Szegedi Tudományegyetem
Informatikai Intézet

Hibás osztály automatikus keresése hibajelentés (bug report) alapján

Szakdolgozat

Készítette:
Krcsmárik Robin
informatika szakos hallgató

Témavezető:
Dr. Vidács László
tudományos főmunkatárs

Szeged
2019

Feladatkiírás

Feladat egy automata ajánló rendszer megvalósítása, mely hibajelentések alapján (pl. https://bugs.webkit.org/show_bug.cgi?id=122565) a forráskódban beazonosítja azon osztályokat, melyek nagy valószínűséggel hibásak, vagyis a hiba javításához ezeket az osztályokat kell módosítani, illetve a rendszer által tett ajánlatok pontosságának kiértékelése valós szoftverrendszeren. Megoldásként szolgálhat a hibajelentésben szereplő szavak és az osztályokban szereplő azonosító nevek közötti egyezések vizsgálata.

Tartalmi összefoglaló

- ***A téma megnevezése:***

Hibás osztály automatikus keresése hibajelentés(bug report) alapján

- ***A megadott feladat megfogalmazása:***

A feladat egy olyan alkalmazás elkészítése, mely a megadott hibabejelentések alapján javaslatot tesz a legvalószínűbb hibás, azaz javításra szoruló osztályokra. Szétbontva áll egy szövegbányászat felhasználásával megközelíthető osztályozási problémára megoldást nyújtó eljárás felhasználásából, illetve egy azt alkalmazó program megvalósításából.

- ***A megoldási mód:***

Az általam választott projekt melyen a feladatban megadott program dolgozik a bugzilla hibakezelő rendszerben kezelt Mozilla Gecko nyílt forráskódú verziókezelt projecthez kapcsolódó bejelentések részhalmaza (https://bugzilla.mozilla.org/). Ezek a részek java programnyelven készültek. A szoftverrendszer forráskódja a https://github.com/mozilla/gecko-dev.git címen elérhető. Első körben az adatok gyűjtéséhez a mozilla project repository-ából a program kigyűjti a bejelentések alapján készített commitokat és a committal módosított fájl listát. Ezt követően a kigyűjtött bug azonosítók alapján a program lekérdezi a hibák adatait és leírásukat a bugzilláról. Következő lépésként szövegbányászati módszereket alkalmazva először Vektortérmodellt állít elő a program, melyen koszinusztávolságot és a dolgozatban, a későbbiekben részletesebben tárgyalt értékeket számol. Végül a megkapott értékek súlyozásával egy rangsor áll elő a szupportvektor-gép rangsoroló osztályozóként való alkalmazása során. Az osztályozó tanulása és tesztelése során a k-szoros keresztvalidációt alkalmazza a program. Eredményként egy lehetségesen javítandó top-k rangsorolt állomány listát ad.

- ***Alkalmazott eszközök, módszerek:***

A program fejlesztése Eclipse 2018-09 segítségével Windows 10 operációs rendszeren készült, java 8 programnyelven. A program az adatbázis kezeléséhez a jdbc sqlite api-t, a git repository kezeléséhez a JGit api-t, a vektortérmodell előállításhoz az Apache OpenNLP api-t és az SVM rangsoroláshoz a C nyelven íródott RankSVM programot használja.

- ***Elért eredmények:***

A megvalósított program képes meghatározni a javítandó osztályokat az azonos területen más kutatási fejlesztés során elért eredményekhez képest hasonló pontossággal.

- ***Kulcsszavak:***

Eclipse, Java 8, Mozilla Gecko, Bugzilla, SVMRank, Apache OpenNLP, JGit

Tartalomjegyzék

BEVEZETÉS

Egy szoftver fejlesztési projekt sikeréhez létfontosságú a szoftver minősége. A színvonal javítása érdekében a fejlesztés során számos szoftverminőség-biztosítási tevékenységet (, mint például teszteléseket, statikus teszteléseket, ellenőrzéseket stb.) hajtanak végre a projekt résztvevői. Az elkészült rendszerek viszont még így is számos hibákat tartalmaznak a valóságban. Egy nagy és fejlődő szoftverrendszer, úgymint a Mozilla Gecko is hosszú időn keresztül nagyszámú hibabejelentést kaphat. Az idei év március hónapjában például a Mozilla Gecko projekt több mint 3000 javításon vagy módosításon esett át. Egy átlagos hibajavítás során a hibabejelentés megtörténte, megerősítése és továbbítása után a fejlesztőknek meg kell találnia azokat a forráskódfájlokat, amelyeket javítani vagy módosítani szükséges a hiba javításának érdekében. Ezen állományok kézi felkutatása egy igen időigényes, ezáltal költséges feladat különösen egy nagy projekt esetében, ahol a kód több ezer állományból áll.

Éppen ezért az utóbbi években számos kutatás indult információ visszakeresésen alapuló technikát alkalmazó módszerek kifejlesztésére irányulóan annak érdekében, hogy a hibabejelentéshez tartozó módosítandó forrásfájlok automatikusan kerüljenek meghatározásra. Ennek alapján ezek a kutatások a hibabejelentést lekérdezésként kezelik, melynek eredménye a javítandó forrásfájlok javasolt listája. A lista alapján javítható a bejelentett hiba, így kikerülve az időigényes forráskód keresést.

Ez a fajta hibakeresési módszer a hibabejelentések és a forráskódok szöveghasonlóságára épülve az információ visszakeresésen (IR) alapszik, így az nem igényel program végrehajtási információkezelést (pl. stack trace kezelést stb.).

Az előzőekben említett kutatások eredménye például a BugScout [1], mely a hibajelentések és a programkód teljes szövege között keres hasonlóságot szövegbányászati módszerrel.

Egy másik kutatás eredménye a BugLocator[2], mely a rendelkezésre álló hibabejelentések alapján bejelentésenként a vektortér-modell előállítása segítségével rangsorolja a forráskódfájlokat relevancia szerint, használva a Látens Dirichlet allokációt (LSI) és a Látens szemantikus indexelést (LSI).

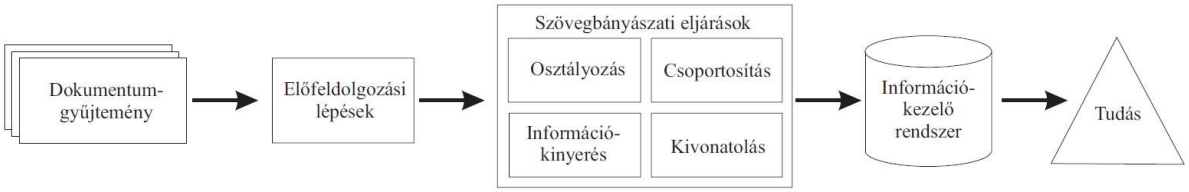
A dolgozatomban egy olyan korábbi Learning to Rank Relevant Files for Bug Reports using Domain Knowledge [3] című kutatásban említett módszert implementáltam, mely hasonlóan az előzőekben említettekhez információ visszakeresésen alapszik. A hibabejelentésekhez a forráskódok relevanciáját szupportvektor-gép osztályozó rangsorolja.

1. A szövegbányászat, mint módszer

1.1. Előfeldolgozás és modellalkotás

1.1.1 Szövegbányászat feladata

A szövegbányászat az emberek közötti szóbeli vagy írásbeli kommunikáció céljára alakult ki, nem a számítógépes feldolgozás szerint. Az emberek könnyedén felismerik és alkalmazzák a nyelvi mintákat. Nem okoznak nekik gondot a helyesírási variációk kezelése, a kontextus felismerése vagy a stilisztikai jelleg azonosítása. Viszont nincs meg bennünk a számítógépeknek az a képessége, hogy a szöveget nagy mennyiségben, vagy nagy sebességgel dolgozzuk fel. A szövegbányászat feladta tehát az emberi nyelvi tudás ötvözése a számítógép nagy feldolgozási képességével.



1.1.1.1 Ábra: A szövegbányászat modellje [4]

1.1.2 Vektortérmodellalkotás

A szövegbányászati feladatoknak két altípusa van a keresés és a rendszerezés. Az előbbi esetben olyan dokumentumokat keresünk, amelyekben egy adott keresőkifejezés fordul elő, utóbbiban pedig dokumentumokat hasonlítunk össze, majd ennek alapján kapcsoljuk őket valamely kategóriarendszer elemeihez. A dolgozatomnál a rendszerezés típusú szövegbányászati feladatot kell végrehajtani, ami más megközelítést igényel a kereséshez képest.

Ahhoz, hogy a dokumentumgyűjtemény elemein valamilyen rendszerezést hajtsunk végre, szükségünk van egy olyan modellre melynek segítségével a dokumentumok hasonlóságát, azaz a távolságát mérni tudjuk. Alapgondolatként azok a dokumentumok hasonlítanak egymásra, amelyeknek a szókészlete átfedik egymást és a hasonlóság mértéke az átfedéssel arányos. Ezt használja az információ-visszakeresésben használt vektortérmodell.

$$\mathbf{D} = \begin{matrix} \mathbf{d}_1 & \mathbf{d}_2 & & \mathbf{d}_N \\ \downarrow & \downarrow & & \downarrow \end{matrix} \begin{pmatrix} d_{11} & d_{12} & \dots & d_{1N} \\ a_{21} & d_{22} & \dots & d_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ d_{M1} & d_{M2} & \dots & d_{MN} \end{pmatrix}$$

1.1.2.1 Ábra: Vektortérmodell Szó-Dokumentum mátrixa [5]

A vektortérmodell egy vektortérben ábrázolja a vizsgálandó illetve összehasonlítandó dokumentumokat. A modell egyik dimenziója a dokumentumgyűjteményben lévő dokumentumok(*N*), míg a másik dimenziója dokumentumgyűjteményben előforduló egyedi szavak(*M*). A vektortérmodell sajátossága, hogy a szavak szövegen belüli pozíciója és sorrendje elvész. Ezt a megközelítést szózsákmodellnek is szokás nevezni.

1.1.3 Dokumentum feldolgozása vektortérmodellé

Ahhoz, hogy egy szöveges dokumentumot vektortérmodellben ábrázolni tudjuk, a súlyozási séma kiválasztásán kívül további előfeldolgozási lépéseket kell megtenni.

Első körben a szöveges dokumentumokat fel kell bontani további egységekre, jellemzően szavakra. Ezt a szakirodalom tokenizálásnak hívja. Tokennek hívjuk egy karaktersorozat konkrét előfordulását, míg típusnak hívjuk az azonos karaktersorozatot tartalmazó tokenek osztályát. A típusok összessége alapján (esetleges további szükséges feldolgozási lépések közbeiktatásával) áll elő a szótár.

Következő lépés az úgynevezett stopszó szűrés, ahol a gyakran előforduló, tartalmi információt egyáltalán nem tartalmazó, megkülönböztető képesség nélküli stopszavak eldobásra kerülnek.

Mindezek után a szavaknak át kell esniük a lemmatizáláson és a szótövezésen. A legtöbb nyelvben a szavak előfordulnak toldalékolt vagy módosított alakban is. A vektortérmodell kialakításánál szükséges az azonos szavaknak különböző szóalakú előfordulásait közös kanonikus alakra hozni. A kanonikus alak meghatározásához a szakirodalom két megközelítést használ. Lemmatizálásnak nevezi a szó lemmájának (normalizált, vagy szótári alakjának) előállítását, illetve meghatározását. Szótövezésnek hívja azt az eljárást, amikor az adott szó szótövének meghatározása a cél. A két eljárás között az a különbség, hogy míg a nyelvészeti motivációjú lemmatizálás mindig értelmes szóalakot állít elő, addig a szótövezés során jellemzően a szó csonkolása történik.

1.1.4 Modell súlyozása

A vektortérmodell azaz a kulcsszó-dokumentum mátrix elemeit (, ahol a sorok a kulcsszavak és az oszlopok a dokumentumok) többféleképpen is lehet súlyozni. Lehet többek között például bináris súlyozással, miszerint ha a szó előfordul a dokumentumban, akkor 1 az érték, egyéb esetben 0. Lehet súlyozni szó előfordulás alapján, ahol már nemcsak az számít, hogy a szót tartalmazza a dokumentum, hanem az is, hogy hányszor. Ez a darabszám lesz az értéke az elemnek. Logaritmikus súlyozásnál az előfordulás alapú súlynak vesszük a logaritmusát és hozzáadunk egyet.

Az eddig említett súlyok nem vették figyelembe a dokumentum hosszát. A nyelvi kifejezés jobb közelítése érdekében helyi és átfogó súlyok bevezetése szükséges, ahol a helyi súly az aktuális dokumentumra tipikusan az előfordulási számra, míg a globális a kulcsszóra vonatkozik. Ennek a súlyozási sémának a szógyakoriság és inverz dokumentumgyakoriság index (TF-IDF index) a neve.

$$d_{ki}^{(6)} = f_{ki} \cdot idf(t_k).$$

1.1.3.1 Ábra: TF-IDF súlyozás képlete [5]

Ahol az f_{ki} a t_k szó d_i dokumentumbeli gyakorisága (azaz a szó előfordulási számát osztjuk a dokumentumbeli szavak számával).

Az $idf(t_k)$ pedig a dokumentumgyakoriság inverze, ami egyenlő a $log(N/n_k)$-val. Itt az n_k azon dokumentumok száma, ahol t_k előfordul, N pedig a dokumentumok száma.

Az így kialakult TF-IDF súlyozás értéke tehát magas lesz azon szavak esetében, amelyek az adott dokumentumban gyakran fordulnak elő, míg a teljes korpuszban ritkán (így nagy a megkülönböztető erejük). Alacsonyabb lesz a súlyozás értéke viszont azon szavak esetében, amelyek a dokumentumban ritkábban, vagy a korpuszban gyakrabban fordulnak elő. Harmadik esetben pedig csekély akár zérus értékű lesz olyan szavaknál, ahol azok a korpusz összes dokumentumában előfordulnak.

A hibabejelentéseknek és forrásállományoknak így előállt vektortérmodelljéből a hasonlóság mérését koszinusz távolság számítással tehetjük meg. A koszinusz távolság előáll a hibabejelentések és a forrásállományok vektorai által bezárt szögek koszinuszával, a szavak által meghatározott dimenziójú térben.

$$s_{cos}(a, b) = \cos\Theta = \frac{\mathbf{a}^{\mathsf{T}}\mathbf{b}}{\|\mathbf{a}\|\|\mathbf{b}\|}$$

$$d_c(D_i, D_j) = \frac{\sum d_{ik} d_{jk}}{\left(\sum {d_{ik}}^2 \cdot \sum {d_{jk}}^2\right)^{0,5}}$$

1.1.3.2. Ábra: Koszinusz távolság vektoros képlete[23]

1.2. Osztályozás

1.2.1. Osztályozás feladata, típusai

A szövegbányászat egyik alapfeladata a szövegek osztályozása. Az osztályozásnak két eltérő megközelítése is létezik. Ez a kettő a klaszterezés és a kategorizálás. A klaszterezés automatikus módszer, ahol az algoritmus maga alakítja ki a kategóriákat. A dolgozat szempontjából az utóbbi megközelítés, azaz a kategorizálás az érdekes.

Formálisan: $\Phi : D \rightarrow 2^C$ osztályozáskor egy osztályozófüggvény megalkotása a cél, amely a D dokumentumtér elemeihez a $C = \{c_1, \ldots, c_{|C|}\}$ kategóriarendszerből vett kategóriák halmazát rendeli. A Φ függvényt röviden osztályozónak nevezik. Az osztályozó létrehozásakor a cél az ismeretlen $\Phi\tilde{} : D \rightarrow 2^C$ célfüggvény minél pontosabb közelítése, azaz a Φ és $\Phi\tilde{}$ eltérésének minimalizálása.

Az osztályozás a kategóriacímke számtól függően lehet egy vagy több címkés, illetve az osztályozás használata szerint kategória vagy dokumentum vezérelt.

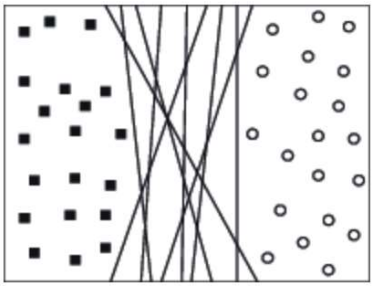
A dokumentum vezérelt osztályozásnál adott $d \in D$ esetén keressük az összes d-hez tartozó kategóriát. A kategória vezérelt osztályozásnál adott $c \in C$ kategória esetén keressük a c-be tartozó összes dokumentumot.

Az eredmény típusa szerint is kétféle osztályozást különböztetünk meg, a kiválasztó és rangsoroló osztályozást. A rangsoroló osztályozásnál a $\Phi : D \times C \rightarrow [0,1]$ függvény nem csak 0 vagy 1 értéket vehet fel, hanem ez az érték egy valós szám lesz. Így a dokumentum vezérelt esetben egy adott $d \in D$ dokumentumhoz rendelt kategóriák relevanciáját a C elemeihez rendelt érték alapján lehet rangsorba állítani.

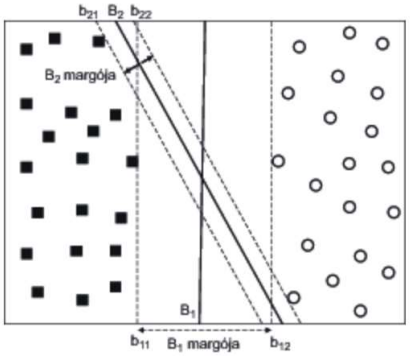
1.2.2. Az SVM osztályozó algoritmus

A szupportvektor-gép (Support Vector Machine – SVM) osztályozó a szövegbányászat terén az egyik leghatékonyabb gépi tanulási módszeren alapuló eljárás. Az SVM alapja a lineáris osztályozók csoportjába tartozik és bináris osztályozási problémák megoldására alkalmas.

Maga az algoritmus egy olyan döntési hipersíkot határoz meg, amely nem csak elválasztja a negatív tanítóadatokat a pozitív tanítóadatoktól, hanem a hipersíknak a mintáktól való távolságát maximalizálja is, mellyel egy maximális margójú hipersíkot állít elő.



1.2.2.1. Ábra: Lehetséges döntési határok lineárisan szeparálható adatok esetén [6]



1.2.2.2.Ábra: Döntési határ margója [6]

A megoldás kialakításában résztvevő tanítóadatokat szupportvektoroknak nevezzük. A hipersík meghatározásában csak a szupportvektorok játszanak szerepet.

A nagy margóval rendelkező döntési határoknak általában jobb az általánosítási hibájuk, mint a kis margóval rendelkezőknek, viszont ha a margó kicsi, akkor a döntési határ bármilyen kis zavarának elég jelentős hatása lehet az osztályozásra. A kis margóval rendelkező döntési határokat létrehozó osztályozók ezért hajlamosabbak a modell túlillesztésre és a korábban nem látott eseteken gyakran rosszul általánosítanak.

Nagy adathalmazok esetén a szeparabilitási feltétel gyakran nem teljesül (nem szeparábilis eset), illetve lineárisan szeparálható halmazok esetén is akadnak olyan kilógó, izolált vagy zajos pontok, amelyeket jobb figyelmen kívül hagyni az osztályozó létrehozásánál.

Ebben az esetben gyengítő változók bevezetése válik szükségessé és a hipersík egyenlet kis módosítással érvényben marad.

$$y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1 - \xi_i, \quad \text{minden } \mathbf{x}_i\text{-re}$$

1.2.2.3.Ábra: Hipersík egyenlete gyengítő változóval [7]

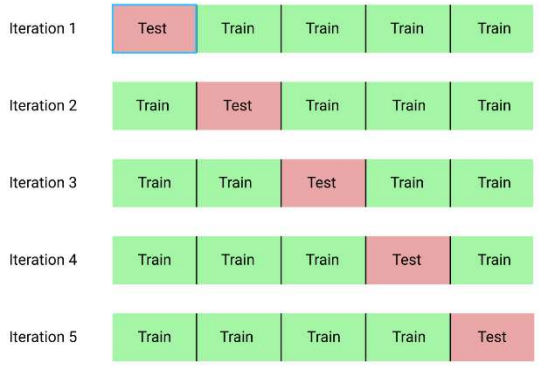
A feladat megoldásánál a tartalék szélessége, a hibásan osztályozott pontok, és a biztonsági sávba benyúló pontok száma közötti optimalizálást hajtunk végre. Fontos megemlíteni az SVM osztályozóknál megadható C regularizációs faktort, amellyel az

adatok fontossága szabályozható. A kicsi érték jobban tolerálja a túllógó pontokat, mint a nagy érték.

1.3. A hatékonyság mérése

1.3.1. Osztályozó kiértékelés k-szoros keresztvalidáció szerint

A feladat megoldásánál magát az osztályozót tanítóhalmazok segítségével tanítottam k-szoros keresztvalidáció szerint. Ez úgy néz ki a gyakorlatban általában, hogy a kiinduló hibabejelentések halmazát 10 diszjunkt részre osztjuk. Az első csomag (fold1) a legújabb hibabejelentéseket, míg a (fold10) a legrégebbieket tartalmazza. Ebből fel lehet építeni 10 darab osztályozót. Az osztályozókat így mindig a k tanítóhalmazon tanítjuk és a k-1 halamazon értékeljük ki. A hatékonyság rátáját a tíz érték átlagaként kapjuk.



1.3.1.1. Ábra: K szoros keresztvalidáció [20]

A dolgozatomban a hatékonysági rátát Top k hatékonyság szerint mértem. Ez akkor számít sikeresnek, ha az osztályozó által az első k pozitívként rangsorolt listában van legalább egy találat.

A hatékonyság mérésén felül többek között még beszélhetünk a szabatosság mértékéről:

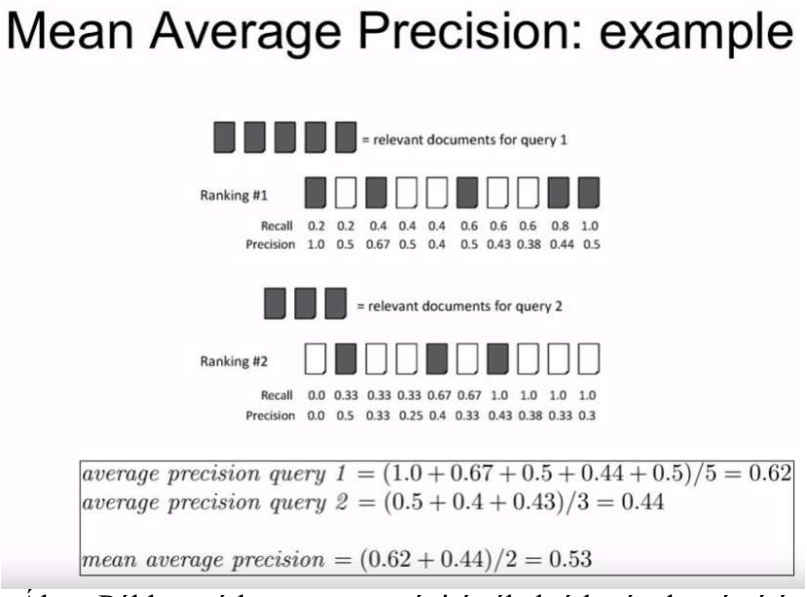
$$\text{szabatosság} = \frac{TP + TN}{TP+FP+TN+FN}$$

1.3.1.2.Ábra: Szabatosság

Ahol a *TP* – a találati listában szereplő releváns találtok száma, az *FP* – a találati listában szereplő nem releváns találtok száma, az *FN* – a találati listában nem szereplő releváns találatok száma és a *TN* – a találati listában nem szereplő nem releváns találatok száma.

1.3.2. Hatékonyság kiértékelés átlagos pontossági értékek átlaga szerint

A Top K hatékonyság mellett kiértékelési szempont lehet az átlagos pontossági értékek átlaga (Mean Average Precision, MAP). Ezt az értéket minden releváns találat megfigyelése esetén mért egyedi pontossági értékek átlagaként számoljuk.



1.3.2.1. Ábra: Példa az átlagos pontossági értékek átlagának számítására[24]

A hibabejelentéshez adott rangsorolt (javítandó forrásfájl) ajánlatokat sorba véve, minden egyes releváns forrásfájlnál számoljuk a pontosságot, azaz az addig talált releváns forrásállomány szám és a találat rangsorának hányadosát, míg el nem érjük az összes javítandó osztályt. Az így összegyűjtött értékek átlagaként kapjuk az átlagos pontossági érték átlagát egy hibabejelentéshez.

2. Felhasznált eszközök

2.1. Fejlesztéshez használt eszközök

2.1.1. Java

A Java [9] egy objektumorientált általános célú programozási nyelv. A hordozhatóság és visszafelé kompatibilitás a legfőbb oka a nyelv népszerűségének. Van hozzá egy hatalmas alap könyvtár, amiben a dátumok és szövegek kezelésétől a gyűjteménytípusokig minden egységesítve van. A Java programozási nyelvhez tartozik egy úgynevezett Java Virtual Machine, ami lehetővé teszi programok futtatását Java bájtkóddal egy köztes rétegen a virtuális gépen keresztül, ami többek között platformfüggetlenné teheti a javában írt programunkat. Tanulmányaim során ezt a nyelvet használtam a legtöbbet, így a dolgozatom készítésénél a célszerűséget is figyelembe véve ezt a nyelvet választottam.

2.1.2. Eclipse IDE

Az Eclipse [10] egy Java alapú keretrendszer. Ennek részét képezi az Eclipse IDE fejlesztői környezet, melyet elsősorban Java alapú alkalmazások fejlesztésére használják, de megtalálható a C++, PHP, Python stb. fejlesztésre optimalizált IDE is.

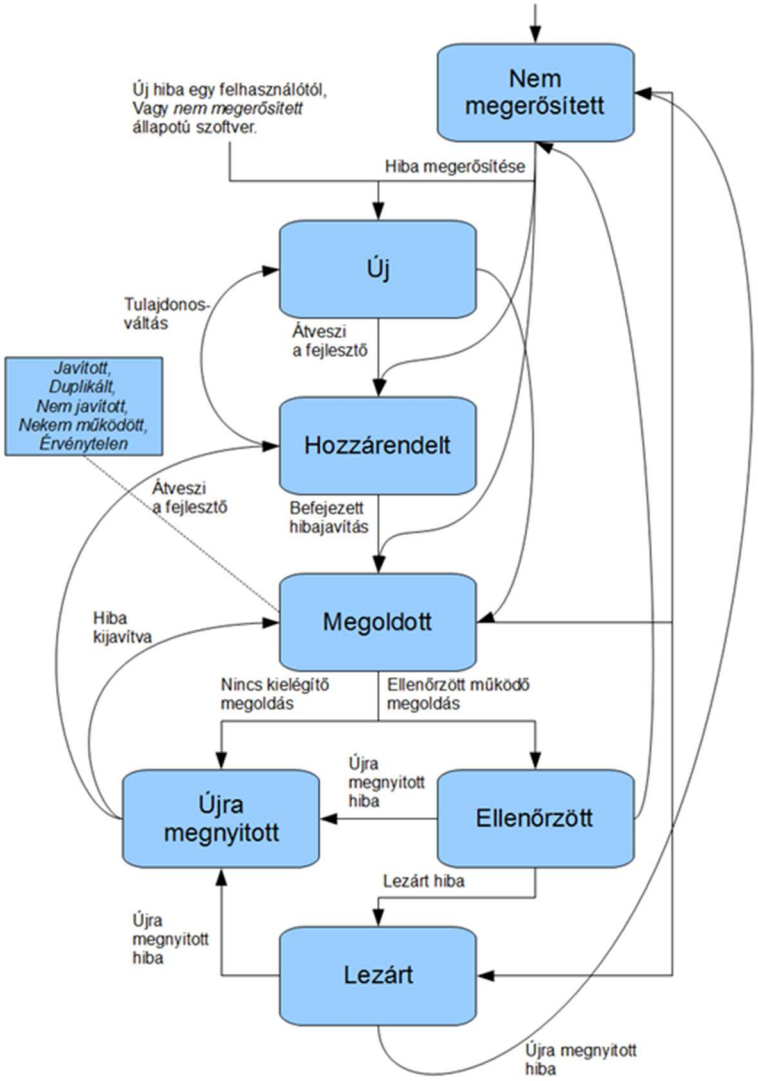
2.2. Vizsgált rendszer

2.2.1. Mozilla Gecko

A Gecko [11] egy a Mozilla Projekt keretében fejlesztett böngészőmotor. A Gecko feladata a webes tartalom feldolgozása és megjelenítése a képernyőn vagy nyomtatásban. A dolgozatomban ezen a nyílt forráskódú rendszeren végeztem a hibakeresést a hibabejelentések alapján. Maga a repository tartalma a https://github.com/mozilla/gecko-dev címen elérhető.

2.2.2. Bugzilla

A Bugzilla [12] egy hibakövető rendszer. A Bugzilla lehetővé teszi a fejlesztők számára, hogy hatékonyan nyomon kövessék a termékek hibáit, problémáit, fejlesztéseit és egyéb változtatási igényeit. A Mozilla Gecko Bugzilla hibakövető rendszere a https://bugzilla.mozilla.org/home címen érhető el.



2.2.2.1. Ábra: A Bugzilla rendszer állapot gépe [13]

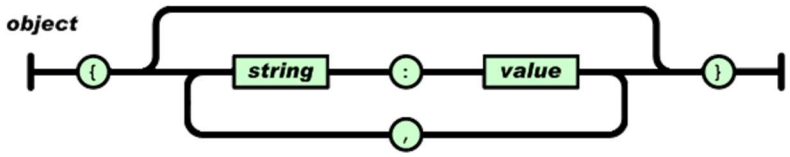
2.3. Felhasznált API-k és programok

2.3.1. JGit

Maga a JGit API-ban [14] implementálva van szinte az összes verziókezeléshez szükséges parancs. A JGit-nek nagyon kevés függősége van, ami alkalmassá teszi bármilyen Java alkalmazás beágyazására, függetlenül attól, hogy az alkalmazás kihasználja-e az egyéb Eclipse technológiákat. A dolgozatomban ahhoz, hogy megállapítsam mely forráskód állományok javítása volt szükséges az adott hibajavításhoz, szükséges volt csatlakozni és adatokat gyűjteni a helyi repository-ból.

2.3.2. JSON

A JSON (JavaScript Object Notation) [15] egy szöveg alapú szabvány, mely alkalmas az ember által olvasható adatcserére. A JavaScript nyelvéből alakult ki egyszerű adatstruktúrák és tömbök reprezentálására. Maga a szabvány nyelvfüggetlen, több nyelvhez is van értelmezője.



2.3.2.1. Ábra: Egy JSON objektum szintaktikája

A JSON szabványt a Bugzilla is használja. A hibaazonosító megadásával lehetőség van ilyen formátumban is lekérni egy hiba teljeskörű adatait és leírásait. A dolgozatomban ezt az API-t használtam a lekérdezett hiba JSON objektumok feldolgozásához.

2.3.3. Apache OpenNLP

Az Apache OpenNLP API [16] egy gépi tanulás alapú eszközkészlet a természetes nyelvű szöveg feldolgozásához. Támogatja a leggyakoribb természetes nyelv feldolgozási feladatokat, többek között a nyelvi felismerést, a tokenizációt, a mondatszegmentációt és a beszédcímkézést. Ezen feladatok segítségével állítja elő a dolgozatomban megírt program a hibabejelentések és a forráskódok szövegéből a vektortérmodellt.

2.3.4. SVMRank

Az SVMrank-ot [17] Thorsten Joachims készítette a hatékony SVM-ek képzéséhez. Ez egy rangsorolási problémát megoldó szupportvektor-gép. Maga a program C-ben íródott, a dolgozatomban ezt a lefordított programot használtam fel.

3. A Hibakeresés végrehajtása a program felépítésén keresztül

3.1. A hibakeresés folyamata a program végrehajtásának szemszögéből

3.1.1. Adatgyűjtés

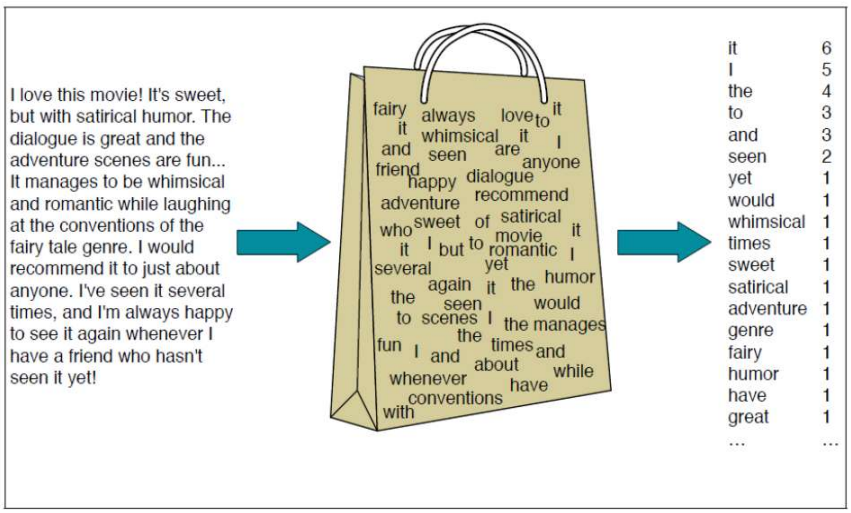
A feldolgozás első lépése az adatgyűjtés. A Mozilla Gecko verziókezelt rendszerében a Bugzillában indított hibajavításnál, miután javították a hibát a javítás commit üzenetében szerepeltetve van a hiba azonosítója.

3.1.1.1. Ábra: Egy hiba javításnak a commit üzenete [18]

A program a JGit API segítségével a letöltött repositoryból kigyűjti a commitokat, melyek üzeneteiből kinyeri a hiba azonosítóját és a módosított állományok listáját. Maga a forrásállomány szövegfeldolgozását java nyelvre optimalizáltam, így a kigyűjtött állományok tekintetében csak a java állományok kerültek figyelembevételre. Az így elkészült hibalista alapján a hibák tételes leírása az azonosítójuk alapján lekérdezhetők a https://bugzilla.mozilla.org címről JSON formátumban, amit a program egyenként kigyűjt és a JSON API segítségével feldolgoz. Az így összeállt adatbázisból készíthető el a vektortérmodell.

3.1.2. Szövegfeldolgozás és vektrotérmodell előállítása

A rendelkezésre álló adatokból (minden hibabejelentés és forrásfájl tekintetében) a program az OpenNLP API segítségével tokenizálás, stop szó szűrés és lemmatizálás után előállítja a szózsákokat. Ez jelen esetben körülbelül 9000 hibabejelentés és körülbelül 3000 java forrásfájl feldolgozását jelentette.



3.1.2.1. Ábra: szózsák modell[19]

A szózsák modell elkészülte után a program előállítja magát a vektortérmodellt és a modellhez tartozó szótárat. Ebben az állapotban a modell jelenleg a szavak előfordulásainak számát tárolja. A tovább lépéshez szükség van ezek súlyozására, ezért a program a modell előállítása után TF-IDF súlyozást hajt végre.

3.1.3. Szupportvektor-gép osztályozó algoritmus alkalmazása

Legyen *br(r,s)* egy hibabejelentés és forrásállomány pár. Az SVM alkalmazásához a program kiszámítja minden *br(r,s)* párhoz az alábbi paramétereket:

- S_1 = Minden *br(r,s)* tekintetében a hibabejelentés és forrásállomány vektora közötti koszinusz távolság {*sim(r,s)*}.
- S_2 = Egy forrásfájlt több hibabejelentés is érinthet. Minden *br(r,s)* tekintetében a program kiszámítja az aktuális hibabejelentés és a forrásfájl javítását érintő megelőző hibabejelentések közötti koszinusz távolságot {*sim(r, br(r,s)*}.
- S_3 = Minden *br(r,s)* tekintetében, ha a hibabejelentés tartalmazza az osztály nevét, akkor egyenlő az osztály nevének hosszával, egyéb esetben pedig 0.
- S_4 = Tekintettel arra, hogy ha egy forrásfájlt frissen javítanak, ott nagyobb valószínűséggel kell ismét javítást végrehajtani, mint a többi fájlon, ezért minden *br(r,s)* tekintetében S_4 előáll a hibabejelentés dátumának és a forrásfájlt megelőző javítás dátumának különbségének reciprokaként {pl.: $(2019.04\text{-}2019.03+1)^{-1}=1/2$}
- S_5 = Minden br(r,s) tekintetében a hibabejelentés megtétele előtti forrásfájlt érintő javítások számával.

A paraméterek előállta után az $f(r,s) = w_1*S_1 + w_2*S_2 + w_3*S_3 + w_4*S_4 + w_5*S_5$ rangsor értéket számítja ki a program a szupportvektor-gép rangsorolási osztályozási algoritmusa segítségével. Egy hibabejelentéshez az *f(r,s)* rangsor értéke adja meg a jóslást, hogy mennyire releváns az adott forrásfájl javítása. Az *f(r,s)* és a w_i súlyok kiszámítását a program az SVM Ranking osztályozó segítségével számítja ki [3].

```
1 qid:7273 1:0.99993604 2:0.99999267 3:0.0 4:1.0 5:69.0 #1#1268125#PermissionBlock.java
0 qid:7273 1:0.99999017 2:0.99998504 3:0.0 4:0.14285715 5:2.0 #0#1268125#Starvation.java
0 qid:7273 1:0.99999017 2:0.9999816 3:0.0 4:0.14285715 5:2.0 #0#1268125#Racerd.java
0 qid:7273 1:0.99999017 2:0.9999778 3:0.0 4:0.14285715 5:1.0 #0#1268125#Eradicate.java
0 qid:7273 1:0.99999017 2:0.9999743 3:0.0 4:0.14285715 5:2.0 #0#1268125#Checkers.java
0 qid:7273 1:0.99999017 2:0.99996614 3:0.0 4:0.14285715 5:2.0 #0#1268125#Biabduction.java
0 qid:7273 1:0.9999877 2:0.9999831 3:0.0 4:0.11111111 5:3.0 #0#1268125#not_packaged.java
0 qid:7273 1:0.9999877 2:0.99994403 3:0.0 4:0.055555556 5:1.0 #0#1268125#UTF16Buffer.java
0 qid:7273 1:0.9999877 2:0.9999526 3:0.0 4:0.14285715 5:1.0 #0#1268125#TreeBuilder.java
0 qid:7273 1:0.9999877 2:0.0 3:0.0 4:0.0 5:0.0 #0#1268125#Tokenizer.java
0 qid:7273 1:0.9999877 2:0.9999778 3:0.0 4:0.14285715 5:1.0 #0#1268125#StateSnapshot.java
0 qid:7273 1:0.9999877 2:0.999984 3:0.0 4:0.14285715 5:1.0 #0#1268125#StackNode.java
0 qid:7273 1:0.9999877 2:0.9999825 3:0.0 4:0.14285715 5:1.0 #0#1268125#Portability.java
0 qid:7273 1:0.9999877 2:0.99998045 3:0.0 4:0.11111111 5:3.0 #0#1268125#MetaScanner.java
0 qid:7273 1:0.9999877 2:0.99998593 3:0.0 4:0.11111111 5:6.0 #0#1268125#ElementName.java
0 qid:7273 1:0.9999877 2:0.9999717 3:0.0 4:0.14285715 5:1.0 #0#1268125#AttributeName.java
```

3.1.3.1. Ábra: Az SVM Ranking egy bemenetjének részlete

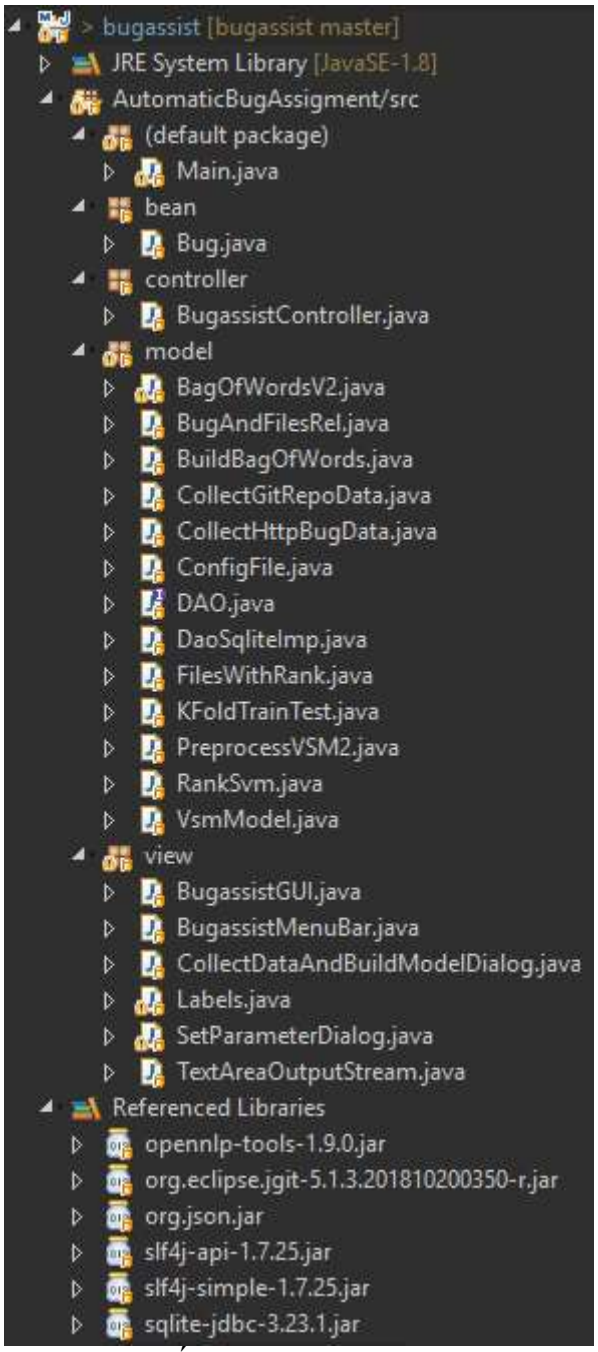
A program a 3.1.3.1. ábra szerinti outputot generálja az SVM Ranking számára. Az ábrán az első érték jelenti, hogy pozitív-e a minta vagy sem. A második érték a csoport szám, amin belül rangsorol. Utána következnek az előzőekben részletezett kiszámított értékek. A # után a programnak a visszaolvasásánál segítséget nyújtó megjegyzések láthatók (pozitív minta-e, hibabejelentés azonosítója, forrásfájl neve).

3.1.4. Eredmény kiértékelése

Az eredmény kiértékelését a program az átlagos pontosság átlaga és a k szoros keresztvalidáció szerint végzi. A rendelkezésre álló hibabejelentéseket 10 egyenlő részre osztva a 3.1.3.1 ábra szerinti 10 text fájlban tárolja. Ezeken végzi el az osztályozást. A megkapott eredményeket visszaolvasva méri a hatékonyságot a Top K hatékonyság szerint. Ez annyit tesz, hogy minden hibabejelentést végig nézve, ha a top k rangsorolt fájlban talál legalább egy ténylegesen javítani szükséges fájlt azt sikeres találatnak értékeli. Ezt átlagolva az összes hibabejelentés számával megadja a program hatékonyságát. Az átlagos pontosságot a program az 1.3.2. pontban foglaltak szerint az összes rendelkezésre álló hibabejelentésen futtatva számolja.

3.2. Program felépítése és működése

3.2.1. A Program szerkezete



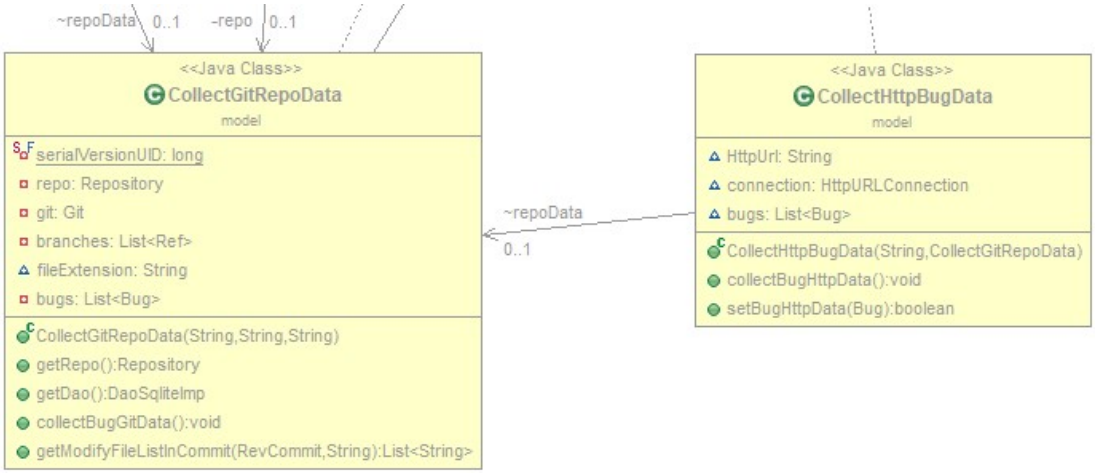
3.2.1.1 Ábra: A program felépítése

A program Eclipse-ből a Main.java fájl futtatásával indul, mely a BugassistController osztály példányosításával elindítja a GUI-t. A programot ez a controller osztály vezérli, illetve fogja össze. A szoftver a bean, model, view és controller csomagokból áll össze.

3.2.2. Adatgyűjtés

A program az adatok gyűjtését sorrendben a CollectGitRepoData és a CollectHttpBugData osztályok segítségével éri el. A CollectGitRepoData

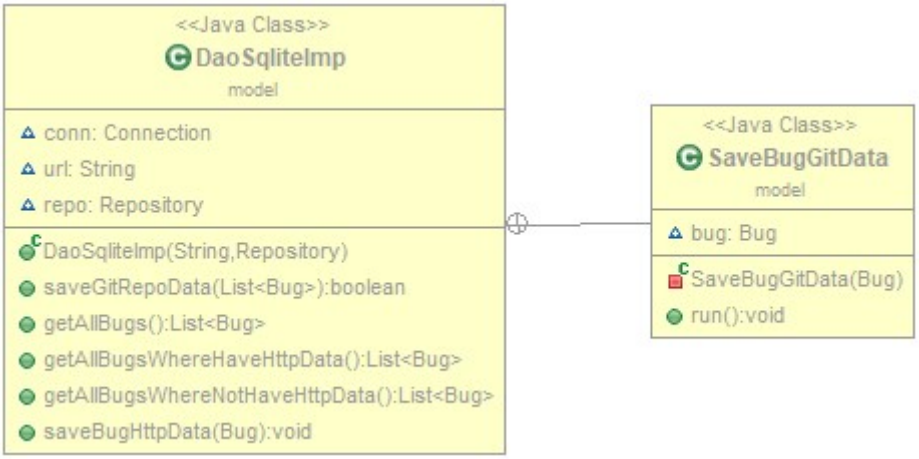
konstruktoraként szükséges megadni a repository helyét, az adatokból létrehozott vagy létrehozandó sqlite adatbázis helyét és a kigyűjteni kívánt forrásfájlok kiterjesztését esetünkben a .java forrásállományokat. Az osztály a collectBugGitData() metódussal a JGit API segítségével kigyűjti az összes branch összes commit-jából a 3.1.1 fejezetben részletezett commit üzenetekből a hibabejelentések azonosítóit, és a commit-tal módosított java fájlok listáját. Ezt egy GetCommitData belső osztály példányosításával oldja meg, ami implementálja a Runnable interfészt. Így a commitok adatainak kigyűjtése szálkezelést használva történik a feladat gyorsítása érdekében.



3.2.1.2 Ábra: A CollectGitRepoData és a CollectHttpBugData osztálydiagram részlete

A program a kigyűjtött adatokat egy Bug bean objektumokból álló listában tárolja, amit a feladat végeztével letárol a DaoSqLiteImp interfész implementálásával létrehozott DAO objektum segítségével, a létrehozott, vagy létrehozandó sqlite adatbázisban. Eddig jelenleg rendelkezésre állnak a hibabejelentések azonosítói és a módosítandó fájlok listája. A DAO osztály által létrehozott adatbázis séma a következő:

```
CREATE TABLE IF NOT EXISTS bug(commitname text, bugid integer);
CREATE TABLE IF NOT EXISTS bugfiles(commitname text, filename text);
CREATE TABLE IF NOT EXISTS bughttpdata(bugid integer, shortdesc text,
longdesc text, productname text, status text, bugdate text, bagofwords
text);
```

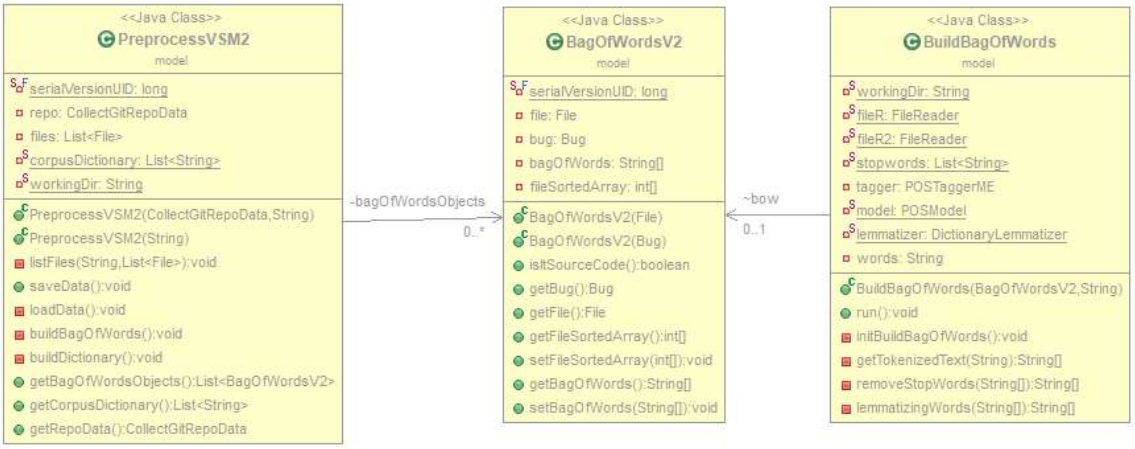


3.2.1.3 Ábra: A DaoSqliteImp osztálydiagram részlete

A git repository adatainak kigyűjtése után, a következő lépcső a hibabejelentések leíró adatainak lekérdezése a https://bugzilla.mozilla.org-ról. Ezt a CollectHttpBugData osztály collectBugHttpData metódusa hajtja végre. Először kigyűjti az összes olyan hibát aminek nincs leírása, majd sorba lekérdezi azokat. A *https:/mozilla/bugzilla.org/rest/bug/bugid/comment* get request-tel, a *bugid* helyébe behelyettesítve a hibabejelentés azonosítóját megkapjuk JSON formátumban a szükséges adatokat, úgymint a hibabejelentés rövid leírását, hosszú leírását, státuszát és a módosítás dátumát. Ezt követően a DAO osztály segítségével hibánként letárolja az adatokat. Az így elkészült adatbázis szolgáltathatja az alapját az előkészítésnek és modellképzésnek.

3.2.3. Előkészítés

Az előfeldolgozást a modellképzéshez a PreprocessVSM osztály hajtja végre. Első körben kigyűjti az összes hibabejelentést, majd kigyűjti a repo könyvtárából az összes java kiterjesztésű fájlt. Minden egyes hibabejelentésből és fájlból egy BagOfWords (szózsák) objektumot állít elő. A BagOfWords objektum fogja magában foglalni, hogy fájlról vagy hibabejelentésről van-e szó, illetve itt lesz letárolva a feldolgozott szavak halmaza (szózsák) is.



3.2.3.1 Ábra: A PreprocessVSM2, BagOfWordsV2 és a BuildBagOfWords osztálydiagram részlete

Ahhoz, hogy ezeknek a BagOfWords objektumoknak létrehozhassuk a szózsákjaikat, a program egyenként szálkezelést használva kiszámítja azt a buildBagOfWords(BagOfWords bo); metódus segítségével. Az utóbbi osztály tartalmazza és alkalmazza a szövegfeldolgozáshoz használt OpenNLP API metódusait. De mielőtt rátérnék ezen metódusokra, fontos megemlíteni, hogy a buildBagOfWords a konstruktorában kapott objektum típusától függően előállítja a folyó szöveghalmazt. Ez a hibabejelentéseknél annyit tesz, hogy a hibabejelentés címét pontosabban rövid leírását összeolvasztja a hosszú leírásával. A forrásfájloknál viszont szükséges a java elnevezési konvencióit követve szétszedni az egybeírt elnevezéseket a nagybetűs karakterek mentén. Ezek rendelkezésre állása után következik az OpenNLP API-ból átvett tokenizálás és lemmatizálás:

- getTokenizedText(String words): Ebben a metódusban van használva az OpenNLP SimpleTokenizer osztálya, ami az adott nyers szöveget a tokenize metódussal karakter osztályokra bontja és egy szavakból álló tömböt ad vissza

Input to Tokenizer	John is 26 years old.					
Output of Tokenizer	John	is	26	years	old	.

3.2.3.1. Ábra: Példa a SimpleTokenizer működésére [21]

- removeStopWords(String[] sourceString): Ez nem az OpenNLP Api része, de mégis itt kell megemlítenem a folyamat megtartása végett. Ezzel a metódussal a szövegtömbből egy angol stop szó szótár lista segítségével, amit a https://gist.github.com/carloschavez9/63414d83f68b09b4ef2926cc20ad641c webhelyről töltöttem le, illetve kiegészítve az általam készített java stop szó

listával, ami tartalmazza például a protected, private, abstract, final stb. szavakat eltávolítja a szükségtelen szavakat és írásjeleket.

- lemmatizingWords(String[] sourceString): Ez a metódus használja az OpenNLP API DictionaryLemmatizer szótár alapú lemmatizálóját, ami a szavak szótári alakra való hozását eredményezi. Ehhez szükség van egy előre definiált szótárra en-lemmatizer.dict fájlra és az API POS Tagger-éhez (en-pos-maxent.bin), ami a szavak szerepét határozza meg a mondatban.

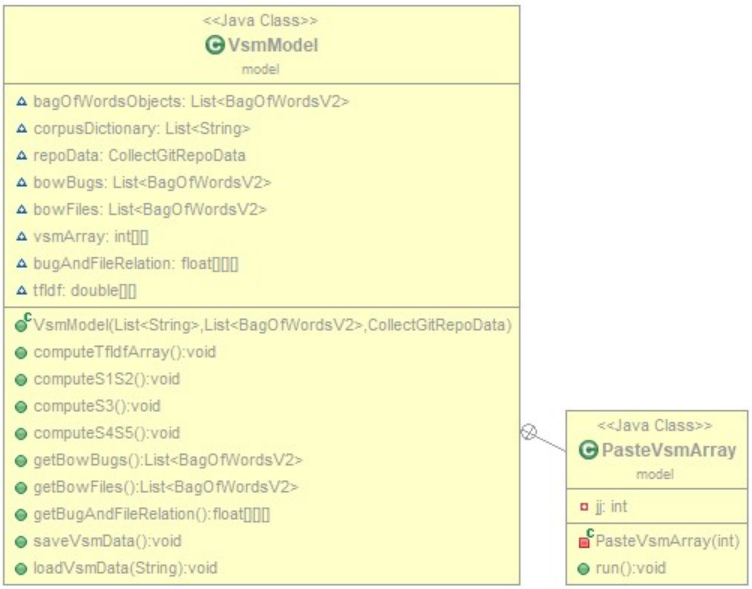
Input to POS Tagger	John is 27 years old.
Output of POS Tagger	John_NNP is_VBZ 27_CD years_NNS old_JJ ._.

3.2.3.2.Ábra: Példa a POS Tagger működésére [22]

A szótár (en-lemmatizer.dict) és a POS Tagger (en-pos-maxent.bin) birtokában a Lemmatizer metódus szótári alakra hozza a szavakat.

3.2.4. Modellképzés

A modellképzést a VSMModel osztály hajtja végre. Ehhez kiindulásként szükség van az előzőleg meghatározott BagOfWords objektumokra, amelyekben már rendelkezésre állnak a „tisztított" szózsákok. A VSMModel a vektortérmodellt adatszerkezetileg egy vsmArray[][] kétdimenziós tömbben tárolja, ahol az első dimenzó a szavak előfordulásainak száma a szótár List<String>corpusDictionary listatömb indexei szerint, míg a második dimenzió a hibabejelentések és forrásfájlok List<BagOfWords> listatömb szózsák objektumaihoz tartozó értékek a listatömb indexei szerint. Ebből a tömbből számolja a tf-idf súlyozást, amiből kiindulva meghatározza a 3.1.3. fejezetben említett S_1,S_2,S_3,S_4,S_5 értékeket. A kiszámított értékeket a program három objektumban kezeli. Első egy ArrayList<String>bowBugs objektum a hibabejelentéskből, a második egy ArrayList<String>bowFiles a java fájlokból és a harmadik egy double[][][] bugAndFileRel háromdineziós tömb. A tömb első két dimenziója a hibabejelentések és a forrásfájlok BagOfWords objektumainak relációja a lista indexük szerint. A harmadikban tárolódik az, hogy az adott hibabejelentés és forrásfájl pár összetartozik-e a javítás szempontjából (0.0 vagy 1.0) értékkel jelölve, továbbá a tf-idf súlyozás, a koszinusz távolság és az S_1,S_2,S_3,S_4,S_5 értékei.



3.2.4.1 Ábra: A VsmModel osztálydiagram részlete

3.2.5. Rangsoroló osztályozás

A szupportvektor-gép osztályozó használatáért a RankSvm osztály a felelős. Konstruktorán keresztül vagy megkapja az előzőekben megalkotott BagOfWords objektumokat és a szükséges értékeket (a tf-idf súlyozást, a koszinusz távolságot, az S_1,S_2,S_3,S_4,S_5 értéket és azt, hogy pozitív-e a minta) a bugAndFileRelation[][][] tömb útján, vagy betölti azokat fájlból. Azért van szükség e két féle módozatra, mert maga a számítás rendkívül időigényes és ha csak a kiértékelt eredményekre van szükségünk, akkor elég fájlból betölteni a VSMModel osztály által kiszámított eredményeket.

Azt osztály példányosításával létrejött objektum először k részre (a továbbiakban 10 részre) osztja a rendelkezésre álló hibabejelentéseket és elmenti a 3.1.3.1. Ábra szerinti formátumban 10 szöveges dokumentumban. Ebben a formátumban tudja kezelni az SVMRanking osztályozó tanításáért felelős lefordított svm_rank_learn.exe program.

Az osztályozónak kiírt 10 csomagban a fájlok rangsorolásához a pozitív minták mellett, minden hibabajelentéshez csak a koszinusz távolság szemszögéből legközelebb álló irreleváns forrásfáljok kerültek kiírásra. Erre azért volt szükség, mert a Mozilla Gecko projekt több, mint 3000 java forrásfájlt tartalmaz aminek a számítása rendkívül időigényes feladat lenne.

A kigyűjtött szöveges állományokat a KFoldTrainTest osztály dolgozza fel úgy, hogy a Process osztály segítségével futtatja az SVMRanking program tanító modulját.

Példa egy feldolgozásra:

```
svm_rank_learn -c 3 example3/train.dat example3/model
```

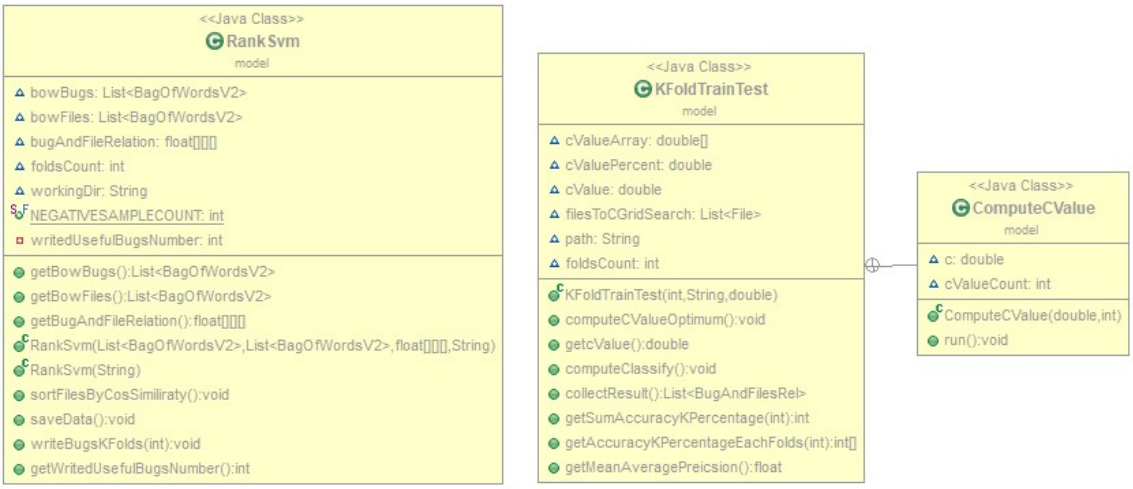
A c regularizációs faktor optimumját a grid search módszer segítségével számoltam ki a computeCValueOptimum metódus segítségével. A metódus a legrégebbi 10. hibacsomagon végigfuttatja az előre megadott értékeket és kiválasztja a legjobb eredményt elérő értéket. Ez a jelenlegi projektben a 0.01 érték.

A rangsor értékeket az SVMRanking program osztályozó modulja számítja ki. Ennek futtatása is a java Process osztálya segítségével történik.

Példa egy feldolgozásra:

```
svm_rank_classify example3/test.dat example3/model
                  example3/predictions
```

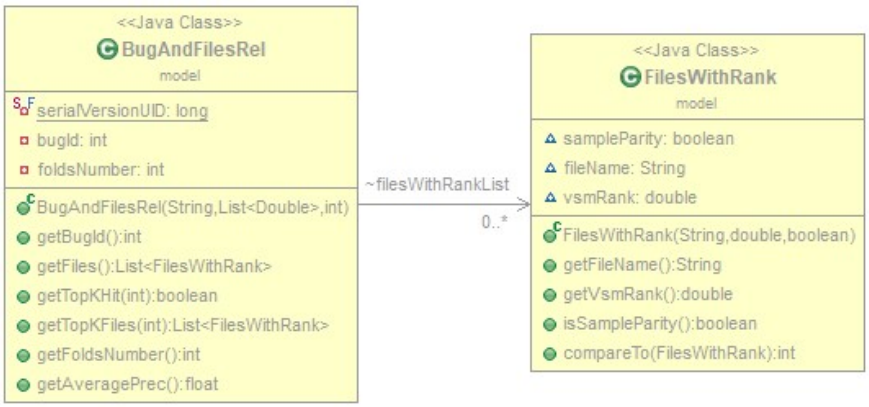
Kimenetként megkapjuk a rangsor értékeket a bemeneti fájl rekordjainak sorrendjében.



3.2.5.1 Ábra: A RankSvm és a KFoldTrainTest osztálydiagram részlete

A program fontosabb összetettebb osztályai közül már csak a BugAndFilesRel osztály maradt, ami az előzőekben említett csomagfájlokból és a rangsor értékeket tartalmazó fájlokból hibabejelentésenként példányosítva összefogja a hibabajelentésekhez tartozó pozitív és negatív mintájú fájlok neveit és rangsorát.

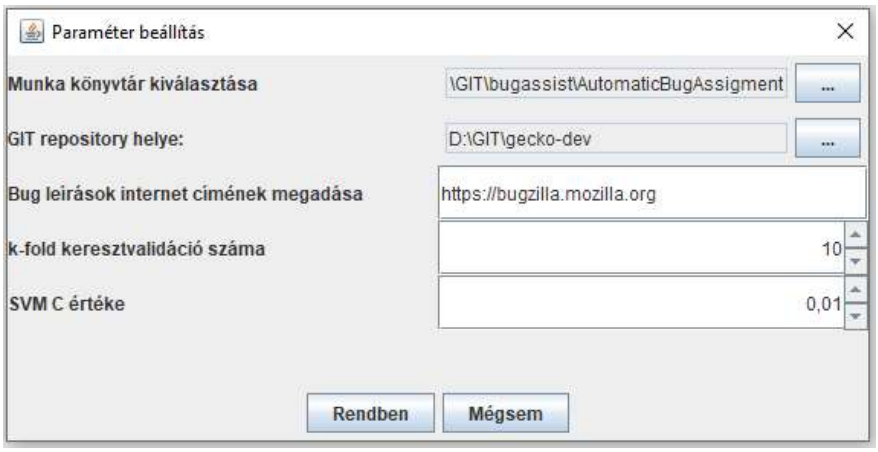
A getTopKHit() metódus segítségével megkaphatjuk, hogy a legjobb rangsorú k állománylistában van-e találat.



3.2.5.2 Ábra: A BugAndFilesRel és a FilesWithRank osztálydiagram részlete

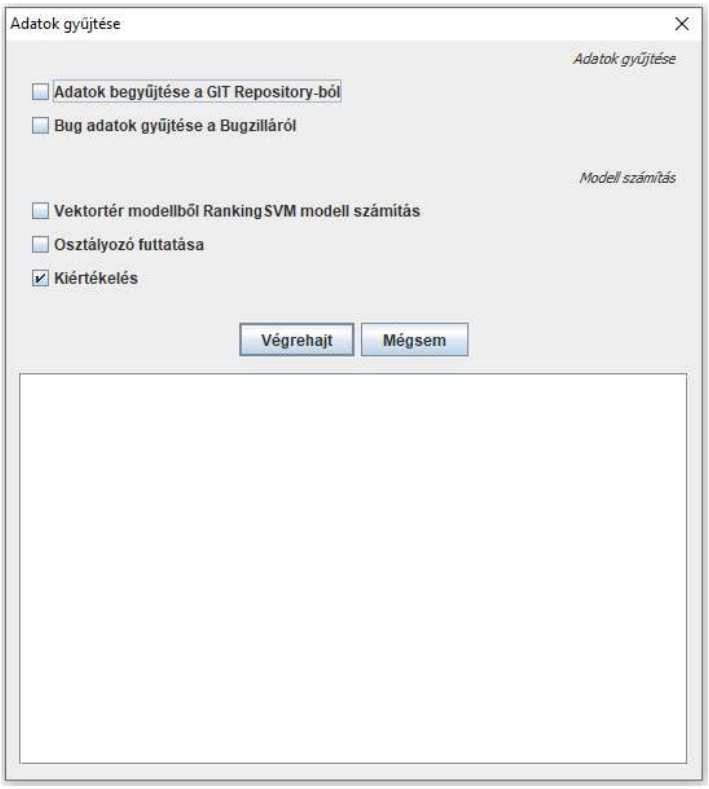
3.2.6. A program használata GUI-n keresztül

A program az Eclipse-ből a Main osztály main metódusával indul. Ezután két ablak megnyitására van lehetőség, a paraméter beállítására vagy a feldolgozásra. A paraméter beállításnál szükséges megadni többek között a munkakönyvtárat, ahol a külső fájlok találhatóak (adatbázis, mentések stb.) és a Mozilla Gecko repository elérési útvonalát.



3.2.6.1. Ábra: Paraméterek beállítása ablak

A feldolgozás ablakban indíthatjuk a tényleges számításokat. Ebben az ablakban állítható be, hogy a program mely részeket futtassa le. Az adatok gyűjtése szekcióban választható, hogy csak a git repository-ban lévő adatokat kérdezze le a program, vagy a hibabejelentések leírását is a Bugzilláról. A GIT repository-ból a hibabejelentések commit-jaiból gyűjthetők az adatok a 3.1.1. fejezetben említett módon. A hibabejelentések lekérdezésének végrehajtása az adatbázis építésekor rendkívül időigényes feladat, mivel a program egyenként kérdezi le a hibabejelentéseket és közte várakozik 3 másodpercet, hogy ne tiltson le a Bugzilla a gyakori kérések miatt. Ez durván hat órát jelent.



3.2.6.2. Ábra: Feldolgozás ablak

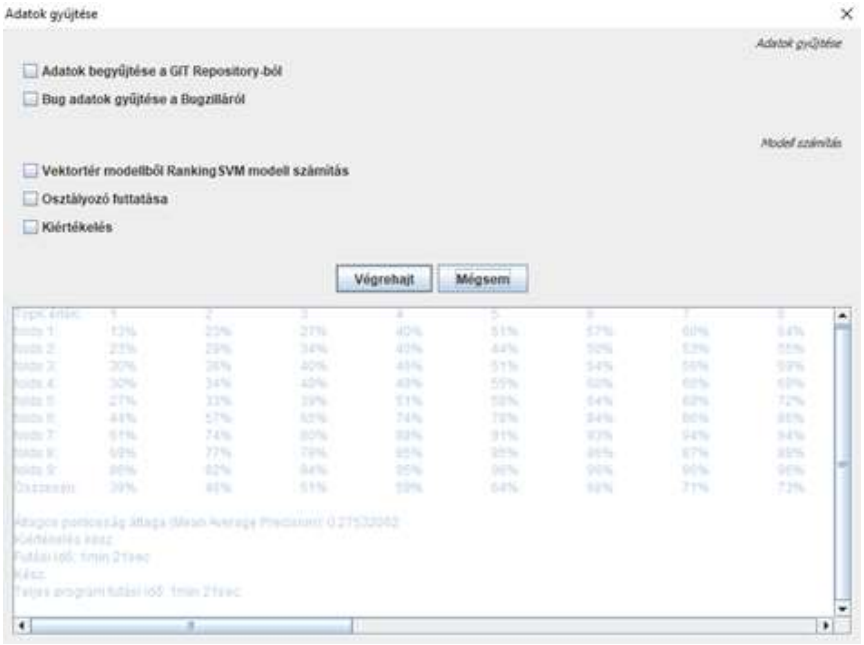
A vektortér modellből RankingSVM modell számítása résznél az előfeldolgozás, a vektortérmodell előállítása és az osztályozáshoz szükséges értékek (tf-idf súlyozás, koszinusz távolság és az S_1,S_2,S_3,S_4,S_5 értékek) kiszámítása történik. Ebben a pontban hajtja végre a program a k-szoros keresztvalidációhot szükséges állományokat is. Az osztályozó futtattásával kerül végrehajtása a szupportvektor-gép tanítása és a rangsor értékek kiszámítása. A kiértékelés fázisában kerül kiszámításra a program hatékonysága a Top-K értékek szerint.

A program futási ideje a futtatott Windows 10 operációs rendszeren, 16 GB memóriájú, AMD FX-8350 8 szálas processzorú konfiguráción a Bugzilláról gyűjtött adatokat nem számítva 70 perc körül alakul.

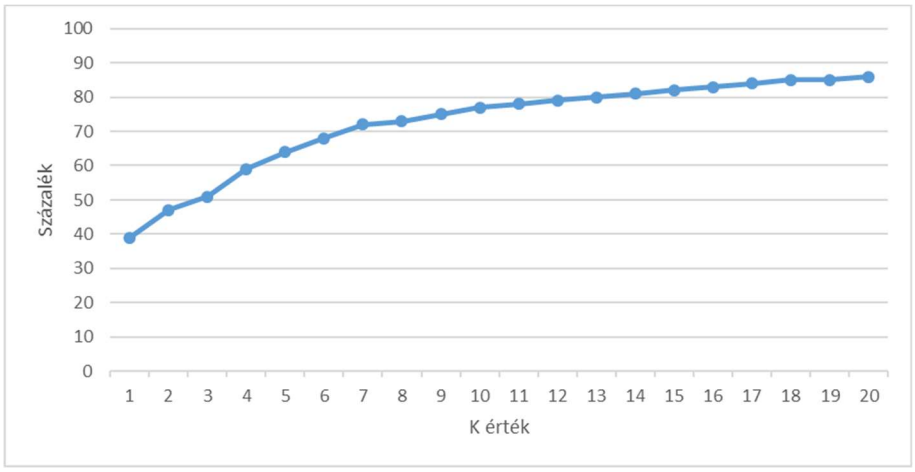
4. Kiértékelés

4.1. Elért eredmények kiértékelése

A program lefuttatása időpontjában (2019. április 28.-án) 5180 hibabejelentés került kigyűjtésre a Mozilla Gecko GIT repository-ából. A vizsgálandó java forrásállományok száma pedig 3030 darab fájl. A nagy számú hibabejelentés és forrásállományból kiindulva a k-szoros keresztvalidációt a program úgy hajtja végre, hogy az osztályozó az előző csomagon tanít és az aktuális csomagon osztályoz. Így kapunk 9 darab egymástól független eredményt. A 9 darab osztályozó által rangsorolt adatokon a Top-K szabatosság mérést alkalmaztam. Ennél módszernél azok a találatok számítanak sikeresnek, ahol a legjobb K rangsorolt fájl közül legalább az egyiket ténylegesen javítani kellet a hiba megoldása érdekében. Ezt több K érték esetén is leválogatja a program és a feldolgozás ablakban meg is jeleníti.

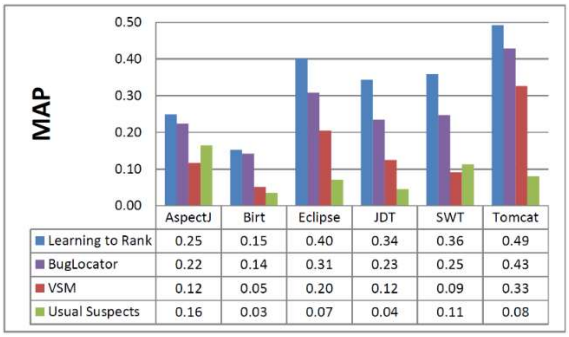


4.1.1. Ábra: A program által kiértékelt eredmények



4.1.2. Ábra: Találati értékek diagramja

A fenti ábrából leolvasható, hogy a program által rangsorolt legjobb 20 forrásállományból 86%-ban az egyiket ténylegesen javítani kellett. Hasonló területen a Xin Ye, Razvan Bunescu, and Chang Liu - Learning to Rank Relevant Files for Bug Reports using Domain Knowledge tudományos cikkében fejlesztett program Top-K hatékonyság szempotjából a legjobb eredményt az SWT projekten érte el k = 20 értéken 80%-os találati aránnyal.
Az átlagos pontosság átlagának (Mean Average Precision) szempontjából az általam fejlesztett program 0.28 pontossági értéket ért el, mely az azonos területen végzett más kutatásokkal összehasonlítva is jó eredménynek könyvelhető el.



	AspectJ	Birt	Eclipse	JDT	SWT	Tomcat
Learning to Rank	0.25	0.15	0.40	0.34	0.36	0.49
BugLocator	0.22	0.14	0.31	0.23	0.25	0.43
VSM	0.12	0.05	0.20	0.12	0.09	0.33
Usual Suspects	0.16	0.03	0.07	0.04	0.11	0.08

4.1.3. Ábra: Átlagos pontosságok átlaga azonos területen végzett más kutatásokkal[3]

A szupportvektor-gép osztályozó alkalmazásánál lehetőség van még a C regularizációs faktor módosítására. Ezt jelenleg 0.01 alapértelmezett értéken futtattam. A programban implementálásra került egy C érték optimalizáló, amely a GridSeacrh elvén a legrégebbi csomagon futtatja az osztályozót különböző C értékekkel, majd ezekből megadja a legjobbat. Az optimalizálót lefuttatva, illetve különböző nagyságrendű {0.001, 0.01, 0.1, 1, 10, 100} értékekkel próbálkozva is futattam az osztályozót. Az eredmények tükrében

megállapítottam, hogy pár százalékos eltérést leszámítva nagy különbség nincs a megválasztott értékek között.

4.2. Összegzés

Összességében a program véleményem szerint sikeresen megvalósítja a jóslást a javítandó forrásfájlok tekintetében. Úgy gondolom, hogy a dolgozatom során kitűzött célokat sikerült teljesíteni, miszerint a program kinyeri az adatokat, feldolgozza őket, majd azokon a szükséges számításokat elvégzi. Végeredményképp pedig ki is értékeli azokat, melyeket meg is jelenít. Az elért eredmények megközelítőleg azonosak más hasonló területen végzett kutatások eredményeivel. A program fejlesztése folyamán nehézség főként a futási idő redukálása folytán adódott. Többször is optimalizálni kellett az osztályok működését.

4.3. Fejlesztési lehetőségek

- A program egyik fejlesztési lehetőségének látom más osztályozó algoritmus felhasználását. Gondolok itt példának a neurális hálózat alapú módszerekre, mint például a Rocchio osztályozóra, mely kötegelt tanulást végez, azaz az összes tanítóadatot egyszerre használja fel.
- Egy másik fejlesztési lehetőség lehetne az, hogy a program bemenetét univerzálissá tegyük más ilyen típusú adatbázisok feldolgozása céljából. Egy másik eshetőségként pedig a feldolgozható forrásfájlok típusainak bővítését látom, hogy ne csak a java forrásállományokat dolgozza fel.

Irodalomjegyzék

[1] Anh Tuan Nguyen, Tung Thanh Nguyen, Jafar Al-Kofahi, Hung Viet Nguyen, Tien N. Nguyen - A Topic-based Approach for Narrowing the Search Space of Buggy Files from a Bug Report
[2] Jian Zhou, Hongyu Zhang, and David Lo - Where Should the Bugs Be Fixed? More Accurate Information Retrieval-Based Bug Localization Based on Bug Reports
[3] Xin Ye, Razvan Bunescu, and Chang Liu - Learning to Rank Relevant Files for Bug Reports using Domain Knowledge
[4] Tikk Domonkos – Szövegbányászat, a szövegbányászat feladata, TypoTex, 2007
[5] Tikk Domonkos – Szövegbányászat, dokumentum reprezentálása vektortérmodellben, TypoTex, 2007
[6] https://www.tankonyvtar.hu/hu/tartalom/tamop425/0046_adatbanyaszat/ch05s05.html
[7] Tikk Domonkos – Szövegbányászat, Osztályozás, TypoTex, 2007
[8] Tikk Domonkos – Szövegbányászat, Osztályozók elemzése, TypoTex, 2007
[9] https://www.oracle.com/java/index.html
[10] https://www.eclipse.org/
[11] https://developer.mozilla.org/hu/docs/Mozilla/Gecko
[12] https://www.bugzilla.org/
[13]https://www.tankonyvtar.hu/hu/tartalom/tamop425/0046_szoftverteszteles/ch09.html#id568730
[14] https://www.eclipse.org/jgit/
[15] http://www.json.org/
[16] https://opennlp.apache.org/
[17] http://www.cs.cornell.edu/people/tj/svm_light/svm_rank.html
[18] https://github.com/mozilla/gecko-dev/commits/master
[19] https://www.inf.ed.ac.uk/teaching/courses/fnlp/lectures/07_slides.pdf
[20] https://towardsdatascience.com/cross-validation-explained-evaluating-estimator-performance-e51e5430ff85
[21] https://www.tutorialkart.com/opennlp/tokenizer-example-in-apache-opennlp/
[22] https://www.tutorialkart.com/opennlp/pos-tagger-example-in-apache-opennlp/
[23] https://www.inf.u-szeged.hu/~berendg/docs/dm/DM_similarity_pf.pdf
[24] https://www.youtube.com/watch?v=pM6DJ0ZZee0

Nyilatkozat

Alulírott Krcsmárik Robin programtervező informatikus BSc szakos hallgató, kijelentem, hogy a dolgozatomat a Szegedi Tudományegyetem, Informatikai Intézet Szoftverfejlesztés Tanszékén készítettem, Programtervező Informatikus BSc diploma megszerzése érdekében.

Kijelentem, hogy a dolgozatot más szakon korábban nem védtem meg, saját munkám eredménye, és csak a hivatkozott forrásokat (szakirodalom, eszközök, stb.) használtam fel.

Tudomásul veszem, hogy szakdolgozatomat / diplomamunkámat a Szegedi Tudományegyetem Informatikai Intézet könyvtárában, a helyben olvasható könyvek között helyezik el.

Szeged, 2019. május 12.

Aláírás

Köszönetnyilvánítás

Köszönöm Dr. Vidács László témavezetőmnek (konzulensemnek) a hasznos és érdekes témát, a szakdolgozat során adott célravezető ötleteit és jó tanácsait.

Köszönöm tanáraimnak az egyetemi éveim alatt szerzett hasznos tudást és a diákokhoz való pozitív hozzáállást.