软件学报 ISSN 1000-9825, CODEN RUXUEW
2026,37(5):2131−2150 [doi: 10.13328/j.cnki.jos.007508] [CSTR: 32375.14.jos.007508]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

CodeLLMTuner: 基于样本重用的代码大模型选择与解码参数调优框架*

曲慕子[1,2], 亢良伊[1,2], 刘 杰[1,2,3,4,5], 王 帅[1,2], 叶 丹[1,2,3,4], 黄 涛[1,2,3,4]

[1](中国科学院 软件研究所, 北京 100190)

[2](中国科学院大学, 北京 100049)

[3](基础软件与系统重点实验室 (中国科学院 软件研究所), 北京 100190)

[4](计算机科学国家重点实验室 (中国科学院 软件研究所), 北京 100190)

[5](中国科学院大学南京学院, 江苏 南京 211135)

通信作者: 亢良伊, E-mail: kangliangyi15@otcaix.iscas.ac.cn

摘 要: 随着大语言模型 (large language model, LLM) 技术的迅速发展, 涌现了众多代码大模型 (Code LLM), 以支持代码生成、代码补全、代码测试和代码重构等任务. 不同模型在处理相同任务时可能表现出显著的性能差异, 且推理阶段的解码参数也会对模型性能产生重要影响. 研究如何为特定代码开发任务高效地选择最佳模型及其最优解码参数. 现有方法通常将模型选择和参数调优分为两个独立阶段, 由于不同阶段的采样策略差异导致无法共享样本数据, 采样与评估计算成本较高. 考虑到不同代码大模型解码参数空间相同, 提出利用倾向评分匹配 (propensity score matching, PSM) 算法加权调整和对齐不同分布的样本数据, 以提高样本数据复用效率、降低计算成本. 由此提出了一个基于样本重用的代码大模型选择与解码参数调优框架 CodeLLMTuner. 该框架包含 3 个阶段: (1) 独立采样阶段, 对多个代码大模型并行执行解码参数调优 (如贝叶斯优化) 并进行数据采样与评估以收集样本数据; (2) 模型选择阶段, 利用 PSM 技术对齐不同模型的样本数据, 从中选出性能期望最优的模型; (3) 获选模型的解码参数调优阶段, 复用获选模型的样本数据, 并在其基础上继续进行解码参数调优, 以全面探索性能空间并显著降低采样成本. 实验结果表明, 在代码生成、代码摘要和测试用例生成这 3 项任务上, CodeLLMTuner 相比于基线方法在相同成本下性能提升 10%–15%, 或在达到相同性能下成本降低超过 20%.

关键词: 代码大模型; 自动代码生成; 模型选择; 解码参数调优; 倾向评分匹配

中图法分类号: TP311

中文引用格式: 曲慕子, 亢良伊, 刘杰, 王帅, 叶丹, 黄涛. CodeLLMTuner: 基于样本重用的代码大模型选择与解码参数调优框架. 软件学报, 2026, 37(5): 2131–2150. http://www.jos.org.cn/1000-9825/7508.htm

英文引用格式: Qu MZ, Kang LY, Liu J, Wang S, Ye D, Huang T. CodeLLMTuner: Code LLM Selection and Decoding Parameter Tuning Framework Based on Sample Reusing. Ruan Jian Xue Bao/Journal of Software, 2026, 37(5): 2131–2150 (in Chinese). http://www.jos.org.cn/1000-9825/7508.htm

CodeLLMTuner: Code LLM Selection and Decoding Parameter Tuning Framework Based on Sample Reusing

QU Mu-Zi[1,2], KANG Liang-Yi[1,2], LIU Jie[1,2,3,4,5], WANG Shuai[1,2], YE Dan[1,2,3,4], HUANG Tao[1,2,3,4]

[1](Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)

[2](University of Chinese Academy of Sciences, Beijing 100049, China)

[3](Key Laboratory of Systems Software (Institute of Software, Chinese Academy of Sciences), Beijing 100190, China)

* 基金项目: 中国科学院软件研究所基础研究项目 (ISCAS-JCMS-202405)
收稿时间: 2024-11-18; 修改时间: 2025-03-03; 采用时间: 2025-08-12; jos 在线出版时间: 2025-12-10
CNKI 网络首发时间: 2025-12-12

[4](State Key Laboratory of Computer Science (Institute of Software, Chinese Academy of Sciences), Beijing 100190, China)
[5](University of Chinese Academy of Sciences, Nanjing, Nanjing 211135, China)

Abstract: With the rapid development of large language model (LLM) technology, many Code LLMs have emerged to support tasks such as code generation, code completion, code testing, and code refactoring. Different models may show significant performance differences when processing the same task, and the decoding parameters at the inference stage will also have an important influence on model performance. This study investigates how to efficiently select the best model and its optimal decoding parameters for a specific code development task. Existing methods generally divide model selection and parameter tuning into two independent stages. Due to the differences in sampling strategies at different stages, sample data cannot be shared, and the computational cost of sampling and evaluation is high. Considering that the decoding parameter space of different Code LLMs is the same, this study proposes the utilization of the propensity score matching (PSM) algorithm for conducting weighted adjustment and aligning sample data of different distributions to improve the reuse efficiency of sample data and reduce computational costs. Therefore, this study proposes a framework CodeLLMTuner for Code LLM selection and decoding parameter tuning based on sample reuse. The framework includes three stages, including the independent sampling stage, which performs decoding parameter tuning (such as Bayesian optimization) on multiple Code LLMs in parallel and conducts data sampling and evaluation to collect sample data. Additionally, the model selection stage adopts PSM technology to align the sample data of different models and selects the model with the optimal performance expectations. The decoding parameter tuning stage of the selected model reuses the sample data of the selected model and continues decoding parameter tuning based on it to fully explore the performance space and significantly reduce sampling costs. Experimental results show that in the three tasks of code generation, code summarization and test case generation, CodeLLMTuner improves performance by 10% to 15% at the same cost compared to the baseline methods, or reduces the cost by more than 20% under the same performance.

Key words: Code LLM; automatic code generation; model selection; decoding parameter tuning; propensity score matching (PSM)

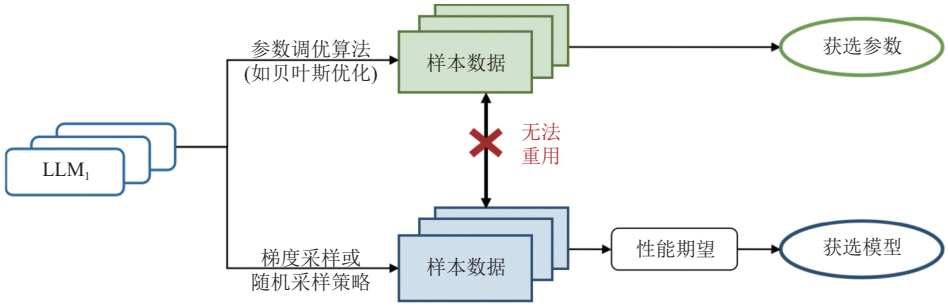
大语言模型技术的快速发展推动了 Codex[1]、StarCoder[2]、Code Llama[3]等代码大模型和 GitHub Copilot[4]等智能开发助手的兴起[5]. 这些专门针对编程语言和软件开发任务训练的代码大模型被广泛应用于自动代码生成[6–9]、代码修复[10,11]、代码测试[12]和代码摘要[13]等任务. 当前代码大模型提供了多种可调的解码参数, 如温度和惩罚参数, 这些参数可以影响代码大模型输出的结果, 从而影响代码开发任务的性能. 本文中“性能”指代码大模型完成特定软件开发任务时的质量, 如代码生成任务中代码的正确性和测试用例生成任务的代码覆盖率. 而不同的训练数据集和模型结构导致不同的代码大模型在完成软件开发任务时的性能有明显差异. 这种模型与参数差异导致的性能差异直接影响软件开发工作的效率与质量, 如表 1 所示, 同一模型的不同解码参数以及不同模型的同一解码参数所对应的代码生成任务正确率可能存在极大差距, 因此用户需要针对特定软件开发需求选择合适的代码大模型与解码参数.

表 1　不同代码大模型在 HumanEval 上采用不同温度解码参数时的性能 (%)

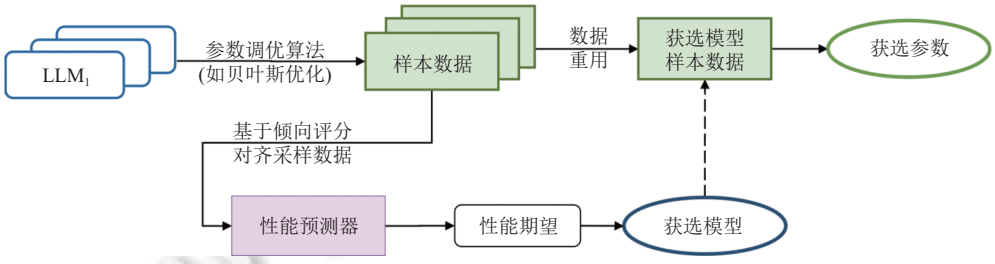
代码大模型	评价指标	temperature=0.2	temperature=0.4	temperature=0.6	temperature=0.8	temperature=1.0
Code Llama 34B	*pass*@1	48.18	45.65	44.90	43.31	36.59
	pass@4	60.42	62.82	67.35	68.83	63.32
	pass@10	66.34	70.10	77.92	81.02	77.66
WizardCoder 34B	*pass*@1	27.95	28.96	26.79	26.92	25.10
	pass@4	39.52	48.42	48.39	55.82	53.85
	pass@10	44.71	58.05	60.01	72.47	69.82
PhindCoder 34B	*pass*@1	49.84	54.77	52.89	52.40	52.34
	pass@4	61.69	71.02	74.19	74.13	78.07
	pass@10	66.71	77.43	81.74	81.77	87.17

通常来讲, 为充分探索解码参数性能空间, 代码大模型的解码参数调优需要对多组解码参数进行采样和评估, 即从解码参数空间选择某组配置, 并在测试基准 Benchmark 上评价所选择的配置对应的性能. 然而对于代码开发任务, 这一过程通常会消耗大量时间和资源, 而庞大的参数空间与候选代码大模型的数量也使得采样与评估成本成倍增加. 以表 1 为例, 该表仅包括 3 个候选模型与 1 维解码参数共计 15 个采样点, 但对于每个采样点均需要为 HumanEval 共 164 项任务中的每项任务重复生成 40 条以上目标代码并进行评估, 以计算生成代码通过率的期望值, 若设最大长度为 400, 则该表需要消耗 15×164×40×400 = 39360000 个 token, 以 Together AI 平台的报价为例约为 31 美元. 而代码大模型选择与参数调优通常需要对数十个模型与上百个采样点进行采样与评估. 因此, 代码大模型选择与解码参数调优的主要挑战在于如何在有限的成本下高效地选择更优的模型和参数[14].

目前解决此类问题的常用方法是分为模型选择与参数调优两个阶段, 如图 1 所示. 为了计算模型在整个参数配置空间中的性能期望以表征模型性能, 模型选择策略通常使用梯度采样或随机采样的方式在不同模型上收集分布相同的样本数据; 为尽可能快地确定最优参数, 参数调优策略通常采用如贝叶斯优化的非随机采样策略收集样本数据, 这些样本数据在不同模型上的分布不同. 不同采样策略得到的样本数据概率空间分布不同, 导致模型选择与参数调优两阶段无法实现数据共享, 均需要大量样本并进行模型的性能评估.



(a) 组合方法流程示意图



(b) CodeLLMTuner流程示意图

图 1　组合方法与 CodeLLMTuner 方法的流程示意图

当前代码大模型都采用 Transformer 类架构, 因此不同的代码大模型有着基本相同的解码参数空间; 而倾向评分匹配 (propensity score matching, PSM) 算法对有相同参数空间的样本, 可以通过加权调整的方式来对齐不同分布的样本数据. 因此, 本文引入 PSM 算法, 将参数调优策略所收集的跨模型异分布样本数据转化为模型选择策略所需要的跨模型同分布样本数据, 以实现不同阶段之间的样本重用并降低计算成本. 基于此, 本文提出了基于倾向评分匹配 (PSM) 方法的代码大模型选择与参数调优框架 CodeLLMTuner, 该框架将代码大模型选择与参数调优过程分为 3 个阶段: (1) 独立采样阶段, CodeLLMTuner 在每个代码大模型上并行执行解码参数调优 (如贝叶斯优化) 以收集样本数据; (2) 在模型选择阶段, 利用 PSM 技术对齐不同模型的样本数据, 通过多任务学习训练性能预测器以计算每个模型的性能期望, 从中选出性能期望最优的候选模型; (3) 获选模型的解码参数调优阶段, CodeLLMTuner 重用获选模型在第 1 阶段的样本数据, 并在其基础上继续进行调优, 不仅充分探索性能空间, 还能显著降低采样成本. CodeLLMTuner 作为一个可扩展的框架, 能够集成多种参数调优策略和模型选择方法, 包括贝叶斯优化和拉丁超立方采样方法.

综上所述, 本文的贡献如下.

- 设计了一个可扩展的代码大模型选择与参数调优框架 CodeLLMTuner, 包括独立采样、模型选择与获选模型解码参数调优这 3 个阶段, 有效复用第 1 阶段样本评估数据, 在有限成本下为性能空间探索分配更多资源, 并支持集成多种参数调优策略和模型选择方法.
- 提出了一种基于倾向评分算法的不同代码大模型不同分布的样本评估数据的对齐方法, 采用多任务学习方法分别对倾向评分预测与性能预测两任务进行训练, 以此构建代码大模型的性能期望预测器, 从而复用第 1 阶段样本评估数据选出候选模型.

● 针对代码生成、代码摘要、测试用例生成等关键任务的 Benchmark 对 CodeLLMTuner 与其他基线方法进行了详尽的实验比较评估. 结果表明, 相较于基线方法所选择的候选模型与参数, CodeLLMTuner 在相同成本下可提高 10%–15% 的性能, 或达到相同性能效果节省 20% 以上的成本.

1 相关工作

本节分别介绍大语言模型参数调优方法与多模型选择及参数调优方法.

1.1 大语言模型参数调优方法

在机器学习与深度学习领域已有大量超参数调优的工作, 在大语言模型领域这方面的研究刚刚起步, 相关研究工作可分为推理过程参数调优与训练过程参数调优.

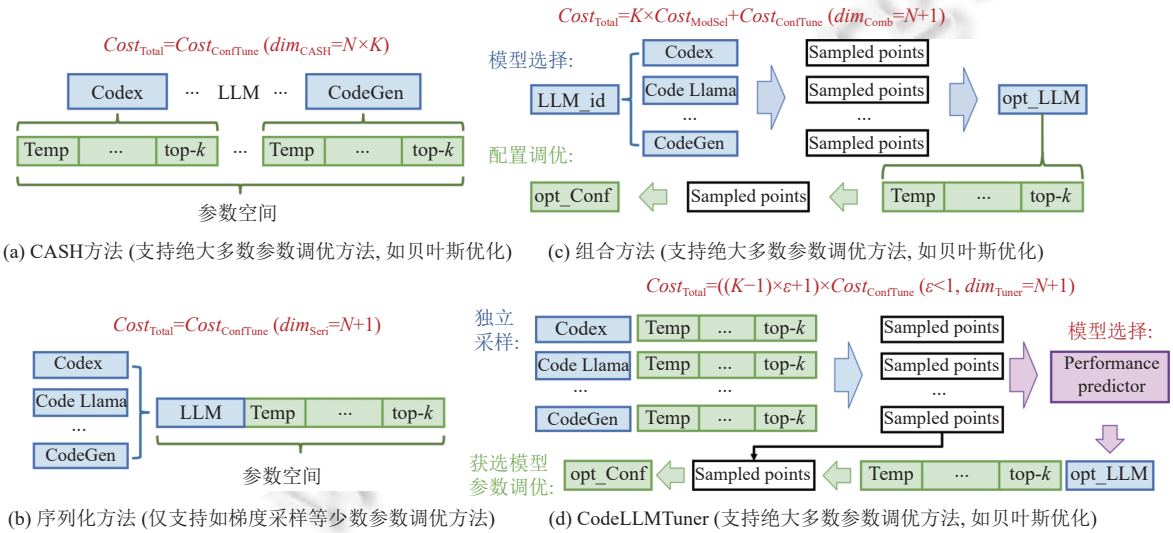
在大语言模型推理阶段, 解码参数会极大影响大语言模型输出结果, 这种对推理时输出的直接控制是确保语言模型在实际应用中实用性的有力工具, 相关工作对如何选择解码参数进行了探索. EcoOptiGen[15]通过高斯过程建立大语言模型性能与其解码配置之间的关系, 并使用贝叶斯优化解决配置调优问题; 文献 [16] 自主确定输入样本的最佳解码策略和配置, 实现大语言模型的自我调优; 文献 [17] 将超参数调整表述为在线多臂老虎机 (multi-armed bandit, MAB) 问题, 并引入两级分层方法以有效探索配置空间. 除此之外, 若将提示词作为参数考虑, 大语言模型推理结果的质量与提供的提示词的信息数量和完善度有关, 因此相关工作对如何选择提示词进行了探索: EvoPrompt[18]借鉴进化算法思想, 通过为提示种群迭代生成新提示以改进种群, 从而实现大语言模型提示调优; MPT[19]从多个源提示中提炼知识并使用迁移学习方法更新可迁移提示以使其适应每个下游目标任务.

大语言模型参数调优还包括训练阶段参数调优, 在大语言模型进行训练或微调时, 超参数会对模型最终性能有重要影响, 因此相关工作对如何选择微调超参数进行了探索: PandaLM[20]提供了一个用于大语言模型微调超参数调优的自动、稳健且可靠的评估 Benchmark; 而文献 [21] 使用 MADS 与 NOMAD 两个黑盒调优算法对低秩自适应 (LoRA) 微调方法进行超参数调优.

然而, 上述方法仅针对特定的单一大语言模型进行配置调优, 而实际应用中常需在多个大语言模型间进行选择, 并同时进行解码配置调优.

1.2 多模型选择及参数调优方法

如图 2 所示, 目前关于多模型选择及参数调优的相关工作可分为 4 类: 联合算法选择和超参数优化 (combined algorithm selection and hyperparameters optimization, CASH)、模型序列化方法、迁移学习方法与多阶段组合方法.



(a) CASH方法 (支持绝大多数参数调优方法, 如贝叶斯优化)　(c) 组合方法 (支持绝大多数参数调优方法, 如贝叶斯优化)

(b) 序列化方法 (仅支持如梯度采样等少数参数调优方法)　(d) CodeLLMTuner (支持绝大多数参数调优方法, 如贝叶斯优化)

图 2　多模型选择及参数调优方法概述

CASH 方法已成为 AutoML[22]中的关键研究问题, 针对该领域目前已经开展了大量工作. 此类研究假设每个候选算法的参数空间不同, 由此将不同算法的参数空间重新组织成一个大型联合参数空间, 并在该联合参数空间上进行参数调优. 例如, Auto-WEKA 2.0[23]采用贝叶斯优化自动搜索最佳联合参数以最大化性能. Rising Bandits[24]将贝叶斯优化与多臂老虎机相结合, 在联合参数空间内分阶段对不同的模型进行调优. DivBO[25]引入了算法间多样性的概念 (即其最优配置的相似性), 并使用该指标作为加权函数来指导基于贝叶斯优化的搜索过程. Auto-Model[26]利用现有研究构建信息网络, 为给定任务选择最合适的机器学习算法, 将贝叶斯优化与遗传算法相结合进行参数调优. 尽管目前研究人员在 CASH 问题上进行了广泛的研究, 但在大模型选择和解码参数调优的组合问题中, 由于每个模型都有着相同的参数空间, 因此直接应用上述算法所构建的联合参数空间所需成本过高.

序列化方法仅支持模型参数空间相同的场景, 该方法将模型序列号作为一维新参数加入参数空间, 以此将多模型的参数调优问题转化为单模型的参数调优问题. 例如, OtterTune[14]提出将多模型作为一个新的特征加入参数空间的特征集; ResTune[27]提出使用 lasso 模型来简化该特征集; 而 Ant[28]使用贝叶斯优化方法实现模型与参数的调优. 由于此类方法为模型序列号分配的参数可能并不恰当, 这类方法在使用贝叶斯优化等复杂方法进行探索时可能困于局部最优, 而在使用网格搜索等简单策略时探索效率低下.

迁移学习方法认为不同模型对应的参数空间上的性能分布是相似的, 以此将模型划分为源环境和目标环境. 该方法在源环境上通过采样与评估收集样本数据以训练性能模型, 然后将源环境上的性能模型经过微调后迁移至目标环境中, 以此预测不同环境的性能并选择最优模型与参数. 例如, Bodin 等人[29]基于随机森林实现性能模型迁移; L2S[30]根据源环境的信息在目标中简化特征集以选择更好的数据点进行采样, 该方法通过缩减性能空间的方式减少成本; 而 Tighineanu 等人[31]使用高斯过程拟合性能模型并使用贝叶斯优化为后续采样提供决策. 然而, 此类方法通常缺少可解释性, 因此难以保证性能预测的准确性; 而且此类算法预测精度受源环境下性能预测模型的准确度影响大, 当源环境与最优环境性能分布不一致时会加大性能预测的误差.

多阶段组合方法将多模型参数调优问题分为模型选择问题和参数调优问题加以解决, 前者需要模型选择方法来选择最优模型, 而后者需要参数调优方法选择最优参数. 对于参数调优方法, 当前通常采用“采样-评估-再采样”的迭代方法收集样本数据并以此选择最优点, 如贝叶斯优化、演化算法等; 对于模型选择方法, 目前已有大量关于如何选择用于定量描述未来观测的最优模型的研究, 如: 赤池信息量准则 (Akaike information criterion, AIC)[32]将样本外预测损失近似为样本内损失和校正项之和; 贝叶斯信息准则 (Bayesian information criterion, BIC)[33]用样本大小的对数代替了 AIC 惩罚项中的常数; 桥接准则 (bridge criterion, BC)[34]还能在渐近状态下结合 AIC 和 BIC 的优势, 有着较好的效果. 然而, 上述方法仅适用于机器学习模型, 而并不适用于大语言模型这一类生成式模型, 这导致大语言模型所能采用的模型选择方法受限, 造成难以承受的调优成本.

考虑到 CASH 方法、序列化方法与迁移学习方法存在的问题, 本文采用组合方法, 将模型选择与参数调优相结合以解决代码大模型选择与参数调优问题. 由于现有的模型选择方法不适用于代码大模型, 考虑到性能期望更高的代码大模型所选择的解码参数在参数调优成本相同的情况下性能最优的概率更高, 本文以模型性能期望作为选择最优模型的指标以减少误差.

2 CodeLLMTuner

本节介绍 CodeLLMTuner 框架, 由图 3 中的 3 个阶段组成. 其中, 绿色三角形表示执行参数调优过程, 橙色长方形表示暂停参数调优过程, 红色正方形表示终止参数调优过程.

独立采样阶段: CodeLLMTuner 对每个候选模型并行独立采用解码参数调优策略 (如贝叶斯优化等方法) 迭代采样和评估以收集样本数据, 在 K 次迭代后暂停上述参数调优过程.

模型选择阶段: 基于 PSM 方法, CodeLLMTuner 使用样本数据构建多任务学习倾向评分与性能预测器, 该预测器输入解码参数并预测每个候选模型上该解码参数的倾向评分和性能, 由此可计算每个模型的性能期望并选择最优模型.

获选模型的解码参数调优阶段: CodeLLMTuner 在获选模型上重用样本数据以继续执行第 1 阶段中暂停的参数调优过程, 在 N 次迭代后停止参数调优过程并选择最优解码参数.

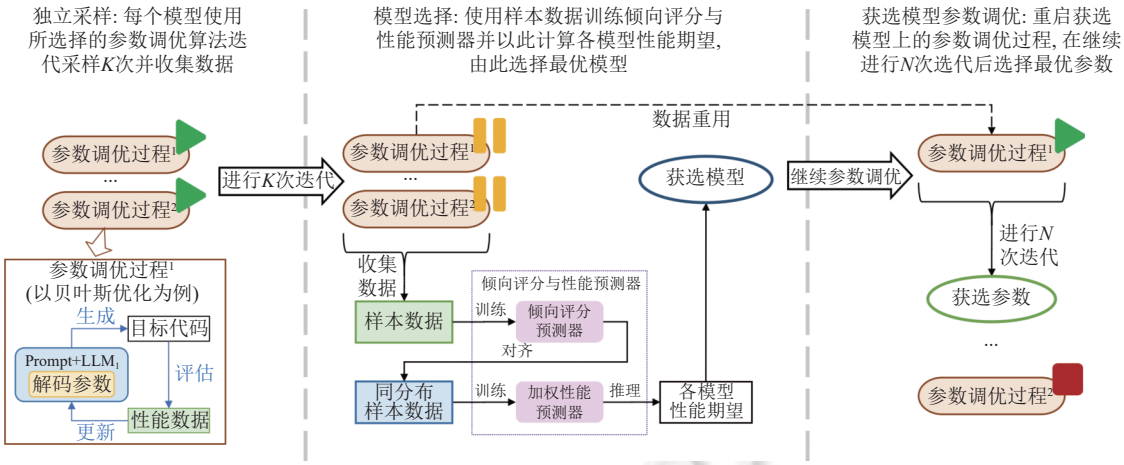
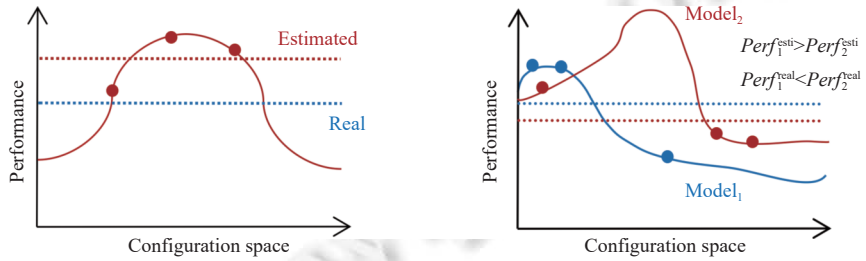


图 3 CodeLLMTuner 架构示意图

2.1 独立采样阶段

如前文所述, 现有的 CASH 方法与序列化方法由于参数空间维度过高或支持调优方法有限等问题均难以满足代码大模型选择与参数调优的需求, 因此本文基于多阶段组合方法, 通过独立采样阶段、模型选择阶段与获选模型的解码参数调优这 3 阶段解决该问题, 其中采样阶段用于收集可供模型选择与参数调优两阶段使用的样本数据. 而由于代码大模型采样成本过高, 分别为两阶段单独采样会导致成本难以接受, 因此本文期望引入数据重用策略, 以提高数据利用率.

然而, 模型选择阶段所需的样本数据分布与参数调优方法所采样的样本数据分布不同, 使得难以直接将参数调优样本数据重用于模型选择. 代码大模型选择通常采用随机采样或梯度采样的方式收集模型间同分布数据以计算代码大模型性能期望; 而参数调优方法由于不同代码大模型间性能分布差异, 通常收集模型间不同分布样本数据. 样本数据分布的差异使得直接重用样本数据会导致性能估计失准. 如图 4 所示, 其中实线表示性能模型, 点表示样本数据, 虚线表示各模型的性能期望. 如图 4(a) 所示, 即使对于单个模型, 由于参数调优阶段所收集的样本数据具有一定的倾斜, 直接使用该样本数据计算出的性能期望与使用梯度采样或随机采样所收集的样本数据计算出的真实值会存在较大差异. 而且, 在多个代码大模型场景中, 如图 4(b) 所示, 模型间样本数据分布的差异增加了预测不同模型性能期望的误差. 而对于参数调优阶段, 现有工作指出, 目前常用的参数调优方法通常在采样之后根据评估结果调整后续采样策略, 而不同分布的样本数据之间不存在逻辑关联, 难以实现数据重用.



(a) 非随机分布下样本数据期望与真实期望的差值 (蓝线表示真实期望)

(b) 模型间样本数据分布差异导致的性能期望估计失准 (不同颜色表示不同模型)

图 4 使用模型间不同分布样本数据估计的性能期望概览

因此本文期望对模型间不同分布样本数据进行处理, 使用匹配方法对齐不同分布的样本数据, 以使用参数调优采样策略收集的样本数据计算模型性能期望并选择最优模型. 具体来说, CodeLLMTuner 在每个模型上使用如贝叶斯优化等方法同时进行解码参数调优, 在该过程中迭代采样和评估以收集样本数据, 在 K 次迭代后暂停参数

调优, 所收集的数据在对齐后被用于计算模型性能期望并选择最优模型.

2.2 模型选择阶段

由于倾向评分匹配算法可以通过加权对齐操作将不同分布数据转化为相同分布的数据, 本文使用该方法对齐独立采样阶段收集的不同分布的数据, 并以此构建倾向评分与性能预测器, 用于预测各模型性能期望以选择最优模型. 其流程如图 5 所示. 对于图 5 中的变量, 数字为下角标表示第几条数据, 数字为上角标表示数据的第几个维度. (1) 为样本数据中的每条数据项计算所属向量; (2) 冻结性能私有层, 同时训练共享层和倾向评分私有层, 构建倾向评分预测器; (3) 然后为样本数据中的每条数据项使用训练好的模型预测倾向评分; (4) 基于倾向评分计算权重值; (5) 冻结共享层, 同时训练性能私有层, 构建加权性能预测器; (6) 使用梯度采样或随机采样为各个模型收集大量解码参数配置并基于性能预测器预测其性能, 以每个模型性能的平均值作为该模型的性能期望, 以此选择最优模型.

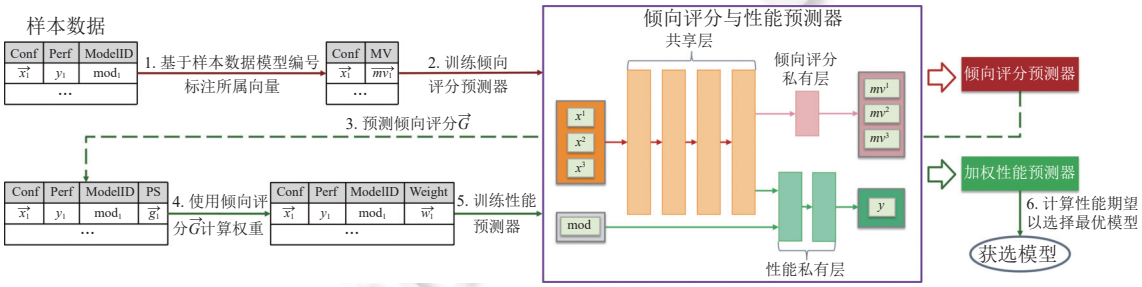
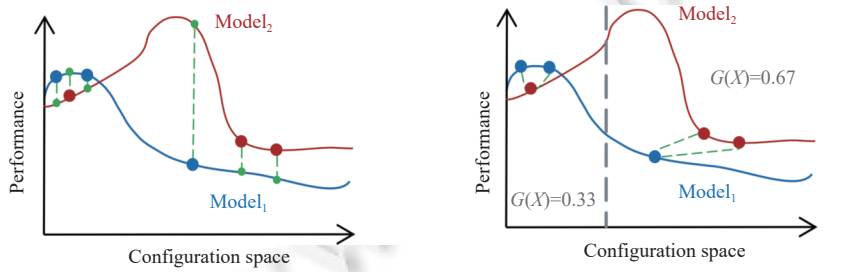


图 5 多任务学习构建模型性能期望预测模型

(1) 基于样本数据被采样时的模型编号标注所属向量

对于如何处理不同分布的样本数据, 以计算平均因果效应 (ATE) 为例: 本文假设因变量为 T, 果变量为 Y, 其他能观察到的变量集合为 X, 而因变量取值范围为 $S=\{0,1\}$. 解决此类问题的直观方法是通过收集尽量多的数据, 以满足条件独立性假设 $T\perp(Y(0),Y(1))|X$, 即 Exact matching 方法, 如图 6(a) 所示 (绿点表示匹配需要的样本点, 绿线表示样本点之间的匹配关系). 然而考虑到统计开销与可实现性后该方法并不可行, 因此倾向评分匹配方法 (propensity score matching, PSM)[35]被用于解决该问题. 如图 6(b) 所示 (绿点、绿线表征内容同图 6(a)), 在 PSM 方法中, 首先对每一个个体计算一个倾向评分 $g(X)$ (propensity score), 此时将条件独立性假设 $T\perp(Y(0),Y(1))|X$ 放松为 $T\perp(Y(0),Y(1))|g(X)$, 接着根据倾向性得分对于个体进行匹配. 此时该方法要求对于每一个 $T=1$ 的个体, 都能从 $T=0$ 的分组里找一个或多个倾向评分相同的个体而非 X 变量一模一样的个体以进行匹配.



(a) Exact matching算法为每条样本收集对应数据点, 以绿点表示

(b) PSM方法按照倾向评分切分参数空间, 并匹配各个子空间内的样本数据

图 6 Exact matching 与 PSM 方法示意图

考虑到当前倾向评分估计方法假设模型总数为 2, 而实际使用中通常无法满足这一要求, CodeLLMTuner 使用倾向评分向量 $\overrightarrow{G(x)}=(G^1(x),G^2(x),\ldots,G^{|M|}(x))$ 而非单个值来表示解码参数配置 $X=x$ 在每个模型上的倾向评分, 其

中 $|M|$ 表示模型总数. 为对倾向评分向量 $\overrightarrow{G(x)}$ 建模, 本文为样本数据集中的每条数据计算所属向量 $\overrightarrow{m(x_i)}=(m^1(x_i), m^2(x_i),\dots,m^{|M|}(x_i))$, 用以表征该条数据是被哪个模型采样, 其中 x_i 表示数据集中的第 i 项数据的解码参数配置, 假设该项数据是由模型 $T=t$ 采样的, 则有:

$$m^j(x_i)=\begin{cases}1, & j=t\\ 0, & j\neq t\end{cases}.$$

(2) 训练倾向评分预测器

使用倾向评分匹配方法对齐不同分布样本数据需要对倾向评分进行建模, 因此本文使用神经网络模型构建倾向评分预测器以估计样本数据倾向评分; 而由于倾向评分方法仅将不同分布的样本数据对齐至相同分布, 而不保证对齐后的分布与随机采样分布相同, 因此 CodeLLMTuner 构建性能预测器以计算各模型性能期望. 考虑到性能和倾向评分建模任务之间的相似性, CodeLLMTuner 采用多任务学习方法 (如图 5 所示), 以节省计算资源并提高准确性. 该模型包括 4 层共享层、1 层倾向评分私有层与 2 层性能私有层, 其中共享层用于提取输入参数在不同模型上与最优解的相似度特征, 倾向评分私有层将该特征转化为在不同模型上被参数调优方法采样的概率, 而性能私有层将其转化为该参数在不同模型上的性能估计.

在构建所属向量后, 以解码参数配置为输入, 所属向量为输出, CodeLLMTuner 训练共享层与倾向评分私有层. 设 n 表示样本数据集大小, x_i 表示数据集中的第 i 条样本数据, 损失函数如下:

$$\hat{\theta}=\arg\min_{\theta}\hat{E}(\theta;X),\ \text{where}\ \hat{E}(\theta;X)=\frac{1}{n}\sum_{i=1}^{n}\sum_{t=1}^{|M|}CrossEntropy(G^t(x_i;\theta),m^t(x_i)) \tag{1}$$

(3) 预测倾向评分 $\overrightarrow{G}$

PSM 方法的关键在于如何构建适当的倾向评分函数 g, 以保证条件独立性假设成立. 本文采用 TARNET 所提出的方法, 该方法要求模型总数为 2, 将倾向评分定义为各解码参数配置被不同模型采样的概率, 即 $g(x)=P(T=1|X=x)$, 此时 $T=1$ 的数据集期望为:

$$\begin{aligned}E[Y|do(T=1)]&=\int_0^1 E[Y|g(X)=g,T=1]P(g(X)=g)\mathrm{d}g\\&=\int_0^1 E\left[\frac{Y}{g(X)}|g(X)=g,T=1\right]P(T=1,g(X)=g)\mathrm{d}g=E\left[\frac{Y}{g(X)}|T=1\right].\end{aligned}$$

若将 $1/g(x)$ 视作权重, 该方法实质上是通过对样本数据加权以对齐不同分布的样本数据, 因此也被称为逆概率加权法. 为计算样本数据中各数据项的权重以对齐数据, CodeLLMTuner 将样本数据输入倾向评分预测器, 预测各数据项的倾向评分 $\overrightarrow{G}$.

(4) 使用倾向评分计算样本数据权重

在模型总数大于 2 时, 考虑到归一化处理, 在对 i、j 两模型的样本数据进行对齐时, i、j 上的数据项权重分别为 $(G^i+G^j)/G^i$ 和 $(G^i+G^j)/G^j$. 为对齐各个模型样本数据的分布, 本文计算该权重的平均值作为统一分布后的数据项权重, 即:

$$W(x,t)=1+\frac{\sum_{t'\neq t}^{|M|}G^{t'}(x)}{(|M|-1)\times G^t(x)}=\frac{(|M|-2)\times G^t(x)+1}{(|M|-1)\times G^t(x)} \tag{2}$$

(5) 训练性能预测器

其后, 本文对样本数据集中的每项数据均使用倾向评分进行加权处理, 对齐模型之间的配置分布以避免样本数据分布的倾斜对性能期望估计结果的影响. 以共享层输出与模型 id 为输入, 以对应模型的性能作为输出, 采用以下损失函数进行训练, 其中 t_i 表示数据集中的第 i 项的所属模型 id, y_i 表示数据集中的第 i 项的性能评分.

$$\hat{\theta}=\arg\min_{\theta}\hat{R}(\theta;X),\ \text{where}\ \hat{R}(\theta;X)=\frac{1}{n}\sum_{i=1}^{n}W(x_i,t_i)\times(Q(x_i,t_i;\theta)-y_i)^2 \tag{3}$$

(6) 计算模型性能期望以选择最优模型

最后, CodeLLMTuner 使用梯度采样为各个候选模型收集大量参数配置, 并基于加权性能预测器预测其性能, 然后为每个模型计算该性能的平均值作为该模型的性能期望, 以此选择最优模型.

2.3 获选模型的解码参数调优阶段

由于独立采样阶段采用了参数调优方法的采样策略, 因此 CodeLLMTuner 可直接利用该样本数据完成解码参数的选择. 然而为进一步探索性能空间以选择更优的解码参数, 本文在选择最优模型后继续进行参数调优. 参数调优方法通常采用“采样-评估-再采样”的迭代流程, 因此虽然此类算法难以重用其他分布的样本数据, 但是可以重用相同采样算法收集的样本数据, 即 CodeLLMTuner 可通过重用第 1 阶段的样本数据与采样策略继续探索获选模型的性能空间.

从模型的角度来看, CodeLLMTuner 实际上是将完整的参数调优过程切分为两个阶段: 在第 1 阶段为每个模型执行参数调优, 然后终止非最优模型上的参数调优过程; 在第 2 阶段, CodeLLMTuner 在获选模型上继续进行参数调优. 具体而言, CodeLLMTuner 首先在参数调优第 1 阶段采用例如贝叶斯优化的采样策略, 以迭代方式为每个模型采样和评估 K 个配置点. 随后, 使用这些模型间不同分布样本数据来估计每个模型的性能期望, 从而选择最优模型. 最后, CodeLLMTuner 在参数调优第 2 阶段重用第 1 步中的采样策略与样本数据继续迭代采样和评估配置 N 次, 以确定最优配置. 由于该方法仅要求参数调优策略采用“采样-评估-再采样”的迭代流程, 因此有较强的适用性.

经实验验证 (见第 3.4 节), 当总成本固定时, 不同的 K、N 比值会导致一定的性能差异. 而考虑到模型选择仅需要对性能期望进行排序, 对性能预测精度的要求不高, 导致模型选择成本增加带来的性能收益不如参数调优. 因此在实际使用中, 用户可以先在第 1 阶段为不同代码大模型并行分配采样成本进行模型选择以收集样本数据, 以此预测不同成本下的获选模型, 当该变量收敛, 即模型选择成本的增加不会改变模型选择的结果时, 停止非获选模型上的参数调优过程, 并将剩余的成本用于探索获选模型上的最优配置.

3 实验分析

3.1 实验数据

为全面评估 CodeLLMTuner 的效果, 本文构建代码大模型选择与解码参数空间.

本文选择 WizardCoder[7]、PhindCoder[36]和 Code Llama[3]作为候选大语言模型, 参数规模覆盖 7B、13B、34B. 如此选择的原因在于: 首先, 这些模型是当前代码生成领域中最具代表性和广泛使用的模型, 能够覆盖不同的技术路线和优化策略; 其次, 这些模型的开源代码和预训练权重易于获取, 便于复现实验和进行比较; 最后, 这些模型在公开基准测试中表现优异, 能够为本文的研究提供可靠的基线.

对于解码参数空间的构建, 本文选择对代码任务完成质量有明显影响的解码参数进行实验: temperature 会影响各 token 的候选概率; top-p、top-k 会直接决定 token 的采样范围; repeatition_penalty 和 length_penalty 在代码摘要中可以有效提高生成结果的可读性, 但在代码生成与测试用例生成时却可能反向影响大模型生成正确的代码. 因此本文选择 temperature、top-p[37]、top-k[38]、repeatition_penalty 和 length_penalty 参与实验, 这些参数是生成任务中最常用的解码参数, 能够全面控制生成文本的质量、多样性、流畅性和长度. 具体参数取值范围如表 2 所示.

表 2 解码参数空间中各参数取值范围

解码参数	取值范围	描述
temperature	uniform(0, 1)	通过调整候选概率影响解码策略的随机性
top-p	uniform(0, 1)	通过影响token list影响生成结果
top-k	lograndint(2, 1 024)	通过影响token list影响生成结果
repeatition_penalty	uniform(0, 1)	通过调整候选概率以避免重复生成多个token
length_penalty	uniform(0, 1)	通过对句子进行归一化处理鼓励模型生成更长的句子

由此构建实验的参数空间, 该空间包含 1 875 个候选模型与解码参数的组合. 大语言模型公共推理云平台 Together AI Platform[39]支持用户对多种大语言模型调用推理服务, 同时提供了包括 temperature 等一系列的解码参数供用户选择, 本文选择该平台进行代码任务的推理. 为探讨 CodeLLMTuner 对不同种类代码任务进行代码大模型选择与解码参数调优的能力, 本文选择代码生成、代码摘要与测试用例生成这 3 项代码任务作为目标任务并为其构建性能空间, 此 3 项任务覆盖了代码任务的不同类型.

代码生成任务: 该任务属于“文本-代码”类型的代码任务, 要求代码大模型基于用户所提出的代码需求描述生成满足需求的代码. 本文选择 HumanEval[1]作为 Benchmark, 该数据集中的各项问题均由人工编写, 是代码生成任务性能评估最为常用的数据集, 有一定的权威性. 为验证输出代码是否正确, 当前通常使用 *pass@k* 表征模型性能, 该方法要求代码大模型为 Benchmark 中的每个任务生成 n 段代码, 计数通过单元测试的代码数量 c, 最终通过公式 (1) 计算每个任务的分数, 其平均值即为代码大模型在 Benchmark 上的性能.

$$pass@k = 1 - \frac{C_{n-c}^{k}}{C_{n}^{k}}. \tag{1}$$

对于 k 的取值, 考虑到模型选择与参数调优目标是在有限的成本下为特定类型的数据集高效地选择更优的模型和参数, 应侧重于代码生成次数较少时的任务场景. 因此本文选择 *pass*@1 以衡量模型在首次代码生成时的性能表现, 选择 *pass*@4 以衡量模型在生成次数较紧凑时的性能表现, 以及选择 *pass*@10 以衡量代码生成更多次时模型的性能表现. 而更大的 k 值通常代表了成本极为充裕情境下的表现, 对研究问题的指导意义较为有限且计算成本极高. 因此, 本文省略了更大的 k 值场景下的实验, 而将精力集中在更具现实应用价值的代码生成次数较少的情境下, 即选择 *pass@k* (k=1、4、10) 作为性能评价指标, 以体现模型在不同代码生成次数下的泛化能力.

代码摘要任务: 该任务属于“代码-文本”类型的代码任务, 要求代码大模型基于提供的代码片段输出该片段的注释或解释. 为验证输出摘要是否准确, 目前通常使用 *BLEU* 分数判断代码大模型输出结果与标准结果的相似程度, 用以表征代码摘要任务的执行质量. CodeXGLUE[40]提供了一个包括 23 007 个代码摘要任务的 Benchmark, 该数据集收集自 GitHub 并经过了规则筛选与调整, 本文从中随机选择 1 000 项作为本实验的 Benchmark. 虽然部分代码摘要的工作采用人工审查以更准确地验证自动生成的摘要是否准确反映了代码的功能与逻辑, 然而模型选择与参数调优场景通常需要数十乃至上百个采样点以充分刻画性能分布, 人工审查成本过高, 且参数调优方法大多采用迭代方法进行采样, 人工审查可能引入人为干扰影响算法实现进而影响实验可靠性, 因此本文选择 *BLEU*_1 与 *BLEU*_2 作为性能评价指标以自动化评估推理结果.

测试用例生成任务: 该任务属于“代码-代码”类型的代码任务, 要求代码大模型基于提供的代码片段输出该片段对应的测试用例, 该任务通常使用生成测试用例的分支覆盖率和行覆盖率评价代码任务完成质量. 不同程序语言的覆盖率测试方法有所不同, 如对于 Python 语言的测试用例, 当前通常使用 pytest 和 slipcover 库进行测试; 对于 JavaScript 语言, 可采用 Mocha 和 Istanbul 工具进行覆盖率测试. CodaMosa[41]经过对多个第三方 Python 库的扫描与 MOSA 测试提供了一个用于生成测试用例的 Benchmark, 本文选择针对 Flask 库的 Benchmark 进行实验.

本文选择并使用了现有的数据集进行性能评估, 虽然大模型可能已经学习过这些数据集中的部分数据, 实验结果可能会高估模型的性能. 然而, 即使存在数据泄露, CodeLLMTuner 与基线方法的对比仍然具有意义, 因为所有基线方法都是在相同的候选模型上进行参数调优. 此外, 数据泄露是领域内的普遍问题, 相关文献也表明通过使用权威评测集可以在一定程度上减轻其影响.

3.2 基线方法

考虑到模型选择与参数调优过程需要同时用到 CPU 与 GPU 资源, 难以使用单独一种资源概括该过程的总成本, 而迭代次数是参数调优算法中衡量成本的常用指标. 因此为便于衡量模型选择与参数调优过程总成本, 本文使用该过程中进行“采样-评估”的迭代次数作为成本的衡量标准. 具体来说, 各基线方法在选择最优模型与参数时, 对多少模型与参数组合进行了代码任务推理与性能评测. 这种定义能够反映模型选择与参数调优的计算开销, 同时确保实验的公平性和可复现性. 本节假设代码大模型选择与解码参数调优的总成本为 I, 模型数量为 $|M|$.

DivBO[25]: 作为 CASH 方法, 该方法将多个代码大模型的解码参数空间重组为一个组合参数空间, 并在其中采

用贝叶斯优化选择最优模型与解码参数. 该方法在每次迭代中为 $|M|$ 个模型分别采样一组样本数据. 在完成 $I/|M|$ 次迭代后, 选择最佳模型和解码参数.

EcoOptiGen[15]: 该方法将模型 id 序列化为一维参数, 将代码大模型选择与解码参数调优转换为单模型参数调优问题. 该方法在新的参数空间上使用贝叶斯优化迭代 I 次以选择最佳模型和解码参数.

Transfer learning[31]: 该方法随机选择某模型作为源环境, 其他模型作为目标环境. 该方法在源环境上通过贝叶斯优化方法迭代采样 K 次并由此构建性能模型, 然后在每个目标环境上采样 $(I-K\times|M|)/(|M|-1)$ 次以迁移性能模型. 之后, 该方法基于各个性能模型选择最优模型与解码参数.

AutoRAG-HP[17]: 该方法首先为每个模型随机采样 K 次, 计算它们的期望值以确定最佳模型. 然后在获选模型上使用贝叶斯优化迭代采样 $I-|M|\times K$ 次, 以选择最优解码参数.

3.3 实验结果

为了评估 CodeLLMTuner 与其他基线方法的性能差异, 本文提出了以下研究问题 (RQ).

- RQ1: 成本相同的情况下, CodeLLMTuner 与基线方法在不同种类的代码任务上的性能差异有多大?
- RQ2: 成本相同的情况下, CodeLLMTuner 与基线方法在组合代码任务上的性能差异有多大?
- RQ3: 为达到相同的性能, CodeLLMTuner 相较于基线方法可以节省多少成本?

RQ1: 成本相同的情况下, CodeLLMTuner 与基线方法在不同种类的代码任务上的性能差异有多大?

代码大模型选择与解码参数调优的首要目标是在成本一定的情况下找到性能更好的代码大模型和解码参数. 以代码生成、代码摘要与测试用例生成为目标任务, 本文比较了 CodeLLMTuner 和基线方法在成本相同的情况下所选择的模型和解码参数所对应性能的差距.

- 代码生成任务

本实验选择 HumanEval 作为 Benchmark, 为测试 CodeLLMTuner 所选择的模型与参数在未经调优任务上的泛化能力, 本实验从 HumanEval 中随机选择 70% 的待测任务组成 HumanEval-70%, 其余 30% 组成 HumanEval-30%. 本文在 HumanEval-70% 上在一定成本下使用各基线方法进行代码大模型选择与解码参数调优并记录所选择的最优模型与参数. 然后比较各基线方法所选择的模型与参数在 HumanEval-70% 上的性能, 用以衡量各基线方法进行代码大模型选择与解码参数调优时的能力; 同时比较上述模型与参数在 HumanEval-30% 上的性能, 用以衡量各基线方法在特定 Benchmark 上选择的模型与参数对其他任务的泛化能力. 考虑到即使总成本相同, 部分基线方法的性能与成本分配策略有一定关系, 如迁移学习中源环境的不同选择会影响该方法的性能, 本文为此类基线方法遍历不同的成本分配策略并计算其性能期望以全面表征此类方法的性能. 同时, 为全面比较 CodeLLMTuner 和基线方法在不同总成本下的性能差距, 本文在不同总成本场景下进行上述实验. 实验结果如图 7 所示, 其中横轴表示总成本大小, 以采样次数进行衡量, 纵轴表示所选择的模型与参数在 Benchmark 上的 *pass@k* (k=1、4、10). 由于参数调优结果均有随机性, 在不同的运行过程中可能会产生不同的结果, 本实验重复 3 次并记录平均性能以避免实验中的随机性误差.

随着 k 的增加, 各方法生成代码的成功率也有所增加; 随着成本的增加, 各基线方法在 HumanEval-70% 与 HumanEval-30% 上所确定的模型与解码参数对应的性能均有提升, 这表示 HumanEval-70% 与 HumanEval-30% 的性能模型有一定相似度, 因此在 HumanEval-70% 上性能更高的模型与参数在 HumanEval-30% 上同样有较好的性能; 不管是在 HumanEval-70% 还是在 HumanEval-30%, CodeLLMTuner 均能达到最高的性能, 这不仅表示 CodeLLMTuner 相较于其他基线方法在代码大模型选择与参数调优问题上更有可能选择性能更优的模型与参数, 也表示 CodeLLMTuner 相较于其他基线方法在泛化能力上的优势: 在特定 Benchmark 上选择的模型与参数在处理其他相似任务时同样有着更好的性能.

除此以外, 为评估 3 次实验间性能波动情况, 本文还以 *pass@1* 作为性能指标, 比较各基线方法在 3 次实验中性能的方差, 如图 8 所示.

首先, DivBO 和 CodeLLMTuner 的方差最低, 其原因在于前者简单地将参数空间进行扩展, 后者能准确选择最优模型, 因此其性能均未根据实验次数的不同而改变. 其次, transfer learning 随着成本的增加, 方差明显降低, 其

原因在于该算法在成本较低时有可能选择次优模型导致性能波动, 而成本充足时能够稳定选择最优解, 致使方差降低. 最后, AutoRAG-HP 与 EcoOptiGen 由于即使成本充足依然有概率选择次优解, 因此性能波动较为明显.

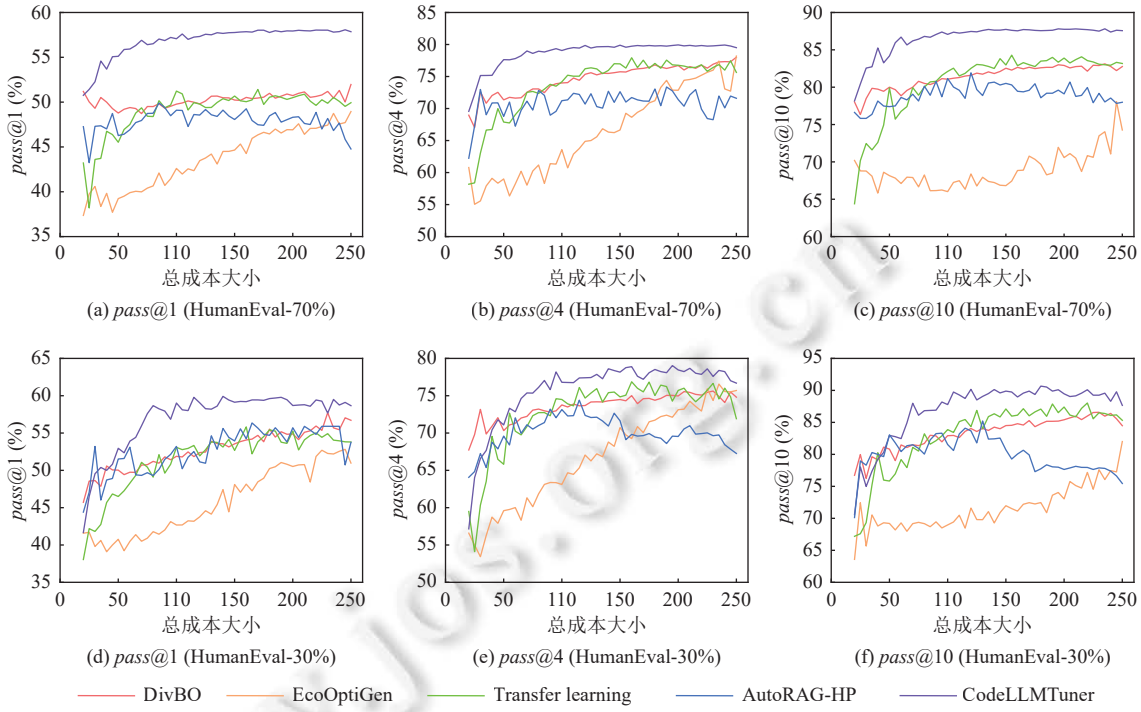


图 7 各基线方法在 HumanEval-70% 和 HumanEval-30% 上所选择的最优模型与参数性能

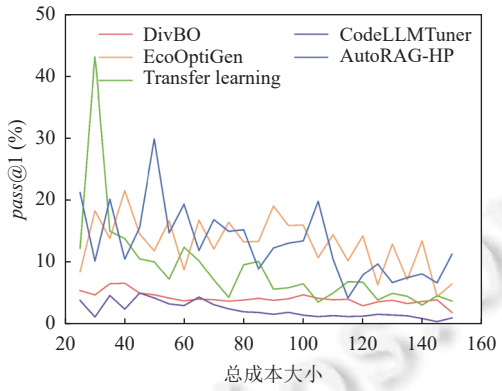


图 8 各基线方法在 HumanEval-70% 上以 *pass*@1 为指标时的方差

● 代码摘要任务

CodeXGLUE 提供了一个包括 23 007 个代码摘要任务的 Benchmark, 本文从中随机选择 1 000 项作为本实验的 Benchmark、并将其以相同方法拆分为 CodeXGLUE-70% 与 CodeXGLUE-30%, 同时采用相同方法进行测试. 其结果如图 9 所示. 其中横轴表示总成本大小, 以采样次数进行衡量, 纵轴表示所选择的模型与参数在 Benchmark 上的代码任务性能, 以 $BLEU\_i$ (i = 1, 2) 进行衡量.

随着成本增加, 各基线方法性能在 CodeXGLUE-70% 上大多有所提升; 在 CodeXGLUE-30% 上, 随着成本增加, 各方法性能均会出现波动的情况, 其原因在于 CodeXGLUE-30% 与 CodeXGLUE-70% 的性能分布有所差异, 在 CodeXGLUE-70% 上性能更优的模型与参数在 CodeXGLUE-30% 上性能可能有所下降; 不管是在 CodeXGLUE-

70% 还是在 CodeXGLUE-30%, CodeLLMTuner 均能达到最高的性能, 表示 CodeLLMTuner 相较于其他基线方法在进行代码大模型选择与参数调优时有着更高的性能与更好的泛化能力.

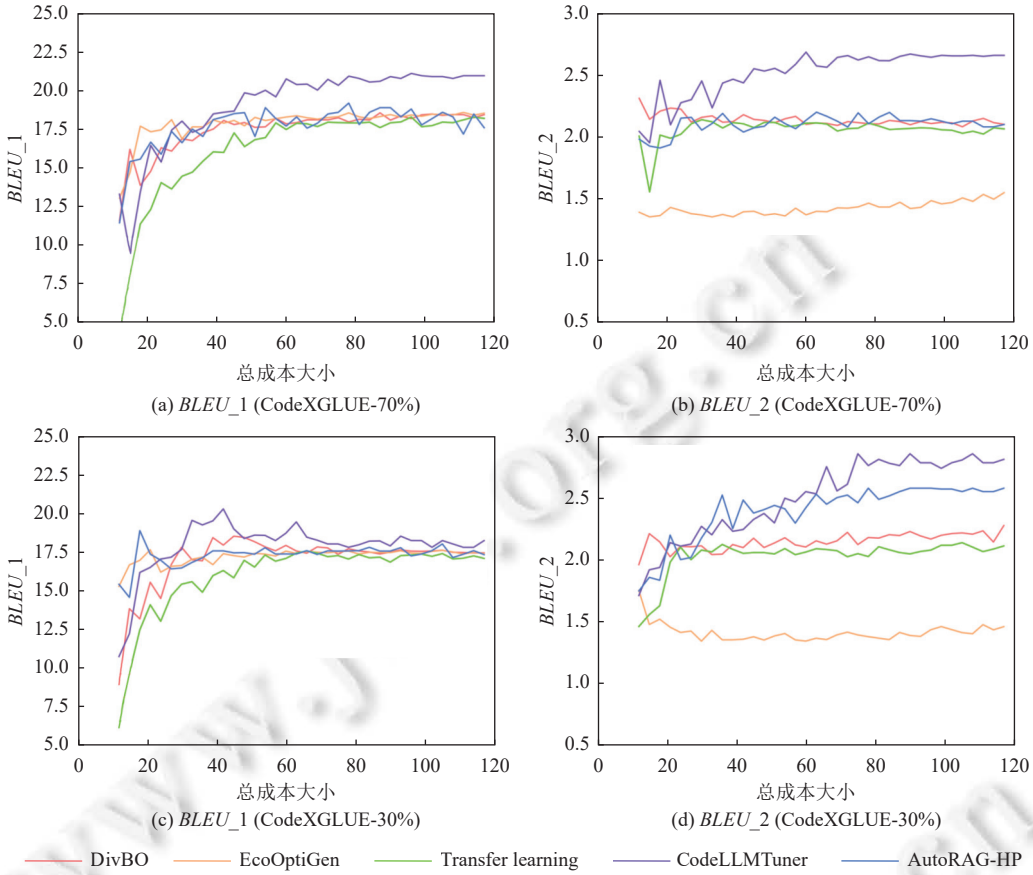


图 9 各基线方法在 CodeXGLUE-70% 和 CodeXGLUE-30% 上所选择的最优模型与参数性能

● 测试用例生成任务

本文选择 CodaMosa 所提出的针对 Flask 库的 Benchmark 上进行测试, 考虑到测试用例生成 Benchmark 中的不同代码任务之间并非独立, 需要整合所有任务的生成结果以评估整体性能, 因此本实验并未对其进行拆分. 本实验在测试用例生成 Benchmark 上使用各基线方法在一定成本下进行代码大模型选择与解码参数调优, 记录各基线方法所选择的最优模型与参数在性能空间上的性能. 本实验同样重复 3 次且遍历不同成本分配策略并计算性能期望, 实验结果如图 10 所示. 其中横轴表示总成本大小, 以采样次数进行衡量, 纵轴表示所选择的模型与参数在 Benchmark 上的代码任务性能, 以行覆盖率和分支覆盖率衡量.

随着成本增加, 各方法均会出现性能波动的情况, 其原因在于测试用例生成任务的性能受随机性影响极大, 即使使用的模型与参数相同, 代码大模型也可能会输出完全不同的结果; 然而, 不管是在行覆盖率还是分支覆盖率, CodeLLMTuner 均能达到最高的性能.

RQ2: 成本相同的情况下, CodeLLMTuner 与基线方法在组合代码任务上的性能差异有多大?

由于代码大模型选择与解码参数调优成本过高, 为每个代码任务都单独进行调优可能是不经济乃至不可行的, 因此用户会将多个代码任务整合为一个代码任务进行调优以节约成本. 本节探索 CodeLLMTuner 相较于其他基线方法在组合代码任务上进行代码大模型选择与解码参数调优的性能差距.

本文使用代码生成, 代码摘要与测试用例生成任务构建组合代码任务, 同时分别使用 *pass*@10、*BLEU*_1 与行覆盖率评估对应代码任务. 由于上述不同任务的性能指标值域范围不同, 在计算加权和之前对 *pass*@10、

*BLEU*_1 与行覆盖率进行正则化处理. 之后使用此 3 项的加权和作为组合任务的评价指标, 其公式如下, 其中 α、β、$(1-\alpha-\beta)$ 分别表示代码生成, 代码摘要与测试用例生成任务性能指标的权重.

$$Score_{\text{multi}} = \alpha \times pass@10 + \beta \times BLEU\_1 + (1-\alpha-\beta) \times Coverage_{\text{line}} \ (\alpha > 0, \beta > 0, \alpha + \beta < 1).$$

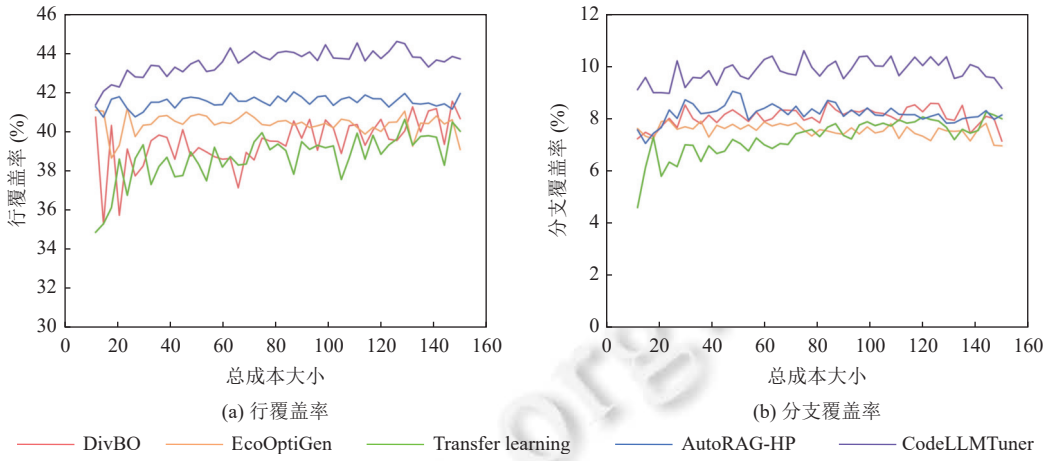


图 10 各基线方法针对测试用例生成任务在 Benchmark 上的最优模型与参数性能对比

本实验将 HumanEval-70%、CodeXGLUE-70% 与 CodaMosa 这 3 个 Benchmark 组合成 MultiTask-70%, 同时将 HumanEval-30%、CodeXGLUE-30% 与 CodaMosa 组合成 MultiTask-30%. 本实验设定以下场景以确定 α、β 组合的值: 平均组合 ($\alpha=\beta=1/3$)、关注代码生成任务 ($\alpha=0.6, \beta=0.2$) 或关注测试用例生成 ($\alpha=0.2, \beta=0.2$). 对于不同的 α、β 组合, 本实验在 MultiTask-70% 上使用各基线方法进行代码大模型选择与解码参数调优, 然后比较各基线方法所选择的模型与参数在 MultiTask-70% 上的性能, 用以衡量各基线方法进行代码大模型选择与解码参数调优时的能力; 同时比较上述模型与参数在 MultiTask-30% 上的性能, 用以衡量各基线方法在特定 Benchmark 上选择的模型与参数对其他任务的泛化能力. 本实验同样重复 3 次且遍历不同成本分配策略并计算性能期望, 实验结果如图 11 所示.

对于模型选择与参数调优性能的比较, 随着成本的增加, 大多数基线方法在 MultiTask-70% 上的性能都有所增加, 然而该增加过程存在一定波动, 其原因在于不同代码任务的性能模型分布不同, 导致组合任务的性能模型存在较多的局部最优点; 组合权重的变化会影响整个组合任务性能模型的分布, 导致各基线方法的模型选择与参数调优结果出现变化, 而 CodeLLMTuner 在各个场景下均能达到最高的参数调优性能.

对于泛化能力的比较, CodeLLMTuner 在平均组合场景下的泛化能力不强, 而在关注特定任务 (代码生成或测试用例生成) 时的泛化能力较强. 其原因在于不同任务的最优模型不同, 平均组合场景下 MultiTask-70% 与 MultiTask-30% 上的最优模型不一致, 导致 CodeLLMTuner 在 MultiTask-70% 上选择的最优模型在 MultiTask-30% 上性能表现不佳; 而在关注特定任务时, MultiTask-70% 与 MultiTask-30% 上的最优模型都是所关注任务的最优模型, 因此 CodeLLMTuner 对其的泛化能力较强.

RQ3: 为达到相同的性能, CodeLLMTuner 相较于基线方法可以节省多少成本?

代码大模型选择与解码参数调优的第 2 个目标是以尽可能低的成本为代码任务找到满足性能需求的模型和解码参数. 为了比较 CodeLLMTuner 相较于基线方法为达到相同的性能要求所消耗的成本, 本实验选择代码生成任务作为目标任务, HumanEval 作为 Benchmark, *pass*@10 作为性能指标以构建性能空间, 并使用相对性能 (*RP*) 来衡量代码大模型选择与解码参数调优过程是否满足性能要求, 其公式为:

$$RP = \frac{Y_{\max} - Y_{\text{pred}}}{Y_{\max}}.$$

该指标表征调优选择的模型与参数所对应的性能相较于真实最优性能的差距, 其中 Y_{pred} 为代码大模型选择与

解码参数调优过程中所选择的模型与解码参数对应的性能, Y_{max} 为各候选模型参数空间中的真实最优性能, 本文通过为各候选模型的参数空间遍历采样并评估 1 875 个参数配置以从中选择最优解的方式确定该数据. *RP* 值越低, 调优过程所选择的模型与解码参数性能越接近理想性能. 本文选择目标相对性能 (0.24、0.20、0.16、0.12、0.08、0.04) 作为性能要求, 比较达到目标 *RP* 值的最少采样次数作为各基线方法成本的衡量标准. 本实验为各个基线方法自低向高地分配成本以进行代码大模型选择与解码参数调优, 并基于上述公式计算各基线方法在不同成本下的相对性能, 由此记录各基线方法满足不同性能要求的最小成本, 即实验相对性能小于目标相对性能的最少的采样次数, 其结果如表 3 所示. 其中, N/A 表示在可控成本范围内该方法无法达到性能要求.

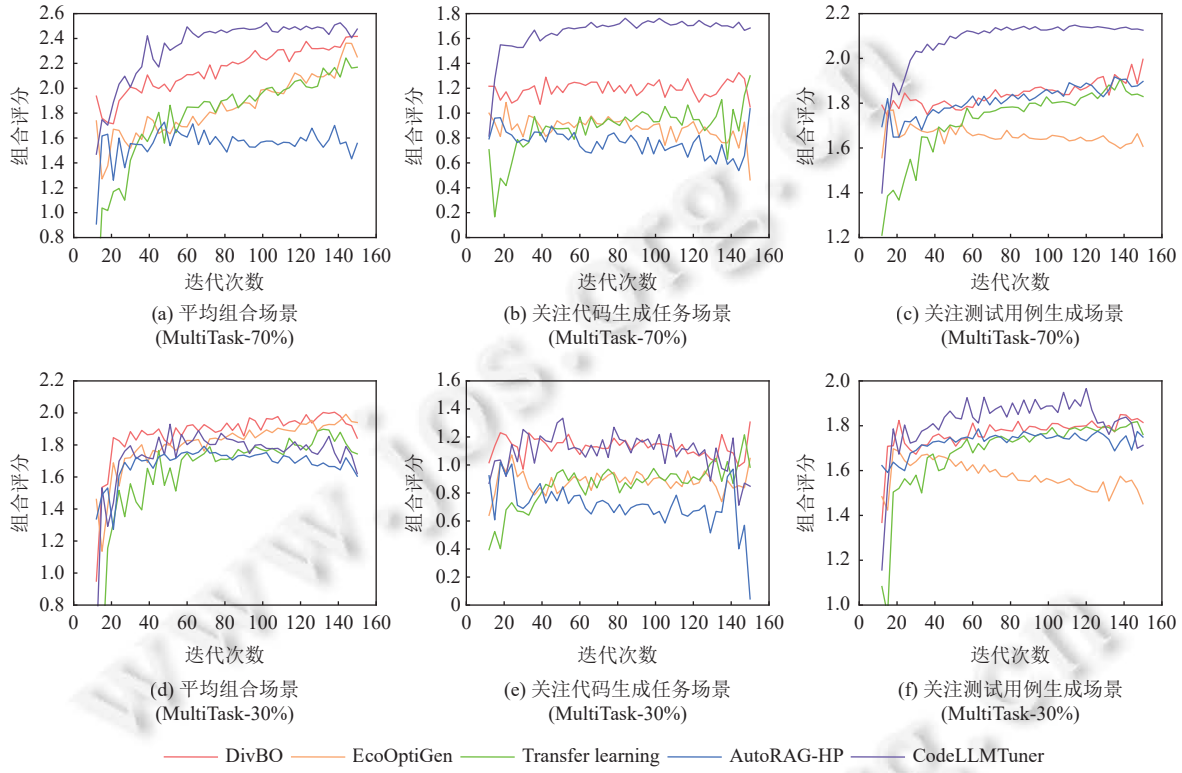


图 11　各基线方法在 MultiTask-70% 和 MultiTask-30% 上所选择的最优模型与参数性能

表 3　各基线方法为达到相同性能要求所需要的最小成本

Baseline method	Relative performance					
	0.24	0.20	0.16	0.12	0.08	0.04
CodeLLMTuner	20	20	20	25	30	55
DivBO	20	20	20	30	135	N/A
EcoOptiGen	20	195	245	N/A	N/A	N/A
AutoRAG-HP	20	20	20	240	N/A	N/A
Transfer learning	25	30	45	50	100	N/A

在不同的性能要求下, CodeLLMTuner 所消耗的成本更少, 而 EcoOptiGen 所消耗的成本最多, 其原因在于序列化方法容易陷入局部最优点导致难以实现全局最优; 在可接受的成本范围内, CodeLLMTuner 可以达到更高的性能要求, 而 DivBO 和 transfer learning 效果次之, 其原因在于 CASH 方法与迁移学习方法虽然会避免陷入局部最优, 但这两种方法会消耗更多成本.

3.4　消融实验

考虑到 CodeLLMTuner 可以与不同的成本分配方式、参数调优方法与模型选择方法相结合, 本节探讨上述因

素对 CodeLLMTuner 的性能影响.

● 不同的成本分配方式对 CodeLLMTuner 性能的影响

如前文所述, CodeLLMTuner 将采样过程分为两个阶段: 在第 1 阶段 CodeLLMTuner 迭代地在每个模型上采样 K 次以选择最优模型; 在第 2 阶段 CodeLLMTuner 在获选模型上采样 N 次以确定最优解码参数. 在总成本一定时, 无法保证两阶段均能获得足够多的成本, 即 K 与 N 呈负相关, 此时成本分配策略, 即 K 和 N 的比率, 会对 CodeLLMTuner 的性能造成重大影响, 因此本节探索 CodeLLMTuner 性能与成本分配策略之间的关系. 为全面探索成本分配策略对 CodeLLMTuner 性能的影响, 本文使用了不同的总成本进行上述实验, 其公式如下所示:

$$TotalCost = K \times |M| + N,$$

其中, $|M|$表示候选模型的数量, 该公式使用总样本数据量表征总成本. 本文选择代码生成作为目标任务, HumanEval 作为 Benchmark, *pass*@10 作为性能评价标准, 在不同的总成本下, 采用具有不同 K 和 N 比率的 CodeLLMTuner 在训练 Benchmark 中进行代码大模型选择与解码参数调优.

其结果如图 12 所示, 其中横轴表示 $\log(K/N)$, 纵轴表示 *pass*@10. 首先, 当总成本一定时, K 和 N 比率过大或过小均会降低 CodeLLMTuner 性能, 其原因在于: 比率过大时, 用于性能空间探索的成本过少, 导致性能空间探索不充分; 而比率过小时, 用于模型选择的成本过少会导致选择次优解. 其次, 随着总成本的增加, 最优性能有所提升, 而性能最优的 K/N 比值有所降低, 其原因在于: 模型选择仅需要对性能期望进行排序, 对性能预测精度的要求不高, 因此模型选择成本增加带来的性能收益不如解码参数调优. 而在进一步实验后, 本文发现不同总成本下性能最优点的 K 值相接近, 即在不同总成本下实现性能最优的分配策略用于模型选择的成本接近, 这为 CodeLLMTuner 的成本分配策略提供了指导: 在保证模型选择准确的同时, 尽可能地分配更多成本用于解码参数调优.

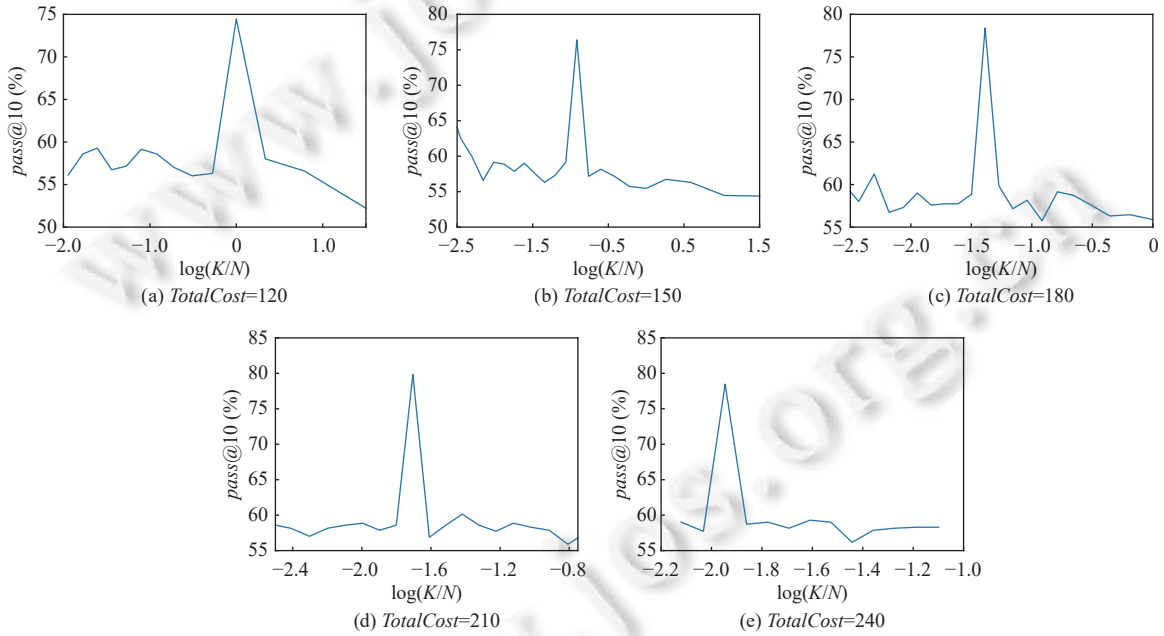


图 12 代码生成任务中不同总成本下的成本分配方式对 CodeLLMTuner 性能的影响

● 参数调优方法对 CodeLLMTuner 性能的影响

考虑到 CodeLLMTuner 可以整合不同的参数调优方法, 本节探索 CodeLLMTuner 性能与不同参数调优方法之间的关系. 以代码生成作为目标任务, HumanEval-70% 作为 Benchmark, *pass*@10 作为性能评价标准, 随机选择、线性回归和贝叶斯优化作为对比方法, 本实验使用结合了不同参数调优方法的 CodeLLMTuner 在训练性能空间中选择最佳的模型和解码参数, 并记录该模型与解码参数在 HumanEval-30% 中的性能结果, 如图 13 所示, 其中横轴表示总成本, 纵轴表示 *pass*@10.

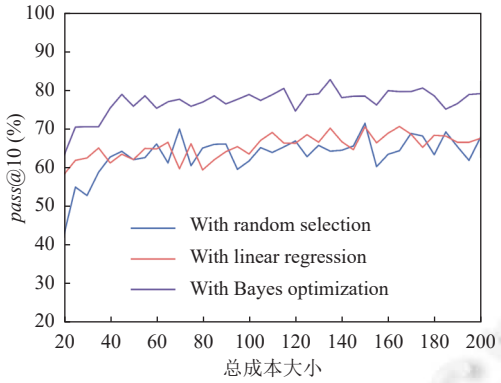


图 13　代码生成任务中参数调优方法对 CodeLLMTuner 性能的影响

由于 HumanEval-70% 与 HumanEval-30% 性能分布的差别, 随着成本的增加, 各对比方法均出现一定的性能波动; 与贝叶斯优化相结合的性能明显优于以随机选择与线性回归作为参数调优方法的性能, 其原因在于贝叶斯优化能更高效地探索配置空间.

● 模型选择方法对 CodeLLMTuner 性能的影响

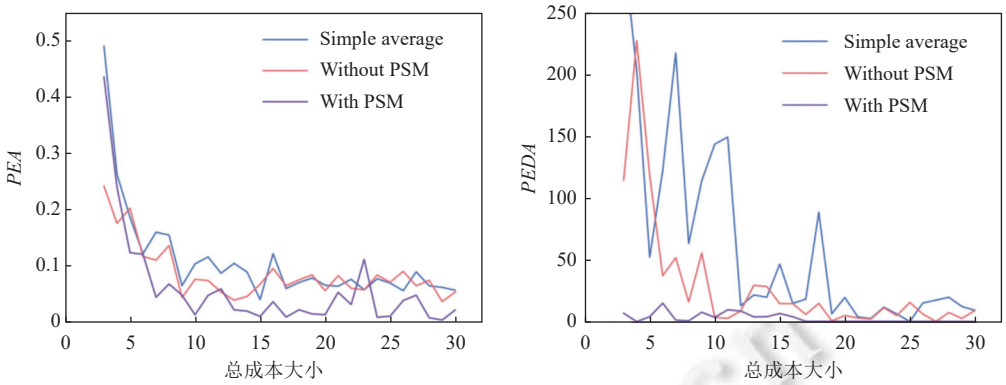
考虑到 CodeLLMTuner 通过倾向评分估计方法使用模型间不同分布样本数据计算各模型性能期望来进行模型选择, 本节对 CodeLLMTuner 进行性能期望预测的准确性进行测试. 本实验以代码生成作为目标任务, HumanEval 作为 Benchmark, *pass*@10 作为性能评价标准, 以 simple average、without PSM 和 with PSM 这 3 种方法作为对比方法: 其中 simple average 直接计算样本数据性能平均值作为性能期望; without PSM 使用样本数据直接训练神经网络, 并由此预测模型性能期望; 而 with PSM 在神经网络训练之前使用倾向评分估计对样本数据进行重加权处理. 为比较不同方法在预测各代码大模型性能期望上的准确率, 本实验从模型的参数空间中遍历采样并评估 1 875 个参数配置以此计算各模型的真实性能期望, 然后使用性能期望预测准确率 (performance expectation accuracy, *PEA*) 与性能期望差值预测准确率 (performance expectation difference accuracy, *PEDA*) 作为衡量模型选择方法准确率的指标, 其公式如下所示:

$$PEA = \frac{1}{|M|}\sum_{i=1}^{|M|}\left(Y_i^{\text{true}} - Y_i^{\text{pred}}\right)^2,$$

$$PEDA = \frac{1}{|M|!}\sum_{i\neq j}^{|M|}\left(D_{ij}^{\text{true}} - D_{ij}^{\text{pred}}\right)^2,$$

其中, $|M|$为模型总数; Y_i^{pred} 表示模型 i 的预测性能期望, D_{ij}^{pred} 表示模型 i、j 之间的预测性能期望差值, 由各基线算法计算得出; Y_i^{true} 表示模型 i 的真实性能期望, D_{ij}^{true} 表示模型 i、j 之间的真实性能期望差值, 由遍历采样样本数据计算得出. 性能期望预测准确率 (*PEA*) 表征各基线方法在预测各代码大模型性能期望上的相对误差, 该指标越小表示预测准确率越高; 而由于模型选择实际上是通过比较模型间性能差距实现的, 本实验还使用性能期望差值预测准确率 (*PEDA*) 表征各基线方法在预测不同代码大模型之间性能期望差值上的相对误差, 指标越小表示预测准确率越高. 本实验在多个模型中使用贝叶斯优化收集不同数量的样本数据, 然后使用各基线方法预测每个模型的性能期望以计算比较预测准确率, 其结果如图 14 所示, X 轴表示总成本大小, Y 轴表示 *PEA* 和 *PEDA* 大小.

首先, 随着样本数据量的增加, 各个方法的性能期望预测准确率和性能期望差值预测准确率都有提升; 其次, 在预测性能期望时, simple average 的效果最差, 而 without PSM 与 with PSM 在部分情况下效果相近, 其原因在于, CodeLLMTuner 仅是将不同分布的样本数据进行对齐, 但是并不保证对齐后的数据点满足随机分布或均匀分布, 因此在计算单一模型性能期望时难以体现性能优势. 但是在预测性能期望差值时, with PSM 相较其他方法有着明显的性能优势, 其原因在于 with PSM 通过倾向评分算法对不同模型的样本数据进行了对齐, 此时引入的其他模型的信息会提高预测的准确率, 即 CodeLLMTuner 相较其他方法更有可能预测出最佳模型.



(a) 各基线方法性能期望预测准确率对比 (b) 各基线方法性能期望差值预测准确率对比

图 14 代码生成任务中模型选择方法对模型性能期望预测准确率的影响

3.5 有效性威胁

本实验可能存在的有效性威胁包括以下几种.

1) 本文仅在单纯的大模型上进行了实验, CodeLLMTuner 的泛化能力在与 RAG、思维链等技术结合后的模型上可能有限. 对于会影响参数空间的优化方法, CodeLLMTuner 效果有限, 但该方法可以与任意不会影响参数空间的优化方法相结合.

2) 本文所使用的性能评价指标仅限于自动化评估指标 (如 *BLEU*), 当使用人工审查作为性能评估方法时, 参数调优效果可能因人工干扰而受限.

3) 尽管本文使用的数据集是广泛使用的基准评测集, 但仍可能存在数据泄露的风险, 即大模型在预训练阶段已经接触过部分数据. 这可能导致实验结果高估模型性能. 然而, 即使存在数据泄露的可能性, CodeLLMTuner 与基线模型的对比仍然具有意义, 因为所有基线方法都是在相同的候选模型上进行参数调优.

4) 实验结果的复现可能受到实验环境 (如硬件配置、软件版本等) 的影响, 导致不同环境下的性能差异.

3.6 局限性

然而, CodeLLMTuner 依然存在部分局限性, 主要包括以下几点.

1) CodeLLMTuner 采用倾向评分匹配算法筛选最优模型, 该算法的应用前提是候选模型的参数空间必须完全一致. 鉴于当前主流代码大模型 (如 DeepSeekCoder 等) 的解码参数空间具有一致性, CodeLLMTuner 的模型选择机制具备良好的泛化能力. 然而, 随着代码大模型的发展, 许多任务的 SOTA 效果往往是通过将基础大模型与 RAG、Agent 等优化技术相结合实现的, 这种结合可能导致模型参数空间发生变化. 在此情况下, CodeLLMTuner 的适用性将受到限制. 因此, 本文的研究范围主要聚焦于基础大模型选择, 暂不涉及结合上述大模型优化技术的复合场景, 未来工作中会考虑在上述结合不同大模型优化方法的场景下进行更全面的对比实验.

2) 模型选择基于期望性能, 无法保证找到最优解. CodeLLMTuner 选择性能期望最优的模型作为最终候选, 但并不能确保该模型的最优参数在所有情况下都优于其他模型. 尽管本文在多个代码任务和数据集上进行了实验, 并取得了良好的效果, 但在某些性能分布特殊的任务中, 最高期望性能的模型可能无法达到其他模型的最优性能, 导致选择出的模型未必是全局最优的.

3) 采用机器学习方法训练的性能模型存在误差. 本文使用倾向评分匹配方法对齐不同分布的样本数据, 并以此构建性能预测器, 用于预测各模型性能期望以选择最优模型, 然而性能模型可能难以准确预测模型性能, 导致选择次优解.

4) 未来强大的通用代码大模型有可能不需要进行模型选择. 目前的实验结果表明, 没有任何单一代码大模型能够在所有代码任务上全面胜出, 因此基于任务特性的模型选择是必要的. 然而, 如果未来出现一个在所有代码任务上均表现卓越的通用代码大模型, CodeLLMTuner 的模型选择策略可能会失去其实际价值.

4 总 结

针对代码大模型选择与解码参数调优问题, 现有组合方法在模型选择与参数调优两个阶段所采样的数据无法重用, 导致两个阶段无法共享样本数据, 计算成本过大. 因此, 本文提出了一个基于样本重用的代码大模型选择与解码参数调优框架 CodeLLMTuner, 该框架利用性能期望选择最优模型, 同时采用倾向评分匹配方法实现样本重用以降低采样成本. 本文在不同类型代码任务上对参与实验的大模型本身的性能表现都进行了实验, 作为本文的基线方法效果. 实验结果表明, 针对代码生成、代码摘要和测试用例生成这 3 项任务, CodeLLMTuner 相比基线方法能够在相同成本下提高 10%–15% 的性能, 或在相同性能下节省超过 20% 的成本, 证明了本文所提出方法的有效性.

References

[1] Chen M, Tworek J, Jun H, *et al*. Evaluating large language models trained on code. arXiv:2107.03374, 2021.

[2] Li R, Allal LB, Zi YT, *et al*. StarCoder: May the source be with you! arXiv:2305.06161, 2023.

[3] Roziere B, Gehring J, Gloeckle F, *et al*. Code Llama: Open foundation models for code. arXiv:2308.12950, 2023.

[4] GitHub Copilot. Your AI pair programmer. 2025. https://github.com/features/copilot

[5] Yang ZZ, Chen SR, Gao CY, Li ZH, Li G, Lü RC. Deep learning based code generation methods: Literature review. Ruan Jian Xue Bao/Journal of Software, 2024, 35(2): 604–628 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6981.htm [doi: 10.13328/j.cnki.jos.006981]

[6] Wang Y, Wang WS, Joty S, Hoi SCH. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In: Proc. of the 2021 Conf. on Empirical Methods in Natural Language Processing. Punta Cana: ACL, 2021. 8696–8708. [doi: 10.18653/v1/2021.emnlp-main.685]

[7] Luo ZY, Xu C, Zhao P, Sun QF, Geng XB, Hu WX, Tao CY, Ma J, Lin QW, Jiang DX. WizardCoder: Empowering code large language models with evol-instruct. In: Proc. of the 12th Int'l Conf. on Learning Representations. Vienna: OpenReview.net, 2024.

[8] Guo DY, Xu CW, Duan N, Yin J, McAuley J. LongCoder: A long-range pre-trained language model for code completion. In: Proc. of the 40th Int'l Conf. on Machine Learning. Honolulu: ACM, 2023. 486.

[9] Wu QY, Bansal G, Zhang JY, Wu YR, Zhang SK, Zhu EK, Li BB, Jiang L, Zhang XY, Wang C. AutoGen: Enabling next-gen LLM applications via multi-agent conversation framework. arXiv:2308.08155, 2023.

[10] Jiang N, Lutellier T, Tan L. CURE: Code-aware neural machine translation for automatic program repair. In: Proc. of the 43rd IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Madrid: IEEE, 2021. 1161–1173. [doi: 10.1109/ICSE43902.2021.00107]

[11] Olausson TX, Inala JP, Wang CL, Gao JF, Solar-Lezama A. Is self-repair a silver bullet for code generation? In: Proc. of the 12th Int'l Conf. on Learning Representations. Vienna: OpenReview.net, 2024.

[12] Li HN, Hao Y, Zhai YZ, Qian ZY. Enhancing static analysis for practical bug detection: An LLM-integrated approach. Proc. of the ACM on Programming Languages, 2024, 8(OOPSLA1): 474–499. [doi: 10.1145/3649828]

[13] Phan L, Tran H, Le D, Nguyen H, Anibal JT, Peltekian A, Ye YF. CoTexT: Multi-task learning with code-text Transformer. arXiv: 2105.08645, 2021.

[14] van Aken D, Pavlo A, Gordon GJ, Zhang BH. Automatic database management system tuning through large-scale machine learning. In: Proc. of the 2017 ACM Int'l Conf. on Management of Data. Chicago: ACM, 2017. 1009–1024. [doi: 10.1145/3035918.3064029]

[15] Wang C, Liu XQ, Awadallah AH. Cost-effective hyperparameter optimization for large language model generation inference. In: Proc. of the 2023 Int'l Conf. on Automated Machine Learning. Potsdam: PMLR, 2023. 21.

[16] Wang SY, Li SM, Sun TX, Fu JL, Cheng QY, Ye JS, Ye JJ, Qiu XP, Huang XJ. LLM can achieve self-regulation via hyperparameter aware generation. In: Proc. of the 2024 Findings of the Association for Computational Linguistics. Bangkok: ACL, 2024. 6632–6646. [doi: 10.18653/v1/2024.findings-acl.396]

[17] Fu J, Qin XT, Yang FK, Wang L, Zhang J, Lin QW, Chen YB, Zhang DM, Rajmohan S, Zhang Q. AutoRAG-HP: Automatic online hyper-parameter tuning for retrieval-augmented generation. In: Proc. of the 2024 Findings of the Association for Computational Linguistics. Miami: ACL, 2024. 3875–3891. [doi: 10.18653/v1/2024.findings-emnlp.223]

[18] Guo QY, Wang R, Guo JL, Li B, Song KT, Tan X, Liu GQ, Bian J, Yang YJ. Connecting large language models with evolutionary algorithms yields powerful prompt optimizers. In: Proc. of the 12th Int'l Conf. on Learning Representations. Vienna: OpenReview.net, 2024.

[19] Wang Z, Panda R, Karlinsky L, Feris R, Sun H, Kim Y. Multitask prompt tuning enables parameter-efficient transfer learning. In: Proc. of the 11th Int'l Conf. on Learning Representations. Kigali: OpenReview.net, 2023.

[20] Wang YD, Yu ZH, Yao WJ, Zeng ZR, Yang LY, Wang CX, Chen H, Jiang CY, Xie R, Wang JD, Xie X, Ye W, Zhang SK, Zhang Y. PandaLM: An automatic evaluation benchmark for LLM instruction tuning optimization. In: Proc. of the 12th Int'l Conf. on Learning Representations. Vienna: OpenReview.net, 2024.

[21] Tribes C, Benarroch-Lelong S, Lu P, Kobyzev I. Hyperparameter optimization for large language model instruction-tuning. arXiv: 2312.00949, 2023.

[22] Thornton C, Hutter F, Hoos HH, Leyton-Brown K. Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms. In: Proc. of the 19th ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining. Chicago: ACM, 2013. 847–855. [doi: 10.1145/2487575.2487629]
[23] Kotthoff L, Thornton C, Hoos HH, Hutter F, Leyton-Brown K. Auto-WEKA 2.0: Automatic model selection and hyperparameter optimization in WEKA. The Journal of Machine Learning Research, 2017, 18(1): 826–830.
[24] Li Y, Jiang J, Gao J, Shao Y, Zhang C, Cui B. Efficient automatic CASH via rising bandits. In: Proc. of the 34th AAAI Conf. on Artificial Intelligence. New York: AAAI Press, 2020. 4763–4771. [doi: 10.1609/aaai.v34i04.5910]
[25] Shen Y, Lu YP, Li Y, Tu YF, Zhang WT, Cui B. DivBO: Diversity-aware CASH for ensemble learning. In: Proc. of the 36th Int'l Conf. on Neural Information Processing Systems. New Orleans: ACM, 2022. 2958–2971.
[26] Wang CN, Wang HZ, Mu TY, Li JZ, Gao H. Auto-model: Utilizing research papers and HPO techniques to deal with the CASH problem. In: Proc. of the 36th IEEE Int'l Conf. on Data Engineering (ICDE). Dallas: IEEE, 2020. 1906–1909. [doi: 10.1109/ICDE48307.2020.00200]
[27] Zhang XY, Wu H, Chang Z, Jin SW, Tan J, Li FF, Zhang TY, Cui B. ResTune: Resource oriented tuning boosted by meta-learning for cloud databases. In: Proc. of the 2021 Int'l Conf. on Management of Data. ACM, 2021. 2102–2114. [doi: 10.1145/3448016.3457291]
[28] Cheng DZ, Rao J, Guo YF, Jiang CJ, Zhou XB. Improving performance of heterogeneous mapreduce clusters with adaptive task tuning. IEEE Trans. on Parallel and Distributed Systems, 2017, 28(3): 774–786. [doi: 10.1109/TPDS.2016.2594765]
[29] Bodin B, Nardi L, Zia MZ, Wagstaff H, Shenoy GS, Emani M, Mawer J, Kotselidis C, Nisbet A, Lujan M, Franke B, Kelly PHJ, O'Boyle M. Integrating algorithmic parameters into benchmarking and design space exploration in 3D scene understanding. In: Proc. of the 2016 Int'l Conf. on Parallel Architecture and Compilation Techniques. Haifa: IEEE, 2016. 57–69. [doi: 10.1145/2967938.2967963]
[30] Jamshidi P, Velez M, Kästner C, Siegmund N. Learning to sample: Exploiting similarities across environments to learn performance models for configurable systems. In: Proc. of the 26th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Lake Buena Vista: ACM, 2018. 71–82. [doi: 10.1145/3236024.3236074]
[31] Tighineanu P, Skubch K, Baireuther P, Reiss A, Berkenkamp F, Vinogradska J. Transfer learning with Gaussian processes for Bayesian optimization. In: Proc. of the 25th Int'l Conf. on Artificial Intelligence and Statistics. PMLR, 2022. 6152–6181.
[32] Akaike H. Fitting autoregreesive models for prediction. In: Parzen E, Tanabe K, Kitagawa G, eds. Selected Papers of Hirotugu Akaike. New York: Springer, 1998. 131–135. [doi: 10.1007/978-1-4612-1694-0_10]
[33] Schwarz G. Estimating the dimension of a model. The Annals of Statistics, 1978, 6(2): 461–464.
[34] Ding J, Tarokh V, Yang YH. Bridging AIC and BIC: A new criterion for autoregression. IEEE Trans. on Information Theory, 2018, 64(6): 4024–4043. [doi: 10.1109/TIT.2017.2717599]
[35] Rosenbaum PR, Rubin DB. The central role of the propensity score in observational studies for causal effects. Biometrika, 1983, 70(1): 41–55. [doi: 10.1093/biomet/70.1.41]
[36] Phind. Phind/Phind-CodeLlama-34B-v2. 2023. https://huggingface.co/Phind/Phind-CodeLlama-34B-v2
[37] Holtzman A, Buys J, Du L, Forbes M, Choi Y. The curious case of neural text degeneration. In: Proc. of the 8th Int'l Conf. on Learning Representations. Addis Ababa: OpenReview.net, 2020.
[38] Fan A, Lewis M, Dauphin Y. Hierarchical neural story generation. In: Proc. of the 56th Annual Meeting of the Association for Computational Linguistics. Melbourne: ACL, 2018. 889–898. [doi: 10.18653/v1/P18-1082]
[39] Together AI. The AI acceleration cloud—Fast inference, fine-tuning & training. 2025. https://www.together.ai/
[40] Lu S, Guo DY, Ren S, Huang JJ, Svyatkovskiy A, Blanco A, Clement C, Drain D, Jiang DX, Tang DY, Li G, Zhou LD, Shou LJ, Zhou L, Tufano M, Gong M, Zhou M, Duan N, Sundaresan N, Deng SK, Fu SY, Liu SJ. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. arXiv:2102.04664, 2021.
[41] Lemieux C, Inala JP, Lahiri SK, Sen S. CodaMosa: Escaping coverage plateaus in test generation with pre-trained large language models. In: Proc. of the 45th IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Melbourne: IEEE, 2023. 919–931. [doi: 10.1109/ICSE48619.2023.00085]

附中文参考文献

[5] 杨泽洲, 陈思榕, 高翠芸, 李振昊, 李戈, 吕荣聪. 基于深度学习的代码生成方法研究进展. 软件学报, 2024, 35(2): 604–628. http://www.jos.org.cn/1000-9825/6981.htm [doi: 10.13328/j.cnki.jos.006981]

作者简介

曲慕子, 博士生, CCF 学生会员, 主要研究领域为软件工程.
亢良伊, 博士, 助理研究员, CCF 专业会员, 主要研究领域为智能软件工程, 自然语言处理应用.
刘杰, 博士, 研究员, 博士生导师, CCF 专业会员, 主要研究领域为软件工程, 系统软件, 机器学习.
王帅, 博士, 高级工程师, 主要研究领域为大数据分析处理, 人工智能, 系统集成.
叶丹, 博士, 研究员, 博士生导师, 主要研究领域为网络分布式系统, 软件工程.
黄涛, 博士, 研究员, 博士生导师, CCF 高级会员, 主要研究领域为网络分布式计算, 软件工程.