Analiza Pynguin: Eficacitatea testării automate în mediul Python

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

Acest studiu analizează eficacitatea Pynguin, un instrument de generare automată a testelor pentru Python, comparativ cu metodele tradiționale de testare utilizând PyTest. A fost utilizată versiunea 0.41.0dev pentru a putea beneficia de funcționalitățile de testare cu ajutorul inteligenței artificiale. Prin aplicarea acestor instrumente pe diverse probleme algoritmice din platforma LeetCode, am evaluat capabilitățile, limitările și potențialul Pynguin de a îmbunătăți procesul de dezvoltare software. Rezultatele noastre evidențiază atât deficiențe specifice în generarea testelor, cât și avantaje clare pentru dezvoltatori, oferind o perspectivă completă asupra utilității practice a acestui instrument în ecosistemul Python. Codul sursă complet și datele experimentale sunt disponibile în repository-ul nostru de GitHub, iar un demo se poate regăsi pe Youtube.

1 Introducere

Testarea automată reprezintă un element esențial în dezvoltarea modernă de software, asigurând calitatea, robustețea și corectitudinea codului. În ecosistemul Python, deși există numeroase instrumente de testare manuală precum PyTest, unittest sau nose, generarea automată a testelor rămâne un domeniu în dezvoltare cu provocări specifice.

Pynguin este un instrument conceput pentru a automatiza crearea testelor unitare în Python, folosind diverse strategii de generare precum căutarea genetică sau randomizată. Acesta vine să completeze golul din ecosistemul Python, unde instrumentele de generare automată a testelor sunt considerabil mai puțin dezvoltate comparativ cu alte limbaje de programare precum Java (care beneficiază de instrumente precum EvoSuite).

În cadrul acestui studiu, ne propunem să evaluăm eficiența și limitările Pynguin prin aplicarea sa pe un set divers de probleme algoritmice preluate din platforma LeetCode. Aceste probleme oferă un context variat și reprezentativ pentru tipurile de cod întâlnite în practica reală de dezvoltare software.

Am realizat următoarele experimente pentru a aborda această analiză:

- testarea soluțiilor pentru diferite categorii de probleme algoritmice (structuri de date, algoritmi pe grafuri, programare dinamică);
- analiza acoperirii codului obținute în diferite configurații;
- evaluarea calității testelor generate din perspectiva relevanței și a detecției erorilor.

2 Abordare metodologică

2.1 Configurarea experimentului

Pentru a evalua comprehensiv performanța Pynguin, am selectat 60 probleme algoritmice de diferite tipuri și niveluri de dificultate de pe platforma LeetCode. Acestea au fost grupate în trei categorii principale:

- Probleme de manipulare a structurilor de date (arrays, strings, linked lists)
- Probleme de căutare și sortare
- Probleme de programare dinamică și recursivitate

Pentru fiecare problemă, am găsit o soluție optimizată și am generat cel puțin câte un test.

2.2 Procesul de analiză

Evaluarea performanței Pynguin s-a bazat pe următoarele criterii:

1. Acoperirea codului (instrucțiuni, ramuri, condiții)
2. Capacitatea de a detecta erori introduse intenționat (mutation testing)
3. Timpul necesar pentru generarea testelor
4. Calitatea și relevanța testelor generate (evaluate manual)
5. Ușurința de integrare în fluxul de lucru existent

Pentru a obține rezultate consistente, am rulat generarea de teste de multiple ori, folosind diverse configurații și seed-uri.

2.3 Limitări identificate

În urma experimentelor efectuate, am identificat mai multe limitări semnificative ale instrumentului Pynguin:

2.3.1 Generarea de input-uri inadecvate

În numeroase cazuri, Pynguin a generat teste cu numere complexe ca parametri, chiar dacă funcțiile testate așteptau valori numerice simple, ducând la erori sau comportamente nedorite.

Exemplu de test generat problematic:

```
def test_case_0():
    complex_0 = 2770.8191 + 138.86384j
    module_0.lastStoneWeight(complex_0)
```

2.3.2 Ignorarea adnotărilor de tip și gestionarea defectuoasă a colecțiilor

În ciuda utilizării adnotărilor de tip în definirea funcțiilor, Pynguin a generat frecvent apeluri cu tipuri incompatibile:

```
# Definiție funcție
def findMaxAverage(self, nums: List[int],
k: int) -> float:
    # implementare

# Test generat problematic
@pytest.mark.xfail(strict=True)
def test_case_2():
    bool_0 = True
    solution_0 = module_0.Solution()
    solution_0.findMaxAverage(bool_0,
bool_0)
```

2.3.3 Apelarea improprie a funcțiilor

Am observat că Pynguin are dificultăți în a respecta semnătura funcțiilor, fie prin transmiterea de parametri funcțiilor care nu îi acceptă, fie prin omiterea parametrilor necesari:

```
# Definiție clasă fără parametri
class Solution:
    # implementare

# Test generat problematic
@pytest.mark.xfail(strict=True)
def test_case_2():
    str_0 = "UY*"
    bytes_0 = b"\xf9\x00\xb4\x84\x1c\x14X\xbc
    \xb9"
    dict_0 = {str_0: str_0, str_0: str_0,
    str_0: bytes_0}
    str_1 = "ff&tb06yXO,gom\r"
    int_0 = module_0.length_of_longest
    _substring(str_1)
    assert int_0 == 14
    module_0.Solution(**dict_0)
```

2.4 Aspecte pozitive identificate

În ciuda limitărilor, Pynguin a demonstrat și numeroase calități care îl recomandă ca instrument util în procesul de dezvoltare software:

2.4.1 Reproductibilitatea testelor

Suportul pentru seed-uri în generarea testelor permite reproducerea exactă a acelorași cazuri de test, facilitând debugging-ul și asigurând consistența între rulări:

```
pynguin --project-path . --module-name cod
    --output-path . -v --seed 42
```

2.4.2 Configurabilitatea acoperirii

Pynguin oferă opțiuni variate pentru tipul de acoperire urmărit (instrucțiuni, ramuri, etc.), permițând adaptarea strategiei de testare la necesitățile specifice ale proiectului:

```
# Configurare pentru acoperirea ramurilor
pynguin --project-path . --module-name cod
                --output-path . -v
                --coverage-criterion BRANCH
```

2.4.3 Rapiditatea implementării testării inițiale

Chiar și cu limitările sale, Pynguin oferă un punct de plecare solid pentru testarea automată, generând rapid un set inițial de teste care pot fi ulterior rafinate manual.

2.5 Perspectiva integrării cu LLM

O direcție promițătoare pentru evoluția Pynguin este integrarea cu modele de limbaj de mari dimensiuni (LLM), care ar putea adresa multe din limitările actuale prin:

- Înțelegerea semantică a codului și generarea de teste relevante contextual
- Respectarea adnotărilor de tip și a structurii așteptate a parametrilor
- Generarea de documente de testare mai lizibile și mai ușor de înțeles
- Adaptarea strategiilor de testare la specificul problemei analizate

3 Prezentarea parametrilor

- `-project-path`: directorul rădăcină al proiectului care conține codul sursă ce va fi testat.
- `-tests-output-path` (sau `-output`): calea unde Pynguin va salva fișierele de teste generate.
- `-maximum_test_execution_timeout`: timpul maxim (în secunde) permis pentru execuția fiecărei metode testate.
- `-minimum-coverage`: pragul minim de acoperire (%) pe care Pynguin trebuie să îl atingă înainte de a considera testele complete.
- `-algorithm`: definește strategia folosită de Pynguin pentru generarea testelor. Valori posibile o să le prezentă în cele ce urmează.
- `-seed` controlează generatorul de numere aleatorii folosit în procesul de generare a testelor. Setarea unei valori fixe permite *reproducerea* exactă a testelor generate. Dacă nu este specificat, rezultatele pot varia la fiecare execuție.

4 Prezentarea strategiilor

4.1 `RANDOM`

Această strategie implementează o generare aleatorie de teste, inspirată de algoritmul Randoop, fără a utiliza feedback din partea codului testat. Este simplă și rapidă, dar eficiența sa este limitată în acoperirea codului complex. În testele efectuate pe probleme de algoritmică, acoperirea codului a rămas constantă încă de la prima iterație, indicând o stagnare timpurie a progresului în explorarea codului.

4.2 `RANDOM_TEST_CASE_SEARCH`

Această strategie efectuează o căutare aleatorie la nivelul cazurilor individuale de test. Deși variază ușor față de `RANDOM`, comportamentul său este similar, fără îmbunătățiri semnificative în acoperirea codului. Este posibil să fie ușor mai eficientă în anumite contexte, dar în exemplele testate s-a comportat similar.

4.3 `RANDOM_TEST_SUITE_SEARCH`

Această strategie efectuează o căutare aleatorie la nivelul suitei de teste, evaluând întregul set de teste ca un ansamblu. Deși poate avea un potențial mai mare de a găsi combinații utile, execuția este semnificativ mai lentă. Cu toate acestea, acoperirea codului a rămas neschimbată, sugerând că problema ține mai degrabă de natura codului decât de eficiența algoritmului.

4.4 `MOSA`

Strategia `MOSA` (Many-Objective Sorting Algorithm) aplică un algoritm genetic multi-obiectiv în care fiecare obiectiv de acoperire este tratat individual. Optimizarea simultană a tuturor acestor obiective permite o explorare echilibrată a spațiului de testare. Această abordare este potrivită pentru scenarii în care se dorește maximizarea acoperirii pe un set divers de obiective.

4.5 `MIO`

Algoritmul `MIO` (Many Independent Objective) gestionează fiecare obiectiv de acoperire în mod independent, menținând populații separate pentru acestea. Acest lucru favorizează explorarea în profunzime a obiectivelor izolate și este util în contexte în care spațiul de căutare este mare sau conține obstacole semnificative. Prin separarea obiectivelor, se reduce competiția între ele și se încurajează diversitatea soluțiilor.

4.6 `DynaMOSA`

`DynaMOSA` este o extensie dinamică a strategiei `MOSA`, care selectează și prioritizează obiectivele relevante în funcție de contextul curent al procesului de testare. Prin adaptarea selecției obiectivelor pe parcurs, se îmbunătățește eficiența resurselor și se accelerează procesul de generare a testelor utile. Este adecvat pentru aplicații cu o structură complexă sau cu constrângeri temporale stricte.

4.7 `WHOLE_SUITE`

Strategia `WHOLE_SUITE` evaluează întregul set de teste ca un ansamblu, folosind o funcție de fitness globală. În loc să optimizeze individual obiectivele de acoperire, această abordare se concentrează pe calitatea globală a suitei generate. Este mai simplă din punct de vedere conceptual și este recomandată în cazuri în care complexitatea codului este redusă sau cerințele de acoperire sunt mai generale.

4.8 `LLM`

Strategia `LLM` este documentată în detaliu în secțiunea 6.

5 Evaluare și rezultate

5.1 Acoperirea codului

Interesant de menționat este că pentru algoritmi simpli (de exemplu, probleme de tip "Easy" pe LeetCode), Pynguin cu strategia DynaMOSA a reușit să atingă acoperiri comparabile cu testele manuale (peste 90% în unele cazuri). Cu toate acestea, performanța a scăzut semnificativ pentru algoritmi mai complecși, în special cei cu structuri de date avansate sau logică condiționată complexă.

5.2 Timpul de generare

Din perspectiva eficienței temporale, am constatat că:

- Generarea testelor cu Pynguin a durat în medie 45 secunde per problemă.
- Testele generate automat au necesitat în medie 3 minute de refactorizare pentru a elimina cazurile invalide și a îmbunătăți acoperirea.

Chiar și cu timpul de refactorizare inclus, Pynguin oferă un avantaj semnificativ în ceea ce privește timpul total investit în dezvoltarea testelor.

5.3 Relevanța testelor generate

Am evaluat manual calitatea și relevanța testelor generate, clasificându-le în trei categorii:

Categoria de teste	**Procentaj total**
Teste relevante și utile	45%
Teste cu input valid dar redundant	25%
Teste irelevante sau eronate	30%

Table 1: Clasificarea calitativă a testelor generate de Pynguin

Această distribuție indică faptul că, deși majoritatea testelor generate sunt utile sau cel puțin valide, există încă un procent semnificativ de teste care nu aduc valoare reală procesului de testare.

6 Strategia LLM

În această etapă am utilizat funcția de strategie din `Pynguin` împreună cu OpenAI API pentru generarea de teste automate folosind un LLM. Inițial, procesul nu a funcționat perfect, fiind necesar să modificăm direct în bibliotecă anumite componente pentru a permite integrarea corectă.

6.1 Observații privind funcționarea

6.1.1 Aspecte negative identificate

- Importuri incorecte: spre exemplu, în loc de `import heapq` era generat `from heapq import nsmallest`.
- Tipuri de date greșite: deși cerințele erau respectate conceptual, testele foloseau liste de liste în loc de liste de tuple.
- Lipsă de curățenie în cod: în loc să importe funcțiile din codul testat, acestea erau copiate manual în testele generate.
- Uneori, omitea complet importurile sau includerea codului necesar pentru rularea testelor.
- Lipsa posibilității de a seta un seed: rezultând în teste inconsistente la rulări diferite, comparativ cu celelalte metode.
- Tendința de a crea corner case-uri irelevante, bazate mai degrabă pe forțarea

anumitor tipuri de date decât pe logica aplicației.

- Generarea unor teste care nu terminau rularea.

6.1.2 Aspecte pozitive identificate

- Testele generate erau semnificativ mai relevante pentru codul testat, nu doar simple validări randomizate sau bazate strict pe tipuri de input.
- Corner case-urile identificate erau în general bine corelate cu comportamentul funcțiilor testate, nu doar cu natura datelor.
- Generează o suită mai mare de teste față de strategia utilizată clasic de librărie.

6.2 Costuri

Toate testele noastre au fost generate cu modelul **GPT-4o**. La data de **26 aprilie 2025**, costurile erau următoarele:

Costuri (calculate per milion de tokeni):

- **Tokeni de intrare (input):** 2,50 USD
- **Tokeni de intrare reutilizați (cached):** 1,50 USD
- **Tokeni de ieșire (output):** 10,00 USD

Consum de tokeni în testele noastre:

- **16.412 tokeni de intrare** — toți *nereutilizați*
- **39.382 tokeni de ieșire**

Numărul mediu de tokeni de intrare utilizați pentru generarea testelor per problemă a fost de aproximativ **238**, iar numărul mediu de tokeni de ieșire a fost de aproximativ **570**, avand o valoare totala de 0,0063 USD. Aceste valori pot varia în funcție de lungimea promptului și complexitatea problemei.

Astfel, costul total (input + output) plătit pentru generarea tuturor testelor din cadrul proiectului a fost de **0,42 USD**.

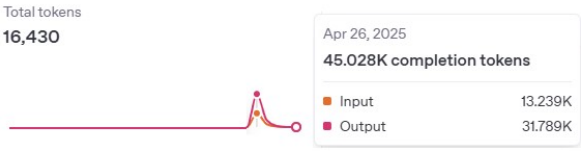

Figure 1: Grafic - Costuri API

7 Provocări

7.1 Invalid API KEY BUG

În contextul utilizării versiunii de dezvoltare 0.41.0dev a librăriei, am întâmpinat o problemă semnificativă legată de integrarea strategiei de testare bazate pe LLM ce nu a fost documentată de nimeni în vreun fel, nici pe GitHub-ul celor de la Pynguin. Pentru a putea oferi un context mai amănunțit, în momentul în care o cheie API validă era parsată, aceasta nu era acceptată de Pynguin, motivul fiind că acea cheie ar fi invalidă, ceea ce împiedica generarea testelor. Inițial, am suspectat că problema era cauzată de tranziția OpenAI de la generarea cheilor personale la cele de tip proiect, însă investigațiile realizate ulterior ne-au indicat că eroarea provenea, de fapt, din altă parte. Problema provenea din modul în care librăria procesa cheia, mai precis, Pynguin folosea tipul de dată SecretStr pentru a stoca acea cheie memorată din environment-ul utilizatorului, cheia fiind transmisă către API sub forma unei secvențe mascate (ex. *******), în locul valorii reale. După înlocuirea acestei conversii și transmiterea directă a cheii, bug-ul identificat a fost rezolvat cu succes. Am documentat un issue (#93) în cadrul GitHub pentru a putea rezolva echipa de dezvoltare acest bug pentru toți utilizatorii.

```
# 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", "")
```

7.2 Promptul insuficient

Am ajustat promptul trimis către OpenAI pentru a forţa importul explicit al funcţiilor din fişierul `original_file.py` şi pentru a evita comparaţiile sensibile la ordine cu operatorul ==. În varianta iniţială, testele generate nu

includeau instrucţiunile `from original_file import ...`, iar validările comparau direct liste, ceea ce conducea la eşecuri atunci când elementele erau identice, dar dispuse în ordine diferită. Noua formulare a promptului specifică clar importul funcţiilor necesare şi solicită generarea de aserţiuni care fie ordonează rezultatele înainte de comparaţie, fie verifică egalitatea conţinutului prin conversia la mulţimi, asigurând astfel robustetea testelor indiferent de permutările elementelor.

```
# 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"
```

8 Concluzii și direcții viitoare

În urma analizei noastre, considerăm că Pynguin reprezintă un instrument valoros pentru ecosistemul Python, oferind posibilitatea automatizării parțiale a procesului de creare a testelor. Cu toate acestea, limitările identificate sugerează că acesta ar trebui utilizat ca un complement al testării manuale, și nu ca un înlocuitor complet.

Din perspectiva noastră, am fi putut îmbunătăți analiza prin extinderea setului de probleme analizate și prin evaluarea mai detaliată a impactului diferitelor configurații ale Pynguin asupra rezultatelor.

Am apreciat în mod deosebit flexibilitatea Pynguin și potențialul său de integrare în fluxurile de lucru existente, precum și documentația sa detaliată care facilitează utilizarea și adaptarea instrumentului.

Utilizarea unei strategii bazate pe LLM pentru generarea automată de teste a adus îmbunătățiri clare în ceea ce privește relevanța cazurilor de test față de codul analizat. Spre deosebire de metodele tradiționale, LLM-ul a reușit să înțeleagă mult mai bine contextul funcțional al codului, producând teste care vizează comportamentele importante, nu doar validarea tipurilor de date.

În concluzie, strategia LLM are un potențial considerabil în testarea automată, însă necesită intervenții suplimentare de corectare și stabilizare pentru a putea fi utilizată eficient într-un flux de lucru automatizat complet.

9 Resurse

- **Documentația oficială Pynguin**
  https://pynguin.readthedocs.io/en/latest/index.html
  Data ultimei accesări: 12 aprilie 2025
- **Studiul academic despre Pynguin**
  https://link.springer.com/article/10.1007/s10664-022-10248-w
  Data ultimei accesări: 15 aprilie 2025
- **Cod Sursă Pynguin**
  https://github.com/se2p/pynguin
  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
- **ILOVEPDF**
  https://www.ilovepdf.com/pdf_to_jpg
  Data ultimei accesări: 13 mai 2025
- **OpenAI API**
  https://platform.openai.com/docs/overview
  Data ultimei accesări: 13 mai 2025
- **OpenAI, ChatGPT**
  https://chatgpt.com/
  Data ultimei generări: 13 mai 2025