TDK dolgozat

Készítette:
Szolnoki Norbert Sándor

Szegedi Tudományegyetem Természettudományi és Informatikai Kar

Informatikai Intézet

Szoftverfejlesztés Tanszék

Kódduplikáció kereső algoritmusok fejlesztése és összehasonlítása Pythonban

Készítette:

Szolnoki Norbert Sándor

programtervező informatikus

II. évf. MSc hallgató

Témavezető:

Dr. Antal Gábor

adjunktus

Szeged

2024

Absztrakt

A dolgozat célja, hogy vizsgálja, mennyire találjuk fel újra a kereket a szoftverfejlesztés területén, különös tekintettel a Python programozási nyelv esetére. A kutatás során több tucat könyvtárat vizsgálunk meg, és elemzés alá vetjük, hogy ezek a könyvtárak tartalmazzák-e az álltalunk megadott példakódokat valamilyen formában. Fontos megjegyezni, hogy a vizsgálat fontos részét képzik, hogy a talált funkció működése egyezzen / hasonlítson az álltalunk megadott funkcióval.

A szoftveres megoldás öt fő különböző elemzésből áll. Minden esetben a bemenet egy példafunkció és könyvtárak, amelyekben keresünk. Az elemzési megoldásaink a következőek:

Chat GPT 4: Egy szoftveresen generált promptban elküldjük a rendelkezésre álló adatokat és elemezzük a választ.

PDG (Program Dependence Graph): programfüggőség gráfmodell segítségével elemezzük a bemenetet és feldolgozzuk az eredményeket.

AST egyezés elemzés: Absztrakt szintaxis fa elemzést végzünk a bemenetben adott könyvtárak funkcióin, ahol egyezést keresünk az álltalunk megadott mintakódhoz.

CodeBert(Graphcodebert): Graphcodebert előtanított tokenizálását felhasználva feldolgozzuk a bemenetet és a talált egyezéseket visszaadjuk.

Előtanított modell (saját fejlesztés): Egy modellt előre betanítottam a könyvtárakban található funkciókkal, majd a megadott példakódot ugyan azon tokenizálás után elkezdi keresni a modellben és visszaadja a találatokat.

A kimenet egy lista lesz, amely tartalmazza a könyvtár nevét, ahol egyezést találtunk, valamint a találati arányt, továbbá a megtalált kódot a könyvtárból. A kutatás eredményei felvázolják, hogy milyen módokon tudunk hatékonyan egyező funkcionalitású kódot keresni és egy lehetséges megoldást nyújtani, amely képes a fejlesztő figyelmét felhívni arra, hogy mi az a funkció, amelyet már egy könyvtár leimplementált. Ezzel a megoldással a kódminőséget lehet javítani, hisz egy sokak által használt könyvtár nagyobb eséllyel biztonságos, mint egy álltalunk újraírt

implementáció.

Tartalomjegyzék

1. fejezet

Bevezetés

Kódunk elemzése és olyan könyvtárak felkutatása, amelyek már megvalósíthatják az általunk írt funkciókat, lehetne a szoftverfejlesztésünk kulcsfontosságú feladata is, hiszen ezzel ha elsőre nem is látjuk, de időt, energiát takaríthat meg nekünk, és a szoftverünk minőségén is tud segíteni.

Gondoljunk bele, hogy megannyi nyilvános szabadon használható könyvtár található meg az interneten Pythonhoz például a PyPI rendszerében [1], amelyekhez bárki hozzáférhet és feltöltheti akár a saját könyvtárát is. Nem biztos, hogy ismerjük az összes funkciót, amely az itt megtalálható könyvtárakban van, és ezért megpróbáljuk magunk lefejleszteni a számunkra szükséges funkciókat. Ha a biztonsági oldalát vizsgáljuk a szoftverfejlesztésnek, akkor könnyen beláthatjuk, hogy a mai digitális környezetben, ahol a kiberfenyegetések nagy számban jelen vannak [2], és az adatszivárgások [3] elszomorítóan gyakoriak, szoftverünk biztonsága rendkívül fontos. A közösség álltal fejlesztett könyvtárak használatával akár meg tudjuk óvni az alkalmazzásunk az XSS-től (Cross-site scripting) [4], SQL injection-től [5] és egyéb biztonsági résektől is. A kódelemzés és a könyvtárak felülvizsgálása alapvetően hozzájárul a szoftverrendszereink hosszú távú karbantartásához és fejlesztéséhez [6], hiszen a későbbeikben bemutatott kódelemzési technikákat bármikor alkalmazni lehet a szoftvereinken. A szoftver természeténél fogva sosincs kész, ahogy ezt a Software evolution [7] című könyvben M. Lehman is leírta, a rendszereknek folyamatosan fejlődniük kell ahhoz, hogy alkalmazkodni tudjanak a változó környezethez, így megértve, hogy a szoftver evolúciója elkerülhetetlen.

Dolgozatunk főbb eredményei:

- Szakirodalom alapján implementáltunk néhány kereső algoritmuist.
- Ezen algoritmusok eredményeit összehasonlítottuk.
- Készítettünk egy Visual Studio Code [8] plugint, ami szabadon konfigurálható és innentől kezdve az IDE tud segíteni, hogy esetleg egy könyvtárból már tudunk használni ilyen funkciót.

A kutatásunkban az alábbi kérdésekre keressünk a válaszokat:

- Milyen kereső algoritmusokat tudunk alkalmazni kódduplikáció keresésére?
- Hogyan tudunk az általunk megírt funkcióhoz hasonló vagy azonos működésű kódot találni a népszerű Python könyvtárakban?
- Mi történik, ha a mi kódunk másképp van megírva, mint ahogy a könyvtárban lévő kódnak?

Tesztjeink a szakirodalom [9] által is elismert első három és egy saját típus mentén végeztük, hogy validáljuk az eredényeink.

Számos algoritmus létezik kódok elemzésére a szöveges szintaktikai elemzéstől kezdve egészen a funkcionalitást megértő modell alapú keresésig. Kutatásunkban fény derült arra, hogy ha jól implementáljuk ezeket az algoritmusokat, akkor egy teljesen megváltoztatott kódtörzsű funkcióhoz is képesek vagyunk találni könyvtárban már implementált funkciót. Azonban nem mindegyik algoritmus képes konstans megbízható találatokkal szolgálni például ha a szintaktika változik. Az általunk vizsgált és implementált kereső algoritmusok az alábbiak voltak:

- Abstract Syntax Tree (AST) [10]
- CodeBert [11]
- Program Dependency Tree (PDG) [12]
- GPT-4 [13]
- TF-IDF vektormodell tanítása és alkalmazása [14]

Fontosnak tartottuk, hogy ne csak egy algoritmust használjunk és értékeljünk ki, mert így betekintést nyerünk az algoritmusok előnyeibe és hátrányaiba miközben összehasonlítjuk őket.

Ha ki kellene emelni egyet az implementációk közül egyet is, az a TF-IDF vektormodell lenne, mert amellett, hogy minden vizsgált típusban 60% felett teljesített, a futási ideje kimagaslóan gyors volt a többihez viszonyítva.

A dolgozatunk felépítése az alábbi: a 2. fejezetben a témához kapcsolódó munkák kapcsolatát ismertetjünk , a 3. fejezetben a kutatásunk módszertanát. A keresési algoritmusok működéséről és az általunk implemetált megoldások leírása a 3. fejezetben kaptak helyet, később az 4 fejezet az eredményeket tárgyalja. Az 5. fejezetben kitérünk az egyik meglepő eredmény miértjére, majd 6. fejezet az eredményeink helyességét veszélyeztető tényezőket vizsgálja és végezetül a 7 fejezetben az összegzés zárja a tanulmányt, az elkészült Visual Studio Code pluginről pedig képernyőfotók találhatóak a Függelékben.

2. fejezet

Kapcsolódó munkák

Tesztjeink az Andreas P. Koenzen és társai által megfogalmazott Code Duplication and Reuse in Jupyter Notebooks című [9] tudományos kutatásukban leírt kódduplikációs első három típusok követve és egy sajátot definiálva végeztük el:

- Type 1: Ezek az eredeti kódrészletek pontos másolatai, kivéve a szóközök és a megjegyzések eltéréseit. Ez a típus a sokszorosítás legegyszerűbb formáját képviseli, ahol a kódot minden változtatás nélkül másolják és illesztik be, kivéve az esetleges formázási változtatásokat.

- Type 2: Ezek szintaktikailag azonos kódmásolatokat tartalmaznak, ahol csak a felhasználó által meghatározott azonosítók, például a változónevek módosultak. Ez a típus megtartja a kód szerkezetét és funkcióját, de módosítja az azonosítókat, hogy lehetőleg illeszkedjenek az új használat környezetéhez.

- Type 3: Ezek a 2. típusú klónok módosított másolatai, amelyekhez utasításokat adnak hozzá, eltávolítanak vagy módosítanak. A 3-as típusú klónok 2-es típusú klónoknak tekinthetők, néhány olyan változtatással a kódban, amelyek túlmutatnak az azonosítók átnevezésén.

- Own Type: Ez a kategória olyan kódrészleteket tartalmaz, amelynek a neve megegyező az eredetivel, de a törzse, argumentumnevei eltérnek valamint a visszaadott érték is különbözik az eredetitől. Ezzel a saját típussal szeretném bemutatni, hogy ezek az algoritmusok egyes esetekben tökéletes eredményt tudnak adni, de amikor a funkcionalitást módosul, akkor képes hibás választ adni.

Peter Bulychev és Marius Minea [15] cikkükben a kódduplikációkat vizsgálják meglévő szoftverekben, és sajnos az eredményeik azt mutatják, hogy a kódduplikációk 6,4%-tól 20%-ig terjedhetnek átlagosan.

Az eredmények azt mutatták, hogy a PDG alapú detektálás segítségével jelentősen növekedett a detektálás pontossága és megbízhatósága a hagyományos szintaktikai összehasonlításokkal szemben, erről H. Nasirloo és F. Azimzadeh [16] írt a tanulmányukban, ahol arra is fény derült, hogy ez a módszer hatékonyan hatékonyabban képes azonosítani az úgynevezett "deep clones"-t is, amely a szakirodalom negyedik típusú kódduplikációjába tartozik. Ebben a kutatásban a két féle módon próbálták detektálni a kódduplikációkat, az egyik a program működése alapján, amely pontos de lassabb mint a másik módszer, ami a memória állapotot vizsgálta. Az implementációnk az első tehát a program működése alapján kereste a duplikációt. Ha ugyan azt a két mérőszámot használjuk, mint H. Nasirloo és F. Azimzadeh, akkor ugyan arra a megállapodásra jutunk, hisz a mi implementációnk is hatékonyan talál duplikációt, de a futásideje inkább a középmezőny végébe tartozik.

A CodeBERT egy következő megoldás, amelyet alkalmaztunk duplikációk megtalálására, azonban ezt a keresési/elemzési módszert nem kifejezetten erre szokták használni, de mivel jól működik programhibák előrejelzésére, így az analízise a módszernek képes egyezést is találni. C. Pan és társai [17] a CodeBERT modellt használja szoftverhibák előrejelzésére. A tanulmány különböző projektek közötti keresztvalidációs modelleket alkalmaz, hogy megvizsgálja, hogyan javíthatja a CodeBERT a hibaelőrejelzési teljesítményt.

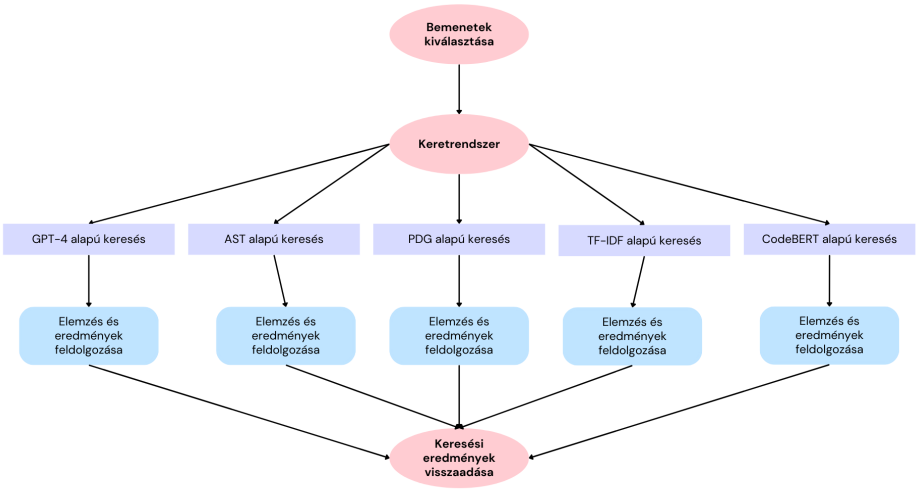
A "A new approach to web application security: Utilizing gpt language models for source code inspection" [18] című cikk, amelyet Szabó Zoltán és Bilicki Vilmos írt, használja a GPT-4 modellt a hasonló funkcióval rendelkező kódok keresésére, fókuszálva különösen a webalkalmazások biztonsági sebezhetőségeinek felismerésére. A cikk részletesen bemutatja, hogy a GPT-4 hogyan képes értelmezni a kódot a sebezhetőségek azonosítására. A modell képességét kihasználva, a kutatás kiemeli, hogy a GPT-4 hatékonyan képes felismerni és összehasonlítani a kódokat a sebezhetőségi minták alapján, amelyek azonos funkciót hajtanak végre, de eltérő biztonsági kockázatokat jelentenek. A Szabó Zoltán és Bilicki Vilmos által készített kutatáshoz nagyon hasonló a mi imlpementációnk is, csak mi Python nyelvvel dolgozunk és más promptot használtunk, valamint más képpen validáltuk az eredményeket.

A TF-IDF módszert alkalmazza a hasonló funkcióval rendelkező kódok keresésére, "Similar words identification using naive and tf-idf method" [19] című tanulmány, amelyet KS Divya,

R Subha és S Palaniswami írtak. A kutatás a TF-IDF módszer alkalmazásával foglalkozik, hogy azonosítsa a hasonló szavakat és kifejezéseket kódokban, ami alapvetően kapcsolódik a kód duplikáció detektálásának problémájához. A kutatás eredményeiből kiderül, hogy a TF-IDF alapú módszer átlagosan 31%-os pontossággal képes egyezésket találni. Ezt az eredményt nekünk sikerült javítani, hisz a mi eredményeinkben 60% fölött teljesített az ezen alapú implementáció.

3. fejezet

Módszertan



3.1. ábra. Folyamatábra

A folyamat öt fő részre bontható. Először kerestük algoritmusokat, amelyek képesek kódok elemzésére és egyezést képes találni vizsgált funkciók között és implementáltuk őket, ezután unit teszteket készítettünk egy szkriptel, ami véletlenszerűen választott egy funkciót minden könyvtárból, hogy minden féle funkciót kerestessünk az algoritmusokkal. Következő lépésben 4 különböző típust generáltunk a tesztesetekből, ezután egy szkript segítségével lefuttattuk az összes tesztesetet az összes algoritmussal, majd végül kiértékeltük az eredményeket.

3.1. Felhasznált keresési algoritmusok működése és megvalósításaink

Természetesen több algoritmus létezik, amely képes kódduplikációt találni, mi ezeket választottuk, hogy legyen néhány népszerű megközelítéshez egy-egy példa. Két dologban minden implementációnk egyezik, az egyik a várt argumentumok, amely nem más mint a keresett funkció, a könyvtárak listája, amelyekben keresünk és egy küszöbszám, amely meghatározta a minimum találati arányt. A másik pedig a keresés eredmény visszaadásának módja, amely minden esetben egy JSON objektum, hogy össze tudjuk őket fűzni az elemzéshez. Összesen 43 darab könyvtárat vontunk be a tesztekbe, és ez a 43 könyvtár 1233 funkciót tartalmaz.

3.1.1. GPT-n alapuló módszer

Promptokat küldhetünk a GPT-nek és a válasz feldolgozásával tudunk kódduplikációkat keresni a promptban megadott könyvtárakhoz.

Megvalósításunk

A bemeneti argumentumokból programatikusan egy promptot generáltunk, amely tartalmazott statikus részeket is, amely azt a célt hivatott szolgálni, hogy megpróbáljuk minél pontosabb eredményre ösztönözni a GPT-t a keresésben. Hogy még sikeresebb legyen a kérésünk, Angol nyelven írtam meg, mert azt tapasztaltam, hogy Magyarul, nem mindig érti meg jól az utasításokat.

3.1. Kódrészlet. Prompt készítő funkció

```
def create_prompt(function_code, libraries, similarity_threshold):
        library_list = ", ".join(libraries)
        prompt = f"""
    Given the following Python function:
    {function_code}
    Please suggest Python functions from each of the following libraries: {library_list} ↩
    that work similarly to the function above.
    It can have different args, different body, but might does the same thing.
    I know the function I provided is very specific, but just give it a try.
    Your response should be a JSON list containing objects with the following keys:
    - library_name: Name of the library
    - similarity_percentage: Similarity percentage of the suggested function round to two ↩
    decimials and add % to the end
    - function_code: Code of the suggested function from the library
```

```
    The similarity percentage should be between {similarity_threshold} and 100, and you can ↩
    provide multiple matches from a single library.
    Only return the json, I don't need nothing else, don't write any comment on your own, ↩
    just json!
    """
        return prompt
```

Ahogy az látható, a statikus szöveg már magában hosszú, mert ebbe még behelyettesítődik a keresett funkció és a könyvtárak listája. Ez egy probléma forrás volt a tesztek során, amelyet ki kellett küszöbölni azzal, hogy egyszerre csak 5 könyvtárat adtunk meg mindig a listából, így egy egy teszthez kilencszer kellett megkérdezni a GPT-t.

3.1.2. Abstract Syntax Tree

Az AST, vagy Absztrakt Szintaxisfa, egy fastruktúra, amely reprezentálja a programkód szintaktikai szerkezetét. Minden csomópont a kód egy elemét képviseli, például utasításokat, kifejezéseket, változókat stb. Az AST segítségével a kódelemzés során könnyen összehasonlíthatóvá válnak a különböző kódok, azonosítva a hasonlóságokat és a különbségeket.
A kódduplikáció keresésére az AST alapú módszerek általában a következő lépéseket követik:

- 1. AST Generálás: Első lépésben az adott forráskódból AST-t generálunk.
- 2. AST Transzformáció: Az AST-t esetleg transzformáljuk, hogy eltávolítsuk a szemantikailag jelentéktelen elemeket, mint például kommentek és felesleges szóközöket.
- 3. Minta-Keresés: Az AST-ben mintákat keresünk, amelyek gyakran ismétlődnek a keresett funkcióban és a könyvtárból nyert funkcióban.
- 4. Összehasonlítás: Az egyező minták összehasonlításra kerülnek az AST-ben, és ha egy bizonyos hasonlósági küszöböt meghaladják, akkor azokat duplikátumnak tekintjük.

Megvalósításunk

Az implementáció az argumentumban kapott funkció kódját megtisztítja és elkezd végig iterálni a szintén argumentumban kapott könyvtárlistán. Vannak olyan könyvtárnevek, amelyeknek más az importálási és telepítési neve, például a pyjwt -> jwt, ezekben az esetekben átírjuk a nevet, hogy a továbbiakban probléma nélkül tudjuk importálni az adott könyvtárat és vizsgálni a funkcióit. Az iteráció a könyvtárak nevein megy végig, és először dinamikusan

importálja a könyvtárat, majd kinyeri az összes hívható funkciót az adott könyvtárból. Amint megvan az összes funkció, ezeken a funkciókon iterálunk végig egy belső for ciklussal, ahol normalizáljuk a funkció kódját, eltávolítjuk a kommenteket, hogy a bemenetként kapott funkció kódjával közel azonos szintre hozzuk. Amint megvan a normalizálás, a difflib könyvtár sequenceMatcher [20] funkcióját használjuk az összehasonlításra, mert ha az elkészített AST-ket szerializáljuk, akkor ezen a szöveges reprezentáción könnyen és hatékonyan el tudja végezni az összehasonlítást az input funkció absztrakt szintaxis fáján és a for cuklusban éppen soron lévő funkció szintaxis fáján. Ha az összehasonlítás mértéke legalább megegyezik az inputként adott küszöbszámmal, akkor egy listába tesszük a funkció és könyvtár adatait a találat százalékával. Amikor leelenőriztük az összes könyvtár összes funkcióját, akkor a találati listánkon szereplő három legmagasabb százalékot elért funkciót adjuk vissza.

3.1.3. TF-IDF saját betanított modell

A TF-IDF (Term Frequency-Inverse Document Frequency) egy statisztikai módszer, amely segítségével megállapítható, mennyire fontos egy szó vagy token egy dokumentumhoz vagy szövegkorpushoz képest. Ez a módszer különösen hasznos szövegfeldolgozási és információkeresési feladatokban, mint például a kódduplikáció keresése a szoftverfejlesztés területén. Alább részletezem a TF-IDF módszer alkalmazását és lépéseit a kódduplikáció keresésére.

A TF-IDF módszer lépései kódduplikáció keresésére:

- Kód Tokenizálása: A forráskódokat elemekre, úgynevezett tokeneke bontjuk, mint például változók, függvények, operátorok és egyéb szintaktikai elemek. Ez a lépés előkészíti a kód szövegét a TF-IDF elemzéshez.

- Term Frequency (TF) Számítása: A TF megmutatja, hogy egy adott token hányszor fordul elő egy dokumentumban (jelen esetben egy kódfájlban). Ez segít kiemelni azokat a tokeneket, amelyek gyakran jelennek meg egy adott kódrészletben.

- Inverse Document Frequency (IDF) Számítása: Az IDF értéke azt mutatja, hogy mennyire ritka vagy egyedi egy token az összes vizsgált dokumentumon (kódfájlon) belül. A ritkább tokenek magasabb IDF értékkel rendelkeznek, ami azt jelzi, hogy jelentősebbek lehetnek a kódszerkezet szempontjából.

- TF-IDF Számítás: Az egyes tokenek TF-IDF értékét úgy kapjuk meg, hogy a token TF

értékét megszorozzuk az IDF értékével. A magasabb TF-IDF értékű tokenek fontosabbak a kódban, és segíthetnek a hasonló funkciókat ellátó kódrészletek azonosításában.

- Kódrészletek Hasonlóságának Mérése: A kódrészletek TF-IDF vektorait használva mérhetjük a hasonlóságot különböző algoritmusokkal, például a koszinusz hasonlósággal. A hasonló TF-IDF vektorok azt jelzik, hogy a kódrészletek hasonló szerepet tölthetnek be, még ha szövegileg nem is egyeznek meg pontosan.

Megvalósításunk

Dolgozatom készítésekor nem találtam az interneten olyan kutatást, ahol TF-IDF alapú vektorizálással készít modellt és azt a modellt használja, hogy kódduplikációt keressen. Ezért hívom ezt a szekciót saját betanított modellnek, és ezért említem úgymond saját fejlesztésnek ezt az implementációt. Tehát először be kell tanítanunk egy modell-t a könyvtárak funkcióival, amelyek rendelkezésre állnak. Ezt az alábi funkció látja el.
Tanítás lépései:

- Argumetumban kapott könyvtárakból kinyerjük az összes funkciót egy listába
- Egy új listába extraháljuk a funkciók kódsorait
- SkLearn TfidfVectorizer segítségével transzformációt illesztünk a funkciók kódjaira
- Lokálisan elmentjük a modelt, hogy később könnyedén hozzáférjünk

Az utolsó lépés amely elmenti a modelt egy fájlba, felelős az algoritmus rentkívül gyors működéséért.

Amikor a modellben keresünk egyező kódot, akkor vektorizáljuk az argumentumban kapott funkciót és hasonlósági százalék határzunk meg a koszinusz hasonlóság [21] segítségével, amely a két vektor álltal bezárt szög meghatározásával tud nekünk segíteni meghatározni a vizsgált funkciók közötti hasonlóság mértékét. A három legmagasabb hasonlósági százalékot kapott funkciót adjuk vissza eredményül.

3.1.4. CodeBERT

A CodeBERT modell a BERT (Bidirectional Encoder Representations from Transformers) architektúrát alkalmazza, amelyet programkódok elemzésére optimalizáltak. A modell előtaní-

tása során nagy mennyiségű forráskódot dolgoznak fel, így a modell képes megragadni a kód szintaktikai és szemantikai mintázatait.

Kódduplikáció detektálási folyamat alkalmazása:

- Kód Tokenizálása: A CodeBERT saját tokenizálót használ, amely képes felismerni a programozási nyelvek kulcsszavait és szerkezeti elemeit.
- Kódrészletek Átalakítása: A tokenizált kódrészleteket átalakítják a CodeBERT bemenetére alkalmas formátumra, amely tartalmazza a tokeneket, pozíciós jelzőket és szegmentálási jeleket.
- Vektorképzés: A CodeBERT a kódrészleteket vektorokká alakítja, amelyek tükrözik a kód strukturális és tartalmi jellemzőit.
- Hasonlóság Mérés: A generált vektorok alapján különböző hasonlósági metrikák (pl. koszinusz hasonlóság) segítségével összehasonlíthatjuk a kódrészleteket, hogy azonosítsuk a potenciális duplikációkat.

Megvalósításunk

Az implementáció fő funkciója először a tokenizálási eszközt állítja be és tokenizálja a bemenetben kapott funkciót, mivel ezt csak egyszer kell megtegyük az egész teszt alatt. Ezen kívül engedélyezi a CUDA használatát. Tesztjeim során észrevettem, hogy hiába egy Core i9 10900K processzoron futtatom, sajnos elég lassú, így feltelepítettem a függőségeket [22] és beállítottam, hogy a CUDA-val használja a gépben lévő RTX 3070 kártyát. Ezzel a módosítással sok időt nyertem a tesztek alatt. Tokenizálónak és modell-nek a Microsoft GraphCodeBert-base fájljait használtam, mert ezeket már előtanították. Először a szükséges elemző eszközök és modellek betöltésével kezdődik a folyamat(graphcodebert-base [11] tokenizer és model). Ezeket a modelleket kifejezetten programkódok elemzésére fejlesztették ki, és képesek a kód szemantikai és szintaktikai jellemzőit vektoros formában reprezentálni. A kód feldolgozása során dinamikusan importáljuk a kívánt könyvtárakat és lekérjük az összes definiált függvényt, amelyeket később elemzésre használjuk.A következő lépésben a bemeneti kódrészletet tokenizáljuk és kódoljuk. A kódolás során a modellt a CUDA eszközre helyezük át, így kihasználva a GPU számítási kapacitását. Minden egyes könyvtári függvényt hasonló módon tokenizálunk és kódolunk, majd a bemeneti kódrészlet vektoros reprezentációját összehasonlítjuk minden egyes könyvtári függvény vektoros reprezentációjával. A hasonlóságot a koszinusz hasonlóság [21] segítségével

határozzuk meg, amely a két vektor közötti szög alapján számolja ki, hogy mennyire hasonlóak a kódok. Amennyiben a hasonlósági érték meghaladja a megadott küszöbértéket, a funkció hozzáadja a hasonló függvények részleteit egy eredménylistához. Majd a legvégén a három legmagasabb hasonlósági számot elért funkciót adjuk vissza eredményül.

3.1.5. Programfüggőségi gráf (PDG)

A Program Dependence Graph (PDG) egy hatékony eszköz a kódduplikáció detektálására, amely lehetővé teszi a kódrészletek közötti függőségi viszonyok és szerkezeti hasonlóságok azonosítását. A PDG a program különböző utasításai közötti adat- és vezérlési függőségeket ábrázolja grafikon formájában, amely segít felismerni a hasonló funkciójú, de különböző módon megírt kódrészleteket.

A PDG alapú kódduplikáció detektálás folyamata az alábbi lépésekből áll:

- PDG Generálása: Minden vizsgált kódrészlethez PDG-t generálnak, amely ábrázolja az egyes kódelemek közötti függőségeket.
- Gráfok Összehasonlítása: A PDG-k összehasonlításával meghatározható, hogy két kódrészlet mennyire hasonló. A hasonlóság mérésére különböző algoritmusokat használnak, mint például a szubgráf-illeszkedés vagy gráfizomorfizmus.
- Klónozott Részletek Azonosítása: Amennyiben a PDG-k között jelentős hasonlóság van, az azt jelzi, hogy a kódrészletek klónozottak lehetnek. Ezt követően részletes elemzés történik a potenciális duplikációk pontos jellemzőinek meghatározására.

Megvalósításunk

Az algoritmusunk a következő lépéseket hajtja végre:

- Kód normalizálása: A kódrészleten először formázási javításokat végezünk, ami így egységesíti a kód stílusát, ami fontos a szintaxisfa (AST) pontos feldolgozásához.
- Szintaxisfa elemzése: A normalizált kódot az ast.parse függvény segítségével elemzik, amely egy AST-t (Abstract Syntax Tree) hoz létre. Ez a fa reprezentálja a kód szerkezetét elemi részekre bontva, mint például függvénydefiníciók, változó hozzárendelések és függvényhívások, ez a lépés szükséges a PDG elkészítéséhez.

- PDG Visitor osztály: Egy saját NodeVisitor osztályt definiáltunk, amely az AST csomópontjait járja be. Ez az osztály felelős a PDG felépítéséért. A PDG-ben minden csomópont egy kódelemet jelöl (például változókat vagy függvényeket), míg az élek az elemek közötti függőségeket (például egy változó egy függvényen belüli használatát) képviselik.
- Függvénydefiníciók kezelése: Amikor egy függvénydefiníciót találunk, azt hozzáadjuk a PDG-hez, és bejárjuk a függvényen belüli további utasításokat.
- Hozzárendelések kezelése: A változó hozzárendeléseknél hozzáadjuk a változókat a PDG-hez, és élekkel jelöljük, hogy ezek a változók mely függvényekben vannak definiálva.
- Változó használatának kezelése: Amikor egy változót "betöltés" kontextusban használnak (azaz olvasnak belőle), az aktuális függvénytől az adott változóhoz élt hozunk létre, jelezve, hogy a változót a függvény használja.
- Függvényhívások kezelése: A függvényhívások esetén élekkel jelöljük, hogy az aktuális függvény mely más függvényeket hívja meg.
- Gráf hasonlóságának vizsgálata:
  - Gráf Izomorfizmus Alapú Hasonlóság: a NetworkX könyvtár DiGraphMatcher osztályát használjuk, amely lehetővé teszi két irányított gráf összehasonlítását. Az izomorfizmus vizsgálat arra szolgál, hogy megállapítsa, vajon a két gráf strukturálisan azonos-e. Ebben az esetben az összehasonlítás két szinten történik:
    - Csomópontok egyezése: A csomópontok egyezését a 'type' attribútum alapján vizsgálják. Ez azt jelenti, hogy csak az azonos típusú (például 'function' vagy 'variable') csomópontokat tekintik azonosnak.
    - Élek egyezése: Az élek egyezését szintén a 'type' attribútum határozza meg (például 'defines', 'used_by', 'calls').
  - Jaccard Index Alapú Hasonlóság: a Jaccard-indexet használjuk a két PDG hasonlóságának mérésére. Ez egy statisztikai módszer, amelyet széles körben alkalmaznak a különbségek és az átfedések mérésére különböző adathalmazok között:
    - Csomópontok és élek átalakítása: A függvény először hash-képessé alakítja a csomópontokat és éleket, ami lehetővé teszi azok összehasonlítását.

    - Metszet és unió kiszámítása: Kiszámítja a két gráf csomópontjainak és éleinek metszetét és unióját.
    - Jaccard-index számítása: A Jaccard-index a metszet és az unió arányaként számítódik. Ez az arány azt mutatja meg, hogy mennyire hasonlóak a gráfok, az eredmény pedig 0 (teljesen eltérő gráfok) és 1 (azonos gráfok) között mozog.

Ahogy a többi esetben, itt is a három legmasabb százalékot elért funkciót adjuk vissza válaszul.

3.2. Tesztesetek kinyerése

Összesen 43 könyvtárt használtunk a tesztekre, és minden könyvtárból véletlenszerűen egy szkript álltal választottunk egy funkciót, ezt tekintjük saját implementációnak, és ezt vagy ennek a módosított változatát kerestetjük az algoritmusokkal. A szkript működése során végigment az összes felsorolt könyvtáron, először lekérte a hívható publikus funkciókat, majd random választott egyet. A választott funkciót egy tömbbe tette, majd expotkor JSON fájlba tette, hogy egyszerűen legyen használható. A JSON listában található funkciókról az alábbi adatokat tároljuk:

- „library”: a könyvtár neve, amely tartalmazza a funkciót
- „function”: a funkció neve amit keresünk
- „code”: a funkció kódja

3.3. Teszttípusok elkészítése

Négy különböző tesztesetet vizsgáltunk, ezek a következőek voltak:

Első típusú teszteset

Ebben a tesztben nem változtatunk a teszteseteken.

```
{
  "library": "xlrd",
  "function": "cellname",
  "code": "def cellname(rowx, colx):\n    ↩
    return '%s%d' % (colname(colx), rowx + 1)↩
    "
}
```

3.2. Kódrészlet. Eredeti funkció

Type 2 típusú teszteset

Változó nevek és funkció név módosításával szintaxisbeli változást eszközöltem, de a funkcionalitáson ezek nem változtatnak, ez álltal szimuláltunk egy esetet, ami arra hasonlít, amikor valaki saját maga készít egy funkciót, és rákeres, hogy létezik már funkció ezzel a működéssel bármelyik könyvtárban.

```
{
    "library": "xlrd",
    "function": "cellname",
    "code": "def cellname(rowx, colx):\n    ↩
    return '%s%d' % (colname(colx), rowx + 1)↩
    "
}
```

3.3. Kódrészlet. Eredeti funkció

```
{
    "library": "xlrd",
    "function": "cell_identifier",
    "code": "def cell_identifier(↩
    row_identifier, col_identifier):\n    ↩
    return '%s%d' % (colname(col_identifier),↩
     row_identifier + 1)"
}
```

3.4. Kódrészlet. Type 2-re módosított funkció

Type 3 típusú teszteset

A kód törzse különbözik az eredetitől, például beraktunk pár véletlenszerű változót, meghívunk funkciókat, de maga a visszaadott érték nem változik.

```
{
    "library": "xlrd",
    "function": "cellname",
    "code": "def cellname(rowx, colx):\n    ↩
    return '%s%d' % (colname(colx), rowx + 1)↩
    "
}
```

3.5. Kódrészlet. Eredeti funkció

```
{
       "library": "xlrd",
       "function": "cellname",
       "code": "def cellname(rowx, colx):\n ↩
      if rowx < 0 or colx < 0:\n        ↩
    return \"Invalid input. Row and column ↩
    indices must be non-negative.\"\n    else↩
    :\n        return '%s%d' % (colname(colx)↩
    , rowx + 1)"
    }
```

3.6. Kódrészlet. Type 3-ra módosított funkció

Own type típusú teszteset

Ebben a tesztesetben megváltoztattunk a kód törzsét, és más eredményt is ad vissza mint az eredeti, hogy megnézzük, mennyire érzékeny fals eredmények adására.

```
{
  "library": "xlrd",
  "function": "cellname",
  "code": "def cellname(rowx, colx):\n    ↩
    return '%s%d' % (colname(colx), rowx + 1)←
    "
}
```

3.7. Kódrészlet. Eredeti funkció

```
{
  "library": "xlrd",
        "function": "cellname",
        "code": "def cellname(rowx, colx):\n ↩
       return f\"{colname(colx)}{rowx + 1}\""
}
```

3.8. Kódrészlet. Own type-ra módosított funkció

3.4. Tesztek futtatása és kiértékelés

A tesztek futtatását automatikusan végeztük, az implementáció során hibatűrő megoldásra törekedtem, ezzel felkészítve az éles környezetben való használatra, hogy ne akadjon meg ha az egyik könyvtárban például problémát talál. A kiértékelést szintén egy szkript végezte, amely végigment az összes elmentett JSON eredménylistán. Mivel a tesztesetek sorrendje minden esetben egyezett, így tudtuk követni az alap tesztesetlistát a validáció során és képes volt megjelölni ha az adott algoritmus megtalálta-e azt a funkciót amit éppen keresünk, attól függetlenül, hogy a funkció neve módosítva lett, mivel mi a könyvtár nevét és a funkció nevét kerestük az eredménylistában.

3.9. Kódrészlet. Eredménylista részlet

```
1  {
2      "arrow": {
3          "original_data": {
4              "library": "arrow",
5              "function": "get",
6              "code": "def get(*args: Any, **kwargs: Any) ->Arrow:\n     \n ↩
       return _factory.get(*args, **kwargs)\n"
7          },
8          "results": [
9              {
10                 "function_name": "Chat_GPT_4_results",
11                 "result": [
```

```
                    {
                        "library_name": "arrow",
                        "similarity_percentage": "80%",
                        "function_code": "def get(*args, **kwargs):\n    ↩
    return arrow.get(*args, **kwargs)"
                    }
                ],
                "time_consumed": 4.585026025772095
            },
            {
                "function_name": "PDG_results",
                "result": [
                    {
                        "library_name": "bokeh",
                        "similarity_percentage": "50.0%",
                        "function_code": "def license():\n    ''' Print ↩
    the Bokeh license to the console.\n\n    Returns:\n        None\n\n ↩
       '''\n    from pathlib import Path\n    with open(Path(__file__).↩
    parent / 'LICENSE.txt') as lic:\n        print(lic.read())\n"
                    }
                ],
                "time_consumed": 8.568641662597657
            },
            {
                "function_name": "CodeBert_results",
                "result": [
                    {
                        "library_name": "arrow",
                        "similarity_percentage": "94.06%",
                        "function_code": "def get(*args: Any, **kwargs: ↩
    Any) -> Arrow:\n    \"\"\"Calls the default :class:`ArrowFactory <↩
    arrow.factory.ArrowFactory>` ``get`` method.\"\"\"\n\n    return ↩
    _factory.get(*args, **kwargs)\n"
                    }
                ],
                "time_consumed": 27.415658473968507
            },
            {
                "function_name": "AST-based-similarity_results",
```

```
                "result": [
                    {
                        "library_name": "arrow",
                        "similarity": "97.25%",
                        "similar_function": "def get(*args: Any, **kwargs:↩
    Any) -> Arrow:\n    \"\"\"Calls the default :class:`ArrowFactory <↩
    arrow.factory.ArrowFactory>` ``get`` method.\"\"\"\n\n    return ↩
    _factory.get(*args, **kwargs)\n"
                    }
                ],
                "time_consumed": 3.466275691986084
            },
            {
                "function_name": "Pre-trained-model_results",
                "result": [
                    {
                        "library_name": "arrow",
                        "similarity": "73.65%",
                        "similar_function": "def get(*args: Any, **kwargs:↩
    Any) -> Arrow:\n    \"\"\"Calls the default :class:`ArrowFactory <↩
    arrow.factory.ArrowFactory>` ``get`` method.\"\"\"\n\n    return ↩
    _factory.get(*args, **kwargs)\n"
                    }
                ],
                "time_consumed": 0.003999471664428711
            }
        ]
    }
}
```


3.5. Visual Studio Code plugin

Készítettünk egy Visual Studio Code plugint, amely kliensként szolgál, és készítettünk egy szervert, ahol az AST alapú és a TF-IDF modell alapú keresési módszereket lehet használni és kódduplikációt keresni a megadott könyvtárakban. A kliens szabadon módosítható beállításai a következőek:

- API endpoint URL, ez lehetővé teszi, hogy céges környezetben akár belső hálózaton is

használjuk

- Keybinding, szükséges billentyűkombináció a keresés indítására
- Search Type, ez lehet AI model, ami a TF-IDF és az AST based search
- Match Threshold, a találat minimális százaléka
- Used libraires, a keresésre használandó könyvtárak

A used libraries beállításhoz, meg szeretném jegyezni, hogy ha egy olyan könyvtárat adunk meg, amely még nem szerepelt eddig a globális listáján a szervernek, akkor újratanítja a frissített listával a modell-t.

Ez a plugin lehetővé teszi, hogy fejlesztés közben vagy refaktorálásnál leelenőrizzük, hogy nincs-e az adott funkció már megvalósítva, beállítás után csak ki kell jelölni a keresendő funkciót és mnegnyomni a beállított billentyűkombinációt, majd a kliens a Visual Studio Code Codelens API-t használva a kijelölt funkció fölött megmutatja a könyvtár nevét, és a találat százalékát, ha pedig rákattintunk, akkor jobb oldalt lenn megjelenik a funkció kódja is, ahonnan másolhatjuk is.

Jelenleg nem érhető el publikusan a plugin és a szerver, de terveim között szerepel a publikálása, ezzel segítve fejlesztőtársaim. Képernyőképeket a beállításokról és a használatról csatoltam a Függelék-be.

4. fejezet

Eredmények

4.1. Eredmények összesítése

Tesztjeink során 43 könyvtár 1233 funkciója között kerestük azt amelyikhez viszonyítva a mi funkciónk duplikáció valamely típusába sorolható. Az alábbi táblázatban látható a teszt eredmények összesítése, amely megmutatja, hogy hány esetben tudott az algoritmus helyes megoldással visszatérni. Az Own type esetében ahogy korábban tárgyaltuk, a funkciók neve megegyezik, de a működésük során eltérés található, vagy csak más dolgot ad vissza mint az eredeti. Ez bizonyos keresési módszerekben normális, hiszen a PDG gráfokat készít és azokat hasonlít össze, ha a torzítás mérete a funkción nem módosítja a gráfot, akkor fals pozitív eredményt fog adni. AST alapú módszer is képes fals pozitív jelzést adni, ha a torzítás mértéke nem haladja meg azt a volumnet, amely még nem tolja az azonossági százalékot a küszöbérték alá. Megfigyelhető, hogy bizonyos algoritmusok nagyon magas százalékban voltak képesek megtalálni a keresett funkciót, például a CodeBert majdnem minden esetben képes volt megtalálni a keresett funkciót, bár a futási ideje minden esetben a legrosszabb volt.

Az AST alapú keresés ahogy azt vártuk, az egyes típusban tökéletesen működött, alig hibázott, ami hiba történt is, betudhatjuk az adatok tisztitása során módosulhatott szintaxisnak, mivel a többi típusban mindig változott a szintaktika, így teljesen elvárt volt, hogy elmaradjon az elsőtől. Mivel a CodeBert nem kódösszehasonlításra lett tervezve, így az átlagos 55%-60%-os találati arány jónak mondható. A Program Függőségi Gráf (PDG) eredményeit a funkció és argumentum módosítás tudta meggátolni a kimagasló eredményekben, az első és harmadik típusban nagyon jól teljesített, de sajnos a másodikban 50% alatt maradt. A TF-IDF vektorokkal előtanított modell nagyon szépen dolgozott minden esetben, és mindemellett a futásideje sosem

érte el a fél másodpercet.

4.1. táblázat. Algoritmusok teszteredményei

Algoritmus	**Összes pozitív találat**	**Teljes futásidő (s)**
	Type 1	
Chat_GPT_4_results	2 (4.65 %)	12.48
CodeBert_results	26 (60.46 %)	1171.40
AST-based-similarity_results	40 (93.02 %)	429.31
Pre-trained-model_results	30 (69.76 %)	0.13
PDG_results	37 (86.04 %)	956.80
	Type 2	
Chat_GPT_4_results	4 (9.3 %)	99.85
CodeBert_results	27 (62.79 %)	1199.87
AST-based-similarity_results	36 (83.72 %)	469.45
Pre-trained-model_results	29 (67.44 %)	0.14
PDG_results	16 (37.20 %)	929.96
	Type 3	
Chat_GPT_4_results	2 (4.65 %)	64.85
CodeBert_results	23 (53.48 %)	1169.23
AST-based-similarity_results	33 (76.74 %)	493.98
Pre-trained-model_results	29 (67.44 %)	0.14
PDG_results	33 (76.74 %)	938.61
	Own type	
Chat_GPT_4_results	1 (2.32 %)	66.16
CodeBert_results	23 (53.48 %)	1190.43
AST-based-similarity_results	26 (60.46 %)	419.91
Pre-trained-model_results	29 (67.44 %)	0.16
PDG_results	30 (69.76 %)	898.08

Válasz a "Milyen kereső algoritmusokat tudunk alkalmazni kódduplikáció keresésére?": Több féle megközelítéssel működő algoritmus megfelel a feladatra:

- Szemtaika alapú keresésre az AST(Abstract Syntax Tree)
- Gráf alapú keresésre a PDG (Program Dependency Graph)
- AI alapúra a GPT-4, manuálisan akár a GPT-3.5
- Előtanított gráf alapú modellre a CodeBERT
- Saját magunk által tanított modellre TF-IDF vektorizálással tanított modell

Válasz a "Hogyan tudunk az általunk megírt funkciókhoz hasonló vagy azonos működésű kódot találni a népszerű Python könyvtárakban?" kérdésre: Bizonyos esetekben egyszerű szemantikai elemzéssel is tudunk találni megegyező vagy hasonló funkciót egy konyvtárban, de ez valószínűleg kevés esetben történne meg a való életben, hisz ha már másképpen nevezünk el egy változót, az már rontja a keresési találati eredményt. Legtöbször a funkciók működésének valamilyen általános reprezenátciójával tudunk nagyobb valószínűséggel a minden napokban azonos működésű funkciót találni.

Válasz a "Mi történik ha a mi kódunk másképp van megírvva mint ahogy a könyvtárban lévő kódnak?" kérdésre: Abban az esetben ha a funkcionalitás identikus, csak a szintaxis más, akkor a PDG, CodeBert és a TF-IDF alapon működő keresés is nagy valószínűséggel segítségünkre lehet.

5. fejezet

Diszkusszió

5.1. GPT-4 alacsony találati arányáról

Sajnos a GPT-4 promptom nem működött olyan jól mint azt vártuk, de az álltala ajánlott funkciók nem lettek letesztelve csak egy példát kiemelve, a GPT-4 által válaszul adottra:

5.1. Kódrészlet. Type 4 kód

```
1
2     "geopy": {
3         "original_data": {
4             "library": "geopy",
5             "function": "get_version",
6             "code": "def get_version():\n    return __version__\n"
7         },
8         "results": [
9             {
10                "function_name": "Chat_GPT_4_results",
11                "result": [
12                    {
13                        "library_name": "django",
14                        "similarity_percentage": "80%",
15                        "function_code": "def get_version():\n    from ↩
    django import get_version\n    return get_version()"
16                    },
17                    {
18                        "library_name": "toml",
19                        "similarity_percentage": "80%",
```

```
                        "function_code": "def get_version():\n    from ↩
    toml import __version__\n    return __version__"
                    },
                    {
                        "library_name": "pandas",
                        "similarity_percentage": "80%",
                        "function_code": "def show_versions(as_json=False)↩
    : \n return pd.show_versions(as_json=as_json)"
                    },
                    {
                        "library_name": "altair",
                        "similarity_percentage": "80%",
                        "function_code": "altair.__version__"
                    },
                    {
                        "library_name": "seaborn",
                        "similarity_percentage": "80%",
                        "function_code": "seaborn.__version__"
                    }
                ],
                "time_consumed": 6.830019950866699
            }
    ]
    }
```


Látható, hogy talált "get_version" nevű funkciót, de sajnos nem a geopy könyvtárban, lehetséges hibaforrás, hogy ott van a listán a geopy is, de mivel a hasonlósági százalékot nem tudta jól megadni, így kiesett a listából, amelyet visszaadott nekünk, valamint mivel nem történt kézzel való validálás a tesztesetek nagy száma miatt, így lehetséges, hogy a GPT-4 eredmény szintén ugyan úgy működő funkcióval tért vissza mint a keresett funkció.

6. fejezet

Eredményeinket veszélyeztető tényezők

6.1. Belső valódiság

6.1.1. Tesztesetek kiválasztása

Mind a 43 könyvtárból egy-egy darab funkciót választpttunk a tesztjeinkhez, ezeket pedig script végezte, amely teljesen véletlenszerűen választott, ezzel elkerülve a választási torzítást.

6.1.2. Eredmény torzulás újra tesztelés során

Mivel a tesztjeinkben már nem volt egy random funkció sem, így kizárt az eredmények torzulása, ha a tesztet újra és újra futtatjuk, ez alól kivételt képez a GPT-4 implementáció, mert van, amikor a mesterséges intelligencia egy értelmezhetetlen választ ad, de ebben az esetben van amikor megoldást nyújt az újra kérdezés.

6.2. Külső valódiság

6.2.1. Implementációk általánosítása

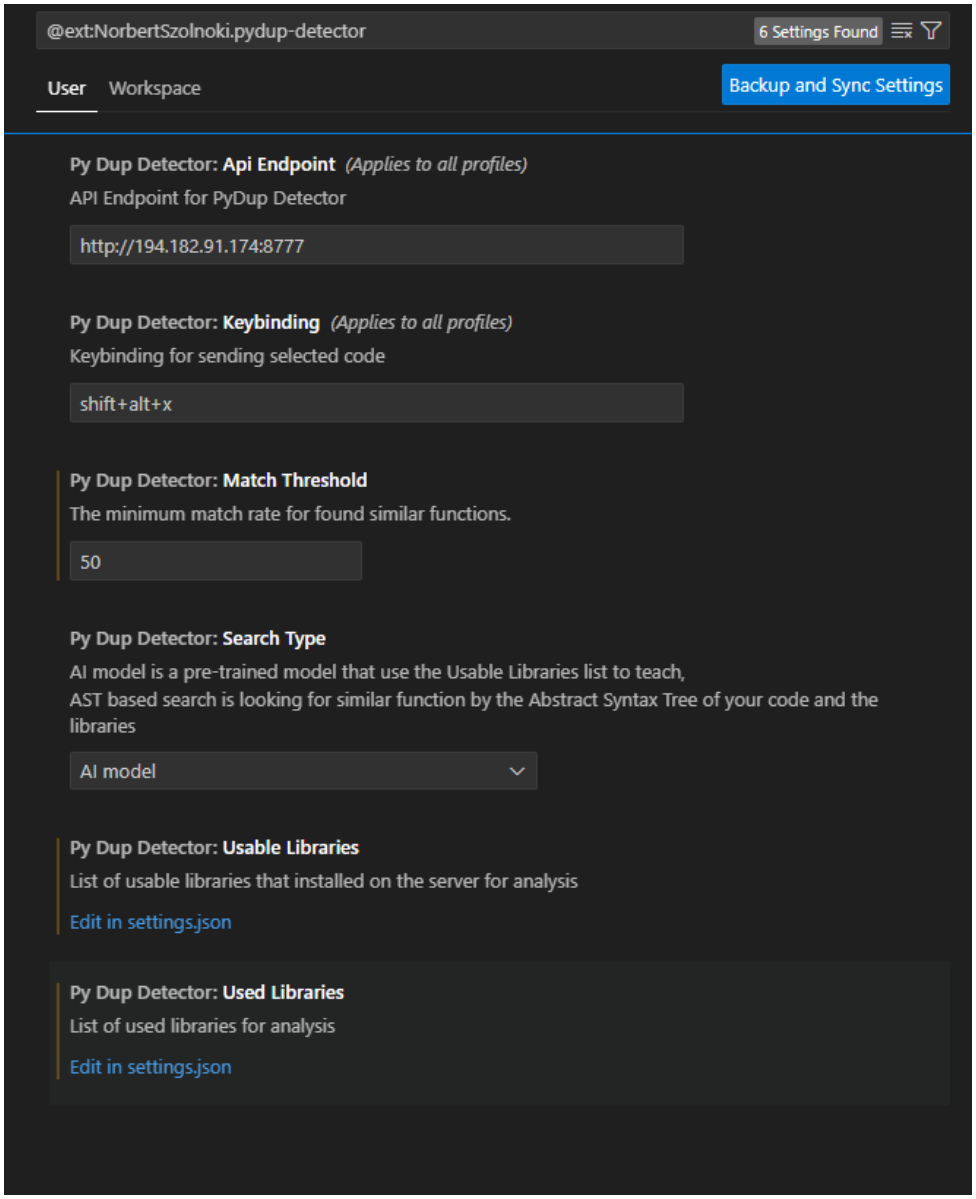
Magabiztosan állíthatjuk, hogy természetesen probléma nélkül működnie kell más könyvtárakon is, hisz a fejlesztés és tesztelés során a kiválasztott 43 könyvtár egy is véletlenszerűen lett kiválasztva az 50 legtöbbet használt Python könyvtár közül.

7. fejezet

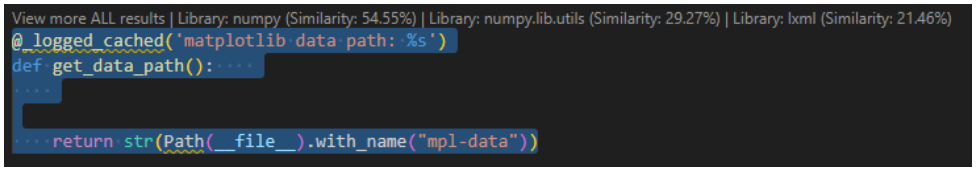
Összegzés

Tapasztalaltaink alapján az elkészített algoritmusok képesek hatékonyan kódduplikációkat keresni, amelyek nem feltétlen rendelkeznek identikus törzzsel, elegendő az, hogy milyen funkciót lát el, és ehhez képesek találni olyan funkciót, ami ugyan ezt a funkcionalitást képviselik. Legfonosabb mérőszámoknak a keresésre fordított időt és a pozitív találot vettem, ebben a viszonylatban a TF-IDF alapú megvalósítás képviseli a legjobb helyet, ha egy olyan kereső algoritmust keresünk, amely képes az esetek 67 százalékában megtalálni a duplikált kódot, a duplikáció tipusától függetlenül. Ha számunkra nagyon fontos a keresési eredmény megbízhatósága, akkor az AST alapon működő algoritmust tudjuk ajánlani, de azt szem előtt kell tartani, hogy, ha túl magas as szemantikai eltérés, akkor valószínű, hogy tévedni fog. Munkám során számtalan olyan problémába ütköztem, melyek megoldása külön külön új dolgokat tanítottak nekem és segítettek jobban megérteni az alkalmazott technológia sajátosságát. Különösen izgalommal töltött el, amikor saját modell-t kezdtem el tanítani és tesztelni, később, amikor elkezdtek érkezni az első találatok, nagyon meglepődtem a sebességen, ami az eredményekben is látható, hogy a pontosság hagy némi kívánnivalót maga után, de ezzel a sebességel egyik implementáció sem tud versenybe kelni.

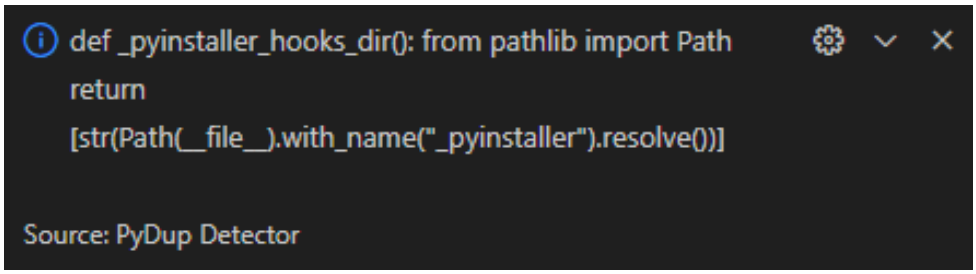
Függelék



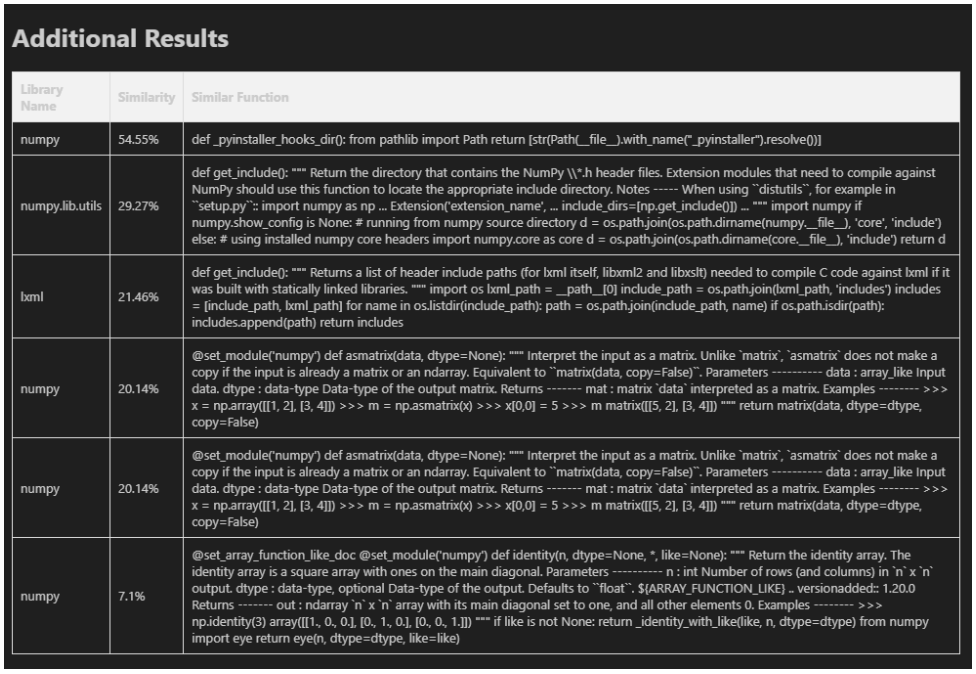
7.1. ábra. VSCode plugin beállításai



7.2. ábra. Találatok mutatása



7.3. ábra. Talált bővebb adatai



Additional Results

Library Name	Similarity	Similar Function
numpy	54.55%	def _pyinstaller_hooks_dir(): from pathlib import Path return [str(Path(__file__).with_name("_pyinstaller").resolve())]
numpy.lib.utils	29.27%	def get_include(): """ Return the directory that contains the NumPy *.h header files. Extension modules that need to compile against NumPy should use this function to locate the appropriate include directory. Notes ----- When using ``distutils``, for example in ``setup.py``: import numpy as np ... Extension('extension_name', ... include_dirs=[np.get_include()]) ... """ import numpy if numpy.show_config is None: # running from numpy source directory d = os.path.join(os.path.dirname(numpy.__file__), 'core', 'include') else: # using installed numpy core headers import numpy.core as core d = os.path.join(os.path.dirname(core.__file__), 'include') return d
lxml	21.46%	def get_include(): """ Returns a list of header include paths (for lxml itself, libxml2 and libxslt) needed to compile C code against lxml if it was built with statically linked libraries. """ import os lxml_path = __path__[0] include_path = os.path.join(lxml_path, 'includes') includes = [include_path, lxml_path] for name in os.listdir(include_path): path = os.path.join(include_path, name) if os.path.isdir(path): includes.append(path) return includes
numpy	20.14%	@set_module('numpy') def asmatrix(data, dtype=None): """ Interpret the input as a matrix. Unlike `matrix`, `asmatrix` does not make a copy if the input is already a matrix or an ndarray. Equivalent to ``matrix(data, copy=False)``. Parameters ---------- data : array_like Input data. dtype : data-type Data-type of the output matrix. Returns ------- mat : matrix `data` interpreted as a matrix. Examples -------- >>> x = np.array([[1, 2], [3, 4]]) >>> m = np.asmatrix(x) >>> x[0,0] = 5 >>> m matrix([[5, 2], [3, 4]]) """ return matrix(data, dtype=dtype, copy=False)
numpy	20.14%	@set_module('numpy') def asmatrix(data, dtype=None): """ Interpret the input as a matrix. Unlike `matrix`, `asmatrix` does not make a copy if the input is already a matrix or an ndarray. Equivalent to ``matrix(data, copy=False)``. Parameters ---------- data : array_like Input data. dtype : data-type Data-type of the output matrix. Returns ------- mat : matrix `data` interpreted as a matrix. Examples -------- >>> x = np.array([[1, 2], [3, 4]]) >>> m = np.asmatrix(x) >>> x[0,0] = 5 >>> m matrix([[5, 2], [3, 4]]) """ return matrix(data, dtype=dtype, copy=False)
numpy	7.1%	@set_array_function_like_doc @set_module('numpy') def identity(n, dtype=None, *, like=None): """ Return the identity array. The identity array is a square array with ones on the main diagonal. Parameters ---------- n : int Number of rows (and columns) in `n` x `n` output. dtype : data-type, optional Data-type of the output. Defaults to ``float``. ${ARRAY_FUNCTION_LIKE} .. versionadded:: 1.20.0 Returns ------- out : ndarray `n` x `n` array with its main diagonal set to one, and all other elements 0. Examples -------- >>> np.identity(3) array([[1., 0., 0.], [0., 1., 0.], [0., 0., 1.]]) """ if like is not None: return _identity_with_like(like, n, dtype=dtype) from numpy import eye return eye(n, dtype=dtype, like=like)

7.4. ábra. Összes találat

Irodalomjegyzék

[1] Python package index (pypi). Online. Accessed: [Add current date here].

[2] G Tsochev, R Trifonov, and O Nakov. Cyber security: Threats and challenges. In *Cyber security*. Publisher or Sponsor, 2020.

[3] Abdulrahman Algarni, Vijey Thayananthan, and Yashwant K. Malaiya. Quantitative assessment of cybersecurity risks for mitigating data breaches in business systems. *Applied Sciences*, 11(8):3678, 2021.

[4] KirstenS, Jim Manico, Jeff Williams, Dave Wichers, Adar Weidman, Roman, Alan Jex, Andrew Smith, Jeff Knutson, Imifos, Erez Yalon, kingthorin, Vikas Khanna, and Grant Ongers. Cross-site scripting (xss). Online, 2023.

[5] kingthorin. Sql injection. Online, 2023.

[6] Andrew J. Ko, Brad A. Myers, and Michael J. Coblenz. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on Software Engineering*, 32(12):971–987, 2006.

[7] M. Lehman et al. *Software Evolution*. Wiley Online Library, 2006.

[8] Microsoft. Visual Studio Code. Visual Studio Code, 2023. Accessed: 2023-04-28.

[9] Andreas P. Koenzen, Neil A. Ernst, and Margaret-Anne D. Storey. Code duplication and reuse in jupyter notebooks. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2020.

[10] Abstract syntax tree. Online. Accessed: [Add current date here].

[11] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[12] V Arora, RK Bhatia, and M Singh. Evaluation of flow graph and dependence graphs for program representation. *International Journal of Computer Applications*, 2012.

[13] E Waisberg, J Ong, M Masalkhi, and SA Kamran. Gpt-4: A new era of artificial intelligence in medicine. *Irish Journal of Medical Science*, 2023(03377):1–10, 2023.

[14] Mukesh Chapagain. Tf-idf vectorizer - scikit learn. Medium, 2022.

[15] Peter Bulychev and Marius Minea. Duplicate code detection using anti-unification. *Lomonosov Moscow State University, Russian Federation Institute e-Austria Timisoara, Romania*, 2008.

[16] H. Nasirloo and F. Azimzadeh. Semantic code clone detection using abstract memory states and program dependency graphs. In *2018 4th International Conference on Web Research (ICWR)*. IEEE, 2018.

[17] C. Pan, M. Lu, and B. Xu. An empirical study on software defect prediction using codebert model. *Applied Sciences*, 11(11):4793, 2021.

[18] Z Szabó and V Bilicki. A new approach to web application security: Utilizing gpt language models for source code inspection. *Future Internet*, 15(10):326, 2023.

[19] KS Divya, R Subha, and S Palaniswami. Similar words identification using naive and tf-idf method. *International Journal of Information Technology*, 2014.

[20] Python Software Foundation. 8.4. difflib – helpers for computing deltas. `https://docs.python.org/3/library/difflib.html`, 2024. Accessed: 2024-04-27.

[21] Gerard Salton. *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer*. Addison-Wesley, Reading, MA, 1989. Chapter on Vector Space Models, where cosine similarity is extensively discussed.

[22] NVIDIA Corporation. Cuda toolkit downloads. `https://developer.nvidia.com/cuda-downloads`, 2024. Accessed: 2024-04-29.