信息科学技术学院

《大模型的代码缺陷检测能力分析报告》

学生姓名 魏鹏超

学生学号 2208010423

计算机科学与技术 **专业** 224 **班**

2025 **年** 03 **月** 18 **日**

摘要

本次测试旨在评估 GPT-4o、DeepSeek-V3 和**文心 X1** 等常见大模型在代码缺陷检测任务中的表现，重点关注它们在不同难易程度和代码量下的检测能力，包括语法错误和逻辑错误等方面。
本次选取了以下三种测试模型：

- **GPT-4o**: OpenAI 的大模型，以强大的自然语言理解和代码生成能力著称，本次测试不采用**深度思考**。
- **DeepSeek-V3**: 专注于代码分析和优化的模型，擅长处理复杂代码逻辑，本次测试不采用**深度思考**。
- **文心 X1**: 百度推出的**深度思考**多模态大模型，具备较强的中文理解和代码处理能力。

每个模型测试 3 段不同难易程度和代码量的代码片段：

- **简单代码**: 10 行左右的代码，包含明显的语法错误或逻辑错误。
- **中等代码**: 50 行左右的代码，包含较为复杂的逻辑错误或潜在的性能问题。
- **复杂代码**: 100 行以上的代码，包含多个模块、依赖关系，以及隐藏的逻辑错误。

对于每个代码片段，我们预设一定数量的代码缺陷，根据大模型的回答结果对大模型进行打分，最终得到模型能力对比图。

关键词: GPT-4o、DeepSeek-V3、文心 X1、软件测试、代码缺陷检测、语法错误、逻辑错误

目录

大模型的代码缺陷检测能力分析报告

1 引言

随着人工智能技术的快速发展，大模型在代码生成、代码分析和缺陷检测等领域的应用日益广泛。OpenAI 的 GPT-4o、DeepSeek-V3 以及百度的文心 X1 等大模型，凭借其强大的自然语言处理和代码理解能力，正在成为开发者辅助工具的重要组成部分。然而，不同模型在代码缺陷检测任务中的表现存在显著差异，尤其是在处理不同难易程度和代码量的任务时，其能力边界尚未被充分探索。

本次测试旨在评估 **GPT-4o、DeepSeek-V3** 和**文心 X1** 在代码缺陷检测任务中的表现，重点关注以下两个方面：

- **语法错误检测**：模型能否快速识别代码中的语法错误并提供修复建议。
- **逻辑错误检测**：模型能否发现代码中的逻辑缺陷，尤其是在复杂逻辑场景下的表现。

2 测试方法

2.1 测试代码设计

为了全面评估模型的缺陷检测能力，测试代码分为三个不同难易程度和代码量的类别，并且我们设置好对应的**预期缺陷**。

1. **简单代码**：计算两个数的和并输出结果。

```cpp
#include <iostream>
using namespace std;

int main() {
    int a = 5
    int b;
    int sum = a + b;
    cout << "The sum is: " << sum << endl;
    return 0;
}
```

预期缺陷：

- 第 5 行：缺少分号（int a = 5 后面缺少分号）。
- 第 6 行：变量 b 未初始化

2. **中等代码**：实现一个简单的学生成绩管理系统，包含添加成绩和计算平均分的功能。

```cpp
#include <iostream>
#include <vector>
using namespace std;

class Student {
public:
    string name;
    double score;

    Student(string n, double s) : name(n), score(s) {}
};

class GradeManager {
private:
    vector<Student*> students;

public:
    void addStudent(string name, double score) {
        students.push_back(new Student(name, score));
    }

    double calculateAverage() {
        double total = 0.0;
        for (int i = 0; i <= students.size(); i++) {  // 逻辑错误：循环条件应为 i < students.size()
            total += students[i]->score;
        }
        return total / students.size();
    }

    ~GradeManager() {
        // 潜在性能问题：未释放动态分配的内存
    }
};

int main() {
    GradeManager manager;
    manager.addStudent("Alice", 90.5);
    manager.addStudent("Bob", 85.0);
    manager.addStudent("Charlie", 92.0);

    cout << "Average score: " << manager.calculateAverage() << endl;
    return 0;
}
```

预期缺陷:

- 第 18 行：循环条件错误（i <= students.size() 应为 i < students.size()）。
- 第 24 行：未释放动态分配的内存（应在析构函数中释放 students 中的指针）。

3. **复杂代码**：实现一个简单的银行账户管理系统，包含存款、取款和转账功能。

```cpp
#include <iostream>
#include <vector>
#include <mutex>
using namespace std;

class Account {
private:
    string accountNumber;
    double balance;
    mutex mtx;

public:
    Account(string accNum, double bal) : accountNumber(accNum), balance(bal) {}

    void deposit(double amount) {
        lock_guard<mutex> lock(mtx);
        balance += amount;
    }

    bool withdraw(double amount) {
        lock_guard<mutex> lock(mtx);
        if (balance >= amount) {
            balance -= amount;
            return true;
        }
        return false;
    }

    double getBalance() {
        lock_guard<mutex> lock(mtx);
        return balance;
    }

    string getAccountNumber() {
        return accountNumber;
    }
};

class Bank {
private:
    vector<Account*> accounts;

public:
    void addAccount(string accNum, double bal) {
        accounts.push_back(new Account(accNum, bal));
    }

    bool transfer(string fromAccNum, string toAccNum, double amount) {
        Account* fromAccount = nullptr;
        Account* toAccount = nullptr;

        for (auto acc : accounts) {
            if (acc->getAccountNumber() == fromAccNum) {
                fromAccount = acc;
            }
            if (acc->getAccountNumber() == toAccNum) {
                toAccount = acc;
            }
        }

        if (fromAccount && toAccount) {
            if (fromAccount->withdraw(amount)) {  // 隐藏逻辑错误：未检查余额是否足够
                toAccount->deposit(amount);
                return true;
            }
        }
        return false;
    }

    ~Bank() {
        for (auto acc : accounts) {
            delete acc;
        }
    }
};

int main() {
    Bank bank;
    bank.addAccount("12345", 1000.0);
    bank.addAccount("67890", 500.0);

    if (bank.transfer("12345", "67890", 1500.0)) {  // 转账金额超过余额
        cout << "Transfer successful!" << endl;
    } else {
        cout << "Transfer failed!" << endl;
    }

    return 0;
}
```

预期缺陷:

- 第 50 行：隐藏的逻辑错误（转账时未检查账户余额是否足够）。
- 第 34 行：依赖关系问题（未处理并发访问，可能导致数据竞争）。

2.2 测试步骤

1. 输入代码: 将代码片段分别输入到 GPT-4o、DeepSeek-V3 和 文心 X1 中。使用统一的提示词（Prompt），例如：“请检测以下代码中的语法错误和逻辑错误，并给出修复建议。”

2. 记录结果 ：记录每个模型的检测结果，包括：正确指出代码缺陷的数量, 正确修正代码缺陷的数量, 错误指出代码缺陷的数量, 错误修正代码缺陷的数量。

3 测试结果

3.1 模型 A (GPT-4o) 测试结果

3.1.1 简单代码测试结果

语法错误：

1. **缺少分号**

```cpp
int a = 5
```

这一行后面缺少分号 `;`，应改为：

```cpp
int a = 5;
```

2. **未初始化变量 `b`**

```cpp
int b;
```

变量 `b` 在使用之前没有被赋值，会导致 `sum = a + b;` 的结果不确定（可能是一个随机值）。

3.1.2 中等代码测试结果

1. 语法和逻辑错误

（1）数组越界访问

```cpp
for (int i = 0; i <= students.size(); i++) {
    total += students[i]->score;
}
```

错误分析：

- `students.size()` 返回的是元素数量，例如 3。
- 由于 `i <= students.size()`，循环的最大 `i` 值会是 `students.size()`（即 3）。
- 但是 `vector` 的索引是 **0 到 size-1**，因此 `students[3]` 超出了范围，导致 **越界访问**（可能崩溃）。

修正方案：

```cpp
for (int i = 0; i < students.size(); i++) { // 修正：i < students.size()
    total += students[i]->score;
}
```

（2）内存泄漏

```cpp
vector<Student*> students;
```

- `addStudent` 里使用 `new Student(name, score)`，但没有在 `GradeManager` 析构函数 `~GradeManager()` 释放这些 `Student` 对象，会导致 **内存泄漏**。

修正方案（释放内存）：

```cpp
~GradeManager() {
    for (Student* student : students) {
        delete student; // 释放动态分配的内存
    }
}
```

或者更好的方式：

- **使用** `vector<Student>` **而不是** `vector<Student*>`，这样 `Student` 对象会自动管理内存，不需要 `new` 和 `delete`。

3.1.3 复杂代码测试结果

1. 逻辑错误

（1）转账逻辑存在竞态条件

问题：多个线程可能同时访问 `fromAccount` 和 `toAccount`，导致数据不一致。

代码分析：

```cpp
if (fromAccount->withdraw(amount)) {
    toAccount->deposit(amount);
    return true;
}
```

假设 **两个线程同时执行转账**：

- 线程 A 可能在 `withdraw()` 之后、`deposit()` 之前被线程 B 打断。
- 线程 B 可能检查 `fromAccount->balance` 时仍然足够，但 `A` 已经减少了余额，导致 `B` 也扣款成功，从而导致 **超额支出**（数据不一致）。

✅ **修正方案**：

- **必须在 `transfer()` 方法中统一加锁**，保证**整个操作的原子性。**

（2）Bank 账户管理存在野指针风险

代码分析：

```cpp
vector<Account*> accounts;
```

- `addAccount()` 中使用 `new` 但 **没有智能指针管理内存。**
- 可能会导致 **悬空指针** 或 **手动管理内存的复杂性。**

✅ **修正方案**：

- **使用 `vector<unique_ptr<Account>>` 来自动管理内存**，避免手动 `delete`。

3.2 模型 B (DeepSeek-V3) 测试结果

3.2.1 简单代码测试结果

1. 语法错误

- **错误位置**: 第 5 行
- **错误描述**: `int a = 5` 后面缺少分号（`;`）。
- **修复建议**: 在 `int a = 5` 后面添加分号。

修复后的代码:

```cpp
int a = 5;
```

2. 逻辑错误

- **错误位置**: 第 6 行
- **错误描述**: 变量 `b` 未初始化，直接用于计算 `sum = a + b`，这将导致未定义行为（`b` 的值是随机的）。
- **修复建议**: 在使用 `b` 之前，为其赋予一个初始值。

修复后的代码:

```cpp
int b = 10; // 或其他合适的初始值
```

3.2.2 中等代码测试结果

1. 逻辑错误

- **错误位置**: 第 18 行
- **错误描述**: 循环条件 `i <= students.size()` 错误，会导致数组越界访问。
  - `students.size()` 返回的是学生数量，而 `students` 的索引范围是 `0` 到 `students.size() - 1`。
  - 当 `i` 等于 `students.size()` 时，`students[i]` 会访问无效内存，导致未定义行为（如程序崩溃）。
- **修复建议**: 将循环条件改为 `i < students.size()`。

修复后的代码:

```cpp
for (int i = 0; i < students.size(); i++) {
    total += students[i]->score;
}
```

---

2. 潜在性能问题

- **错误位置**: 第 24 行（析构函数）
- **错误描述**: `GradeManager` 类中使用了动态内存分配（`new Student`），但在析构函数中未释放这些内存，导致内存泄漏。
- **修复建议**: 在析构函数中释放 `students` 中每个 `Student` 对象的内存。

3.2.3 复杂代码测试结果

1. 语法错误

- **错误位置**: 第 44 行
- **错误描述**: `if (fromAccount->withdraw(amount))` 多了一个右括号 `)`。
- **修复建议**: 删除多余的右括号。

修复后的代码:

```cpp
if (fromAccount->withdraw(amount)) {
```

---

2. 逻辑错误

- **错误位置**: 第 44 行
- **错误描述**: 在转账逻辑中，`fromAccount->withdraw(amount)` 可能会失败（例如余额不足），但代码仍然会尝试 `toAccount->deposit(amount)`，这会导致逻辑不一致。
- **修复建议**: 在 `withdraw` 成功后，再进行 `deposit` 操作。

修复后的代码:

```cpp
if (fromAccount->withdraw(amount)) {
    toAccount->deposit(amount);
    return true;
}
```

---

3. 潜在性能问题

- **错误位置**: 第 50 行（析构函数）
- **错误描述**: `Bank` 类中使用了动态内存分配（`new Account`），但在析构函数中未释放这些内存，导致内存泄漏。
- **修复建议**: 在析构函数中释放 `accounts` 中每个 `Account` 对象的内存。

修复后的代码:

```cpp
~Bank() {
    for (auto acc : accounts) {
        delete acc; // 释放动态分配的内存
    }
}
```

3.3 模型 C (文心 X1) 测试结果

3.3.1 简单代码测试结果

语法错误：

1. **分号缺失**

   `int a = 5` 语句末尾缺少分号。

   修复：在行尾添加分号 `int a = 5;`

逻辑错误：

2. **未初始化的变量**

   变量 `b` 未被初始化，其值是不确定的（随机垃圾值），导致 `sum` 的计算结果不可预测。

   修复：为 `b` 赋予明确的初始值，例如 `int b = 0;`

3.3.2 中等代码测试结果

语法错误：

1. **缺少头文件**

   `Student` 类使用了 `string` 类型，但未包含 `<string>` 头文件。

   修复：在文件开头添加 `#include <string>` 。

逻辑错误：

2. **越界访问**

   `calculateAverage()` 函数中的循环条件 `i <= students.size()` 会导致越界访问（索引范围是 `[0, size-1]` ）。

   修复：将循环条件改为 `i < students.size()` 。

3. **内存泄漏**

   `GradeManager` 的析构函数未释放动态分配的 `Student` 对象。

   修复：在析构函数中添加内存释放逻辑：

3.3.3 复杂代码测试结果

以下是代码中的错误及修复建议：

语法/逻辑错误分析：

1. **死锁风险**

   `transfer()` 中分别锁定两个账户的互斥锁，若其他线程以相反顺序请求锁会导致死锁。

   修复：使用 `std::lock` 同时锁定两个互斥锁，并按固定顺序（如账号大小）加锁。

2. **允许转账给自己**

   当前代码允许将钱转给自己，导致余额不变但返回成功。

   修复：添加检查，若 `fromAccount == toAccount` 则返回失败。

3. **未原子化转账操作**

   `withdraw()` 和 `deposit()` 分开加锁，其他线程可能中间修改余额。

   修复：在 `transfer()` 中同时锁定两个账户的互斥锁，确保原子性。

4. **冗余遍历**

   单次循环查找两个账户，若账号不唯一可能错误匹配。

   修复：分开两次循环或使用 `find_if` 查找账户。

4 结果分析

对于大模型给出结果中，我们忽略一些**建议性的意见，**因为这些意见不影响代码的实际运行。

设大模型正确指出的缺陷数量为 α_1，正确修正的缺陷数量为 α_2，错误指出的缺陷数量为 β_1，错误指出的缺陷数量为 β_2，$Y_{A,B,C}$ 分别对应 chatgpt-4o，DeepSeek-V3，文心 X1 的得分，且 $Y=(\alpha_1+\alpha_2)-\frac{(\beta_1+\beta_2)}{2})$。

最终，根据如下数据，得出 $Y_1=4.0, Y_2=4.5, Y_3=5.0$。

简单代码

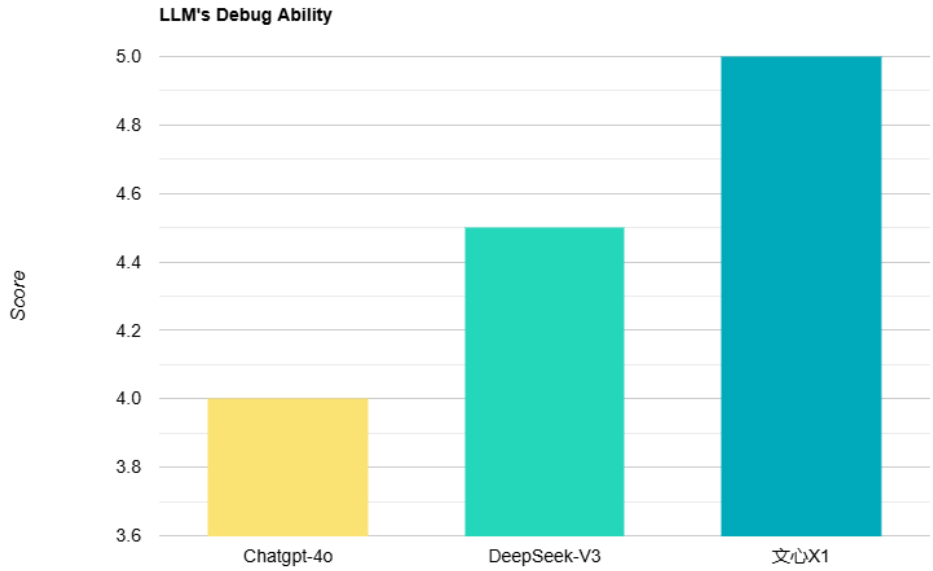
模型/指标	正确指出缺陷数量	错误指出缺陷数量	正确修正缺陷数量	错误修正缺陷数量
模型A(GPT-4)	2	0	2	0
模型B(DeepSeek-V3)	2	0	2	0
模型C(文心一言)	2	0	2	0

中等代码

模型/指标	正确指出缺陷数量	错误指出缺陷数量	正确修正缺陷数量	错误修正缺陷数量
模型A(GPT-4)	2	0	2	0
模型B(DeepSeek-V3)	2	0	2	0
模型C(文心一言)	2	0	2	0

复杂代码

模型/指标	正确指出缺陷数量	错误指出缺陷数量	正确修正缺陷数量	错误修正缺陷数量
模型A(GPT-4)	0	0	0	0
模型B(DeepSeek-V3)	1	1	1	1
模型C(文心一言)	1	0	1	0



综上，可以得出结论，**带有深度思考的文心 X1 模型**在总体表现上略优于目前因为服务器过载而下调了能力的 **DeepSeek-V3 模型**，同时 DeepSeek-V3 模型的总体表现略优于 **Chatgpt-4o 模型**。

不过需要指出的是，三种 LLM 在中低难度代码的纠错上都得到满分，只是在复杂任务中表现出微小差距。