact learner. Otherwise, we show lower bounds on the probabilities that ExPairT-LLM chooses the correct cluster and identifies that a cluster has to be refined. The number of pairwise membership queries and pairwise equivalence queries is at most $\binom{|\mathcal{P}|}{2}$ for each type, where $|\mathcal{P}|$ is the number of candidate programs.

We evaluate ExPairT-LLM on code datasets: *HumanEval* (Chen et al. 2021), *MBPP-sanitized* (Austin et al. 2021), *APPS* (Hendrycks et al. 2021), and *LiveCodeBench* (Jain et al. 2025). Its pass@1 (success rate) exceeds $\mathcal{B}^4$ (Chen et al. 2024a), the state-of-the-art, by $+13.0\%$ and CodeT (Chen et al. 2023a) by $+16.6\%$. ExPairT-LLM also improves the pass@1 of LLMs performing complex reasoning: by $+32.8\%$ for *OpenAI o1-mini* (OpenAI 2024), by $+20.4\%$ for *DeepSeek-R1* (DeepSeek-AI 2025), and by $+18.9\%$ for *Gemini 2.5 Flash* (Google-DeepMind 2025). Both types of pairwise queries are important: the pass@1 of ExPairT-LLM is higher by $+7.7\%$ as a result of pairwise equivalence queries and by $+24.6\%$ as a result of pairwise membership queries.

Related Work

Code Selection. Several code selection algorithms rely on LLMs to generate input-output examples and cluster programs by their consistency with the examples (Chen et al. 2023a, 2024a; To, Nguyen, and Bui 2024). Other approaches generate inputs, group programs by their outputs, and select the cluster with the most programs (Li et al. 2022; Shi et al. 2022). However, they can fail for various reasons: if the input-output examples are incorrect, if the selected cluster is incorrect, or if the selected cluster contains nonequivalent programs. LLMCodeChoice uses fuzzing and symbolic execution to find inputs that differentiate candidate programs (Fan et al. 2024); however, it is unsuitable for tasks with complex input constraints and assumes the LLM is always correct. Others use neural networks to estimate the correctness of a program for a task without executing the program on test inputs (Inala et al. 2022; Zhang et al. 2023b).

LLM as an Oracle. Relying on LLMs as oracles (judges) for selection between two candidates has been proposed for text-related tasks, including summarization (Liusie, Manakul, and Gales 2024; Liusie et al. 2024; Liu et al. 2024), chat assistants (Liusie, Manakul, and Gales 2024; Liusie et al. 2024; Zheng et al. 2023), content generation (Liu et al. 2024), and information retrieval (Qin et al. 2024). CodeRL (Le et al. 2022) employs reinforcement learning to enhance code-generating LLMs during training and inference by utilizing a separate LLM trained to assess the functional correctness of generated code samples. Other works utilize LLMs for self-repair in code generation models, providing feedback that identifies errors from execution results and generates explanations to fix the code (Chen et al. 2024b; Zhang et al. 2023a; Olausson et al. 2024).

Program Synthesis. Our work is related to *oracle-guided inductive synthesis* (OGIS) (Jha and Seshia 2017; Ji et al. 2020, 2023), which generates a program meeting a specification by interacting with a SAT/SMT oracle. In particular, it is related to *counterexample-guided inductive synthesis* (CEGIS) (Solar-Lezama et al. 2006; Alur et al. 2013; Abate et al. 2018), where if the oracle eliminates a candidate program, it also returns a counterexample.

Problem Definition

Tasks and Programs. A coding task consists of a natural language description (e.g., a docstring describing the function) and possibly input-output examples. Figure 2 shows task examples. If a program p satisfies a task t, we say p is correct and write $p \models t$. For each task, we are given a finite set of *candidate programs* $\mathcal{P}$. We assume that there exists $p \in \mathcal{P}$ that is correct and that the programs are deterministic and terminate for every input (stochastic or nonterminating programs can be removed in a preprocessing step by running them on several inputs with a reasonable timeout). Obtaining the set of candidates is orthogonal to our problem.

Code Selection. Given a task t and a set of programs $\mathcal{P}$ that contains a correct program, a code selection algorithm $\mathcal{A}$ aims to return $p \in \mathcal{P}$ satisfying t. We evaluate algorithm $\mathcal{A}$ by its pass@1 (Chen et al. 2021) (success rate). Given

```
def amicable_numbers_sum(limit):
    '''Write a function to sum all amicable numbers from 1
    to a specified number'''
```

(a) MBPP-sanitized

```
'''The teacher gave Andrew an array of n numbers a_1,
..., a_{n}. After that he asked Andrew for each k from
1 to n - 1 to build a k-ary heap on the array and count
the number of elements for which the property of the
minimum-rooted heap is violated, i.e. the value of an
element is less than the value of its parent. ... for a
non-root element v the property of the heap is violated
if a_{v} < a_{p}(v). Help Andrew cope with the task!
-----Input-----
The first line contains a single integer n (2 ≤ n ≤
2·10^5). The second line contains n space-separated
integers a_1, ..., a_{n} (- 10^9 ≤ a_{i} ≤ 10^9).
-----Output-----
in a single line print n - 1 integers, separate the
numbers with a single space — the number of elements
for which the property ... is violated ...'''
def solution(stdin: str) -> str:
```

(b) APPS

Figure 2: Tasks from (a) *MBPP-sanitized* (Austin et al. 2021), and (b) *APPS* (Hendrycks et al. 2021).

a set of task-program-set pairs $\mathcal{T}$, the pass@1 of $\mathcal{A}$ is the percentage of tasks for which it returns a correct program:

$$\text{pass@1}(\mathcal{A}) = \frac{100}{|\mathcal{T}|} \sum_{(t_i, \mathcal{P}_i) \in \mathcal{T}} \mathbb{I}\left[\mathcal{A}(t_i, \mathcal{P}_i) \models t_i\right] \quad (1)$$

Challenges. This problem is challenging. First, tasks are described by natural language and examples, which may be ambiguous, under-specified, or require domain knowledge, e.g., mathematics (Figure 2(a)). Second, the candidate programs are often too complex (Dou et al. 2024) to be analyzed by existing program analyzers, especially if the tasks pose complex input constraints (e.g., Figure 2(b)), which are not easily expressed with existing tools. Third, the algorithm cannot interact with a user to obtain more information. Our problem is not a classical program synthesis problem (Gulwani, Polozov, and Singh 2017; Alur et al. 2013). First, the program space can be complex, consisting of hundreds of lines of code, spanning a wide range of operations, data structures, and sometimes uncommon libraries. Second, the task may include a lengthy natural language description that is difficult to translate into a formal specification. Third, the tasks can vary significantly, e.g., mathematics (Figure 2(a)) and algorithms and data structures (Figure 2(b)), whereas program synthesizers often focus on related tasks.

Pairwise Membership & Equivalence Queries

In this section, we present our idea: exact learning by posing new queries to an oracle.

Exact Learning. In exact learning (Angluin 1987a,b), a learner identifies a target concept (a set) by interacting with an oracle. Typically, the learner poses two types of queries. A membership query asks whether an element is a member of the target, and the oracle replies with *yes* or *no*. An equivalence query asks if a candidate concept is equivalent to the target. If the oracle replies *yes*, the learner terminates. Otherwise, the oracle returns a counterexample element. Often, exact learning algorithms identify the target concept if it is in the search space and the oracle answers correctly. In our setting, a membership query asks whether an input-output pair aligns with the given task, and an equivalence query asks whether a program satisfies the given task. However, membership queries may require an excessively large number of interactions to determine the correct output for a single input, and equivalence queries are too challenging for LLMs.

Pairwise Comparisons. LLMs have been shown to be effective as judges for selecting between two candidates in text-related tasks (Liusie, Manakul, and Gales 2024; Liusie et al. 2024; Liu et al. 2024; Qin et al. 2024; Zheng et al. 2023). Additionally, several works rely on pairwise comparisons to identify the best candidate from a set of candidates (Wauthier, Jordan, and Jojic 2013; Jamieson and Nowak 2011; Shah and Wainwright 2017; Cattelan 2012; Newman 2022). One robust approach is *Copeland's method* (Copeland 1951), a voting system that determines the winner by head-to-head comparisons between each pair of candidates. In every comparison, one candidate gains a point. The winner is the candidate with the most points. Theoretically, we could rely on Copeland's method to identify a correct program by posing to the LLM oracle a series of queries, each of which presents two programs and asks the oracle to select the better program for the given task. However, such pairwise queries are still challenging for LLMs.

Pairwise Queries. Recent advancements have significantly enhanced LLM capabilities in complex reasoning (Wei et al. 2022; Huang and Chang 2023; Wang et al. 2023), leading to remarkable performance in coding problems. While the previous query types are challenging for them, we observe that LLMs can reliably answer *pairwise membership* and *pairwise equivalence* queries. A pairwise membership query consists of a task, a list of k inputs, and two lists of k outputs $(t, \mathcal{I}, O_1, O_2)$, asking which outputs are more suitable for t and $\mathcal{I}$: O_1 or O_2. A pairwise equivalence query consists of a task and two programs (t, p_1, p_2), asking whether the programs are semantically equivalent with respect to t. The oracle replies with *yes* or *no*. If it replies *no*, it provides a differentiating input `x`, i.e., $p_1(\texttt{x}) \neq p_2(\texttt{x})$. As an example, consider the task t of computing the length of a string `s`. A pairwise membership query is `(`t`,['Banana'],[5],[6])`, whose answer is `[6]`. A pairwise equivalence query is (`$t$`,return length(s),return 6`). The answer is *no*, and a possible counterexample is `'Apple'` (the first program returns `5` and the second program returns `6`).

Exact Learning by Pairwise Queries to LLMs

In this section, we introduce `ExPairT-LLM`.

Pseudocode. Algorithm 1 shows its pseudocode. The inputs are a coding task in natural language t, a list of inputs $\mathcal{I}$ (provided by the user or LLM-generated), a list of candidate programs $\mathcal{P}$, and an LLM $\mathcal{L}$. It returns a program $p \in \mathcal{P}$, intended to satisfy t. It first initializes the selected cluster C^* to $\mathcal{P}$ (Line 1). Next, it iteratively refines C^* until

Algorithm 1: ExPairT-LLM $(t, \mathcal{I}, \mathcal{P}, \mathcal{L})$

Input: Task t, inputs $\mathcal{I}$, programs $\mathcal{P}$, and LLM $\mathcal{L}$.
Output: A program $p \in \mathcal{P}$.

```
1:  C* = P   ▷ The selected cluster; initially all programs
2:  while True do
3:      C, O = cluster(I, C*)   ▷ Clusters and outputs
4:      scores = [0] × |C|   ▷ Zero scores for clusters
5:      for i = 1, ..., |C| do   ▷ Tournament
6:          for j = i + 1, ..., |C| do
7:              out = L.membership(t, I, O[i], O[j])
8:              if out == O[i] then scores[i] += 1
9:              else scores[j] += 1
10:     C* = C[argmax(scores)]   ▷ Selected cluster
11:     eq = True
12:     for i = 2, ..., |C*| do
13:         x = L.equivalence(t, C*[1], C*[i])
14:         if x ≠ ⊥ and C*[1](x) ≠ C*[i](x) then
15:             I = [x]; eq = False; Break
16:     if eq then return C*[1]
17: Function cluster(I, P)
    Input: Inputs I and candidate programs P.
    Output: Clusters of programs C and outputs O.
18: C = []; O = []   ▷ Lists of clusters and their outputs
19: for all p ∈ P do
20:     outputs = [p(i) | i ∈ I]   ▷ The outputs of p
21:     found = False
22:     for i = 1, ..., |C| do
23:         if outputs == O[i] then
24:             C[i].append(p); found=True; Break
25:     if not found then
26:         C.append([p]); O.append(outputs)
27: return C, O
```

C^* contains only equivalent programs (Line 2). Each iteration begins by invoking the `cluster` function (Line 3). This function partitions C^* into clusters $\mathcal{C}$ and their respective outputs $\mathcal{O}$, based on the outputs of programs in C^* for the inputs $\mathcal{I}$ (Line 18–Line 27). Then, ExPairT-LLM initializes the `scores` array, which records the clusters' scores (Line 4). Then, Copeland's method is executed by posing a pairwise membership query for each pair of clusters (Line 5–Line 9). The cluster with the maximal score is set to C^* (Line 10). Next, ExPairT-LLM checks if the programs in C^* are equivalent. It initializes the `eq` flag to `True` (Line 11). It then poses pairwise equivalence queries to the LLM between the first program in C^* and every other program in C^* (Line 12–Line 13). If the LLM finds a differentiating input `x` for two programs, ExPairT-LLM validates it by executing both programs on `x` (Line 14). If the validation succeeds, ExPairT-LLM sets the inputs $\mathcal{I}$ to [x], negates the flag `eq`, and begins another iteration to refine C^* (Line 15). Otherwise, if `eq` remains `True`, ExPairT-LLM returns the first program in C^* (Line 16). The extended version shows the query prompts.

Example. Figure 3 shows a running example, for the task of computing the length of a string `s`, the inputs $\mathcal{I}$ = [`'Banana'`], and four candidate programs (Figure 3(a)). ExPairT-LLM runs every program on the input and clusters the programs based on their outputs (Figure 3(b)): A (for `6`), B (for `5`), and C (for `4`). ExPairT-LLM identifies the correct cluster by three pairwise membership queries to determine the most suitable output for `'Banana'` (Figure 3(c)). Accordingly, it selects cluster A. Note that when two incorrect outputs are compared (e.g., `5` and `4`), the oracle selects one of them. Next, it poses a pairwise equivalence query to the LLM for the two programs in A (Figure 3(d)). Since the programs are not equivalent, the LLM returns a differentiating input `'Apple'`. ExPairT-LLM then refines cluster A into two clusters A_1 and A_2 based on the output for `'Apple'` (Figure 3(e)). It then selects the correct cluster by a pairwise membership query to determine the more suitable output for `'Apple'` (Figure 3(f)). Accordingly, it selects A_1. Since A_1 contains only one program, ExPairT-LLM returns this program (Figure 3(g)).

Advantages. ExPairT-LLM has several advantages. First, unlike approaches that return a program from the first selected cluster, it checks if the selected cluster contains nonequivalent programs. Second, unlike most approaches where the input examples are not adapted to the candidate programs, it generates inputs for distinguishing nonequivalent programs. Third, unlike approaches in which a single incorrect answer by the LLM can lead to failure, ExPairT-LLM is robust to some LLM errors: (1) it selects a cluster by the aggregated score in Copeland's method, and (2) it validates that differentiating inputs are indeed differentiating by running the programs. Fourth, unlike approaches that rely on fuzzing and symbolic execution, it can handle tasks with complex input constraints. Fifth, it leverages recent advancements enabling LLMs to perform exceptionally well in pairwise comparisons (Liusie, Manakul, and Gales 2024; Liusie et al. 2024; Liu et al. 2024; Qin et al. 2024).

Limitations. Since ExPairT-LLM compares the outputs of candidate programs, it targets tasks that can be implemented as stateless functions, common in many widely used code generation benchmarks (Chen et al. 2021; Austin et al. 2021; Hendrycks et al. 2021; Jain et al. 2025). It is not designed for programs that do not take any input (e.g., retrieving static configuration settings), do not generate explicit outputs (e.g., updating storage without direct feedback), or exhibit nondeterminism or nontermination (relevant for specialized code generation benchmarks, e.g., DS-1000 (Lai et al. 2023) and BIGCODEBENCH (Zhuo et al. 2025)).

Guarantees. We next present theoretical guarantees and proof sketches. Full proofs are in the extended version. Recall that the program space $\mathcal{P}$ is finite, contains a correct program, and consists of terminating and deterministic programs. We first show that ExPairT-LLM terminates. Second, if the oracle $\mathcal{L}$ is accurate, ExPairT-LLM returns the correct program. Otherwise, we show a lower bound on the probability of it choosing the correct cluster and show the probability of identifying that a cluster contains nonequivalent programs. Lastly, we analyze the number of queries.

Lemma 1. *Algorithm 1 terminates within $|\mathcal{P}|$ iterations.*

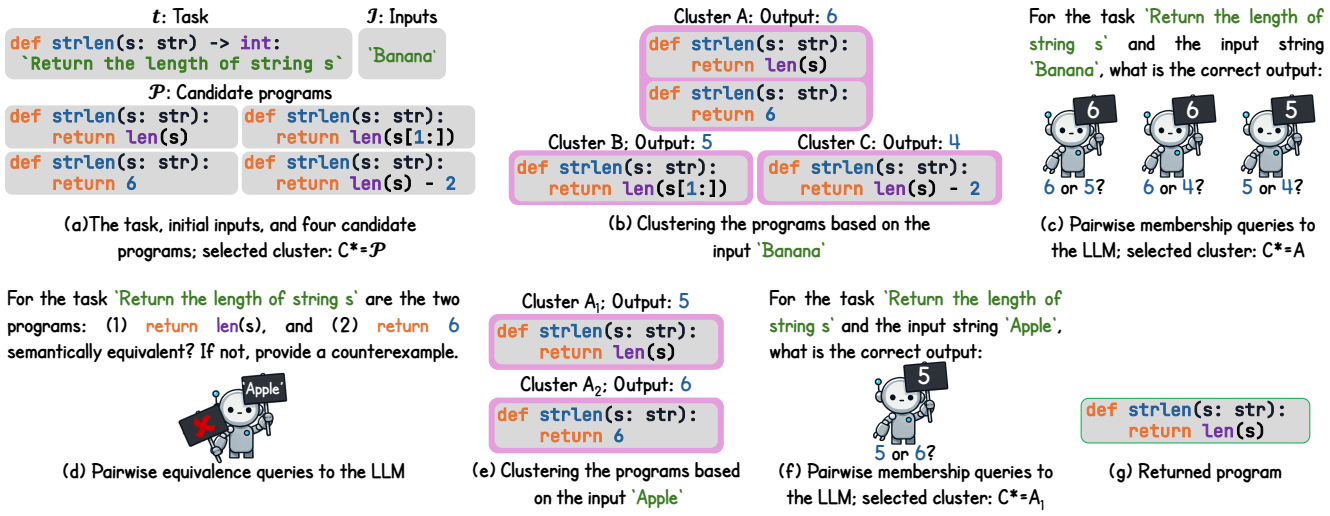

Figure 3: A running example of ExPairT-LLM.

It follows since the size of the selected cluster C^* decreases at every iteration, starting from the second one.

We next assume an accurate oracle that answers correctly.

Lemma 2. *If $\mathcal{L}$ is accurate, at every iteration (Line 2), a correct program $p^* \in \mathcal{P}$ is in the selected cluster C^*.*

The proof is by induction. Base: $C^* = \mathcal{P}$. Step: since $\mathcal{L}$ is accurate, the cluster containing p^* obtains the maximal score in Copeland's method and is thus selected as C^*.

Lemma 3. *If $\mathcal{L}$ is accurate, at any iteration that C^* contains an incorrect program, $\mathcal{L}$ returns a differentiating input.*

By the assumption and Lemma 2, C^* has an incorrect program p' and a correct one p^*. ExPairT-LLM poses a pairwise equivalence query for the first program in C^* and every other program in C^*. Since $\mathcal{L}$ is accurate and $p' \not\equiv p^*$, $\mathcal{L}$ must return a differentiating input.

Theorem 4. *If $\mathcal{L}$ is accurate, ExPairT-LLM returns a correct program.*

Since ExPairT-LLM terminates, it returns a program. By Lemma 2 and Lemma 3, it must be a correct program.

Next, we assume an inaccurate oracle. We begin with a lower bound on the probability that, at any iteration, C^* is set to the cluster containing a correct program out of all clusters $\mathcal{C} = \{C_1, \ldots, C_{n+1}\}$ (Line 10). Without loss of generality, we assume it is C_{n+1}. Since C^* is set to the cluster with the maximal score in Copeland's method, we consider the event A that C_{n+1} has the maximal score and provide a lower bound on its probability. We make two assumptions, common in tournaments (Trawinski and David 1963; Huber 1963). First, the oracle's responses are independent. Second, the probability that the oracle's response to a pairwise membership over (the outputs of) C_i and C_j is correct equals: (1) p, for $p > 0.5$, if C_i or C_j is C_{n+1} or (2) 0.5, otherwise. These assumptions hold in practice. First, queries are independent API calls to the LLM, which has no memory between them. Second, to illustrate that $p > 0.5$ holds

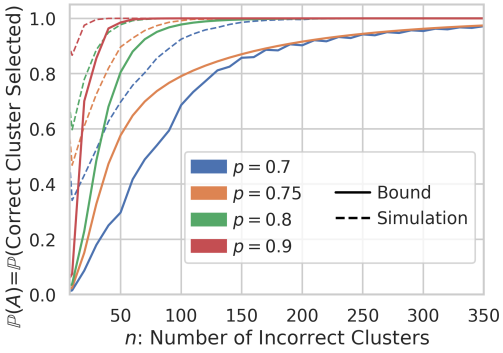

Figure 4: The empirical probability (dashed) and our lower bound (solid) for different p.

in practice, we measure the proportion of correct responses of *OpenAI o1-mini* (OpenAI 2024) in pairwise membership queries over programs generated by *CodeLlama* (Rozière et al. 2023) and *DeepSeek-Coder* (Guo et al. 2024) on the *HumanEval* dataset (Chen et al. 2021). The results estimate $p = 0.87$ for *CodeLlama* and $p = 0.96$ for *DeepSeek-Coder*.

Theorem 5. *Under the tournament's assumptions, a lower bound on the probability that the correct cluster C_{n+1} has the maximal score in Line 10 is:*

$$\mathbb{P}(A) \geq \left[\sum_{k=\lceil j \rceil+2}^{n} \binom{n}{k} p^k \, q^{n-k}\right] \cdot \left[1 - n\left(\tfrac{1}{2}\right)^{n-1} \sum_{k=\lfloor j \rfloor+1}^{n-1} \binom{n-1}{k}\right],$$

where $q \triangleq 1-p$ and $j \in [1, n-1]$ is a real-valued parameter.

The proof defines the event B that the maximum score over all clusters but the correct one is at most j, and lower

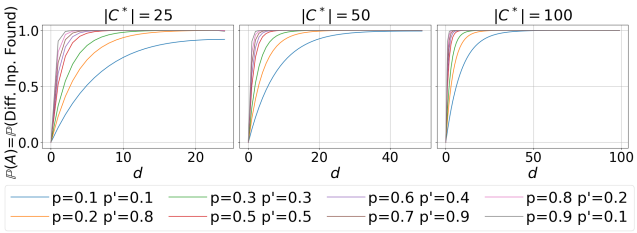

Figure 5: The probability of finding a differentiating input as a function of the number of incorrect programs d.

bounds $\mathbb{P}(A)$ with $\mathbb{P}(A|B)\cdot\mathbb{P}(B)$. It then shows that $\mathbb{P}(A|B)$ and $\mathbb{P}(\overline{B})$ are sums over binomially distributed random variables with probabilities p and 0.5, respectively. Figure 4 compares the empirical $\mathbb{P}(A)$ to the lower bound as a function of n, for different values of p and $j \in [\frac{n-1}{2}, n-1]$. $\mathbb{P}(A)$ is computed by simulations with 10^4 repetitions for every n and p. The figure shows that the larger the n, the tighter our lower bound and that $\mathbb{P}(A) \geq 0.89$ for $p \geq 0.9$ and $n \geq 25$.

Next, we show the probability that, at any iteration, `ExPairT-LLM` finds a differentiating input if C^* has nonequivalent programs. We assume that the oracle's responses to pairwise equivalence queries are independent, and that the probability that it returns a differentiating input for a pairwise equivalence query is p if one of the programs is correct and the other one is incorrect, or p' if both programs are incorrect. In the following, we denote by $d \in [|C^*|]$ the number of incorrect programs in C^* and by A the event of `ExPairT-LLM` finding a differentiating input.

Theorem 6. *The probability of `ExPairT-LLM` finding a differentiating input is:*

$$\mathbb{P}(A) = \left(1-(1-p)^d\right)\frac{|C^*|-d}{|C^*|} + \left(1-(1-p)^{|C^*|-d}(1-p')^{d-1}\right)\frac{d}{|C^*|}.$$

The proof defines the event B that the first program in C^* is correct and shows expressions for $\mathbb{P}(A|B)\cdot\mathbb{P}(B) + \mathbb{P}(A|\overline{B})\cdot\mathbb{P}(\overline{B})$, which yield the above $\mathbb{P}(A)$. Figure 5 shows $\mathbb{P}(A)$ as a function of the number of incorrect programs d, for different values of $|C^*|$, p and p'. For $p \geq 0.5$ and $d \geq 5$ (even for low p'), the probability of `ExPairT-LLM` identifying that C^* has incorrect programs is high (> 0.9). Namely, if the selected cluster has a correct program, the probability of returning an incorrect program is low. Empirically, $p \geq 0.5$ in practical settings. For example, on *HumanEval*, with *OpenAI o1-mini* as the oracle, we observe $p = 0.72$ for *CodeLlama* and $p = 0.61$ for *DeepSeek-Coder*.

Lastly, we show that the maximal number of queries of each type is $\binom{|\mathcal{P}|}{2}$. This polynomial complexity is consistent with exact learners in other domains (Hermo and Ozaki 2020; Marusic and Worrell 2015; Hellerstein et al. 1996).

Theorem 7. *`ExPairT-LLM` poses up to $\binom{|\mathcal{P}|}{2}$ pairwise membership queries and $\binom{|\mathcal{P}|}{2}$ pairwise equivalence queries.*

The proof shows that each pair of programs can be compared by each type of query at most once.

Evaluation

Here, we evaluate `ExPairT-LLM` and show that: (1) it outperforms existing code selection algorithms, including the state-of-the-art, (2) it significantly improves the pass@1 of LLMs with complex reasoning capabilities, (3) both pairwise membership and pairwise equivalence queries improve its performance, and (4) it poses 25.7 queries on average.

Setup. We implemented `ExPairT-LLM` in Python. The experiments were run on an Ubuntu 20.04.6 OS on a dual AMD EPYC 7742 server with 1TB RAM. Unless otherwise specified, each task has $|\mathcal{P}| = 25$ candidate programs, an initial input list $\mathcal{I}$ with five inputs generated by *GPT-4o* (OpenAI 2023), and *OpenAI o1-mini* (OpenAI 2024) as the LLM oracle. We evaluate on three datasets: *HumanEval* (Chen et al. 2021), which has 164 coding tasks; *MBPP-sanitized* (Austin et al. 2021) (denoted as *MBPP*), which has 427 coding tasks; and *APPS* (Hendrycks et al. 2021), whose test set has 5,000 coding tasks. To fairly compare with prior works (Chen et al. 2024a, 2023a), for *HumanEval*, we exclude input-output examples provided within the tasks. These datasets are the most popular benchmarks for evaluating code generation models (Rozière et al. 2023; Guo et al. 2024; Hui et al. 2024). Each of these datasets provides test input-output examples for each task (hidden during the selection), which we use to evaluate the correctness of the selected programs. For each dataset, we consider several models for generating the program space $\mathcal{P}$: *Codex* (Chen et al. 2021) (code-davinci-002 version), *CodeLlama* (Rozière et al. 2023) (7B-Python version), *StarCoder* (Li et al. 2023), *DeepSeek-Coder* (Guo et al. 2024) (6.7B Instruct version), and *CodeGen* (Nijkamp et al. 2023) (MONO-16.1B version). Due to the substantially larger number of coding tasks in *APPS*, we follow previous works (Chen et al. 2024a, 2023a) and consider for its evaluation only the *Codex* model. The program space generated by each model is obtained from the repository provided by Chen et al. (2024a). We evaluate a code selection algorithm by the popular pass@1 metric (Equation (1)). Like Chen et al. (2024a, 2023a), we exclude from the pass@1 pairs $(t, \mathcal{P})$ where all programs in $\mathcal{P}$ satisfy t, or where no program in $\mathcal{P}$ satisfies t.

Baseline Comparisons. We compare `ExPairT-LLM` with two code selection algorithms. *First,* $\mathcal{B}^4$ (Chen et al. 2024a), the state-of-the-art, defines the optimal selection strategy using a Bayesian framework, where the posterior probability of passing states between solutions and tests guides the selection. It approximates this posterior using Bayesian statistics and reformulates the problem as integer programming. $\mathcal{B}^4$ has three variants. In our comparison, for each dataset and model, we define its pass@1 by the best variant. *Second,* CodeT (Chen et al. 2023a), which generates test cases for each task, executes the candidate programs on these cases, and selects a program based on dual execution agreement. This agreement considers both the consistency of the outputs with the generated test cases and their agreement with the outputs of other candidate programs. For both $\mathcal{B}^4$ and CodeT, we use the authors' code, with the generated input-output examples provided in the repository

Dataset	Model	ExPairT-LLM	$\mathcal{B}^4$	CodeT	Orig
Human-Eval	Codex	**91.3**	80.5	74.1	38.7
	CodeLlama	**89.1**	66.6	62.5	40.3
	StarCoder	**91.7**	70.5	65.4	37.6
	DeepSeek-Coder	**91.6**	80.8	80.9	65.4
	CodeGen	**94.1**	64.0	55.1	42.4
MBPP	Codex	**84.4**	**84.4**	83.4	54.6
	CodeLlama	**81.7**	76.2	76.7	50.0
	StarCoder	**81.4**	72.7	72.4	43.6
	DeepSeek-Coder	**78.9**	76.8	76.8	58.3
	CodeGen	**82.6**	78.1	67.7	43.6
APPS	Codex	**81.4**	54.3	50.2	25.8

Table 1: The pass@1 (%) of ExPairT-LLM and baselines.

of Chen et al. (2024a). We also compare with the *Original* baseline, i.e., the LLM which returns a program given a task. Table 1 shows the results. ExPairT-LLM outperforms all baselines across all datasets and models. On average, its pass@1 exceeds $\mathcal{B}^4$ by $+13.0\%$, CodeT by $+16.6\%$, and *Original* by $+40.7\%$. We assess the statistical significance of these improvements with McNemar's test (McNemar 1947), suitable for comparing two algorithms on the same set of tasks with a binary success metric (e.g., pass@1). It confirms that ExPairT-LLM significantly outperforms each baseline on every dataset (p-value < 0.001).

LLMs with Complex Reasoning. We next compare ExPairT-LLM to LLMs with strong reasoning capabilities: *OpenAI o1-mini* (OpenAI 2024), *DeepSeek-R1* (DeepSeek-AI 2025), and *Gemini 2.5 Flash* (Google-DeepMind 2025). For each, we run ExPairT-LLM and use the LLM to generate 10 candidate programs (for each task), to generate 5 initial input examples, and as the oracle. We run this experiment on 150 randomly selected tasks from *APPS* and on *LiveCodeBench* (Jain et al. 2025), with 511 tasks. Table 2 shows the pass@1 of ExPairT-LLM and the original pass@1 of the LLM (i.e., the first program it returns). ExPairT-LLM improves the original pass@1 by $+32.8\%$ for *OpenAI o1-mini*, by $+20.4\%$ for *DeepSeek-R1*, and by $+18.9\%$ for *Gemini 2.5 Flash*. For both datasets, a McNemar's test confirms that ExPairT-LLM significantly outperforms the original pass@1 (p-value < 0.001).

Importance of Our Queries. Next, we study the importance of both pairwise membership and pairwise equivalence queries. We consider two variants. *First,* ExPairT-LLM_no_equ, which relies solely on membership queries and skips the pairwise equivalence queries (Line 12–Line 15). That is, the programs are clustered based on their outputs for the initial inputs, then a cluster is selected by pairwise membership queries (i.e., by majority vote), and lastly the first program in it is returned. *Second,* ExPairT-LLM_no_mem, which skips the pairwise membership queries (Line 5–Line 10) and instead selects the cluster with the most programs as C^*. Figure 6 presents

Dataset	Model	ExPairT-LLM	Original
APPS	OpenAI o1-mini	**91.2**	54.7
	DeepSeek-R1	**64.3**	56.3
	Gemini 2.5 Flash	**70.3**	54.1
LiveCode-Bench	OpenAI o1-mini	**95.8**	66.7
	DeepSeek-R1	**88.4**	55.7
	Gemini 2.5 Flash	**92.9**	71.4

Table 2: The pass@1 (%) of ExPairT-LLM and LLMs with complex reasoning.

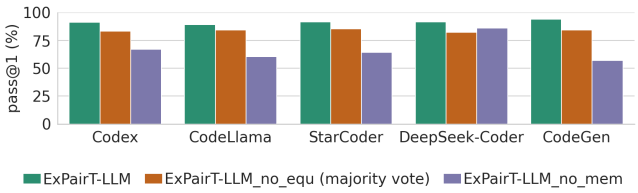

Figure 6: The pass@1 of ExPairT-LLM and variants without equivalence or membership queries, on *HumanEval*.

Model	P. Mem.	P. Eq.
Codex	12.4	9.1
CodeLlama	20.8	8.2
StarCoder	16.6	8.2
DeepSeek-Coder	5.2	14.9
CodeGen	25.4	7.6

Table 3: The average #queries on *HumanEval*.

the results. On average, on *HumanEval*, the pass@1 of ExPairT-LLM is higher than ExPairT-LLM_no_equ by $+7.7\%$ and ExPairT-LLM_no_mem by $+24.6\%$. The McNemar's test confirms that ExPairT-LLM significantly outperforms both variants (p-value < 0.001).

Total Queries. Lastly, Table 3 shows the number of queries ExPairT-LLM poses on *HumanEval*, for five models and $|\mathcal{P}| = 25$. On average, it poses 16.1 pairwise membership queries and 9.6 pairwise equivalence queries, i.e., 5.3% and 3.2% of the worst-case bounds $\binom{25}{2} = 300$ (Theorem 7). That is, the average total number of queries is $\approx |\mathcal{P}|$.

Conclusion

We present ExPairT-LLM, an exact learning algorithm for selecting the correct program. ExPairT-LLM poses pairwise membership and pairwise equivalence queries to an LLM oracle. If the LLM is accurate, ExPairT-LLM returns the correct program; otherwise, we present lower bounds on the probability of identifying the correct program. ExPairT-LLM outperforms the state-of-the-art, on average by $+13.0\%$, and improves the pass@1 of LLMs with strong reasoning capabilities by an average of $+24.0\%$.

References

Abate, A.; David, C.; Kesseli, P.; Kroening, D.; and Polgreen, E. 2018. Counterexample Guided Inductive Synthesis Modulo Theories. *In CAV*.

Alur, R.; Bodík, R.; Juniwal, G.; Martin, M. M. K.; Raghothaman, M.; Seshia, S. A.; Singh, R.; Solar-Lezama, A.; Torlak, E.; and Udupa, A. 2013. Syntax-guided synthesis. *In FMCAD*.

Angluin, D. 1987a. Learning Regular Sets from Queries and Counterexamples. *In Inf. Comput.*

Angluin, D. 1987b. Queries and Concept Learning. *In Mach. Learn.*

Angluin, D. 2004. Queries revisited. *In Theoretical Computer Science*.

Austin, J.; Odena, A.; Nye, M. I.; Bosma, M.; Michalewski, H.; Dohan, D.; Jiang, E.; Cai, C. J.; Terry, M.; Le, Q. V.; and Sutton, C. 2021. Program Synthesis with Large Language Models. *CoRR*, abs/2108.07732.

Bshouty, N. H. 2013. Exact Learning from Membership Queries: Some Techniques, Results and New Directions. *In Algorithmic Learning Theory*.

Cattelan, M. 2012. Models for Paired Comparison Data: A Review with Emphasis on Dependent Data. *In Statistical Science*.

Chen, B.; Zhang, F.; Nguyen, A.; Zan, D.; Lin, Z.; Lou, J.; and Chen, W. 2023a. CodeT: Code Generation with Generated Tests. *In ICLR*.

Chen, M.; Liu, Z.; Tao, H.; Hong, Y.; Lo, D.; Xia, X.; and Sun, J. 2024a. B4: Towards Optimal Assessment of Plausible Code Solutions with Plausible Tests. *In ASE*.

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

Chen, X.; Lin, M.; Schärli, N.; and Zhou, D. 2024b. Teaching Large Language Models to Self-Debug. *In ICLR*.

Chen, Y.; Wang, R.; Jiang, H.; Shi, S.; and Xu, R. 2023b. Exploring the Use of Large Language Models for Reference-Free Text Quality Evaluation: An Empirical Study. *In IJCNLP-AACL (Findings)*.

Copeland, A. H. 1951. A Reasonable Social Welfare Functio. *In University of Michigan Seminar on Applications of Mathematics to the social sciences*.

DeepSeek-AI. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. *CoRR*, abs/2501.12948.

Ding, B.; Qin, C.; Liu, L.; Chia, Y. K.; Li, B.; Joty, S.; and Bing, L. 2023. Is GPT-3 a Good Data Annotator? *In ACL*.

Dou, S.; Jia, H.; Wu, S.; Zheng, H.; Zhou, W.; Wu, M.; Chai, M.; Fan, J.; et al. 2024. What's Wrong with Your Code Generated by Large Language Models? An Extensive Study. *CoRR*, abs/2407.06153.

Fan, Z.; Ruan, H.; Mechtaev, S.; and Roychoudhury, A. 2024. Oracle-Guided Program Selection from Large Language Models. *In ISSTA*.

Feng, Z.; Guo, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, T.; Jiang, D.; and Zhou, M. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. *In EMNLP*.

Gao, M.; Ruan, J.; Sun, R.; Yin, X.; Yang, S.; and Wan, X. 2023. Human-like Summarization Evaluation with ChatGPT. *CoRR*, abs/2304.02554.

Google-DeepMind. 2025. Gemini 2.5 Flash.

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

Guo, D.; Zhu, Q.; Yang, D.; Xie, Z.; Dong, K.; Zhang, W.; Chen, G.; Bi, X.; Wu, Y.; Li, Y. K.; Luo, F.; Xiong, Y.; and Liang, W. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming - The Rise of Code Intelligence. *CoRR*, abs/2401.14196.

He, X.; Lin, Z.; Gong, Y.; Jin, A.; Zhang, H.; Lin, C.; Jiao, J.; Yiu, S. M.; Duan, N.; and Chen, W. 2024. AnnoLLM: Making Large Language Models to Be Better Crowdsourced Annotators. *In NAACL (Industry Track)*.

Hellerstein, L.; Pillaipakkamnatt, K.; Raghavan, V.; and Wilkins, D. 1996. How Many Queries Are Needed to Learn? *In J. ACM*.

Hendrycks, D.; Basart, S.; Kadavath, S.; Mazeika, M.; Arora, A.; Guo, E.; Burns, C.; Puranik, S.; He, H.; Song, D.; and Steinhardt, J. 2021. Measuring Coding Challenge Competence With APPS. *In NeurIPS*.

Hermo, M.; and Ozaki, A. 2020. Exact Learning: On the Boundary between Horn and CNF. *In ACM Trans. Comput. Theory*.

Huang, J.; and Chang, K. C. 2023. Towards Reasoning in Large Language Models: A Survey. *In ACL (Findings)*.

Huber, P. J. 1963. A Remark on a Paper of Trawinski and David Entitled: "Selection of the Best Treatment in a paired-Comparison Experiment". *In The Annals of Mathematical Statistics*.

Hui, B.; Yang, J.; Cui, Z.; Yang, J.; Liu, D.; Zhang, L.; Liu, T.; Zhang, J.; Yu, B.; Dang, K.; Yang, A.; Men, R.; Huang, F.; Ren, X.; Ren, X.; Zhou, J.; and Lin, J. 2024. Qwen2.5-Coder Technical Report. *CoRR*, abs/2409.12186.

Inala, J. P.; Wang, C.; Yang, M.; Codas, A.; Encarnación, M.; Lahiri, S. K.; Musuvathi, M.; and Gao, J. 2022. Fault-Aware Neural Code Rankers. *In NeurIPS*.

Jain, N.; Han, K.; Gu, A.; Li, W.-D.; Yan, F.; Zhang, T.; Wang, S.; Solar-Lezama, A.; Sen, K.; and Stoica, I. 2025. LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code. *In ICLR*.

Jamieson, K. G.; and Nowak, R. D. 2011. Active Ranking using Pairwise Comparisons. *In NIPS*.

Jha, S.; and Seshia, S. A. 2017. A theory of formal synthesis via inductive learning. *In Acta Informatica*.

Ji, R.; Kong, C.; Xiong, Y.; and Hu, Z. 2023. Improving Oracle-Guided Inductive Synthesis by Efficient Question Selection. *In OOPSLA*.

Ji, R.; Liang, J.; Xiong, Y.; Zhang, L.; and Hu, Z. 2020. Question selection for interactive program synthesis. *In PLDI*.

Lai, Y.; Li, C.; Wang, Y.; Zhang, T.; Zhong, R.; Zettlemoyer, L.; Yih, W.; Fried, D.; Wang, S. I.; and Yu, T. 2023. DS-1000: A Natural and Reliable Benchmark for Data Science Code Generation. *In ICML*.

Le, H.; Wang, Y.; Gotmare, A. D.; Savarese, S.; and Hoi, S. C. 2022. CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning. *In NeurIPS*.

Li, R.; Allal, L. B.; Zi, Y.; Muennighoff, N.; Kocetkov, D.; Mou, C.; Marone, M.; Akiki, C.; et al. 2023. StarCoder: may the source be with you! *In Trans. Mach. Learn. Res.*

Li, Y.; Choi, D.; Chung, J.; Kushman, N.; Schrittwieser, J.; Leblond, R.; Eccles, T.; Keeling, J.; Gimeno, F.; Lago, A. D.; Hubert, T.; Choy, P.; de Masson d'Autume, C.; Babuschkin, I.; Chen, X.; Huang, P.-S.; Welbl, J.; Gowal, S.; Cherepanov, A.; Molloy, J.; Mankowitz, D. J.; Robson, E. S.; Kohli, P.; de Freitas, N.; Kavukcuoglu, K.; and Vinyals, O. 2022. Competition-level code generation with AlphaCode. *In Science*.

Liu, J.; Xia, C. S.; Wang, Y.; and Zhang, L. 2023. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. *In NeurIPS*.

Liu, Y.; Zhou, H.; Guo, Z.; Shareghi, E.; Vulic, I.; Korhonen, A.; and Collier, N. 2024. Aligning with Human Judgement: The Role of Pairwise Preference in Large Language Model Evaluators. *CoRR*, abs/2403.16950.

Liusie, A.; Manakul, P.; and Gales, M. J. F. 2024. LLM Comparative Assessment: Zero-shot NLG Evaluation through Pairwise Comparisons using Large Language Models. *In EACL*.

Liusie, A.; Raina, V.; Fathullah, Y.; and Gales, M. J. F. 2024. Efficient LLM Comparative Assessment: A Product of Experts Framework for Pairwise Comparisons. *In EMNLP*.

Luo, Z.; Xu, C.; Zhao, P.; Sun, Q.; Geng, X.; Hu, W.; Tao, C.; Ma, J.; Lin, Q.; and Jiang, D. 2024. WizardCoder: Empowering Code Large Language Models with Evol-Instruct. *In ICLR*.

Marusic, I.; and Worrell, J. 2015. Complexity of equivalence and learning for multiplicity tree automata. *In J. Mach. Learn. Res.*

McNemar, Q. 1947. Note on the sampling error of the difference between correlated proportions or percentages. *In Psychometrika*, 12.

Newman, M. E. J. 2022. Ranking with multiple types of pairwise comparisons. *In Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*.

Nijkamp, E.; Pang, B.; Hayashi, H.; Tu, L.; Wang, H.; Zhou, Y.; Savarese, S.; and Xiong, C. 2023. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. *In ICLR*.

Olausson, T. X.; Inala, J. P.; Wang, C.; Gao, J.; and Solar-Lezama, A. 2024. Is Self-Repair a Silver Bullet for Code Generation? *In ICLR*.

OpenAI. 2023. GPT-4 Technical Report. *CoRR*, abs/2303.08774.

OpenAI. 2024. OpenAI o1-mini.

Qin, Z.; Jagerman, R.; Hui, K.; Zhuang, H.; Wu, J.; Yan, L.; Shen, J.; Liu, T.; Liu, J.; Metzler, D.; Wang, X.; and Bendersky, M. 2024. Large Language Models are Effective Text Rankers with Pairwise Ranking Prompting. *In NAACL (Findings)*.

Rozière, B.; Gehring, J.; Gloeckle, F.; Sootla, S.; Gat, I.; Tan, X. E.; Adi, Y.; Liu, J.; et al. 2023. Code Llama: Open Foundation Models for Code. *CoRR*, abs/2308.12950.

Shah, N. B.; and Wainwright, M. J. 2017. Simple, Robust and Optimal Ranking from Pairwise Comparisons. *In J. Mach. Learn. Res.*

Shi, F.; Fried, D.; Ghazvininejad, M.; Zettlemoyer, L.; and Wang, S. I. 2022. Natural Language to Code Translation with Execution. *In EMNLP*.

Solar-Lezama, A.; Tancau, L.; Bodík, R.; Seshia, S. A.; and Saraswat, V. A. 2006. Combinatorial sketching for finite programs. *In ASPLOS*.

Tan, Z.; Li, D.; Wang, S.; Beigi, A.; Jiang, B.; Bhattacharjee, A.; Karami, M.; Li, J.; Cheng, L.; and Liu, H. 2024. Large Language Models for Data Annotation and Synthesis: A Survey. *In EMNLP*.

To, H.; Nguyen, M.; and Bui, N. 2024. Functional Overlap Reranking for Neural Code Generation. *In ACL (Findings)*.

Trawinski, B. J.; and David, H. A. 1963. Selection of the Best Treatment in a Paired-Comparison Experiment. *In The Annals of Mathematical Statistics*.

Wang, X.; Wei, J.; Schuurmans, D.; Le, Q. V.; Chi, E. H.; Narang, S.; Chowdhery, A.; and Zhou, D. 2023. Self-Consistency Improves Chain of Thought Reasoning in Language Models. *In ICLR*.

Wauthier, F. L.; Jordan, M. I.; and Jojic, N. 2013. Efficient Ranking from Pairwise Comparisons. *In ICML*.

Wei, J.; Wang, X.; Schuurmans, D.; Bosma, M.; Ichter, B.; Xia, F.; Chi, E. H.; Le, Q. V.; and Zhou, D. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. *In NeurIPS*.

Zhang, K.; Li, Z.; Li, J.; Li, G.; and Jin, Z. 2023a. Self-Edit: Fault-Aware Code Editor for Code Generation. *In ACL*.

Zhang, T.; Yu, T.; Hashimoto, T.; Lewis, M.; Yih, W.; Fried, D.; and Wang, S. 2023b. Coder Reviewer Reranking for Code Generation. *In ICML*.

Zhang, X.; Yu, B.; Yu, H.; Lv, Y.; Liu, T.; Huang, F.; Xu, H.; and Li, Y. 2023c. Wider and Deeper LLM Networks are Fairer LLM Evaluators. *CoRR*, abs/2308.01862.

Zheng, L.; Chiang, W.; Sheng, Y.; Zhuang, S.; Wu, Z.; Zhuang, Y.; Lin, Z.; Li, Z.; Li, D.; Xing, E. P.; Zhang, H.; Gonzalez, J. E.; and Stoica, I. 2023. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. *In NeurIPS*.

Zhuo, T. Y.; Vu, M. C.; Chim, J.; Hu, H.; Yu, W.; Widyasari, R.; Yusuf, I. N. B.; Zhan, H.; He, J.; Paul, I.; Brunner, S.; Gong, C.; Hoang, J.; Zebaze, A. R.; Hong, X.; Li, W.; Kaddour, J.; Xu, M.; Zhang, Z.; Yadav, P.; and et al. 2025. BigCodeBench: Benchmarking Code Generation with Diverse Function Calls and Complex Instructions. *In ICLR*.