人工智能基础大作业报告

陈凯丰[1] 队员姓名 2[2] 队员姓名 3[2] 队员姓名 4[3] 队员姓名 5[3]

Abstract

很短的项目摘要

1. 主题

项目的主题和需求来自朱汶宣同学安装 Jupiter Notebook 的经历，由于一些依赖和 WSL 的问题，他希望能够有一款轻量级的产品能够帮助他定制化地精细分析运行命令过程中的问题。

随后我们将项目主题定为了 LLM 集成的轻量级终端。

1.1. 需求分析

为了帮助用户解决在命令行下的问题，首先应当监控命令行的输出，然后在接到用户指令之后将这些信息处理并调用 LLM 获得建议，最后将建议呈现在用户界面上，供用户选择采用。

我们决定将开发目标平台定在 windows 上，因为小组几位同学使用的都是 windows 电脑，并且 windows 原生的命令行生态并不成熟，相对于 Linux 和 Mac 缺少统一的管理器，遇到的环境和配置问题可能更多。

1.2. 技术选型

为了能够在短时间内进行开发，我们小组决定使用 python 进行开发，优先采用已有的开源库来提高开发效率。

在查阅资料之后，我们决定使用 pywinpty 库来实现 windows 下和终端的交互，这个封装了系统调用，维护了一个伪终端（Pseudo Terminal）对象，使得程序可以和运行当中的命令行进行交互。

对于和 LLM 交互的部分，我们决定使用较为成熟的 openai 库来实现和远程 API 的交互。

由于实现一个 GUI 过于复杂，需要实现的终端渲染内容过多（如虚拟控制字符等），我们决定实现一个 CLI，直接利用 windows 已经实现好的终端应用（如 windows terminal 等）。

1.2.1. 总体架构

项目总体包括如下模块：

- 模拟终端模块：封装和 shell 的交互。
- CLI 模块：实现用户交互，维护命令行历史内容。
- 记忆模块：实现和 LLM 交互的短期和长期记忆。
- LLM 模块：封装和 LLM 的交互。
- agent 模块：总结和处理信息，将上下文、记忆和问题交给 LLM 模块。
- 部署模块：针对部署项目（尤其 git 仓库）要求生成特别的部署计划。
- 安全模块：检测命令安全性。
- utils 模块：工具模块。

【这里需要插入一个图】

1.2.2. 模块简介

CLI 模块是整个程序的入口以及交互界面，其中维护了一个内建的模拟终端对象和 Agent 对象，CLI 负责捕获用户的输出，判断是否是询问指令，并且将输入交给模拟终端或 Agent，同时异步地监听终端的输出。

utils 模块实现了一些独立的命令行工具和功能，如

行内刷新输出等。

模拟终端模块实现了一个模拟终端类，在设定启动命令后异步地读取内建终端的输出，将输出异步地将输出回传给 CLI。

LLM 模块，封装了获取 API Key 已经和远端 API 的调用，以及从 LLM 中获取建议的 prompt engineering。

记忆模块（memory）封装了更新短，中和长期记忆的过程，以及总结生成新记忆的 prompt 工程。

安全模块（security）封装了对命令安全性的两步检验。

部署建议模块（deploy）封装了对于 github 仓库给用户提供部署建议的方法。

Agent 模块统筹规划了上述各个模块的工作，并在得到用户输入后进行调用。

上述中重要的模块将在实现细节中展开。

1.3. 实现细节

1.3.1. 模拟终端和 CLI

pywinpty 的伪终端支持向 stdin 写、从 stdout 阻塞地读。

windows 下的 shell（如 powershell）本身维护了一个缓冲区，接受用户输入的字符，包括可显示字符和不可见的控制字符，如 Ctrl-C、Backspace、左键等，并且实时向 stdout 输出带有控制台虚拟终端序列的 ANSI 字节流。

windows 下的终端，如 conhost、windows terminal、git bash 等均实现了控制台虚拟终端序列的控制和渲染，事实上每输入一个字符，windows 下的终端软件会接收到 shell 输出的 ANSI 字节流，要求软件重新渲染最后一行，以此达到动态输入的目的。因此直接捕获字符写入伪终端这可能导致大量重复的输出，为了解析这些输出需要较大的工作量。

为了解决这个问题，我们决定牺牲部分终端的便捷性，采用一次输入一行的方法，如此输入不会重复出现，虚拟终端序列的去除也较为方便，同时还能够直接利用 windows 下终端对 ANSI 字节流的支持。

因此启动 CLI 的主进程就是一个循环，阻塞地读取用户一行的输入，并且将其经过处理交给 Agent 或写入内建伪终端。

由于 shell 的输出时间并不规律，从内建伪终端的读取必须是异步的，模拟终端类中新建了一个进程进行读取和传回。在 CLI 中，一个回调函数被传递给模拟终端类，在读取到输入后将新输入的内容加到历史队列中并输出到用户界面上，其中数据结构的锁都由 python 内置库管理。

在用户输入调用 Agent 时，CLI 会将目前所有的历史记录和目录等信息传递给 Agent，Agent 经过处理以及和 LLM 的交互后返回一个命令列表，用户确认后在模拟终端中执行。

1.3.2. Agent 实现

Agent 是负责连接表面的和用户交互的 CLI，以及内部的需要利用语言模型的各个模块的枢纽。

在接受到普通命令时，CLI 会将命令发送给 Agent，Agent 将会将命令发送给记忆模块，更新记忆。

在接收到问询命令（即?? 开头的命令），CLI 会将该命令发送给 Agent，Agent 将调取记忆模块的所有记忆，并结合获取的环境信息，发送给 LLM 的 Handler，之后得到一个命令建议的列表，或者想要重新询问用户的请求，呈现给用户。若是重新询问用户的请求，则将不断重复上述的操作，直到用户选择退出，或者从 Handler 得到命令建议的列表。询问用户想要采取哪个方案后，Agent 会将该命令发送给安全模块，由安全模块做执行前的最后一次确认。确认完毕后，Agent 会将这个命令发送给 CLI 进行执行，并发送给记忆模块更新记忆。

在接收到部署建议（即 deploy 开头的命令），CLI 会将该命令发送给 Agent，Agent 会将该命令转发给部署建议模块，获取建议后呈现给用户。

1.3.3. LLM 接口处理

LLM_core 中含有 API 密钥管理，LLM 调用函数以及流输出的函数。这几个函数都是平凡的。

API 密钥管理负责接收用户的 API，将其加密存储在文件系统中后，并在需要的时候返回。

LLM 调用则负责给定 prompt，向 LLM 获取回答。

流式输出则负责向命令行流式输出 LLM 的回答。

1.3.4. Handler 与 Prompt 工程

Handler 负责整理两个内容：整理各种信息并向 LLM 提问，以及根据格式解析 LLM 的返回结果。

Prompt 首先给出记忆与环境信息，然后给出用户的需求以及过往用户对 LLM 所提问题的回答，然后再向 LLM 描述任务以及格式要求。

由于我们需要 LLM 学会向用户提问，所以 Prompt 工程中强调了何时应当向用户提问，而何时应当用已知的信息直接给出回答："... 如果你没有收到请求，或者觉得用户用词模糊，或不足够清楚自己要做什么任务，或者在执行中遇到困难，或者遇到了未能处理的问题。请先返回一个'0'，然后返回一个疑问句，即你的询问，然后不再给出任何内容！询问后也不用添加 ʊ！否则先返回一个'1'，然后给我提供若干可供接下来执行的选项。"

一个细节是，由于命令的特殊性（包含大量如反斜杠等字符），经过了多次测试，为了避免生成信息内容对解析的干扰，我们采用了以特殊字符'ʊ' 作为分割符号来进行 Prompt 工程："... 选项不用太多，要求内容精简直接，数量最多五个。每一个选项的格式是三元组：该选项对应的可供执行的 Powershell cmd 命令（一定是能在终端执行的命令！）ʊ 该选项的说明 ʊ 该选项的注意事项 ʊ。并且三元组都要给出内容，且以 ʊ 分隔，注意每一项后面都要加 ʊ。"。采用此方案后，没有遇到过解析出错的情况。

在出现问题的情况，将抛出错误。用户再次输入"??"即可重试。

1.3.5. 安全检查

当 Agent 决定直接执行某个语言模型获取的命令建议的时候，这个命令会发送给安全检查。安全检查模块负责二次判断语言模型给出的命令是否有潜在的安全隐患。

安全检查采用两步的检查方案。第一步采用直接进行关键词检索，若命令不含有可能危险的关键词（如 rm 等），则直接通过安全检测。

未能通过第一步安全检查的命令，将询问语言模型该命令的安全性。若安全，则直接通过安全检测，否则将同时返回一个警告，告知用户该命令可能会产生什么样的潜在安全隐患。

1.3.6. 历史记忆与用户画像

记忆模块维护了用户的短期记忆（近期的若干条命令），中期记忆（当前命令行的操作历史总结）与长期记忆（用户画像）。

每次用户进行命令的输入，Agent 都将会把命令发送给记忆模块，记忆模块将根据该命令更新三个记忆。

短期记忆将记录用户最近的若干条命令。

中期记忆：而对于提前设定的参数 SHORT_HISTORY（默认为 10），每隔这么多条消息之后，就将通过语言模型生成截至到目前为止的所有命令的总结，以节省记忆大小。

长期记忆：当用户关闭客户端，将调用记忆模块的长期记忆更新操作。记忆模块会利用语言模型，将之前的用户画像和此次命令行的中短期记忆进行总结，得到一份新的用户画像。用户画像包括了用户在命令行的行为习惯与偏好，熟悉程度，以及之前所在命令行做过的一些重要操作。

在需要的时候，记忆模块会将所有的记忆整合后发送给 Agent。

1.3.7. 部署建议模块

部署模块的 deploy 功能分为两种。当 deploy 的内容并非 github 仓库，则直接尝试通过问询语言模型，用语言模型本身的知识库获取部署的方案。

否则，若接收的是一个 github 仓库的 url，通过访问该 url 拉取仓库中的 readme 后。随后，函数将结合用户记忆以及环境信息，向语言模型索取部署建议。

语言模型将分析该 readme 中是否含有部署信息，或者对于较为知名的项目，动用本身的知识给出部署的建议方案。由于多数情况下 readme 中含有的信息不足以直接给出若干个命令，故仅返回较为具体的部署方式，由用户自己进行执行。

部署建议为一个有序列表，返回给 Agent。

1.3.8. 模型选择

考虑在不同的使用场景，对语言模型的精度，速度，推理能力等有不同的需求，故使用了不同的语言模型。

命令建议：获取建议的命令操作时，使用 DeepSeek-R1。这是因为该步骤对输出格式的要求较高且较复杂，并且命令可能较为复杂，需要足够准确。所提供的信息（如过往的记忆，用户的想法等）较多，需要一定的推理能力。

安全模块：在安全模块的第二次检验中，使用 Qwen2.5-7B-Instruct。这是因为判断命令是否足够安全的任务简单，所输出的格式要求简单，在这一步采用更小的模型，减少用户在已知应该使用什么命令的情况下的等待时间。

记忆模块：在记忆模块的所有语言模型均使用 Qwen2.5-7B-Instruct。这是因为中长期记忆并不需要完全准确细致地进行总结，只需记录下用户做了些什么。并且当 SHORT_HISTORY 较小时，中期记忆可能会造成平凡的“卡顿”现象，使用更快的语言模型有助于缓解卡顿时间。

部署模块：部署模块所使用的语言模型使用 DeepSeek-R1。这是因为 README 文件可能较长，需要较好的文本理解与推理能力才能找到有用信息。

1.3.9. 参数调优

1.4. 评估对比

1.5. 反思

1.5.1. 项目要求对照

在与项目要求进行检查的过程中，由于该项目的特殊性，有两个基础要求我们认为并不适用于此项目：

- ”实现核心功能闭环，需定义明确的输入输出规范并实现校验机制”。由于命令行的特殊性，不符合输入规范的命令将直接由命令行返回错误信息，这也符合用户在日常使用中的习惯。在已有足够完善的命令行自带的规范的情况下，我们认为额外引入除了”??” 与”deploy” 之外的其他特殊的规范不会起到积极的作用。
- ”实现输入预处理（敏感词过滤/指令注入防护）”。除去对用户输入命令的解析，我们并未做敏感词过滤等等其他的非平凡的预处理。这是因为我们希望至少用户能将其作为一个正常的命令行使用，并且用语言模型减少其使用门槛，而并非通过各种方式减少用户命令的自由度。

参考文献

Langley, P. Crafting papers on machine learning. In Langley, P. (ed.), Proceedings of the 17th International Conference on Machine Learning (ICML 2000), pp. 1207–1216, Stanford, CA, 2000. Morgan Kaufmann.

A. 附录

可以将一些额外的内容放在这里