

Raport despre folosirea tool-urilor AI în testare

Mai 2025

Andrei-George Popescu andrei-george.popescu@s.unibuc.ro	Adrian-George Leventiu adrian-george.leventiu@s.unibuc.ro
Mihai-Leonard Ionescu mihai-leonard.ionescu@s.unibuc.ro	Alexandru-Iulian Anca alexandru-iulian.anca@s.unibuc.ro
Marin-Adrian Beșel marin-adrian.besel@s.unibuc.ro	

Abstract

Pynguin este un instrument care ajută la generarea automată a testelor unitare pentru codul scris în Python, folosind metode precum căutarea genetică sau generarea aleatorie. A fost creat pentru a acoperi un gol din ecosistemul Python, unde astfel de unelte nu sunt la fel de dezvoltate ca în alte limbaje, cum e cazul Java, care are la dispoziție soluții avansate precum EvoSuite. În acest studiu, ne-am propus să vedem cât de bine funcționează Pynguin și unde întâmpină dificultăți, testându-l pe mai multe probleme algoritmice de pe platforma LeetCode. Am ales aceste probleme pentru că sunt variate și reflectă situații reale din munca unui programator.

1 LLM în cadrul Pynguin

1.1 Introducere

În cadrul acestui proiect ne-am concentrat pe testarea automată a codului Python folosind componenta bazată pe inteligență artificială (LLM) oferită de biblioteca Pynguin care permite generarea de teste unitare fie prin analiză statică și tehnici evolutive, fie prin intermediul unui model AI capabil să analizeze codul și să sugereze cazuri de test relevante. Practic, am selectat strategia de generare a testelor prin AI și am lăsat modelul să genereze testele unitare pe baza codului nostru. Scopul a fost să evaluăm cât de bine poate un model de inteligență artificială să înțeleagă și să testeze codul în mod autonom și să observăm diferențele dintre celelalte strategii bazate pe analiza statică și aceasta. În acest raport descriem modul în care am folosit această strategie, pașii realizați, avantajele și limitele întâlnite, precum și concluziile trase din această abordare.

1.2 Cum funcționează

Pentru a utiliza această strategie, am parcurs următorii pași:

1. Obținerea cheii API de la OpenAI
2. Setarea variabilei de mediu OPENAI_API_KEY
3. Setarea parametrului -algorithm
4. Generarea testelor

1.3 Rezultate

În urma rulării cu strategia LLM, am obținut următorul tip de teste:

```
def test_search_insert_large_list
(solution):
    # Test case with a larger list
    # List with even numbers from
    # 0 to 999998
    nums = list(range(0, 1000000, 2))
    target = 500001
    assert solution.searchInsert(nums,
    target) == 250001

def test_search_insert_insert_position_start
(solution):
    # Test case where the target should
    # be inserted at the beginning
    assert solution.searchInsert([1, 3,
    5, 6], 0) == 0
```

1.4 Provocări

1.4.1 Problema cu cheia API invalidă

Atunci când am folosit versiunea de dezvoltare 0.41.0dev a Pynguin, am întâlnit o eroare destul de frustrantă legată de noua funcție care folosește modele LLM pentru generarea testelor. Problema nu era explicată nicăieri, nici măcar pe [GitHub-ul oficial Pynguin](#).

Concret, chiar dacă foloseam o cheie API validă, Pynguin spunea că e invalidă și nu putea continua generarea testelor. La început am

crezut că e din cauza schimbărilor de la OpenAI, care a trecut de la chei personale la chei de tip proiect. După ce am investigat mai mult, am descoperit că problema era, de fapt, în Pynguin.

Librăria prelua cheia din variabilele de mediu, dar o trata greșit: în loc să trimită cheia reală către API, trimitea o versiune ascunsă (cum ar fi *****), din cauza modului în care era stocată cu `SecretStr`. După ce am scos acea conversie și am trimis cheia așa cum e, totul a funcționat normal.

Am raportat problema printr-un [issue \(#93\)](#) pe GitHub, ca să poată fi rezolvată și pentru ceilalți utilizatori.

```
# Definiția lor inițială (problema la linia 116)
```

```
116     OPENAI_API_KEY = SecretStr(
os.environ.get("OPENAI_API_KEY", ""))
```

```
# Rezolvarea oferită de noi
```

```
116     OPENAI_API_KEY =
os.environ.get("OPENAI_API_KEY", "")
```

1.4.2 Promptul nu era suficient de clar

Am fost nevoiți să modificăm promptul trimis către OpenAI pentru a ne asigura că testele generate importă corect funcțiile din fișierul `original_file.py` și că nu se bazează pe comparații sensibile la ordinea elementelor folosind operatorul `==`.

Inițial, testele nu includeau deloc linii de genul `from original_file import ...`, iar rezultatele erau comparate direct ca liste. Asta ducea la erori atunci când listele conțineau aceleași elemente, dar într-o ordine diferită.

După ce am reformulat promptul, am cerut explicit ca funcțiile necesare să fie importate corect și ca verificările să fie făcute într-un mod mai sigur, fie prin sortarea rezultatelor înainte de comparație, fie prin transformarea lor în mulțimi (`set`), astfel încât ordinea să nu mai conteze. În felul acesta, testele au devenit mai robuste și mai corecte.

```
# Prompt original
```

```
    user_prompt = "Generate test cases
for the following Python code:\n\n"
```

```
# Prompt modificat
```

```
    user_prompt = "Generate test cases
for the following Python code, import
the necessary classes and function from
```

```
the file named:" + config.configuration
.module_name.replace(".", "/") +", be
careful with the original code output
type (if the output type is mentioned in
the function, do not try to put something
else instead of what it is required, make
them so that the order of the result
does not matter), the code:\n\n"
```

1.5 Concluzii

Utilizarea componentei LLM din Pynguin a adus atât avantaje semnificative, cât și limitări notabile.

Pe partea pozitivă, testele generate de modelul AI au fost mai relevante și mai adaptate logicii codului testat, comparativ cu metodele clasice. LLM-ul a demonstrat o capacitate bună de a identifica corner case-uri relevante, nu doar situații aleatorii sau bazate strict pe tipuri de input.

Totuși, au existat și aspecte problematice. Modelul a făcut uneori importuri greșite sau incomplete, a generat teste cu structuri de date nepotrivite și nu a oferit un mod de control al determinismului (ex. lipsa unui seed), ceea ce a dus la rezultate diferite între rulări.

De asemenea, lipsa de „curățenie” a codului generat și tendința de a forța anumite tipuri de input fără legătură cu logica aplicației au fost limitări importante. În concluzie, LLM-ul are potențial real în generarea inteligentă de teste, dar are nevoie de rafinări pentru a produce cod robust, curat și consistent.

2 AI în restul proiectului

În afară de utilizarea AI-ului în cadrul generării testelor, am folosit ChatGPT pentru refrazare în general, cât și pentru aflarea unor informații relevante proiectului, pentru a eficientiza procesul.

Câteva exemple de prompturi și răspunsurile aferente:

- "avand un paper despre Pyguin, biblioteca de python ce are in vedere testarea unitara in python, cum as putea numi un capitol ce are in vedere documentarea unui issue/bug gasit la librerie (dat fiind faptul ca lucrez pe versiunea 0.41.0dev care nu este cea stabila, deoarece ea este singura in care se regasesc functionalitatile AI pe care le vreau testate)?"

Pentru un capitol care documentează un **issue/bug** descoperit într-o versiune de dezvoltare a unei biblioteci (în cazul tău, *Pyguin 0.41.0dev*), titlul ar trebui să fie clar, formal și să reflecte:

1. natura instabilă a versiunii;
2. identificarea și documentarea unei probleme;
3. contribuția ta în procesul de testare sau investigare.

Iată câteva sugestii de titluri pentru acel capitol:

Variante formale și clare:

- Documentarea unui Bug în Pyguin 0.41.0dev
- Investigarea unei Probleme în Versiunea de Dezvoltare Pyguin
- Identificarea și Analiza unui Comportament Neașteptat în Pyguin AI
- Limitări și Probleme Identificate în Pyguin 0.41.0dev
- Studiu de Caz: Bug Descoperit în Versiunea Experimentală a Pyguin

Variante mai tehnice și orientate pe proces:

- Debugging și Raportare de Bug-uri în Pyguin 0.41.0dev
- Diagnosticarea unei Probleme în Pyguin: Studiu de Caz
- Testare Exploratorie a Funcționalităților AI în Pyguin și Identificarea unui Bug
- Impactul Instabilității în Versiunile Dev: Cazul Pyguin 0.41.0dev

Variante mai narative (pentru un stil mai personal sau orientat pe experiență):

- Provocări întâlnite în utilizarea versiunii de dezvoltare a Pyguin
- Dincolo de documentație: Ce am descoperit în Pyguin 0.41.0dev
- Un pas în față, doi înapoi: Explorarea unui bug în Pyguin

Variante mai narative (pentru un stil mai personal sau orientat pe experiență):

- Provocări întâlnite în utilizarea versiunii de dezvoltare a Pyguin
- Dincolo de documentație: Ce am descoperit în Pyguin 0.41.0dev
- Un pas în față, doi înapoi: Explorarea unui bug în Pyguin

Dacă ai un stil mai academic sau structurat, o bună alegere ar fi:

Identificarea și Documentarea unui Bug în Pyguin 0.41.0dev

Dacă preferi un titlu mai aplicat și contextual:

Studiu de Caz: Eșecul Execuției Testelor AI în Pyguin 0.41.0dev

Spune-mi dacă vrei să personalizez titlul în funcție de natura bug-ului (ex: legat de integrarea AI, generarea testelor, fallback-ul pe modele tradiționale etc).

- "cum functioneaza dynamosa pe scurt in cadrul libreriei pyguin"

Pe scurt, **DynaMOSA** este una dintre strategiile de generare a testelor implementate în Pyguin, și se bazează pe **optimizare multi-obiectivă dirijată dinamic**. Este o extensie a algoritmului **MOSA (Many-Objective Sorting Algorithm)**, adaptată pentru testarea automată.

Cum funcționează DynaMOSA în Pyguin:

- Scopul este să genereze teste unitare care acoperă cât mai bine codul sursă (ex. ramuri, condiții, linii de cod).
- În loc să aibă un singur obiectiv (de exemplu, maximizarea acoperirii de cod), DynaMOSA tratează fiecare obiectiv de acoperire (ex. fiecare ramură) ca un **obiectiv separat**, folosind o abordare **multi-obiectivă**.
- **Dynamic** înseamnă că nu consideră toate obiectivele de la început. În schimb, introduce **treptat** obiectivele pe măsură ce codul este explorat — ceea ce reduce spațiul de căutare și face algoritmul mai eficient.
- Este un algoritm **evolutiv**, adică începe cu o populație de teste generate aleator și le evoluează (prin selecție, încrucișare, mutație) pentru a îndeplini cât mai multe obiective de acoperire.

Pe scurt:

DynaMOSA în Pyguin este un algoritm evolutiv care generează automat teste unitare, optimizând simultan pentru mai multe obiective de acoperire, introduse dinamic pe măsură ce testarea progresează.

Dacă vrei, pot face și o comparație între DynaMOSA și LLM în contextul Pyguin.

3 Resurse

- **Documentația oficială Pyguin**
<https://pyguin.readthedocs.io/en/latest/index.html>
Data ultimei accesări: 12 aprilie 2025
- **Studiul academic despre Pyguin**
<https://link.springer.com/article/10.1007/s10664-022-10248-w>
Data ultimei accesări: 15 aprilie 2025
- **Cod Sursă Pyguin**
<https://github.com/se2p/pyguin>
Data ultimei accesări: 13 mai 2025
- **Colecția de probleme algoritmice testate**
<https://algomap.io/>
Data ultimei accesări: 27 aprilie 2025
- **Documentația PyTest**
<https://docs.pytest.org/>
Data ultimei accesări: 10 mai 2025
- **Python Type Hints PEP 484**
<https://peps.python.org/pep-0484/>
Data ultimei accesări: 22 aprilie 2025
- **EvoSuite**
<https://www.evosuite.org/>
Data ultimei accesări: 22 aprilie 2025
- **OpenAI API**
<https://platform.openai.com/docs/overview>
Data ultimei accesări: 13 mai 2025
- **OpenAI, ChatGPT**
<https://chatgpt.com/>
Data ultimei generări: 16 mai 2025