ISTRUZIONI PRINCIPALI

Inizio dicendo che per generare test junit con gli LLM ho dovuto innanzitutto capire come è organizzato l’insieme dei file messo a disposizione dai ricercatori.

La cartella principale è scripts, essa contiene degli script in python che ci permettono di generare test junit con codex, gpt o codegen, poi ci sono altri tipi di script per fixare i test generati la prima volta con delle euristiche che vengono applicate alle classi di test. Infine c’è anche un file config che serve a specificare una secret key (che è necessaria per le comunicazioni delle api di openai) e la directory principale che contiene tutti i file e le cartelle. Poi c’è un altro script molto usato che è utils, in cui ci sono molte funzioni utili agli altri script per diversi motivi.

Dopodichè abbiamo la cartella RQ1\_Test\_Generation e RQ2\_Prompt\_Elements che contengono tutti i prompt di input per ogni modello e per ogni scenario e contengono anche gli output che ogni modello restituisce per ogni scenario. Ogni prompt (sia di input che di output) presente in queste cartelle è sia in json che in csv. Quelli che ci interessano principalmente sono i file json.

EvoSuiteBenchmark e HumanEvalJava contengono tutti i contesti che ci permettono la compilazione ed esecuzione di tutti i progetti dei dataset e i relativi test(rispettivamente sono progetti apache ant e maven).

JavaContextRetriever dovrebbe servire alla generazione dei prompt di input che mandiamo agli LLM per fargli generare codice.  
  
Step per l’esecuzione

1. Per prima cosa ho cercato di mandare in esecuzione *generate\_tests\_model.py* (dove model indica il modello con cui eseguiamo la generazione) tramite il terminale con il comando:

*python generate\_tests\_codex.py -t 2000 -d SF110 -q RQ1 -p RQ1\_Test\_Generation\OpenAI\_Data\SF110\_input\2\_a4j\_prompt.json*

1. Da questo comando parte una comunicazione con l’api del modello che permette la generazione di ogni prompt che si trova all’interno del file json del percorso indicato come prompt -p.
2. Quando la generazione si conclude nella cartella che si riferisce all’output di quel dataset che abbiamo usato riguardo alla domanda a cui stiamo rispondendo, viene generato un file json che contiene gli output per ogni metodo riguardo alla generazione automatica e in una cartella sempre a questo percorso vengono generate anche le classi di test vere e proprie.
3. A questo punto c’è bisogno di eseguire lo script *fix\_tests\_openai.py* che ci permette di andare a risolvere alcuni problemi molto comuni che impediscono la compilazione dei test, qui viene creato un altro file json sempre nella cartella degli output.
4. Dopo vado ad usare lo script *genera\_file\_java\_fixati.py* per prendere in input il file json appena generato nell’output e generare i file java che poi vado ad inserire nel progetto che mi interessa nella cartella src/test.
5. A questo punto è possibile compilare il progetto per cui si stanno generando test unitari, con il rischio molto alto che ci siano altri errori di compilazione che a questo punto devono essere risolti manualmente (i più comuni sono import mancanti o nomi uguali di due o più metodi nella classe di test).
6. Una volta che anche questi errori di compilazione sono stati risolti nella cartella target del progetto troverò una cartella site che contiene il report di jacoco (se non c’è fare jacoco:report) riguardo alla copertura di linee e rami. Invece sempre in target/surefire-reports troverà la correttezza (in termini di failure) di ogni metodo testato.
7. Ho creato anche uno script di nome *calcola\_correttezza.py* che dà una percentuale generale tenendo conto di tutti i metodi testati.

Risultati ottenuti

Ovviamente non ho generato tutti i test per tutti gli scenari con ogni modello per entrambi i dataset, ma ho fatto delle prove per verificare la fattibilità della cosa:  
La generazione è stata fatta con codex\* per il dataset HumanEval, sia per lo scenario originale che per gli scenari 1 e 2 (per lo scenario 3, la generazione è stata effettuata correttamente, solo che essendo basata su classi che riguardano solo la signature del metodo, non si riesce a compilare). I risultati che ho ottenuto con codex sono: **branch coverage = 92% e line coverage = 91 %** per lo scenario originale, poi rispettivamente **87 e 88 % per lo scenario 1 e 87 e 88 % per lo scenario 2**. Questi risultati risultano essere anche leggermente migliori di quelli trovati nello studio. Per quanto riguarda il tasso di correttezza invece sia aggira attorno al **66% per lo scenario originale e 40 e 37 % per gli scenari 1 e 2**. Anche questi risultati sono leggermente migliori rispetto a quelli che avevamo già.

L’utilizzo di gpt anche ha mostrato dati simili a quelli già ottenuti per entrambi i dataset. E con gpt sono riuscito anche a generare test per il dataset SF110, con codex non ci ero riuscito perché c’era un errore con i token, che risultavano essere troppo pochi per l’input/output che doveva gestire il modello. Di conseguenza si riusciva a generare test con codex per SF110 solo impostando il limite di token a 1000.

\*Per quando riguarda codex, gli autori di questo studio avevano usato il modello code-davinci-002 che al momento però è deprecato, di conseguenza sul sito di openai ho trovato una sezione che consiglia l’uso di specifici modelli al posto di quelli deprecati e ho usato gpt3.5-instruct. Questo cambiamento forse ha portato all’errore con i token che impedisce la generazione di test con codex per SF110.

Per ultimo, il modello di StarCoder. La generazione con questo modello funzionava, ma i tempi di generazione risultavano essere eccessivamente lunghi. Per generare la metà dei test per lo scenario originale di HumanEval, in risposta quindi a RQ1, ho impiegato quasi una giornata intera, questo mi ha portato a evitare per il momento l’uso di questo modello.