Die Hausaufgaben müssen von jedem Studierenden einzeln bearbeitet und abgegeben werden. Für die Hausaufgabe sind die aktuellen Informationen vom Blog https://seblog.cs.uni-kassel.de/ss24/generative-ki-in-der-software-technik/ zu berücksichtigen.

Abgabefrist ist der 25.08.2024 – 23:59 Uhr

Abgabe

Wir benutzen für die Abgabe der Hausaufgaben Git. In eurem Repository soll sowohl der aktuelle Stand des Berichtsheftes, als auch die zu den jeweiligen Aufgaben gehörenden Ergebnisse an generiertem Code oder anderen Artefakten persistiert werden.

Wir nutzen **weiterhin** das Repository, was in Hausaufgabe 1 bereits unter folgendem Link angelegt wurde:

https://classroom.github.com/a/XeAtEq1Z

Die Bearbeitung der gestellten Aufgaben muss wie in Vorlesung und Übung besprochen in einem einheitlichen Berichtsheft dokumentiert werden.

- **Handschriftliche (Teil-)Berichte werden nicht gewertet.**
- **Nicht dokumentierte (Teil-)Berichte werden mit 0 Punkten bewertet!**
- **Nicht persistierte Lösungen für Aufgaben wie Code oder andere Artefakte führen zu 0 Punkten für diese Aufgabe.**
- **Das Berichtsheft muss als PDF-Datei (.pdf) abgegeben werden. Jedes Experiment ist in einem eigenen Kapitel anzulegen.**

Bericht

Jedes Experiment soll im Berichtsheft auf einer neuen Seite beginnen, sodass eine klare Unterteilung zwischen den einzelnen Experimenten gegeben ist. Ein Experiment ist schriftlich in die Abschnitte **Vorbereitung**, **Durchführung** und **Auswertung** zu unterteilen (je nach Aufgabentyp kann dies ergänzt werden).

Die Gesamtlänge eines Berichts (**minus der Abbildungen**) sollte in etwa bei 1–2 Din A4 Seiten (*Schriftgröße 11–12*) liegen. Wir empfehlen die Anfertigung des Berichtsheftes mit Latex.

Aufgabe 1 – Automatisierung von Mutationstesten – Fortsetzung

In dieser Aufgabe soll Hausaufgabe 05 fortgesetzt werden. Nach der Automatisierung des Mutationstestens sollen nun aus überlebenden Mutanten Tests generiert werden, die diese Mutanten aufdecken. Wir setzen also voraus, dass ein Werkzeug existiert, das aus gegebenem Code Mutanten erzeugt, kompiliert, mit der Testsuite des gegebenen Codes überprüft, als getötet oder lebendig kategorisiert und überlebende Mutanten archiviert.

Das zu entwickelnde System soll dann das Folgende leisten:

- Überlebende Mutanten sollen genutzt werden, um die Testsuite zu erweitern. (Also wieder aus der Datei einlesen und in einen Prompt einbauen.)
- Der daraus entstandene Mutanten-Test soll in eine Datei geschrieben und automatisch auf Lauffähigkeit geprüft werden. Wenn (i) der Test durchläuft, (ii) der Ausgangscode den Test erfüllt und (iii) der Test die Branch-Coverage erhöht, soll er in die existierende Testsuite integriert werden. Andernfalls wird er verworfen.
- Das Gesamtsystem soll vollautomatisiert nutzbar sein, um die Testsuite zu erweitern. Der Benutzer soll als Parameter eine anvisierte Branchcoverage und ein Limit für zu sendende bzw. zu empfangende Token angeben können. Das System soll dann iterativ Mutanten und aus überlebenden Mutanten Tests generieren, bis eins der Abbruchkriterien erreicht ist.
- **Optional:** Optimiert eure Implementierung von Blatt 5 auf die folgende Weise: Mutanten sollen nur noch durch diejenigen Tests überprüft werden, die die Codezeile auch ausführen, die zur Erzeugung des Mutanten geändert wurde.

Experiment: Wendet euer System auf die Codebasis an, die auf der folgenden Seite präsentiert wird (gleiche, wie auf dem vorherigen Blatt). Wählt dabei als Ziel eine Branchcoverage von 90 % und Tokenlimits, die euch nach ersten Tests realistisch erscheinen, sodass das System garantiert mehrere Iterationen durchlaufen kann.

Codebasis

Mit der folgenden, bereits auf Blatt 3 verwendeten Codebasis sollt ihr euer entwickeltes Programm am Ende laufen lassen und über die Ergebnisse berichten. **Für Tests während der Entwicklung empfiehlt es sich, auch andere Java-Programme zu verwenden, um das System nicht aus Versehen auf den Beispielcode zu optimieren. Hier dürft ihr frei eigene Programme verwenden.**

Die zugrundeliegende Javadatei können hier heruntergeladen werden:

`https://github.com/sekassel/gkistss24-files`

Folgende Dateien sind für die Aufgabe relevant:

- `Aufgabenblatt 3/sekasselmaps/MapService.java`
- `Aufgabenblatt 3/sekasselmaps/MapServiceTest.java`
- `Aufgabenblatt 3/sekasselmaps/model/City.java`
- `Aufgabenblatt 3/sekasselmaps/model/Location.java`
- `Aufgabenblatt 3/sekasselmaps/model/Order.java`
- `Aufgabenblatt 3/sekasselmaps/model/Street.java`

Programmverhalten
Das Programm sucht einen möglichen Pfad von einer Stadt zu einer anderen.

Bericht

Das programmierte Projekt soll im Berichtsheft kurz (max. 2 Seiten) dokumentiert werden. Hierbei soll ein Durchlauf des Programmes und dessen Ergebnis an einem Beispiel beschrieben werden. Eine detaillierte Quelltextdokumentation ist nicht notwendig. Dokumentieren Sie in Ihrer Evaluation explizit, inwieweit die im Szenario festgelegten Ziele erreicht werden konnten (eventuelle weitere Qualitätseigenschaften der Tests können informell diskutiert werden).

Aufgabe 2 – Parametereinstellungen

In dieser Aufgabe sollt ihr euch etwas mit den verschiedenen Parametern der Open AI API auseinandersetzen und ausprobieren, welche Auswirkungen sie jeweils haben. Die Parameter sind unter https://platform.openai.com/docs/api-reference/chat/create dokumentiert; einen kurzen Überblick gab es auch in der Einführungsvorlesung.

Führt für die gleiche Codebasis wie bei Aufgabe 1 die folgenden Aufgaben aus:

1. Wiederholt einzelne Prompts aus Aufgabe 1 (zur Mutanten- und/oder Testgenerierung) und ändert dabei jeweils den Wert **eines** selbst ausgewählten Parameters im Vergleich zur Durchführung bei Aufgabe 1. Erklärt in der Auswertung eures Experiments jeweils, welche Unterschiede bei den Ergebnissen ihr erwartet hättet und welche ihr tatsächlich beobachten konntet. Führt mindestens 10 solcher Prompts durch.
2. Wiederholt das Experiment aus Aufgabe 1 mit einer abweichenden Einstellung der Parameter und beschreibt die beobachteten Unterschiede.

Aufgabe 3 – Reduktion der Testfälle – Für Masterstudenten

Aufgabe für Masterstudenten!

Neben der hohen Laufzeit und dem Problem äquivalenter Mutanten kann beim Mutationstesten auch problematisch sein, dass Mutanten überleben, für die keine Tests geschrieben werden sollen, weil diese Tests die Qualität der Testsuite eher verringern als erhöhen würden: Sie testen Triviales und vergrößern dafür die Testsuite (d. h., sie erhöhen deren Laufzeit und machen sie schwerer wartbar). Eine schöne Beschreibung dieses Problems (und eine Beschreibung, wie Google damit umgeht), gibt es in dem Paper "Practical Mutation Testing at Scale. A view from Google" von Petrovic et al.

Ziel dieser Aufgabe ist es, auszuprobieren, ob LLMs in der Lage sind, Mutanten zu erkennen, die zu unnötigen Testfällen führen würden. Geht in den folgenden Schritten vor:

- Lest Ausschnitte aus dem Paper "Practical Mutation Testing at Scale. A view from Google", besonders die Abschnitte 1–3, um euch mit dem Problem und einer möglichen, nicht LLM-basierten Lösung vertraut zu machen.
- Passt eure Prompting Strategie an, um zu versuchen zu vermeiden, dass unproduktive Tests generiert werden. Hier könnt ihr sowohl beim Generieren der Mutanten als auch beim Generieren von Tests ansetzen.
- Wiederholt die Evaluation aus Aufgabe 1; benutzt dabei einen Timeout und/oder ein auf der Anzahl der Token basierendes Terminationskriterium (statt Coverage). Beobachtet, ob und wie sich die generierten Tests ändern.