

TDK-dolgozat

Kolláth István Tibor

**Egy reprodukálható, Python nyelvű hibákat tartalmazó adatbázis
készítése és felhasználása az automatikus hibajavítás területén**

Készítette:

Kolláth István Tibor

II. évf. programtervező informatikus MSc hallgató

Témavezető:

Dr. Antal Gábor

adjunktus

Szegedi Tudományegyetem Természettudományi és Informatikai Kar

Informatikai Intézet

Szoftverfejlesztés Tanszék

Szeged

2023

Tartalomjegyzék

1. Bevezetés	2
2. Kapcsolódó munkák	3
3. Módszertan	7
3.1. Projektek kiválasztása	7
3.2. Lehetséges issue-k keresése	8
3.3. Lehetséges hibák manuális felülvizsgálata	9
3.4. Folyamatos release (kiadás)	11
3.5. Séma	11
3.6. PyBugHive bővített verziója	13
3.6.1. PyBugHive architektúrája	13
3.6.2. Offline verzió	15
3.6.3. Elérhetőség	15
4. Karakterisztikák, jellemzők	16
4.1. PyBugHive karakterisztikája	16
4.2. Adatokra vonatkozó karakterisztikák	17
4.3. Limitációk	19
5. Használati eset	21
5.1. Hibakeresés	22
5.2. Hibajavítás	25
6. Következtetések és jövőbeli tervek	27

1. fejezet

Bevezetés

Az utóbbi években a Python programozási nyelv az egyik legnépszerűbb programozási nyelv lett. Ehhez hozzájárult, hogy a Python számos területen alkalmazható, például adattudományra (data science), webfejlesztésre és mesterséges intelligencia algoritmusok használatára. Az egyre növekvő érdeklődés miatt egyre több kutató figyelmét keltette fel a nyelv, ezzel felfedezve a benne rejlő potenciált és új felhasználási területeket. Azonban annak ellenére, hogy a Python népszerűsége folyamatosan nő, mind a kutatók, mind a fejlesztők körében, a magas minőségi tanulmányok számának változásával ez a növekedés nem tart lépést. Erre a hatalmas szakadékra egy lehetséges magyarázat, hogy nagyon kevés a hozzáférhető, magas minőségű hibaadatbázis, adathalmaz és ez megnehezíti a kutatók számára a reprodukálható kísérletek végzését, és azok kiértékelését. Sajnos csak néhány olyan hibaadatbázis létezik, amely reprodukálható, hibás Python kódot tartalmaz. A hibaadatbázisok széles körben alkalmazhatóak automatizált programjavításra, hibakalibrációra, hiba-előrejelzésre.

Annak érdekében, hogy enyhítsük a fent említett problémát, bemutatjuk az általunk készített, különböző Python projektekből származó, manuálisan ellenőrzött, reprodukálható hibákat tartalmazó adatbázist, melyet PyBugHive-nak nevezünk. A PyBugHive célja, a Python népszerűsége és a minőségi kutatási adathalmaz hiányának áthidalása, lehetővé téve a kutatók számára egy megbízható forrást kísérleteikhez. Munkánkat más nyelvek hasonló adatbázisainak példájára építettük, a két leginkább említésre méltó ilyen példa a Defects4J és a BugsJS volt. A PyBugHive tervezését ezeknek a népszerű és széles körben elfogadott adatkészletek architektúráis elveinek betartásával végeztük.

2. fejezet

Kapcsolódó munkák

Hibaadatbázisok

A SIR benchmark [4] egyike volt az első valós hibákat tartalmazó adathalmazoknak. Kilenc C programot és hét Java programot adott hozzá a jól ismert Siemens benchmark-hoz [10]. Automatizálási folyamatot segítő szkripteket, teszteseteket és hibaadatokat is tartalmaz. A SIR benchmarkban valós és mesterséges hibák is megtalálhatók, azonban a mesterséges hibákból több található meg a hibaadatbázisban.

Az egyik leggyakrabban használt adathalmaz a Defects4J [12], amely széles körben használatos a szakirodalomban [24, 29, 11, 8, 9, 35, 5, 21]. Jelenleg 835 valós hibát tartalmaz, mely 17 valós életből vett, nyílt forráskódú Java projektből származik. A szerzők egy teljes keretrendszert hoztak létre, így az adathalmaz használata és bővítése rendkívül egyszerű és kis erőfeszítést igényel.

A Bugs.jar [27] Java projektekből származó valós hibákat tartalmaz. 1158 hibát és javítás van benne, nyolc nagy, népszerű, nyílt forráskódú Java projektből. A Bugs.jar-hoz sajnos nem tartozik API (Alkalmazásprogramozási Felület), de egy README fájl segít az adatbázis használatában.

JavaScript nyelvű hibákból a BugsJS [33] az első és egyben az egyik legnépszerűbb hibaadatbázis. A BugsJS 453 valós, manuálisan ellenőrzött hibát tartalmaz, 10 népszerű JavaScript projektből. A BugsJS egy letisztult, könnyen használható API-t is kínál a felhasználói számára. Minden hibához tartozik hibajegy (issue), a javító módosítás (commit) és a teszteset(ek). A szerzők bemutattak egy példa használatot is, ahol a BugsJS-t

hiba detektáláshoz használták [32].

A ManyBugs és az IntroClass [17] 1183 hibát tartalmaz 15 C programból. Ebből 185 valós életből vett hiba van, mely kilenc nyílt forráskódú programból származik. A fennmaradó 998 hiba, diákok C programozási feladatából származik, melyek egy bevezető C programozási kurzuson keletkeztek. A szerzők kifejezetten az automatizált programjavítás céljából hozták létre ezt a benchmarkot. Hasonlóan az adathalmazunkhoz, a szerzők szkripteket is biztosítanak az adathalmaz könnyű használatához. Az IntroClass egy része Java-ra is adaptálva lett [7].

A BugSwarm [31] a Travis CI build adatait használja fel a hibák feltárásához. A BugSwarm egy sikertelen build-et hibának tekint (azaz olyan hibának, amely megakadályozza a rendszer sikeres összeállítását vagy a tesztek lefutását), míg a sikeres build a hibás verzió javítása. A BugSwarm-ban 3091 sikertelen-sikeres pár található. A BugSwarm egy Docker környezetet kínál, így minden build könnyen megismételhető, reprodukálható. A feltárt hibák Java és Python projektekből származnak. Egy kritikus értekezés [6] az adatbázis néhány pontatlanságára hívta fel a figyelmet: ismétlődő hibák, nem reprodukálható build-ek, nem létező Docker fájlok és olyan hibák, amelyek nem kapcsolódnak a forráskódhoz. Ezzel szemben, mi minden hibát kézi módon validáltunk az adatbázisunkban és létrehoztunk egy "offline" verziót is az adathalmazból, azért, hogy ne tűnjenek el a hibák.

A QuixBugs [20] 40 hibás és helyes verziót kínál ugyanazon programokból. A benchmark sorozat a Quixey Challenge [16] problémáin alapul. Mivel a kihívás során egy perces feladat volt a hiba kiszűrése a kódsorban, így minden program csak egy hibás sort tartalmaz. A QuixBugs a hibákat kiváltó teszteseteket is biztosítja számunkra. Az adathalmaz ugyanazt a 40 algoritmust tartalmazza Java és Python nyelven is. A szerzők nem hoztak létre saját API-t a QuixBugs-hoz, azonban a tesztek futtatásához használható a Gradle (Java-hoz) és a pytest (Python-hoz). Ellentétben QuixBugs-al a mi adatbázisunk nem csak egy sorban javítható hibákat tartalmaz.

Python hibaadatbázisok

Bár hibaadatbázisok már régóta léteznek, viszonylag kevés van belőlük Pythonhoz. Az első ilyen hibaadathalmaz a BugsInPy [34] volt. Munkájukban Widyasari és munkatársai egy benchmarkot hoztak létre, amely 493 hibát tartalmaz 17 valós Python projektből, emellett a benchmark használatának könnyítése érdekében támogató szkripteket is készítettek. Munkájuk során heurisztikák és kézi ellenőrzés segítségével azonosították a valódi hibákat. Ellentétben a munkájukkal, mi a GitHub hibakezelő rendszerét használtuk a potenciális hibák gyűjtéséhez. Így biztosak lehetünk benne, hogy az adatbázisunkban lévő hibák valósak, továbbá, hogy a fejlesztők javítani szeretnék volna a hibát. Ezenkívül, ahogy Rezaalipour és munkatársai is beszámoltak [26], a BugsInPy-ban olyan hibák is vannak, amelyek nem reprodukálhatóak. A hibák kezeléséhez a BugsInPy shell szkriptet használ. A PyBugHive ezzel ellentétben Python szkripteket használ, mely platformfüggetlenséget tesz lehetővé. Ellentétben a BugsInPy-al, mi Python virtuális környezeteket is használunk, amelyek biztosítják, hogy minden teszt a megfelelő Python verzióval és függőségekkel fusson. Továbbá elkészítettünk egy olyan verziót is, melyben minden függőség előre telepítve van, hogy megelőzzük a függőségek változásából eredő problémákat.

	BugsInPy	Közös	PyBugHive
Projekt	17	4	10
Hibák száma	500	5	105

2.1. táblázat. BugsInPy és a PyBugHive összehasonlítása

A két adatbázist összehasonlítottuk, amit a 2.1 táblázatban láthatunk. Mint látható, a két adatbázis 4 közös projektet tartalmaz, azonban mindössze 5 issue-t (hibajegyet) találunk meg mindkét adatbázisban, ami 4.7% hasonlóságot jelent. Ez azt mutatja, hogy a megoldásunk kiegészíti a BugsInPy-t, így hosszú távú terveink közé tartozik a BugsInPy-ban található hibák importálása a saját adatbázisunkba.

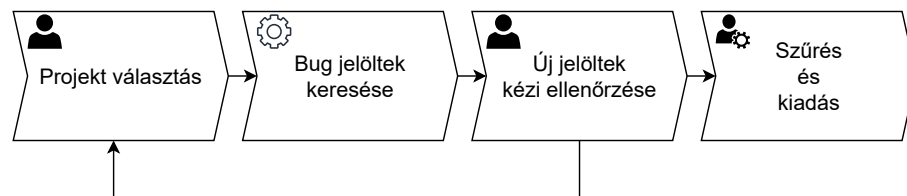
A PyTraceBug [1] egy Python hibaadathalmaz, amely 5,7 millió nem hibás forráskódrészletet és 24 ezer hibás kódrészletet tartalmaz, amelyeket automatikusan gyűjtöttek publikus GitHub projektekből. Az adatbázis tartalmazza az issue-kat, használatot segítő további információkat, mint például a feldolgozott forráskódokat. A PyTraceBugs metó-

dus szintű hibás kódrészleteken alapul. Ennek oka az, hogy a szerzők egy olyan adatbázist akartak létrehozni, amelyet könnyen lehet mélytanuláshoz használni. Az adatok Pandas DataFrames objektumokban vannak tárolva. Ellentétben ezzel a megoldással, a mi adatbázisunk hibaszintű és a PyTraceBugs-tól eltérően, a mi hibáinkat kézi validálás alapján készítettük (kiemelt figyelemmel a reprodukálhatóságra, míg a PyTraceBugs esetében ez nem volt szempont). Emellett mi API-t is kínálunk az adatbázisunk egyszerű használatához.

A Defectors [23] a legújabb adathalmaz, amely hibás Python nyelvű projektekből származó kódrészleteket tartalmaz. Körülbelül 213 ezer forráskódfájlt tartalmaz (ebből körülbelül 93 ezer hibás), mindezt 18 különböző területről származó, 24 népszerű Python projektből. A szerzők a SZZ algoritmust [28] is használták a hibákat előidéző változások azonosításához. Bár a Defectors jó választás lehet gépi tanulási modellek számára, a hibák reprodukálása bonyolult és nehézkes, mivel sem a reprodukciós lépések, sem a reprodukáláshoz szükséges környezet nem áll rendelkezésre. Ellentétben ezzel, a mi adatbázisunkban lévő hibákat reprodukálni tudjuk, így az algoritmusok helyességét ellenőrizhetjük.

3. fejezet

Módszertan



3.1. ábra. Módszertanunk áttekintése

A PyBugHive tervezése során célkitűzésünk volt, hogy a lehető legjobban felhasználóbarát legyen, ugyanakkor lehetővé tegye a fejlesztők számára, hogy minimális erőfeszítéssel egyszerűen tudják bővíteni, karbantartani. A hibák gyűjtésének és az adathalmazba való felvételének módszere magas pontosságot garantál (azaz a bekerült hibák valódi hibák lesznek), és így a hibák reprodukálása, ellenőrzése sokkal könnyebbé, gyorsabbá válik. A 3.1. ábra áttekinti a hibagyűjtési módszerünk fő lépéseit. Ebben a fejezetben bemutatjuk az adataink gyűjtésének egyes lépéseit, az adatbázis séma részleteit, a szűrési feltételeket, a reprodukálási és ellenőrzési folyamatot, valamint az API-k és parancssoros eszközök felépítését és alapvető használatát.

3.1. Projektek kiválasztása

Az első lépésben olyan projekteket kellett kiválasztanunk melyek megfelelően dokumentált issue kezeléssel rendelkeznek. A projektek a GitHub-on legnépszerűbb Python projektek közül kerültek kiválasztásra. Ezek közül azokat választottuk, amelyek nagy akti-

vitással rendelkeznek, így valószínűleg nagy fejlesztői közösséggel is. Kifejezetten olyan projekteket választottunk, amelyek teljesítik az alábbi kritériumokat:

- **Népszerűség:** Stargazer-ek (csillagok) száma $\geq 2,000$
- **Érettség:** Commitok száma $\geq 1,000$
- **Aktivitás:** Az utolsó commit éve ≥ 2022
- **Közösség:** Forkok száma ≥ 200
- **Szervezettség:** A problémák kezelése a GitHubon keresztül történik

Ahogy az azt később részletesen is kifejtjük, szigorú követelményeket alkalmazunk. A legtöbb, ezer potenciális problémával rendelkező projekt esetében is kevesebb mint száz potenciális hibát kaptunk. Azonban ezek a jelöltek a kézi validáció során valódi hibáknak bizonyultak, tehát a szigorú szűrőmechanizmus segített a valós (true positive) példák azonosításában. A bekerült projektek végső listája megtalálható a 4.1. táblázatban.

3.2. Lehetséges issue-k keresése

Miután kiválasztottuk a potenciális projekteket, elkezdtuk a fő kiválasztási, keresési folyamatot. Ebben a fázisban automatikusan ellenőriztük az összes projekthez kapcsolódó hibajegyet. A GitHub API segítségével minden issue-t összehasonlítottunk a követelményeinkkel, és azokat tároltuk el egy adatbázisban melyek megfeleltek az összes követelményünknek. Ezen az adatbázison kezdtük el a munkát, ebből az adatbázisunkból szűrtük ki végül a valós (true positive) hibákat. Minden hibának meg kellett felelnie a következő követelményeknek ahhoz, hogy belekerüljön az adathalmazba:

- **Az issue hibának van jelölve.** Ez csökkenti annak valószínűségét, hogy a hibajegy-hez tartozó commit új funkciót ad hozzá. A projektek jelöléseit (label) manuálisan áttekintettük, és hozzáadtuk őket a adatbányász szkriptünkhöz.
- **A hiba egyetlen committal van javítva.** Ez segíti a validálást és a tesztelést, valamint megkönnyíti a hibák megértését és reprodukálását.

- **A hibajegy le van zárva.** Ez biztosítja, hogy a problémát sikeresen megoldották.
- **A javító commit tartalmaz mind kód- mind tesztváltozásokat.** A tesztfájlok változásai akkor fordulnak elő, amikor egy új funkcionalitás (például input ellenőrzés) kerül hozzáadásra egy issue megoldásához, vagy az előző tesztek nem fedték le teljesen a funkció egy bizonyos részét (például egy függvényt nem teszteltek érvénytelen adatokkal).

Ezen követelmények kombinációja maximalizálja a valódi hibajavítások valószínűségét, valamint viszonylag könnyen reprodukálhatóvá teszik azokat. A kiválasztott projektek népszerűek és valószínűleg nem rendelkeznek rosszul megjelölt hibajegyekkel, azonban biztosan nem zárhatjuk ki annak az eshetőségét, hogy a lezárt hibajegy rosszul volt jelölve és nem hibajavító commitot tartalmazott. A probléma mitigálásának érdekében az adatokat kézi ellenőrzésnek is alávetettük.

Fontos megjegyezni, hogy az információgyűjtési fázisokat a könnyebb érthetőség érdekében egymást követő lépésekként mutatjuk be, de ezek párhuzamosan is elvégezhetőek a különböző projektek vizsgálata alkalmával.

3.3. Lehetséges hibák manuális felülvizsgálata

Ez a feladat volt az adatbázis létrehozásának legnehezebb és legerőforrás-igényesebb fázisa. Ennek részeként minden *bug* jelöltet manuálisan meg kellett vizsgálni, és validálni, hogy eldönthessük a hiba valóban valódi hiba találat-e, és hogy a commit változásai nem tartalmaznak semmi mást, csak a javítást. Annak érdekében, hogy a hibákat reprodukáljuk, két lépésre volt szükség: (1) környezet beállítása, a szükséges függőségek telepítése és a program build-elése (felépítése); (2) a tesztek futtatása és a hibajavítás validálása a forráskódban lévő javított tesztek futtatásával.

Ehhez Ubuntu telepítést, virtuális környezeteket és Docker konténereket használtunk annak érdekében, hogy minimalizáljuk annak a lehetőségét, hogy a telepítési lépések más platformokon ne sikerüljenek. Bár csak Linux rendszereken próbáltuk ki őket, a reprodukciós lépéseket platformfüggetlenek. Ezért az API megpróbálja reprodukálni a hibákat a használt operációs rendszertől függetlenül. Természetesen lehetnek olyan projektek,

melyek nem minden rendszeren működnek.

A begyűjtött hibajelöltek manuális vizsgálata során minden hibát ellenőriztünk és megpróbáltunk reprodukálni. Először elemeztük a commit tartalmát és a megfelelő metaadatokat annak érdekében, hogy megállapítsuk, valóban hiba-e a megadott probléma. Ha valóban hiba volt, megpróbáltuk reprodukálni a telepítéshez szükséges függőségeket telepítve. A projektek fejlesztőinek README-jeire és a telepítési útmutatóira támaszkodtunk, mivel nem rendelkezünk mély ismeretekkel az összes projektről, így azok telepítését is a fejlesztői útmutatások alapján végeztük. A telepítési és teszt lépések precízen való követése volt a cél, de sok esetben ezeket módosítani kellett, főleg visszafelé kompatibilitás miatt. Például sok esetben egy projektnek minimális verziókövetelménye volt egy adott függőségre, de a pontos (vagy maximális) verzió nem volt meghatározva, ebben az esetben a függőség legújabb verzióját telepítettük. Ez néhány esetben problémához vezetett, hiszen a legújabb verziókban már kompatibilitási gondok voltak, így ezeket kézzel megvizsgáltuk, és a megfelelő verziót telepítettük.

A reprodukációs folyamat a következő lépéseket tartalmazza:

- Első lépésként fel kell telepíteni a projekt javított verzióját, és ellenőrizni, hogy a tesztek sikeresen lefutnak-e.
- Második lépésként fel kell telepíteni a hibás verziót.
- Harmadik lépésként le kell futtatni a teszteket a hibás verzión és ellenőrizni, hogy mindegyik sikeresen lefut-e.
- Negyedik lépésként át kell másolni a hibás verzióhoz azokat a tesztfájlokat amiket a javító commit tartalmazott.
- Ötödik lépésként újra le kell futtatni a teszteket a hibás verzión annak érdekében, hogy megtudjuk az új tesztek hibásan (failed) futnak-e a projekt régebbi (hibás) verzióján.

A reprodukációs folyamat célja ellenőrizni, hogy a hiba javítva lett-e a commit után, valamint, hogy a biztosított tesztek valóban lefedik-e a hibás viselkedést. Ha minden fenti lépés sikeres volt egy hibánál, akkor nemcsak sikerült telepítenünk a projekt előző

verzióját, hanem ellenőriztük egy hiba-teszt páros érvényességét is, így azt a konkrét hibát érvényesnek és reprodukálhatónak jelöltük.

Miután reprodukáltunk egy hibát, feljegyeztük a reprodukálásának lépéseit. Ezután ezeket a lépéseket használva újra átnéztük a hibákat, a hiba reprodukálásának ellenőrzéseként. Emellett létrehoztunk a PyBugHive offline verzióját (lásd a 3.6.2 fejezet is) mellyel a felhasználóknak nem kell kézi módon telepíteniük a projekteket, biztosított a virtuális környezet és a tesztek futtathatóak.

3.4. Folyamatos release (kiadás)

Annak ellenére, hogy most publikáltuk a PyBugHive első verzióját, karbantartásra és bővítésre folyamatosan szükség van. A kiadási folyamat során az adatbázis munkaverzióját egy automatizált szkripttel szűrjük meg, mely segítségével csak a valóban ellenőrzött hibák kerülnek kiadásra. A manuálisan ellenőrzött hibákat ezután átmásoljuk egy új, üres adatbázisba. A létrehozott adatbázis lesz az új verzió, amelyet manuálisan publikálunk a GitHubon. Szemantikus verziózást alkalmazunk az adatbázis különböző verzióinak megkülönböztetésére; adott egy {major}.{minor}.{patch} verzió, az inkrementumok a következő esetekben történnek:

- **Major:** Egy vagy több hiba hozzáadásra/törlésre került.
- **Minor:** Python API/Docker környezet változások.
- **Patch:** Hibajavítások (például korlátozásokat állítunk be a bug telepítési lépéseiben, függőségeket adott verzióhoz rögzítünk), dokumentációs változások (amelyek nem érintik magát a hibaadatbázist).

Minden verzióváltás okát megjegyzések formájában részletezni fogjuk a kiadás során.

3.5. Séma

A adatbázisban minden rekord egy projektet reprezentál, ezek mindegyikét egy felhasználónévvel és egy repository mezővel azonosítjuk. Ezek a mezők a projekt tulajdonosát

és a projekt nevét reprezentálják. Ez a jelölés nemcsak az adatbázisunkban azonosítja a projektet, hanem lehetővé teszi a felhasználóknak, hogy további információk nélkül klónozzák a projektet, mivel ezek az azonosítók megegyeznek a GitHub-on található azonosítókkal. Ezek mellett minden projekt tartalmaz egy `installSteps` mezőt, amely leírja, hogyan kell telepíteni a projektet, valamint egy hiba-listát, amely tartalmazza a megadott projekt hibáit, amelyek megfelelnek a követelményeinknek (lásd a 3.2 fejezetben.).

Minden bug (azaz hiba) azonosítása `id` mezővel történik (melynek értéke a GitHub-on lévő issue azonosítója). A bugok mindegyike tartalmazza a hozzájuk tartozó commit-ot is, beleértve a hash-t, a commit dátumát, a szülő (parent) hash-ét, a megváltozott forráskódot és teszt fájlokat tartalmazó listát (`files` és `tests`), valamint a teljes javítást. Elhatároztuk, hogy az egyes fájlokhoz közvetlen URL-t is tárolunk, így azok egyszerűen elérhetők anélkül, hogy az egész repository-t klónoznunk kellene. A manuális validációs fázis során a módosult/új teszt fájlok tesztéseit ellenőriztük annak biztosítása érdekében, hogy minden hiba reprodukálható legyen.

A rekordok további metaadatokat is tartalmaznak a hibákról, például a címüket, létrehozás dátumát, a GitHub-on található hiba címkéket. Természetesen további adatok is elérhetők a GitHubon, melyek könnyen lekérdezhetők a GitHub API segítségével. Például a 1234 számú *projX* projekt *ownerY* tulajdonában lévő issue teljes leírása megtalálható itt: <https://github.com/ownerY/projX/issues/1234>.

Az `installSteps` mező megjelenhet az issue szintjén is, ha a hiba más telepítési lépéseket igényel. Ez könnyen előfordulhat, mivel lehetnek olyan hibák, amelyek csak egy specifikus újabb/régebbi Python verzióval működnek, vagy az adott hiba a függőség más verzióját igényli.

Minden ellenőrzött hibának van egy `testSteps` és egy `testStepsFull` mezője is, amelyek leírják, hogyan futtathatók a megfelelő tesztek. A `testSteps` csak a javításhoz kapcsolódó tesztfájlokat futtatja, míg a `testStepsFull` a projektben található összes tesztet futtatja.

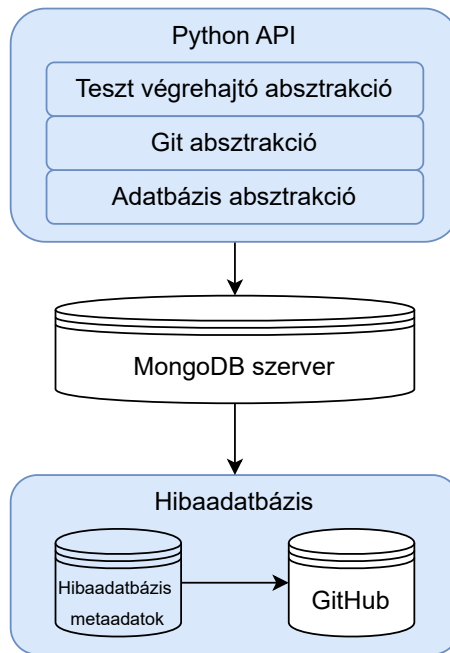
3.6. PyBugHive bővített verziója

Míg a PyBugHive csak manuálisan validált bejegyzéseket tartalmaz, kínálunk egy kiterjesztett verziót az adatbázisunknak, amit PyBugHiveX-nek nevezünk. A `manuallyChecked` mező azt mutatja, hogy egy problémát manuálisan ellenőriztek és reprodukáltak-e, vagy sem. Ha ez `true`, akkor a megadott probléma nemcsak két manuális ellenőrzés során igazoltan hiba, hanem reprodukálható is (természetesen ez az érték *igaz* minden PyBugHive hibára).

Ezenkívül lazítottunk a hibák felvételekor alkalmazott követelményeken, beleértve azokat a problémákat is, amelyeket kifejezetten nem jelöltek meg hibaként, de a leírásuk tartalmazta a `bug` szót. Ezek a bugok kevésbé valószínűek, hogy helyesek, mivel még jól menedzselt projektekben is előfordulhat, hogy egy fejlesztő olyan problémát hoz létre, amelynek nem reprezentatív a leírása. Ezeket a problémákat egy külön mezővel jelöltük meg, amit `text_based`-nek hívunk, így könnyen elkülöníthetők a címkéken (`labelen`) alapuló hibajelöltektől. Ha ennek a mezőnek az értéke `true`, akkor a megadott hibát kihagytuk volna a szigorúbb felvételi követelményekkel, és így nagyobb a valószínűsége annak, hogy hamis (*false positive*) a találat.

3.6.1. PyBugHive architektúrája

A 3.2 ábra bemutatja a PyBugHive architektúráját. Az általunk készített részek kék színnel vannak kiemelve (a Python API és a hibaadatbázis), míg a rendszerünktől független elemek fehérrel vannak színezve (MongoDB Server, GitHub). Ahogy látható, a magas-szintű Python API teljes absztrakciót biztosít a tesztesetekhez. Az API lefedi a fő funkcionalitásokat, melyek az alábbiak: (1) parancsok a megfelelő tesztek futtatásához, (2) interakció a verziókezelő rendszerrel (a projektjeink a GitHubon vannak tárolva, de ez az absztrakció lehetővé teszi más verziókezelő rendszerek használatát is a jövőben), (3) adatbázis interakció. Mivel az adatok egy Mongo adatbázisban vannak tárolva, az API futtatásához szükség van egy adatbázis szerverre is (kivéve az offline verziót, lásd 3.6.2).



3.2. ábra. PyBugHive architektúrája

PyBugHive használata

A PyBugHive használata egyszerű, mivel az összes parancs (a *clean* parancson kívül) ugyanazt a mintát követi: *python pybughive.py {parancs} {projekt}-{issue}*, ahol *{projekt}* a kiválasztott projekt neve, és *{issue}* a kiválasztott issue sorszáma.

A lehetséges parancsok a következők:

- **checkout:** Klónozza a kiválasztott projektet és a megfelelő commit hash-re állítja. Ez az első parancsunk kell, hogy legyen, amikor az eszközt használjuk.
- **install:** Megkeresi a megfelelő telepítési lépéseket és végrehajtja azokat. Ez a második parancs.
- **test:** Végrehajtja a megfelelő teszteket. A `--all` flag-gel használva, a *issue*-nál szereplő fájlok helyett az összes tesztet futtatja.
- **fix:** Kijavítja a forráskódban lévő hibát.
- **clean:** Ennek a parancsnak nincs szüksége projekt vagy az *issue* sorszáma. Törli az összes futtatás során létrehozott ideiglenes fájlt. A `--all` flag-gel használva mindent töröl, amit az eszköz letöltött, beleértve a projekteket is.

Mielőtt a PyBugHive-ot használni lehet, konfigurálni kell a `INSTALL_DIRECTORY`-t, amely megadja, hogy hova kell letölteni a projekteket, valamint a `MONGO_URL`-t, amely megadja a MongoDB kapcsolat URL-jét. A felhasználó három módon adhatja meg ezeket:

1. Hozzáadhatja azokat a rendszer környezeti változóihoz.
2. Létrehozhat egy `.env` fájlt a PyBugHive projekt gyökérkönyvtárában, és hozzáadhatja ezeket a fenti változókat.
3. Létrehozhat egy `config.py` fájlt a PyBugHive projekt gyökérkönyvtárában.

Kínálunk egy Docker image-et, melyben mindent előre telepítettünk. Biztosítunk továbbá egy telepítőszkriptet (`setup.sh`), amely telepíti az adatbázisunkban szereplő bugok sikeres futtatásához szükséges Python verziókat és az egyéb szükséges függőségeket. Mindkét opció tökéletes választás lehet, mi azonban inkább a Docker környezet használatát javasoljuk.

3.6.2. Offline verzió

A hibaadatbázisok egyik fő problémája a hibák folyamatos reprodukálhatóságának biztosítása [36, 26], például a folyamatosan változó függőségverziók miatt.

Ez az a fő ok, amiért létrehoztunk a PyBugHive offline verzióját. Ennek a változatnak nincs szüksége adatbázis kapcsolatra, de internetkapcsolatra sem, így a hibaadatbázist teljesen offline környezetben is lehet használni. Minden hiba esetében letöltöttük a konkrét projektet, mind a hiba előtt, mind utána, létrehoztuk a megfelelő környezetet, majd mindent egy zip fájlba tömörítettünk. Ezen módszerrel a felhasználóknak nem kell kézi módon telepíteniük a projekteket, biztosított a virtuális környezet és a tesztek futtathatóak. Ennek a verziónak a hátránya azonban a nagy mérete.

3.6.3. Elérhetőség

Az API, az adatbázis, a szkriptek, a projektek offline verziójához szükséges linkek és a reprodukcióhoz szükséges megfelelő Docker környezet elérhető itt:

<https://pybughive.github.io>

4. fejezet

Karakterisztikák, jellemzők

Ebben a fejezetben a PyBugHive-al kapcsolatos karakterisztikákat, alapvető jellemzőket és annak lehetséges korlátait mutatjuk be.

4.1. PyBugHive karakterisztikája

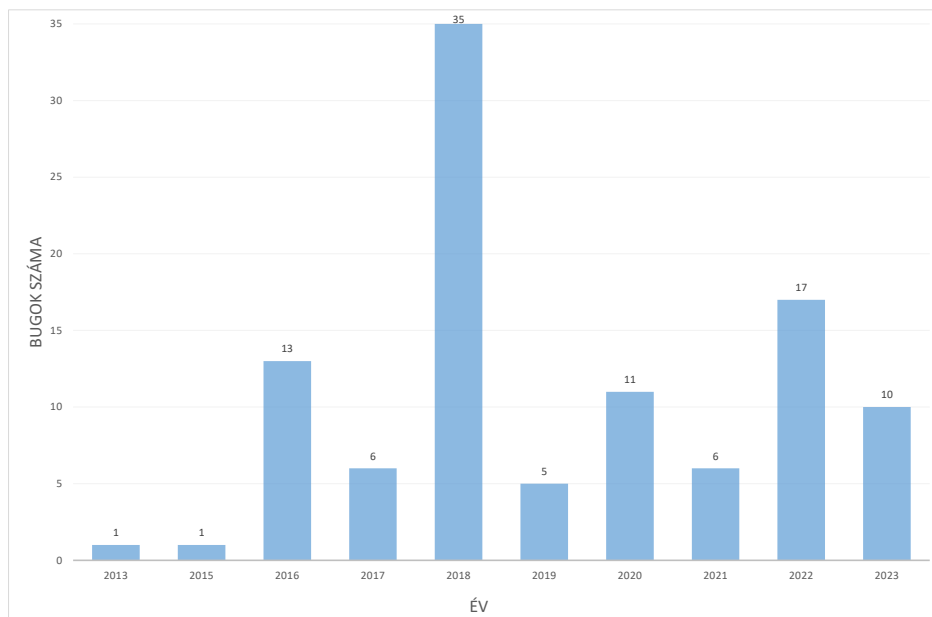
4.1. táblázat. PyBugHive és a benne lévő projektek karakterisztikája (2023. július 31.)

Név	Issue-k	Domain (fő profil)	LOC	Teszttek (LOC)	Csillagok	Fork-ok	Commit-ok	Aktív évek
psf/black	38	kód formázás	303 466	12 709	33 389	2 161	1 627	5
cookiecutter/cookiecutter	2	projekt generálás template-ből (sablonból)	5 631	4 055	20 162	1 892	3 011	10
Rapptz/discord.py	2	Discord bot	37 960	1 464	13 368	3 773	4 906	8
freqtrade/freqtrade	15	kriptovaluta kereskedés	73 015	41 765	22 755	5 213	22 035	6
numpy/numpy	1	numerikus analízis	170 179	91 891	24 344	8 459	33 396	13
pandas-dev/pandas	16	adatelemzés	394 739	268 554	39 550	16 639	33 195	13
python-poetry/poetry	11	csomagkezelés	39 411	24 504	26 358	2 050	2 938	5
saltstack/salt	7	rendszerfelügyelet	747 255	307 831	13 472	5 471	118 331	12
scrapy/scrapy	2	webes adatgyűjtés	45 052	29 105	48 267	10 127	10 032	13
explosion/spaCy	11	természetes nyelvfeldolgozás	89 644	30 771	26 954	4 232	15 984	9
Átlag	10		190 635	81 264	26 861	6 001	24 545	9

Az adathalmazunk kezdeti verziójához 105 hibát gyűjtöttünk össze 10 nyílt forráskódú GitHub projektből.

A 4.1 táblázat felsorolja a projekteket és azok statisztikáit, mint például a reprodukált hibák száma, a fő profil (domain), a kódsorok száma (LOC) vagy a projekt fejlesztése óta eltelt évek.

Szándékosan törekedtünk arra, hogy olyan projekteket válasszunk, amelyek széles körű spektrumot fednek le; előnyben részesítettük azokat, amelyek elérték egy bizonyos érettségi szintet, amelyet a 3. fejezetben jellemeztünk. Ezek a projektek népszerűségüket



4.1. ábra. PyBugHive-ban összegyűjtött bug-ok az évek függvényében

a GitHub-on elért csillagok számával is igazolták.

A 4.1 ábra az évek során keletkezett bug-ok eloszlását mutatja be. Ahogy látható, az adatbázis legrégebbi hibája 2013-ból származik, míg a legtöbb hibajavítás 2018-ból származik.

4.2. Adatokra vonatkozó karakterisztikák

Egy magas minőségű hibakészlet létrehozása jelentős időt és emberi erőfeszítést igényel. Ahhoz, hogy egy adathalmaz hasznos legyen a közösség számára, tisztában kell lennünk az abban található adatok alapvető jellemzőivel. Ezért elemzést végeztünk a PyBugHive-ban található hibákon annak megállapítására, mennyire "triviálisak". Egy hibát akkor tekintünk triviálisnak, ha azt megfelelő eszközökkel gyorsan és emberi beavatkozás nélkül automatikusan lehet észlelni. A triviális hibák könnyen gyűjthetők nagy mennyiségben is, de korlátozott felhasználhatósággal rendelkeznek az ipari és kutatási közösségek számára. Ezért biztosítani akartuk, hogy a PyBugHive nem triviális hibákat tartalmazzon.

Setup

Két széles körben használt statikus elemző eszközt alkalmaztunk hibakeresőként, nevezetesen a Pylint¹ és Bandit² eszközök. Ezeket az eszközöket széles körben használják mind az ipari, mind az akadémiai világban [14, 19]. Egy hibát triviálisnak tekintettünk, ha azt legalább az egyik statikus elemző eszközzel ki lehetett mutatni.

Mindkét eszköz esetében az elemzési módszer ugyanaz volt. Egyszerű szkriptet hoztunk létre a futtatásukhoz, először a hibát tartalmazó forráskódon, majd a javított verzión. A szkript végigiterált az adatbázis minden problémáján, klónozte a megfelelő projektet, majd kiválasztotta a projekt hibás verzióját. Ezután egymás után futtatatta a statikus elemző eszközöket a hibás fájlokon, majd megismételte ugyanezt, a projekt javított verziójával.

Az elemzések után összehasonlítottuk a kimeneteket mindkét verzióra és ellenőriztük, hogy a javított forráskód eltávolította-e az előzőleg a statikus elemzők által jelzett problémákat. A Pylint esetében az eszköz warning-okat jelzett, míg Bandit a hibás sorokat jelentett, így ezeket az eredményeket mentettük és automatizált módon hasonlítottuk össze. A hibás és javított verziók eredményeinek különbségét rögzítettük, így csak a különbségeket kellett ellenőrizni. Ezeket az összegyűjtött eredményeket manuálisan ellenőriztük. A Pylint és a Bandit futtatásához további beállításokra nem volt szükség, csak meg kellett adnunk a megfelelő fájl(ok) elérési útvonalát.

Eredmények

Az összehasonlítás során azt találtuk, hogy az elemzők egyik bugot sem találták meg. A Bandit esetében csak néhány olyan eset volt, ahol a jelzett hibák megváltoztak a javítás előtt és után is, de még ezek az esetek sem találták meg a konkrét hibát. A Pylint esetében csak kisebb különbségek voltak az eredményekben, de ezek a jelzett hibák nem hivatkoztak valós hibákra.

Egy ilyen jelentés-pár látható a 4.1 kódrészletben. Az első sor a hibáról szóló figyelmeztetés a javítás előtt, a második sor pedig a figyelmeztetés a javítás után. Ahogy látható, csak a számok változtak, de ezenkívül a Pylint jelentése pontosan ugyanaz a hibajavítás

¹<https://www.pylint.org>

²<https://bandit.readthedocs.io>

előtt és után. Tehát a javítás nem orvosolta a korábban jelzett figyelmeztetést. Így ez azt is jelenti, hogy a Pylint nem találta meg a hibát.

```
Too many branches (13/12) (too-many-branches)
Too many branches (15/12) (too-many-branches)
```

Kódrészlet 4.1. Pylint által adott jelentések

4.3. Limitációk

Olyan projekteket választottunk ki, amelyek különböző területeket fednek le, sok commitot tartalmaznak, és népszerűek a GitHub-on, de minél több projektet adunk hozzá a datasetünkhöz annál robosztusabbá válik. Így tervezzük, hogy folyamatosan bővítjük és új verziókat adunk ki az adatbázisunkból.

Kihagyott hibák

Bár a projektek rengeteg hibát tartalmaznak, csak egy részüket vettük bele, ennek oka főként a reprodukálhatóság hiánya volt. Sok olyan hiba volt, amely az adott projektjük régi verzióiban jelent meg és a függőségeik nem szerezhetők be automatikusan. Egy megoldás lehetett volna a függőségek kézi beszerzése (például kézi fordítás és telepítés), de ez veszélyeztette volna az adathalmazunk reprodukálhatóságát és rontotta volna a projektek függőségeinek telepítési idejét.

Továbbá a csak Python 2-ben fellelhető hibákat is figyelmen kívül hagytuk. Kizárólag a Python 3 projektekre összpontosítottunk, mivel a Python 2.7 hivatalosan már nincs támogatva³. Így ezen hibák valószínűleg már nem lesznek a jövőbeni kutatások középpontjában.

Meglepő módon néhány olyan hibába ütköztünk, ahol a javított verzió tesztjei a hibát tartalmazó verzióban is hibátlanul futottak. Ennek természetesen nem szabadna megtörténnie. Az anomália kivizsgálása érdekében több virtuális környezetet és több tesztkeretrendszert használtunk, de mindegyik ugyanazt az eredményt hozta, ami így ezen hibák kihagyásához vezetett.

³<https://www.python.org/doc/sunset-python-2/>

Volt néhány olyan hiba is, ahol a tesztek minden futtatásnál különböző eredményeket produkáltak (race condition), ezeket a hibákat sem célszerű felhasználnunk, így ezeket az eseteket is kihagytuk.

Összességében, bár a hibák egy jelentős részét ki kellett hagynunk a felmerült problémák miatt, még mindig sok potenciális hibát tervezünk ellenőrizni a jövőben. Mivel az adatbázisunk bővítése egyszerű, a rendszeres frissítés lehetséges és a jövőbeli terveink között is szerepel.

Platform

Bár a PyBugHive platformfüggetlen, érdemes megjegyezni, hogy az adatbázisunkban szereplő néhány projekt kifejezetten Linux környezetre van tervezve. Ezért az adathalmazunkat Linux környezetben (Ubuntu) hoztuk létre, mivel a benne szereplő projektek általában Linux operációs rendszeren voltak tesztelve, és a fejlesztők is csak ezt az operációs rendszert veszik figyelembe. Ezért a hibák reprodukálási folyamata elsősorban a Linux platformra összpontosult.

Ezen felül néhány projekt olyan függőségeket igényelt, amelyek csak Linux platformon elérhetők. A PyBugHive API-ja továbbra is alkalmazható különböző platformokon (például statikus elemzési célokra), azonban egyes projektek tesztjei nem biztos, hogy teljes mértékben működnek nem-Linux környezetekben.

5. fejezet

Használati eset

Ebben a fejezetben bemutatjuk a PyBugHive használatát egy olyan használati eset révén, amely egy nagy nyelvi modellt, a GPT-3.5¹-öt használja. A Nagy Nyelvi Modelleket (LLM) egyre több területen használják.

Az LLM-eket már tanulmányozták a szoftverfejlesztés területén [22, 30, 15, 13], ezért mi is elhatároztuk, hogy egy kísérletet végzünk a GPT-3.5 segítségével a hibaadatbázisunkban található hibákon.

A GPT, azaz a Generative Pre-trained Transformer [3, 25], az OpenAI által fejlesztett Nagy Nyelvi Modell (LLM) család. A Nagy Nyelvi Modellek mesterséges intelligencia modellek, amelyek különféle mesterséges intelligencia technikákat, kifejezetten mélytanulást [18] és neurális hálókat [2], alkalmaznak emberi szövegek megértésére és generálására.

A GPT egy egyszerű API-n keresztül érhető el, amelyet könnyedén használhatunk egy Python könyvtár segítségével². Ebben a fejezetben a GPT-3.5 API-ra az egyszerűség kedvéért *modell*-ként, *AI*-ként is hivatkozunk.

¹A pontos verzió a *gpt-3.5-turbo-16k* volt, további információkért lásd: <https://openai.com/blog/openai-api>

²<https://pypi.org/project/openai>

5.1. Hibakeresés

Előkészület

Először arra kértük a modellt, hogy viselkedjen úgy, mint egy tapasztalt fejlesztő, aki alapos kódfelülvizsgálatot (code review-t) végez. Végül négy olyan promptot választottunk, amelyek arra összpontosítanak, hogy a GPT-3.5 képes-e megtalálni a hibákat a kódban. Mivel az AI válaszai nem mindig következetesek, ezek a promptok többé-kevésbé azonosak, de kisebb változtatásokkal. Mindegyik megkérdezi, hogy van-e valamilyen jelentős hiba a kódban, de különböző utasításokat ad arra, hogyan válaszoljon (például csak egy *yes* vagy *no* válasszal, vagy esetleg adjon részletes magyarázatot is). A négy konkrét prompt, amit használtunk:

- **Prompt 1:** „Do you think the following code contains any major bugs? You must give a definitive answer.”
- **Prompt 2:** „Do you think the following code contains any major bugs? You have to start your answer with a yes or no, indicating the contents of your answer!”
- **Prompt 3:** „Do you think the following code contains any major bugs? Your answer must be a single word, which can only be yes or no!”
- **Prompt 4:** „Do you think the following code contains any major bugs? You have to start your answer with a yes or no, and provide a detailed list of the steps you took to come to your conclusion.”

A promptok véglegesítése után meghatároztuk a kiértékelési folyamatot. A kiértékelés során megállapítottuk, hány alkalommal válaszolt a modell *igen*nel a különböző hibakereső prompt-okra. Ez a folyamat félig automatikusan történt, mivel manuálisan is átnéztük az összes GPT-3.5 választ, és megvizsgáltuk, hogy azok valós hibákat tekintve relevánsak-e.

Eredmények

A 5.1 táblázat projektek szerint csoportosítva összefoglalja a GPT-3.5 által elért eredményeket. A fent említett négy különböző prompt-ot használtunk arra, hogy a modelltől

5.1. táblázat. Prompt-ok eredményei

Projekt neve	Bugok száma	Prompt 1		Prompt 2		Prompt 3		Prompt 4		Legalább 1	Több	Magyarázat
		Yes	No	Yes	No	Yes	No	Yes	No	Yes	Yes	
black	38	0	38	10	28	0	38	9	29	17	2	0
cookiecutter	2	0	2	1	1	0	2	1	1	1	1	1
discord.py	2	0	2	0	2	0	2	1	1	1	0	0
freqtrade	15	1	14	4	11	1	14	5	10	10	1	2
numpy	1	0	1	0	1	0	1	0	1	0	0	0
pandas	16	0	16	4	12	1	15	5	11	8	2	1
poetry	11	0	11	1	10	0	11	4	7	5	0	0
salt	7	0	7	1	6	0	7	5	2	5	1	0
scrapy	2	0	2	1	1	0	2	1	1	1	1	1
spaCy	11	1	10	2	9	1	10	8	3	8	3	2
Σ	105	2	103	24	81	3	102	39	66	56	11	7

megkérdezzük, talált-e hibát a megadott forráskódban. A táblázat négy oszlopa azt mutatja, hány alkalommal adott GPT-3.5 *yes* (a kód hibás) vagy *no* (a kód nem tartalmaz hibát) választ egy adott prompt-ra. A következő oszlop megmutatja azoknak az eseteknek a számát, amikor a modell legalább az egyik prompt-ra *yes* választ adott. Az utolsó előtti oszlopban felsoroljuk azoknak az eseteknek a számát, amikor több mint egy prompt-ra adott *yes* választ. Végül az utolsó oszlop azt mutatja, hány esetben volt képes a modell magyarázatot adni az észlelt hibára. Ahhoz, hogy eldöntsük, hogy magyarázat helyes-e vagy sem, manuálisan elemeztük a válaszokat.

Ahogy látható, az 1. és a 3. prompt nagyon kevés *yes* választ eredményezett. Ezek nem voltak hatékonyak a hibák észlelésére. A 2. és különösen a 4. prompt már jobban teljesített. A 4. esetben a modell szignifikánsan több *yes* választ adott. Amennyiben a minimum egy *yes* választ számoljuk, akkor 56 bug-ot sikerült megtalálnia a modellnek. Azonban a modell csak 11 esetben észlelte a hibás kódot legalább 2 prompt segítségével. Ebből csupán 7 esetben volt képes megmagyarázni az észlelt hibát (ebbe beleértve a részben helyes magyarázatokat is).

Fontos megjegyezni, hogy a modell soha nem válaszolt *yes* válasszal mind a 4 prompt-ra. Az #1612-es issue esetében a modell 3 prompt-ra is *yes* választ adott. Az #1612-es issue esetében a modell által adott magyarázat a következő: *"Overall, the code contains syntax errors, missing parameters, and inconsistent method signatures. The deprecation message for the 'upper_' property could also be clearer."*

Ez arra utal, hogy a Python dokumentáció szerint a `__get__` metódusnak további két paraméterre van szüksége, az *instance* és az *owner*. Azonban ebben a hibában a fájl

Cython-t is használ, ahol ez nem kötelező.

A hiba leírása³ és a spaCy API dokumentációja⁴ alapján a `text` és `orth_` értékeinek azonosnak kellene lenniük. Azonban a bugos implementáció nem felelt meg ennek a követelménynek, ahogyan azt a 5.1 is szemlélteti. A modell magyarázata így helytelen. A hibás kódrészletet a 5.2. kódrészletben látható.

```
>>> import spacy

>>> nlp = spacy.load('en_core_web_sm')
>>> doc = nlp('The black cat purrs.')
>>> span = doc[1: 3]
>>> span.text
'black cat'
>>> span.orth_
'blackcat'
```

Kódrészlet 5.1. A #1612-es issue-ban található bug előidézése³

```
property orth_:
    """Verbatim text content (identical to Span.text).
    Exists mostly for consistency with other attributes.
    RETURNS (unicode): The span's text."""
    def __get__(self):
        return ''.join([t.orth_ for t in self]).strip()
```

Kódrészlet 5.2. #1612-es issue-ban található bug részlete

Összességében a GPT-3.5 a 105 hibából, 56-ot (53.3%) képes volt megtalálni, legalább az egyik prompt-ra adott *yes* válasszal. Azonban csak 11 esetben (10.5%) válaszolt több, mint egy prompt-ra *yes* válasszal, és csak 7 esetben (6.7%) volt képes magyarázatot adni az észlelt hibára. A GPT-3.5 ígéretes eredményeket mutat a hibák észlelésében, azonban még nem tekinthető annyira megbízhatónak, hogy a gyakorlatban is használható legyen.

³<https://github.com/explosion/spaCy/issues/1612>

⁴<https://spacy.io/api/span#attributes>

5.2. Hibajavítás

Előkészület

Ezt követően a GPT-3.5 -öt arra kértük, hogy viselkedjen úgy, mintha egy tapasztalt fejlesztő lenne, aki a forráskód hibáit szeretné kijavítani.

Kezdetben hat promptot használtunk: kettő közülük ugyanazt a kérdést tette fel, mint amit a hibakeresés során használtunk, de azt is elmondta a modellnek, hogy javítsa ki azokat. Három prompt nem tett fel kérdést, csak annyit mondott a mesterséges intelligenciának, hogy javítsa ki a hibákat és magyarázza meg a javításának lépéseit. Az utolsó prompt azt mondta a mesterséges intelligenciának, hogy egy másik programozó már talált hibákat a kódban és létrehozott egy GitHub issue-t a hiba címével és a GPT-3.5 feladata az, hogy kijavítsa azt. Próbálgatások után finomhangoltuk azt a prompt-ot, amely a legjobb eredményeket hozta. A végleges prompt, amelyet az adatbázis teljes értékeléséhez használtunk, a következő volt:

- **Prompt:** „Do you think the following code contains any major bugs? If so, you have to provide a fixed code. You must return the entire code with your fixes included. Do not include the explanation in your answer, you must provide the source code only (without any formatting). If you think that the code does not contain any bugs, please return the original source code I have provided. The other expert reported this bug and gave the following description: {issue címe}.”

A prompt pontosan meghatározza, hogyan kell viselkednie a modellnek, és mit kell visszaadnia, még akkor is, ha a GPT-3.5 úgy gondolja, hogy a kód nem hibás. Azonban a GPT-3.5 sokszor nem ezeknek a kéréseknek megfelelően válaszolt és sok esetben megváltoztatta a visszaadott kód indentálását.

A kérések során csak a hibás metódusokat küldtük el, de ez lehet, hogy nem elegendő a modell számára ahhoz, hogy megértse a problémát. Bizonyára a GPT-3.5 jobb eredményeket érhetne el, ha rendelkezne az összes releváns kódrészlettel, az összes sikertelen teszttel, vagy akár az egész projekttel. Jövőbeli terveink közt szerepel ezek további vizsgálata.

Eredmények

Projekt	Issuek száma	Sikertelen	Nincs változás	Kisebb változás	Részleges javítás	Teljes javítás
black	38	19	15	4	0	0
cookiecutter	2	0	1	1	0	0
discord.py	2	2	0	0	0	0
freqtrade	15	11	2	1	1	0
numpy	1	1	0	0	0	0
pandas	16	8	8	0	0	0
poetry	11	9	2	0	0	0
salt	7	3	4	0	0	0
scrapy	2	2	0	0	0	0
spaCy	11	6	4	0	0	1
Σ	105	61	36	6	1	1

5.2. táblázat. Hibajavítás GPT-3.5 API-val

A GPT-3.5-öt használtuk az adatbázisunkban található hibák kijavítására. A 5.2 táblázat projekt szerint csoportosítva mutatja be a hibajavítás eredményeit. Ahogy látható, a legtöbb javítási kísérlet eredménytelen volt. Ez azt jelenti, hogy a javítások vagy egyáltalán nem voltak valós javítások (például a modell csak törölte a megjegyzéseket, módosította az indentálást stb.), vagy értelmetlenek voltak (például a *return* megváltoztatása *return None*-ra).

Meglepő módon olyan esetek is előfordultak, ahol a modell, annak ellenére, hogy a prompt kijelentette, hogy a kód biztosan hibás, arra a következtetésre jutott, hogy nincsenek hibák, és a prompt-nak megfelelően egyszerűen visszaadta az eredeti kódot. Ezeket az eseteket a `No change` címkével jelöltük

Összesen csak 6 javítás esetén értünk el `Kisebb változás`-t. Csak egy esetben sikerült részleges javítást és szintén egy esetben értünk el teljes javítást.

6. fejezet

Következtetések és jövőbeli tervek

Bár a Python népszerűsége folyamatosan nőtt az utóbbi években, a Python nyelvű projektek körében végzett kísérletek száma nem növekedett hasonló mértékben. Ennek többek között valószínűleg az az oka, hogy a kutatók rendelkezésére álló benchmarkok és adathalmazok száma nagyon csekély.

Ezen probléma enyhítésére létrehoztuk a PyBugHive nevű, gondosan összeállított hibaadatbázist, ami reprodukálható, Python nyelvű hibákat tartalmaz. Az adatbázis 105 valós, manuálisan validált, reprodukálható hibát tartalmaz. Ezek 10 különböző területről származó Python projektből erednek. Biztosítunk egy parancssoros felhasználói felületet, a PyBugHive-ot, amely lehetőséget nyújt a tesztesetek végrehajtásához, a projektek klónozásához és azok specifikus verzióinak kipróbálásához. Ez biztosítja, hogy a felhasználók könnyen hozzáférhetnek a projektek hibás és javított verzióihoz. A keretrendszerünk könnyedén alkalmazható például automatizált programjavításra, hibalokalizációra, hiba-előrejelzésre.

Elemzéseink azt mutatták, hogy az általunk gyűjtött hibák nem triviálisak, a statikus elemzők egyiket sem tudták megtalálni. Bemutattunk egy valós életbeli alkalmazást az elkészült adathalmazon, amely az egyik legnépszerűbb nagy nyelvmodellen, a GPT-3.5 -ön alapul.

Először értékeltük a modell hibafelismerő képességeit több különböző prompt használatával. Eredményeink azt mutatták, hogy a modell legalább egy *yes* választ adott több mint a hibák felére. Azonban egynél több *yes* válasz ritka volt, összesen csak 7 esetben (6,7 %) sikerült a GPT-3.5 -nek helyes magyarázatot adnia. A hibajavítás során a legtöbb

esetben a modell nem tudott valós javítást adni a hibás kódhoz. A GPT-3.5 egy esetben részleges javítást, egy másikban szemantikailag helyes javítást készített.

A GPT számos lehetőséget rejt még, azonban a modellek jelenlegi fejlettsége még nem eredményez teljesen megbízható működést. További céljaink között szerepel a kísérlet GPT-4-el és a GPT-4 Turbóval történő megismétlése és a prompt-ok tökéletesítése.

Jelenlegi és jövőbeli terveink magában foglalják a PyBugHive bővítését, további alrendszerek hozzáadásával (természetesen további hibákkal történő bővítéssel). Új hibák hozzáadása mellett tervezzük a BugsInPy és a QuixBugs integrálását is az adatbázisunkba.

Publikáció

A bemutatott munkánkból egy nemzetközi Q1-es folyóiratcikk (IEEE Access) is készült, amelyet a bírálók elfogadtak, jelenleg megjelenés alatt van.

Irodalomjegyzék

- [1] Elena N Akimova, Alexander Yu Bersenev, Artem A Deikov, Konstantin S Kobylkin, Anton V Konygin, Ilya P Mezentsev, and Vladimir E Misilov. Pytracebugs: A large python code dataset for supervised machine learning in software defect prediction. In *2021 28th Asia-Pacific Software Engineering Conference (APSEC)*, pages 141–151. IEEE, 2021.
- [2] Chris M Bishop. Neural networks and their applications. *Review of scientific instruments*, 65(6):1803–1832, 1994.
- [3] Robert Dale. Gpt-3: What’s it good for? *Natural Language Engineering*, 27(1):113–118, 2021.
- [4] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10:405–435, 2005.
- [5] Alexandru Dura, Christoph Reichenbach, and Emma Söderberg. Javadl: automatically incrementalizing java bug pattern detection. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–31, 2021.
- [6] Thomas Durieux and Rui Abreu. Critical review of bugswarm for fault localization and program repair. *arXiv preprint arXiv:1905.09375*, 2019.
- [7] Thomas Durieux and Martin Monperrus. Introclassjava: A benchmark of 297 small and buggy java programs. 2016.
- [8] Gregory Gay and René Just. Defects4j as a challenge case for the search-based software engineering community. In *Search-Based Software Engineering: 12th In-*

- ternational Symposium, SSBSE 2020, Bari, Italy, October 7–8, 2020, Proceedings* 12, pages 255–261. Springer, 2020.
- [9] Andrew Habib and Michael Pradel. How many of all bugs do we find? a study of static bug detectors. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 317–328, 2018.
- [10] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments on the effectiveness of dataflow-and control-flow-based test adequacy criteria. In *Proceedings of 16th International conference on Software engineering*, pages 191–200. IEEE, 1994.
- [11] Jiajun Jiang, Yingfei Xiong, and Xin Xia. A manual inspection of defects4j bugs and its implications for automatic program repair. *Science china information sciences*, 62:1–16, 2019.
- [12] René Just, Darioush Jalali, and Michael D Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*, pages 437–440, 2014.
- [13] Junaed Younus Khan and Gias Uddin. Automatic code documentation generation using gpt-3. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–6, 2022.
- [14] Divya K Konoor, Rajyalakshmi Marathu, and Prashanth Reddy. Secure openstack cloud with bandit. In *2016 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, pages 178–181. IEEE, 2016.
- [15] Márk Lajkó, Dániel Horváth, Viktor Csuvik, and László Vidács. Fine-tuning gpt-2 to patch programs, is it worth it? In *International Conference on Computational Science and Its Applications*, pages 79–91. Springer, 2022.
- [16] Ryan Lawler. How do you hire great engineers? give them a challenge. <https://web.archive.org/web/20160305163006/https://gigaom.com/2012/01/19/quixey-challenge/>, 2012.

- [17] Claire Le Goues, Neal Holtschulte, Edward K Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. The manybugs and introclass benchmarks for automated repair of c programs. *IEEE Transactions on Software Engineering*, 41(12):1236–1256, 2015.
- [18] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.
- [19] Suryadiputra Liawatimena, Harco Leslie Hendric Spits Warnars, Agung Trisetiyarso, Edi Abdurahman, Benfano Soewito, Antoni Wibowo, Ford Lumban Gaol, and Bahtiar Saleh Abbas. Django web framework software metrics measurement using radon and pylint. In *2018 Indonesian Association for Pattern Recognition International Conference (INAPR)*, pages 218–222. IEEE, 2018.
- [20] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN international conference on systems, programming, languages, and applications: software for humanity*, pages 55–56, 2017.
- [21] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. Avatar: Fixing semantic bugs with fix patterns of static analysis violations. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 1–12. IEEE, 2019.
- [22] Stephen MacNeil, Andrew Tran, Dan Mogil, Seth Bernstein, Erin Ross, and Ziheng Huang. Generating diverse code explanations using the gpt-3 large language model. In *Proceedings of the 2022 ACM Conference on International Computing Education Research-Volume 2*, pages 37–39, 2022.
- [23] Parvez Mahbub, Ohiduzzaman Shuvo, and Mohammad Masudur Rahman. Defectors: A large, diverse python dataset for defect prediction. *arXiv preprint arXiv:2303.04738*, 2023.

- [24] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset. *Empirical Software Engineering*, 22:1936–1964, 2017.
- [25] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.
- [26] Mohammad Rezaalipour and Carlo A Furia. An empirical study of fault localization in python programs. *arXiv preprint arXiv:2305.19834*, 2023.
- [27] Ripon K Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul R Prasad. Bugs. jar: A large-scale, diverse dataset of real-world java bugs. In *Proceedings of the 15th international conference on mining software repositories*, pages 10–13, 2018.
- [28] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? *ACM sigsoft software engineering notes*, 30(4):1–5, 2005.
- [29] Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and Marcelo de Almeida Maia. Dissection of a bug dataset: Anatomy of 395 patches from defects4j. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 130–140. IEEE, 2018.
- [30] Nigar M Shafiq Surameery and Mohammed Y Shakor. Use chat gpt to solve programming bugs. *International Journal of Information Technology & Computer Engineering (IJITC) ISSN: 2455-5290*, 3(01):17–22, 2023.
- [31] David A Tomassi, Naji Dmeiri, Yichen Wang, Antara Bhowmick, Yen-Chuan Liu, Premkumar T Devanbu, Bogdan Vasilescu, and Cindy Rubio-González. Bugswarm: Mining and continuously growing a dataset of reproducible failures and fixes. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 339–349. IEEE, 2019.
- [32] Béla Vancsics, Péter Gyimesi, Andrea Stocco, Davood Mazinanian, Arpád Beszédes, Rudolf Ferenc, and Ali Mesbah. Poster: Supporting javascript experimentation

- with bugsjs. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 375–378. IEEE, 2019.
- [33] Béla Vancsics, Attila Szatmári, and Árpád Beszédes. Relationship between the effectiveness of spectrum-based fault localization and bug-fix types in javascript programs. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 308–319. IEEE, 2020.
- [34] Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, et al. Bugsinpy: A database of existing bugs in python programs to enable controlled testing and debugging studies. In *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pages 1556–1560, 2020.
- [35] Chi Zhang, Yu Wang, and Linzhang Wang. Infrared: A meta bug detector. *arXiv preprint arXiv:2209.08510*, 2022.
- [36] Hao-Nan Zhu and Cindy Rubio-González. On the reproducibility of software defect datasets. *ICSE. IEEE*, 2023.