Szegedi Tudományegyetem

Természettudományi és Informatikai Kar

Sérülékenység előrejelzés JavaScript programokban folyamat metrikák segítségével

Diplomamunka

Készítette:

Viszkok Tamás

programtervező informatikus

hallgató

Témavezető:

Dr. Ferenc Rudolf

tanszékvezető egyetemi docens

Dr. Hegedűs Péter

tudományos munkatárs

Szeged

2020

Tartalomjegyzék

Feladatkiírás

A jelenleg használt, szoftverhibák előrejelzését végző mesterséges intelligencia modellt építő algoritmus paraméterezéseinek, futtatásainak, illetve eredményeinek tárolása, visszakeresése még nem megoldott. A feladat az algoritmus paraméterezésének automatizálása, az eredmények tárolása és az elvégzendő részfeladatok párhuzamosítása A feladat megoldásához szükséges framework elkészítése ezen szakdolgozat témája.

Az elkészült szoftver segítségével egy webes felületen lehet a tanuló algoritmusok, illetve a tanulás alapjául szolgáló adatok előállításának paramétereit megadni, visszakeresni, illetve a futtatott feladatok eredményeit újra felhasználni. Az ily módon megadott paraméterekkel ellátott feladatokat a szerver feladata kiosztani az alá regisztrált worker gépekre, ezáltal párhuzamosítva a megadott feladatokat.

Tartalmi összefoglaló

- A téma megnevezése:

  Sérülékenység előrejelzés JavaScript programokban folyamat metrikák segítségével

- A megadott feladat megfogalmazása:

  A szakdolgozat témája egy keretrendszer fejlesztése tanulóalgoritmusok használatának megkönnyítésére. A hallgató feladata bemutatni a fejlesztés lépéseit, az algoritmikus részek implemen-tálásától a grafikus felület létrehozásának mikéntjéig.

- A megoldási mód:

  A keretrendszer nagyobb része saját munka eredménye, a worker gépek logikáját viszont egy munkatársam valósította meg.

- Alkalmazott eszközök, módszerek:

  Az alkalmazás fejlesztése a Pycharm nevű fejlesztői környezetében történt a Python-Flask framework segítségével. Az adatbázis elasticsearchben íródott.

- Elért eredmények:

  A szoftvert bemutató cikk egy Q1-es tudományos folyóirat második körös elbírálása alatt áll, azaz hamarosan publikálásra kerül.

- Kulcsszavak:

  Python, Python-Flask, Elasticsearch, Mesterséges Intelligencia, Tanuló algoritmusok

Bevezetés

A számítógépes rendszerek ellen elkövetett újabb és újabb támadások egyre növekvő száma miatt kiemelt figyelmet kell fordítanunk a szoftvereink biztonságossá tételére. A lehető legmegbízhatóbb működés érdekében fontos lenne kiszűrni a folyamatból az emberi tényezőt. Ha sikerülne előállítani egy mesterséges intelligencia modellt, ami valós időben meg tudja állapítani egy adott kódrészletről, esetünkben egy adott metódusról, hogy az sérülékenységet fog-e okozni a későbbiekben, azzal rengeteg időt, egyúttal rengeteg erőforrást lehetne spórolni. Egy ilyen irányú kutatás [2] továbbgondolásáról szól ez a dolgozat. A kutatók JavaScript nyelven írt szoftverek sérülékenységét próbálták előre jelezni statikus kódmetrikák segítségével. Én ezt az imént említett kutatás során létrehozott JavaScript függvényeket tartalmazó adatbázist bővítettem ki az úgynevezett folyamat metrikákkal.

Szerencsére két igen fontos jelenség együttállása segít a probléma kezelésében. Egyrészt korábbi kutatások szerint [6] az említett támadások megközelítőleg 90 százaléka ismert sérülékenységek kihasználásával dolgozik, így ezen sérülékenységek keresése a kódunkban, majd a bevett védekezési módszerek alkalmazása hatékony megoldásnak ígérkezik a támadások megfékezésére tett erőfeszítéseink kivitelezésében. Másrészt a JavaScript nyelv egyre növekvő népszerűségnek örvend, újabban már nem csak kliens oldali webfejlesztés területén, de asztali és szerver alkalmazások, illetve mobilfejlesztés területén is. Ennek következtében, az interneten tömérdek mennyiségben találhatunk különböző JavaScript nyelven írt szoftvereket, amiket összegyűjtve jelentős méretű adathalmazt készíthetünk, ami számosságából adódóan remek lehetőséget biztosít a mesterséges intelligencia területén alkalmazott tanulóalgoritmusok betanításához, teszteléséhez, illetve a sérülékenység előrejelzéséhez szükséges modellünk előállítására felmerülő ötleteink teszteléséhez.

Az előállított metrikák közül néhány, csak említés szintjén:

- létrehozás és módosítások közt eltelt idő súlyozott átlaga
- függvény létrehozása óta létrejött szoftver verziók (kommitok) száma
- az adott kódrészletet érintő hibajavítások száma

Ahogy a folyamat metrikák használatának létjogosultságát már korábbi kutatások is bizonyították [5, 7], úgy esetemben, a sérülékenységek előrejelzésében is jelentős mértékű javulást hozott, ahogy az a későbbiekben látható lesz.

A módszer hatékonyságának ellenőrzéséhez 10 különböző algoritmust használtam, melyek közül a paraméterezhető modellek 10-10 különböző paraméterezését hasonlítottam össze. Többek közt használtam neuronhálóra épülő modelleket, döntési fa alapú osztályozókat, mint például a Random Forest, illetve néhány egyszerűbb algoritmust is, mint például a Naive Bayes módszer. Ezek közül a legjobb eredményt a Random Forest érte el, 85,75%-os F-measure értékkel (93,1% precision, 79,5% recall).

A dolgozat felépítése

A következő fejezetben ismertetem, hogy milyen igények vezettek a módszer kifejlesztéséhez. Majd a 2. fejezetben további sérülékenység előrejelző módszerekről lesz szó, hangsúlyozva az általam használt folyamat metrikák és ezen módszerek közötti hatékonyságbeli különbséget.

Az utána következő fejezetekben az általam alkalmazott módszer egyes lépéseit mutatom be. Először, a 3. fejezetben a modellek betanításához összegyűjtött adatbázis előállításáról, illetve az általam használt metrikákkal való kiegészítéséről lesz szó. Majd a 4. fejezetben a különböző algoritmusok paraméterezéséről, betanításáról és kiértékeléséről írok, valamint a használt keretrendszer is említésre kerül. Továbbá, az 5. fejezetben a kapott modellek eredményeit hasonlítom össze. Végezetül a 7. fejezetben összegzem az elért eredményeket és néhány jövőbeli fejlesztési ötletet vázolok fel.

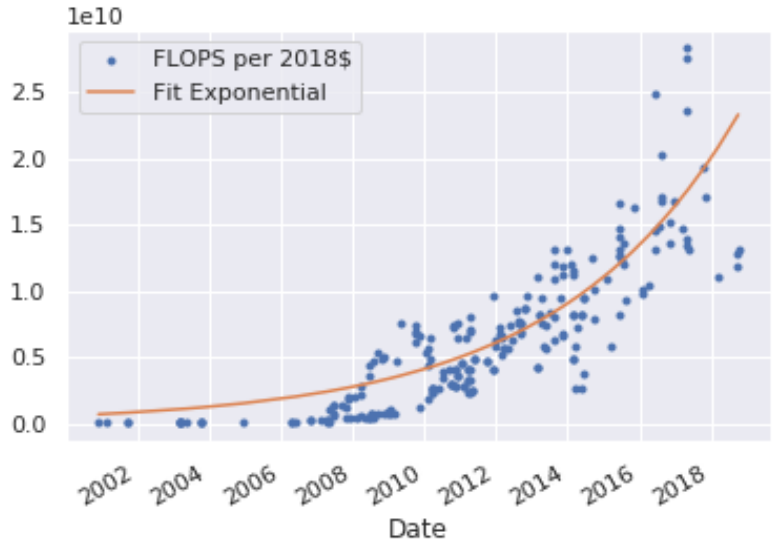
1. fejezet

Motiváció

Napjainkban, a technológiai fejlődésnek köszönhetően egyes szoftverek már nem csak a hagyományos számítógépes környezetben érhetőek el. Ott vannak a zsebünkben az okostelefonokon, a konyháinkban a hűtőkön, de még a kezünkön hordott órákon is. Ahogy egyre jobban a mindennapi életünk szerves részévé válnak ezek az eszközök, úgy egyre gyorsabban nő a sebezhető felület a rosszindulatú támadások számára is. Értelemszerűen, ezzel egyidejűleg rohamosan nő az igény a biztonságos szoftverek írására, a szoftverek sebezhetőségeinek minél hatékonyabb felderítésére.

Ennek ellenére az ilyen sebezhetőségek felderítésére sok esetben a vállalatok nem fordítanak elegendő erőforrást a szűkös határidők miatt, vagy akár teljesen elhanyagolják és a fejlesztőkre bízzák a dolgot. De még azon cégek esetén is ahol külön szakembert foglalkoztatnak a szoftvereik biztonságossá tételének érdekében, az emberi tényező továbbra is problémát jelenthet. Ennek kiküszöbölésében segítene egy automatizált módszer bevezetése.

Másrészt, egy szoftver sérülékenységeit minél hamarabb sikerül felderíteni, annál nagyobb lesz az a költségvetésbeli, illetve erőforrásbeli megtakarítás ami ellenkező esetben kárba veszne. Illetve, ami talán még ennél is sokkal fontosabb, hogy egy kiadott szoftver, ami sérülékenységeket tartalmaz, jelentős mértékben csorbíthatja a fejlesztők, de még inkább a cég hírnevét. Emellett, egy ilyen eset akár hosszas pereskedéseket is maga után vonhat, amit ideális esetben szeretnénk elkerülni. Éppen ezért, ha sikerülne létrehozni egy olyan eszközt, ami képes a szoftverfejlesztési folyamatok során valós időben megállapítani egy adott kódrészletről, ha az az ismert sérülékenységek valamelyikét tartalmazza, még mielőtt éles verzióban kiadásra kerülnének, felbecsülhetetlen értékű lenne.



1.1. ábra. Adott áron elérhető számítási kapacitás növekedése az utóbbi pár évben[1].

1.1. A mesterséges intelligencia térhódítása

Szerencsére a mesterséges intelligencia virágkorát éljük, a tanulóalgoritmusok használatához szükséges nagy számítási kapacitású eszközök egyre elérhetőbb áron szerezhetők be (lásd 1.1 ábra), az interneten fellelhető tömérdek mennyiségű, bárki számára elérhető adat pedig tökéletes táptalajt biztosít a nagyobb méretű tanító adathalmazt igénylő algoritmusok számára is. Adja magát a lehetőség, hogy a sérülékenység előrejelzés területén is próbára tegyük a tudomány ezen ágazatát.

A már létező hibadetektáló modellek többsége főleg általános szoftverhibák előrejelzésére fókuszál. A sérülékenység, mint olyan, sok esetben nem tekinthető a hagyományos értelemben vett hibának, így ezek a módszerek módosítás nélkül nem alkalmazhatóak megfelelő hatékonysággal a sérülékenységek ellen. A megoldás egy új modell betanítása lenne.

[1] A kép forrása: http://mediangroup.org/gpu.html

2. fejezet

Kapcsolódó munkák

A mesterséges intelligencia használatával történő hibadetektálás életképességét korábbi kutatások már bizonyították [4, 1]. Viszont ezen megoldások a program helyes működését akadályozó hibalehetőségek előrejelzésére szolgálnak. A sérülékenységek sok esetben nem sorolhatóak ebbe az általánosított kategóriába. Emiatt a létező hibadetektáló modellek helyett be kell tanítanunk egy új, speciálisan a sérülékenységek detektálására alkalmas modellt.

2.1. Sérülékenységeket tartalmazó adatbázis

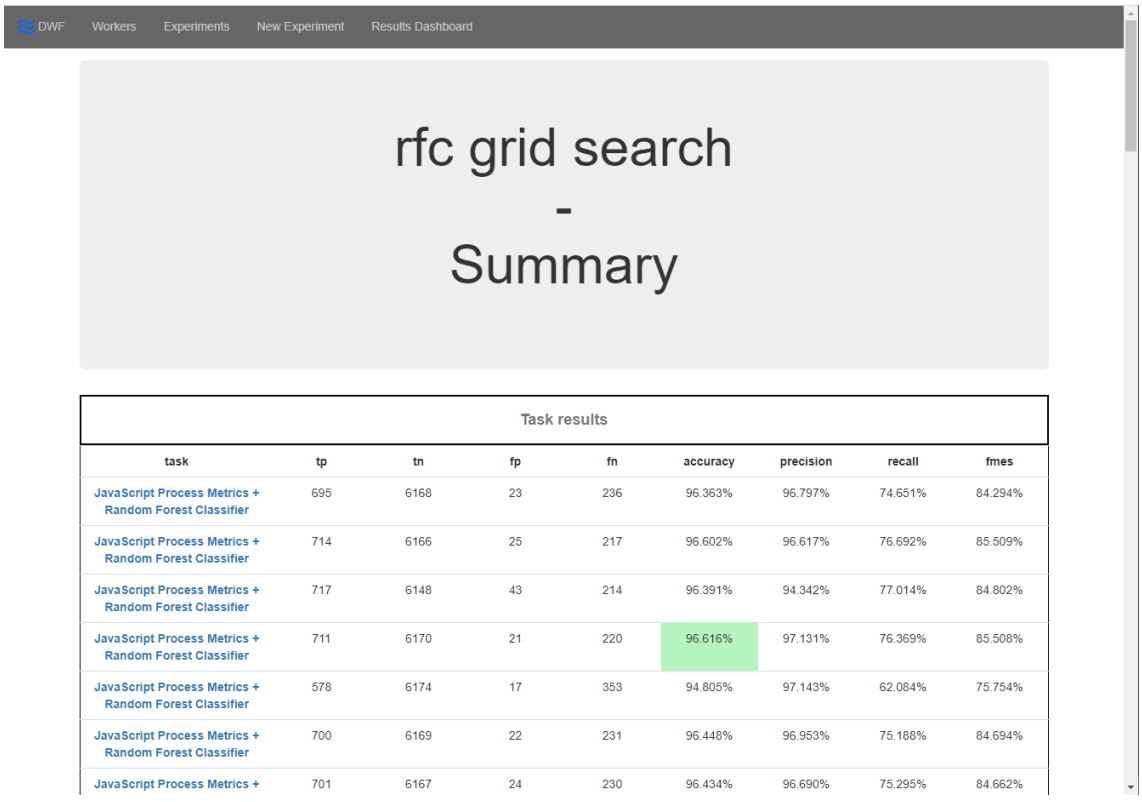
Ezzel a céllal jött létre egy függvényszintű, azaz függvényeket, illetve azok esetleges sérülékenységeit tartalmazó adatbázis egy korábbi kutatás keretében [2]. Az adatbázis JavaScript nyelvű projektek függvényeiből lett előállítva, ami egy nagyszerű választás volt a kutatók részéről. Annak ellenére, hogy eleinte a JavaScript főként a kliens oldali webfejlesztés nyelve volt, az utóbbi időben más területeken is meglátták a benne rejlő potenciált, illetve annak az előnyeit, ha minden platformon, kliens és szerver oldalon, mobil applikációk fejlesztésénél, vagy akár a beágyazott rendszereknél is ugyanazt a nyelvet használják. Az egységesítés előnyeire most nem térnék ki, részben azért, mert elég nyilvánvalóak, de leginkább azért, mert nem ez a dolgozatom témája. Ami a mi esetünkben fontos az ennek a jelenségnek a következménye, mégpedig a nyilvánosan elérhető JavaScript nyelvű projektek egyre gyorsabban növekvő száma. Ugyanis minél nagyobb számosságú az adathalmaz, amit a modellünk tanításához tudunk felhasználni, annál jobb hatékonysággal lesznek képesek működni a napjainkban egyre nagyobb népszerűségnek

örvendő mesterséges neuronhálókra épülő algoritmusok.

Az említett sérülékenység adatbázis előállításakor statikus kódmetrikákat tároltak el minden egyes függvényről, mint például a ciklikus komplexitás, paraméterek száma, vagy utasítások száma, stb. Az adatbázis használatával 76%-os F-measure értéket értek el (91% Precision-el és 66% Recall értékkel), ami már eleve nagyon ígéretesen hangzik.

2.2. Eszköz a folyamat metrikák kinyeréséhez

A bevezető részben említett, folyamat metrikák használhatóságáról szóló kutatások [5, 7] eredményei arra engednek következtetni, hogy az adatbázisunkat kibővítve ezen metrikákkal akár nagyságrendekkel jobb eredmények is elérhetőek lennének. Az imént említett adatbázis kibővítéséhez *Gyimesi Péter* [3] módszerét alkalmaztam, melynek során a sérülékenység adatbázisban található projektek *git*[1] verziókövető rendszer használatával létrejött gráfját kell a legelső kommittól kezdve végig elemezni a kívánt verzióig.



2.1. ábra. Deep Water Framework - Summary nézet

[1] https://git-scm.com/

2.3. Framework a modellek betanításához

A létrejött kibővített adatbázissal lehetőségem nyílt betanítani különböző osztályozó algoritmusokat, eltérő paraméterezésekkel. Többek közt két neuronhálós implementációt, de néhány egyszerűbbet is, mint például a Naive Bayes. Kilenc algoritmus, a paraméterezhetőek közül egyenként húsz különböző paraméterezését hasonlítottam össze, ami több, mint 120 tanítást jelent. Ezt a 120 tanítást kézzel elindítani, majd a kapott eredményeket szintén kézzel összegyűjteni és összehasonlítani elég kényelmetlen és hosszadalmas művelet lenne. Szerencsére a Deep-Water Framework[2] épp az ilyen feladatok leegyszerűsítésére jött létre. Segítségével az algoritmusok futtatása gond nélkül lezajlott és a kapott eredmények összehasonlítása is gyerekjáték volt a felület Sumary nézetének segítségével (lásd 2.1 ábra).

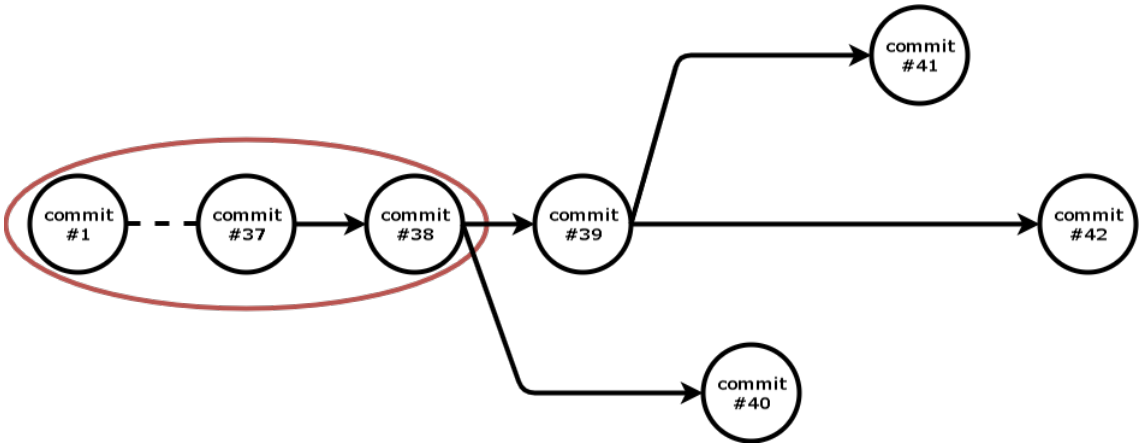
[2]https://github.com/sed-inf-u-szeged/DeepWaterFramework

3. fejezet

Folyamat metrikák kinyerése

Az első lépés a dolgozatom elkészítése felé vezető úton a a folyamat metrikák kinyerése volt. Mindenek előtt, először is a már létező adatbázisban lévő adatokat kellett végig járni, hogy még megfelelnek-e a valóságnak.

Egy python szkript segítségével végig jártam az összes git repo azon kommitjait, amiből az adatbázis eredetileg feltöltésre került. Amennyiben az adott sha-1 kódhoz tartozó szoftververzió már nem volt elérhető a verziókövető rendszer gráfjában (5 darab ilyen kommit volt), úgy a hozzátartozó adatbázis bejegyzéseket semmisnek tekintettem, csak a dolgozat készítésének időpontjában elérhető verziók bejegyzéseit tartottam meg.



3.1. ábra. Közös szülő kommitok

A folyamat metrikák kinyeréséhez a sérülékenység adatbázisunkban szereplő összes kommit összes szülőkommitját végig kell elemezni, egészen az adott repo legelső kommitjától kezdődően. Mivel egy repository-hoz több különböző kommit is tartozhat az adatbázisban, így előfordulhat, hogy néhány kommit szülőkommitjai nagyrészt meg-

egyeznek (lásd 3.1 ábra), így az egy repoba tartozó kommitok közül időrendileg a legújabbat megelőző összes revízió számát összegezve egy egész jó alsó becslést kaphatunk az összes végig elemzendő szoftververzió számát illetően, amit később az elemzéshez szükséges erőforrások becslésére használtam. Tehát második lépésként az első szűrőn átjutott kommitok szülő revízióinak számát összesítettem, úgy, hogy az azonos repository alá tartozó kommitok közül csak az időrendben legújabbat vizsgáltam. A végeredmény egy 300000 körüli revíziószám lett, azaz háromszázezer különböző szoftververzióra kell az elemzést egyesével végig futtatni a későbbiekben.

3.1. Forráskód elemző keretrendszer használata

Az elemzéshez a QualityGate[1] rendszerét használtam, mely rendszer megvalósítja *Gyimesi Péter* [3] módszerét. Egy 4 magos, 8 szálas Intel i7-7700-as cpu-val és 16 GB ram-al rendelkező konfiguráción. Azt kaptam eredményül, hogy percenként 6 verziót tud végig elemezni ez a konfiguráció 8 szálas beállítással. Ilyen sebességgel az összes verzió végigelemzése 35 napot venne igénybe, ha csak ezt az egy gépet használnám. Szerencsére sikerült ez a gép mellett még 5 VM-et is munkára fogni, így a becslés 35 napról kevesebb, mint egy hétre redukálódott. Valójában sajnos a teljes művelet jóval több időt vett igénybe, mivel az első néhány kielemzett verzióban még jelentősen kevesebb kódot kellett végig járnia az elemzőnek, mint az időrendben újabb verziókban, így az általam kezdetben mért percenkénti 6 verzió idővel a töredékére csökkent.

Az elemzések futtatása eleinte nem ment zökkenőmentesen, ugyanis a QualityGate jelenleg is fejlesztés alatt álló, legújabb verzióját használtam. Ennek ellenére, miután a néhány nagyobb kezdeti nehézséget sikerült áthidalni, már szinte probléma nélkül végig ment az elemzés minden verzióra. Ez alól kivételt képez öt nagyobb projekt, melyeknek a végigelemzését méretükből adódóan már túl sok időbe telt volna kivitelezni. Ezen projekteken egy verziónak az elemzése már 30 percet, vagy annál is többet vett igénybe úgy, hogy még több ezer revízió lett volna hátra, amit idő szűkében már nem engedhettem meg magamnak. Az ily módon kieső adatmennyiség sajnos elég számottevő volt. Az eredeti adatbázis 12,125 függvény sérülékenységinformációját tartalmazta, míg a kibővített adatbázisba az el nem készült elemzések miatt mindössze 7,123 függvény metrikáit

[1] https://quality-gate.com/

Példa 3.1. run_agents.py

```
import paramiko
# ...

def run_commands_on_agent(ip, user, pwd):
    client = paramiko.SSHClient()
    client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    client.connect(ip, username=user, password=pwd)
    channel = client.invoke_shell()
    # ...

    for command in commands:
        channel.send(command + "\n")
        # ...

    channel.close()
    client.close()

# ...
for ip, usr, pwd in agents:
    run_commands_on_agent(ip, usr, pwd)
```

gyűjtöttem össze.

Az összesen hat gépen való futtatás megkönnyítésére egy Python szkriptet írtam, amiből a 3.1 példában látható egy kis részlet. A szkript SSH kapcsolat segítségével belép az egyik gépre, a szükséges utasítások listáját végrehajtja, majd ugyanezt a műveletsort végrehajtja az összes többi gépen is. Ehhez a Paramiko nevű könyvtárat használtam, ami egy SSHv2 protokollt megvalósító interfészt ad a Python nyelvű programozáshoz. A QualityGate futtatásához szükséges fájlokat feltöltöttem egy git repositoryba, így az előbb említett utasítások többek közt egy git clone parancsból, pár futást ellenőrző szkript és a docker agent elindításából állnak.

3.2. A folyamat metrikák kigyűjtése csv fájlba

Az általam használt mesterséges intelligencia modelleket betanítani képes keretrendszer két fő lépésre bontja a tanulási feladatokat. Az első lépés a Feature Assembling, melynek outputja egy olyan, kizárólag számokat tartalmazó csv file, amit a konkrét tanuló algoritmusok egy az egyben fel tudnak használni a működésük során, a második lépés pedig maga a kívánt algoritmus kiválasztása, paraméterezése és futtatása. A keretrendszer alapvetően csak egy felületet ad az elvégzett feladatok, illetve azok eredményeinek csoportosítására. Igaz, a második lépéshez szükséges tanuló algoritmusok közül a legismertebbek alapértelmezetten meg vannak valósítva benne, viszont, ha valami egyedi dol-

got szeretnénk kivitelezni, például oszloponként egyedi előfeldolgozó lépéseket akarunk végrehajtani az adatainkon, akkor azt egy saját Feature Assembling modul implementálásával tehetjük meg. Az én esetemben is erre volt szükség. A fentebb említett QualityGate forráskód elemző szoftver futtatásának eredményét kellett lekérdeznem az általa biztosított API végpontokról, majd azokat egybegyűjtve csv formátumban elhelyezni a workerek számára is elérhető helyen.

Ez a feladat egy saját modul implementálását kívánta meg. A következő fejezet ezt a részfeladatot mutatja be részletesebben.

4. fejezet

Saját Feature Assembling modul implementálása

A Deep-Water Framework saját modullal való kiegészítéséhez két dologra van szükség. Először is az algoritmusunkat úgy kell megírnunk, hogy megvalósítsa azt az interfészt amit a keretrendszer előír ahhoz, hogy be lehessen kötni a kódunkat a többi előredefiniált modul közé. Második lépésként össze kell állítanunk egy konfigurációs fájlt, ami meghatározza a modul paraméterező felületének kinézetét, valamint az algoritmusunk szükséges, illetve opcionális paramétereit. Miután a leíró fájl elkészült, és az algoritmusunkat bemásoltuk a worker moduljai közé a felület használatának segítségével képesek vagyunk kiadni az elvégzendő feladatot egy worker gépnek.

4.1. Az adatbázis felépítése

Az algoritmusom megtervezésének első lépése az összefésülendő adatbázisok felépítésének megismerése volt. Az függvényekről elérhető információk a 4.1 példában láthatóak felsorolva.

A [2] kutatás eredményeként létrejött adatbázis a *raise_info_columns* listában található oszlopokban tartalmaz a függvények azonosításához szükséges információkat. Erre az adatbázisra a későbbiekben az egyszerűség kedvéért csak eredeti adatbázisként fogok hivatkozni, ugyanis a kutatásom témája ezen adatoknak a kibővítése, illetve az így kapott bővített adatbázis felhasználása a sérülékenység előrejelzés területén. A *name* oszlop a függvények nevét, a *longname* egy jóval hosszabb, adott projekten belül teljesen egyedi

Példa 4.1. CSV info columns

```
raise_info_columns = ['name', 'longname', 'path', 'full_repo_path', 'line', 'column', ↩
    'endline', 'endcolumn']
proc_info_columns = ['hash', 'longname', 'type', 'path', 'start_line', 'start_column',↩
     'end_line', 'end_column']
```

nevet, a *path* oszlop pedig a forrásfájl projekten belüli elérési útját tartalmazza, amiben az adott függvény megtalálható. A *full_repo_path* az imént említett fájlra mutató github linket tárolja az adott vulnerability fixhez tartozó revízióval, később ebből a linkből nyertem ki a kommitot azonosító egyedi SHA-1 kódot. A *line*, *column*, *endline* és *endcolumn* oszlopok pedig rendre a függvényt tartalmazó fájlban a függvény első sorának fájlon belüli relatív sorszámát, oszlopszámát, illetve az utolsó sorának sorszámát és oszlopszámát tárolják.

A QualityGate rendszerből érkező adatokról elérhető információk oszlopnevei a *proc_info_columns* listában vannak felsorolva. A *hash* oszlop a függvényt tartalmazó revízió SHA-1 kódját, a *longname* pedig a fentiekhez hasonlóan egy adott projekten belül egyedi nevet tartalmaz. A *type* az adatok granularitásának mértékét jelzi (Package, Component, File, Class, Function, stb.), esetemben ez a függvényszintű információkat jelenti. A *path* oszlopban itt is a függvényt tartalmazó fájl projekten belüli elérési útját találjuk. A *start_line*, *start_column*, *end_line* és *end_column* oszlopok úgyszintén, a fentiekhez hasonlóan a függvényt tartalmazó fájlban a függvény első sorának sorszámát, oszlopszámát, illetve az utolsó sorának sorszámát és oszlopszámát tárolják.

4.2. Az algoritmus implementálása

Esetemben az algoritmusnak négy darab paramétere van. Először is meg kell adni az eredeti CSV formátumú adatbázis fájlra mutató útvonalat, illetve egy könyvtár elérési utat, ahova az elkészült bővített adatbázis kerül. Ezek a paraméterek nem opcionálisak, megadásuk elengedhetetlen az algoritmus működéséhez. Emellett még két opcionális paramétere van az algoritmusomnak, egyikkel a kívánt metrikákat lehet megadni (a paraméter elhagyásával minden elérhető metrika bekerül az eredményül kapott adatbázisba), a másik paraméterrel pedig megadható, hogy mely revíziószámú projekteket szeretnénk felhasználni az algoritmus működése során (a paramétert elhagyva szintén az összes elér-

Példa 4.2. data dictionary

```
data = {
  "project_hash_1": {
    "start_line-start_column-end_line-end_column_1": vuln_data_1,
    "start_line-start_column-end_line-end_column_2": vuln_data_2,
    ...
  },
  "project_hash_2": {
    "start_line-start_column-end_line-end_column_3": vuln_data_3,
    ...
  },
  ...
}
```

hető projekt bekerül az adatbázisba, akárcsak a metrikák esetében).

Maga az algoritmus működését tekintve annyit tesz, hogy minden projektre meghívja a QualityGate által biztosított API végpontot, melynek eredménye egy a folyamat metrikákat tartalmazó csv fájl zip formátumba tömörítve. Az így kapott tömörített állományokat kicsomagolja és összefésüli az eredeti, sérülékenység információkat tartalmazó adatbázissal. Ezután az összefésült fájlt elhelyezi a megosztott hálózati meghajtón, majd pedig visszatér az eredményül létrejött fájl elérési útjával.

Ehhez első lépésként meg kellett találnom azokat az oszlopokat, amik alapján a két helyről érkező információkat össze tudom egyeztetni. Első ránézésre az adatbázis *longname* nevű oszlopa kézenfekvő megoldásnak tűnt, ám hamar kiderült, hogy a két adatbázis a SourceMeter forráskódelemző szoftver két eltérő verzióját használta, így ezek az értékek teljes mértékben eltérőek voltak. Egyéb próbálkozások után a működő megoldás végül az lett, hogy adott projekten belül a sor és oszlop információkat használom egyedi kulcsként *start_line-start_column-end_line-end_column* formában. Az egyező sorok megtalálásához a Python nyelv egy beépített adattípusát, a *Dictionary*t használtam. Pythonban a Dictionary adattípus hash táblát használ a kulcsok tárolására, így maximalizálva a keresés hatékonyságát. Kulcsoknak az adott revíziókat azonosító SHA-1 kódokat adtam meg. Minden kulcs azonosít egy-egy Dictionary-t, amiben a kulcsok adott függvényeket azonosítanak, értékeik pedig az adatbázisból az adott függvényekhez tartozó statikus és folyamat metrikák.

Ezek után az eredeti adatbázist beolvastam és elkészítettem a hozzátartozó adatstruktúrát, ami a 4.2 példában látható. Ezután a QualityGate felől érkező adatokon végig iterálva elmentettem a mindkét adatbázisban egyaránt megtalálható függvényekről elérhető adatokat. Az így előállt összefésült adatbázist kiírtam egy csv fájlba *pro-*

cess_metrics_merged_full.csv néven. Erre azért volt szükség, mivel a DWF keretrendszer egy olyan CSV fájlt vár bemenetéül, ami kizárólag numerikus értékeket tárol. Ahhoz tehát, hogy a tanulásokhoz fel lehessen használni a létrejött új adatbázist, el kellett távolítanom a szöveges adatokat tartalmazó, illetve a tanulás szempontjából irreleváns oszlopokat. Ez utóbbiak a kezdő sor, kezdő oszlop, utolsó sor és utolsó oszlop értékek voltak, melyek segítségével az egyedi kulcsot állítottam elő az adatok összefésüléséhez. Ezen adatok igaz numerikus értékek, viszont a tanulás szempontjából irrelevánsak. A szükségtelen oszlopok eltávolításával létrejött fájlt *process_metrics_merged_data.csv*-nek neveztem el, jelezve, hogy ez a fájl, csak a nyers adatokat tárolja. Az így kapott adatbázisban viszont elvesznek a konkrét függvények beazonosításához szükséges információk, ennek a problémának az orvoslására szolgál a korábban említett *process_metrics_merged_full.csv* nevű CSV állomány.

4.3. A felületet leíró fájl elkészítése

Az algoritmus DWF-be integrálásának végső lépése egy a felületet leíró fájl elkészítése. Ez a lépés abból áll, hogy adunk egy nevet az algoritmusnak, ami a felületen fog megjelenni, majd egy listában felsoroljuk az algoritmus paramétereit. Esetemben a *JavaScript Process Metrics* elnevezést adtam meg az algoritmus DWF felületen megjelenített nevének. A paraméterek listájába 4 elem került, két útvonal, illetve két szöveg típusú paraméter. A két útvonal közül az egyik az eredeti adatbázist tartalmazó CSV fájlra kell mutasson, a másik pedig egy könyvtárra, ahova a kimeneti CSV fájl kerül. Ez a két paraméter kötelezően kitöltendő, az algoritmus nem is tudna ezek nélkül működni. A másik két paraméter csak opcionális, a kitöltésük nem kötelező, az eredményül kapott adatbázist tudjuk velük szűrni. Az első segítségével megadhatóak azok a metrikák, amiket szeretnénk az eredményben látni, a második pedig a projektekre való szűrést teszi lehetővé azáltal, hogy segítségével megadhatók a tanuláshoz felhasználni kívánt revíziókat azonosító SHA-1 kódok.

Így a folyamat metrikákat kinyerő DWF modul fejlesztésének lépésével el is készültem. Következő feladatként a mesterséges intelligencia modellek betanítására, kiértékelésére, valamint összehasonlítására fókuszáltam.

5. fejezet

Kiértékelés a DWF használatával

A következőkben az általam fejlesztett tanuló keretrendszer használatát mutatom be részletesebben egy konkrét példán keresztül. Egyúttal a folyamat metrikák jelentőségét is megfigyelhetjük a kapott eredményekből.

5.1. DWF szerver indítása, saját modul importálása

Ahhoz, hogy az elkészült modult használni tudjuk, az elkészült felület leíró fájlt el kell helyezzük a *DWF-server/strategies*, illetve a *DWF-client/strategies* mappában egyaránt. Az algoritmust tartalmazó Python fájlt pedig a *FeatureAssembler/fstrategies* könyvtárban. Miután ezzel megvagyunk, az adatok tárolásáért, rendszerezéséért, illetve a feladatok párhuzamosításáért felelős szervert kell elindítanunk. Ezt a *DWF-server* mappán belüli *build_and_run.bat* (windows), vagy *build_and_run.sh* (linux) script-el tehetjük meg. A script egyrészt elindítja az adatbázist szolgáltató Elasticsearch konténert, valamint az adatok vizualizációját lehetővé tevő Kibana konténert. Továbbá elkészít egy Dockerimage-et, amibe bemásolja az általunk készített modullal kibővített szerver kódját, majd elindítja azt. Ezek után a szerver elérhető a localhost:4000-es címen, a Kibana felülete pedig a localhost:5631 címen.

5.2. Workerek indítása

Workereket kétféleképpen csatlakoztathatunk a klaszterbe. Egyik lehetőség, hogy a *DWF-client* mappában levő *dwf_client.py* szkriptet kézzel futtatjuk, vagy a másik lehe-

tőség, hogy a projekt könyvtár gyökerében található *run_workers.bat* (windows), vagy *run_workers.sh* (linux) szkriptek egyikével egyszerre több workert is indíthatunk egy adott számítógépen. Amennyiben a windows-os verziót használjuk, parancssori argumentumként megadhatjuk hány workert szeretnénk indítani az adott gépen, míg a linuxos verzió fixen négyet indít.

5.3. Experiment létrehozása

Az experiment egy a keretrendszer által definiált fogalom, mely tanítások egy adott halmazát fogja össze. Új experiment létrehozását a fejlécben található *New Experiment* gombra kattintva indíthatjuk el. A nevének egyedinek kell lennie, amiről a név megadása után azonnal kapunk is visszajelzést a felülettől.

A Priority lenyíló menüben az experimentünk fontosságát állíthatjuk be. Alapértelmezetten több párhuzamosan futó experimet esetén a feladatok egyforma gyakorisággal kerülnek kiosztásra. Abban az esetben, ha több különböző prioritással hozunk létre experimenteket, akkor a szerver egy alacsony, három normál, illetve hat magas prioritású feladatot oszt ki egy időben. Az immediate prioritású experimentek viszont a klaszter minden erőforrását megkapják, tehát mikor folyamatban van egy ilyen experimentbeli feladat, akkor az minden alacsonyabb prioritású experimentbeli feladattal szemben előnyt élvez.

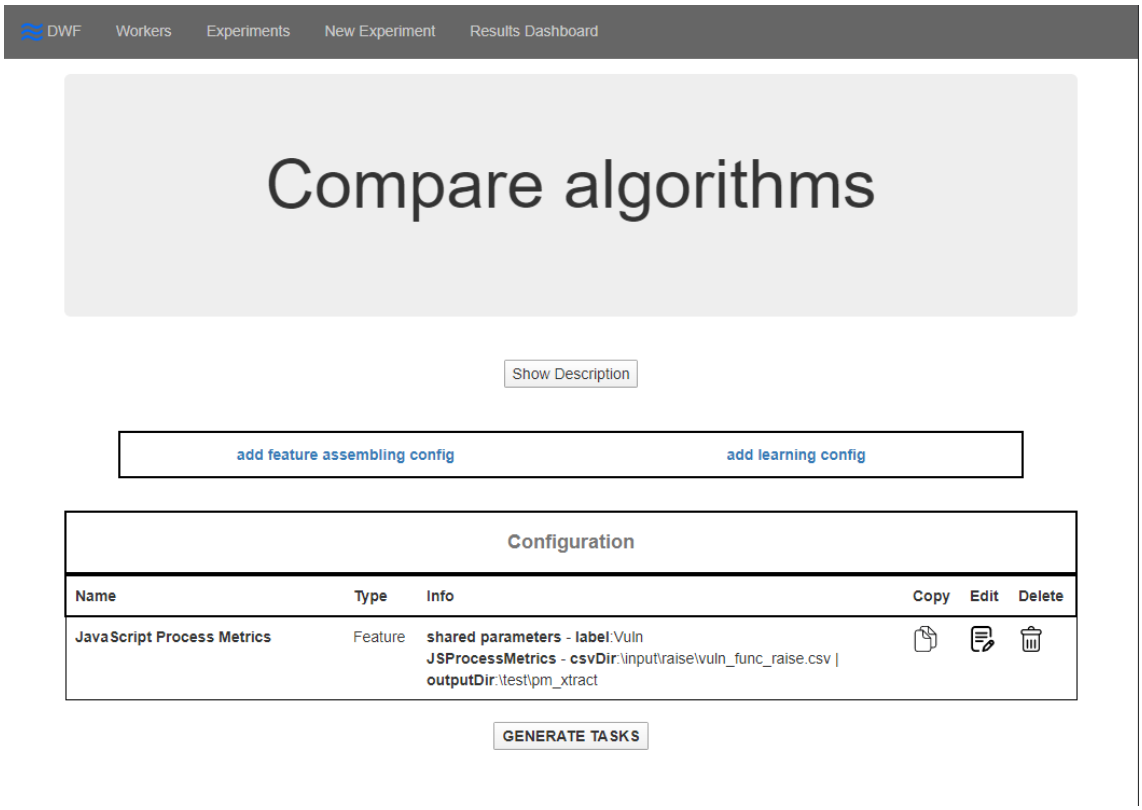
A Markdown bemeneti mezőben pedig megadhatunk egy leírást az experimentünkről, amiben leírhatjuk, hogy miről is szól az adott kutatási feladatunk. A formázás eredménye valós időben látható a bemeneti mező alatt, ezzel segítve a szövegszerkesztést arra az esetre, ha kézzel szeretnénk azt megadni.

Én most az előző fejezetben elkészült modult használva szeretnék futtatni néhány tanítást azzal a céllal, hogy a tanuló algoritmusok hatékonyságáról kapjak egy megközelítőleges eredményt. Az experimentemnek a *Compare algorithms* nevet adtam, a prioritást pedig normal-on hagytam. Ezután a következő lépés a feladatok felvétele az újonnan létrehozott experimentbe.

5.4. Feature Assembling lépés hozzáadása

A tanulási feladatok két lépésből tevődnek össze, az első lépés a tanító adatokat tartalmazó input fájl előállítása, a második lépés pedig a tanuló algoritmusok betanítása. Az első lépés más néven a *Feature Assembling* kimenete egy csv fájl, ami kizárólag numerikus adatokat tartalmaz. Amennyiben rendelkezünk egy ilyen fájllal és semmilyen előfeldolgozó lépést nem kívánunk rajta elvégezni, akkor az első lépés megadásakor választhatjuk a *Manual File Input* lehetőséget is.

Esetemben minden egyes feladat bemenetét az előző fejezetben elkészített algoritmus által előállított CSV fájl adja. Ha megfelelően importáltuk a modult, a Feature Asembling konfiguráció hozzáadása gomb megnyomása után a lenyíló menüben meg is jelenik az algoritmus. Először még egy fontos információt meg kell adni ezen a felületen, mégpedig annak a kimeneti fájlban található oszlopnak a nevét, amely az osztálycímkéket tartalmazza, azaz, hogy melyik oszlopra szeretnénk úgymond rátanulni. A választott oszlop nevét a *Label column* mezőbe írhatjuk.



5.1. ábra. Konfigurációk hozzáadása

A címke oszlop, illetve az algoritmus paramétereinek kitöltése, majd a *Submit* gomb megnyomása után az első lépés megadásával el is készültem (lásd 5.1 ábra). Következő

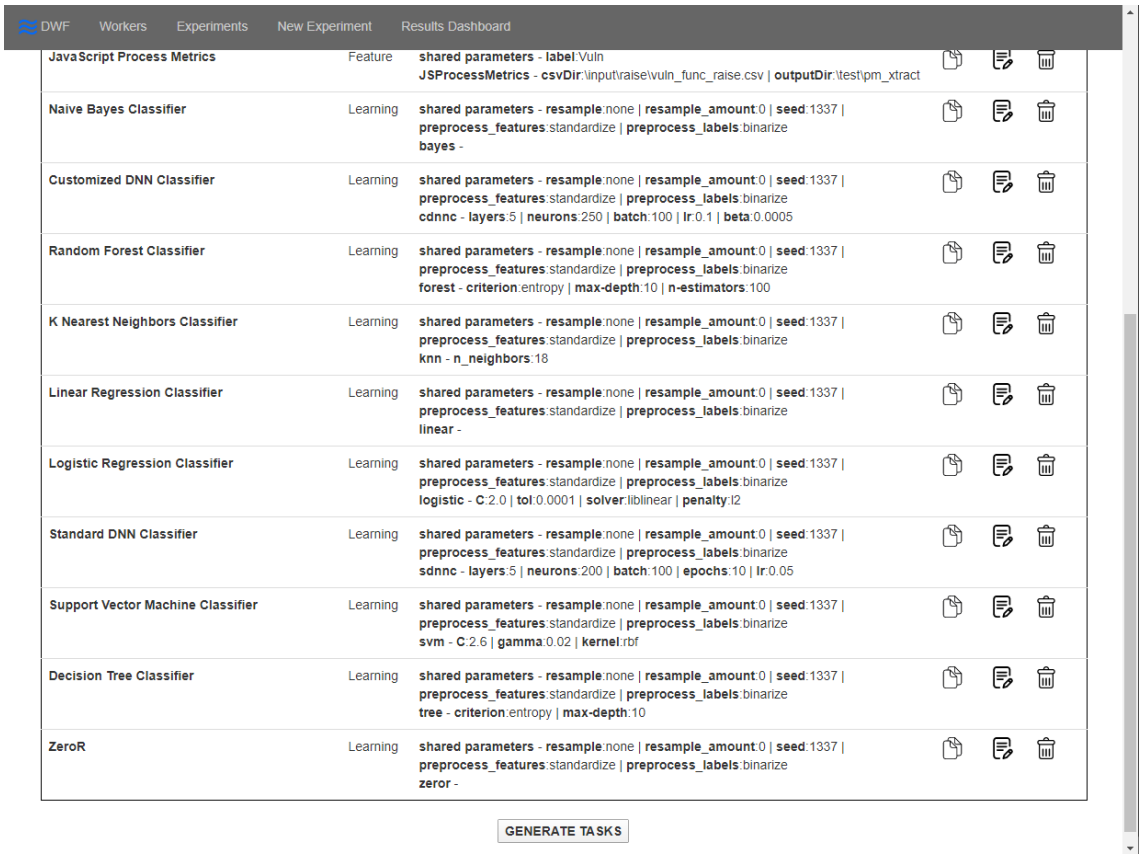
lépés a tanuló algoritmusok, illetve azok paramétereinek definiálása.

5.5. Tanuló algoritmusok hozzáadása

Tanuló algoritmus hozzáadása az *add learning config* gombbal lehetséges (lásd 5.1 ábra). A gomb megnyomását követően egy felület fogad minket, ahol az elérhető algoritmusokból választhatunk, illetve megadhatjuk a választott algoritmus paramétereit. Ehhez a *Strategy parameters* rész *Select strategy* lenyíló menüjéből kell választanunk egyet. Tízféle algoritmusból választhatunk, többek közt két neuronhálós implementációból, két döntési fa alapú algoritmusból, de akár a legegyszerűbb ZeroR algoritmust is választhatjuk, ami minden példányt nullával osztályoz. A kívánt algoritmust kiválasztva megjelennek annak paraméterei, melyek közül a kötelező paramétereknek meg vannak adva alapértelmezett értékek, így akár azonnal rá is nyomhatunk a *Submit* gombra. Igaz van pár algoritmus ami nem rendelkezik paraméterekkel, a többi algoritmus esetében egy extra kényelmi funkció érhető el *Configuration presets* néven. Három eltérő méretű előre megadott konfigurációs halmaz közül választhatunk, név szerint kis-, közepes- és nagy méretű halmazok közül. A könnyebb paraméter megadás érdekében a lenyíló menükből több elem is választható egyidejűleg, illetve a numerikus értékeket lehetőség van intervallumként is megadni kezdő-, befejező- és lépésköz értékekkel.

Ezen kívül a *Shared parameters* rész első paraméterként választhatunk egyet a *Resample Type* lenyíló menüből. Resample alatt a bináris osztályok esetén történő "újra mintavételezést" értjük. Felfelé történő újra mintavételezés esetén a kisebb példányszámú osztály elemeiből adott mennyiséget véletlenszerűen kiválasztva, a választott elemeket megduplázzuk, így csökkentve az elemszámbeli eltérést, lefelé irányban hasonló dolog történik, a nagyobb példányszámú osztály elemeiből törlünk adott mennyiséget véletlenszerűen. Ezt a mennyiséget a *Resample Amount* mezőben adhatjuk meg, amely mező csak akkor jelenik meg, ha választottunk egy irányt a lenyíló menüből. Az alapértelmezetten kiválasztott érték a *None*, ezt akár így is hagyhatjuk. A *Random Seed* érték megadásával biztosíthatjuk, hogy az elért eredmények megismételhetőek legyenek. A *Preprocess Features* és *Preprocess Labels* lenyíló menükből extra előfeldolgozó műveleteket definiálhatunk.

Miután megadtunk minden szükséges paramétert, a *Submit* gombra kattintva felve-



5.2. ábra. Tanulóalgoritmusok hozzáadva

hetjük a konfiguráció(kat) az experimenthez hozzáadni kívánt feladatok közé. Én mind a 10 elérhető algoritmushoz felvettem az alapértelmezett értékekkel egy-egy konfigurációt (lásd 5.2 ábra).

Ebben a fázisban még az experiment szerkeszthető, azaz lehet hozzáadni, módosítani, vagy törölni konfigurációkat. A *GENERATE TASKS* gombra kattintva elkészülnek a konfigurációk alapján a tanulási feladatok. Ezután az experiment már nem szerkeszthető. A legenerált feladatok egyesével, vagy akár egyszerre is elindíthatóak. A feladatok sorában található zöld színű indítás gombbal feladatonként, vagy a táblázat fejlécében található *Run all* felirat melletti zöld, 3-as indítás gombra kattintva egyszerre is elindíthatjuk az experiment feldolgozását.

Ezután a következő lépés a kapott eredmények kiértékelése.

5.6. A tanulások eredményeinek kiértékelése

Amint a feladatok táblázatának *Status* oszlopában minden sorban a *Task Completed* feliratot látjuk, neki is láthatunk az eredmények kiértékelésének. Ezt a *Summary* gombra

5.1. táblázat. Tanuló algoritmusok összehasonlítása 1/2.

classifier	true positive	true negative	false positive	false negative
Random Forest	698	6169	22	233
Decision Tree	**719**	6119	72	**212**
Customized DNN	697	6143	48	234
Standard DNN	699	6117	74	232
K Nearest Neighbors	609	6171	20	322
Support Vector Machine	559	6170	21	372
Logistic Regression	345	6128	63	586
Linear Regression	282	6163	28	649
Naive Bayes	82	5976	215	849
ZeroR	0	**6191**	**0**	931

5.2. táblázat. Tanuló algoritmusok összehasonlítása 2/2.

classifier	accuracy	precision	recall	fmes
Random Forest	**96.420%**	**96.944%**	74.973%	**84.555%**
Decision Tree	96.012%	90.898%	**77.229%**	83.508%
Customized DNN	96.040%	93.557%	74.866%	83.174%
Standard DNN	95.703%	90.427%	75.081%	82.042%
K Nearest Neighbors	95.198%	96.820%	65.414%	78.077%
Support Vector Machine	94.482%	96.379%	60.043%	73.991%
Logistic Regression	90.887%	84.559%	37.057%	51.531%
Linear Regression	90.494%	90.968%	30.290%	45.447%
Naive Bayes	85.060%	27.609%	8.808%	13.355%
ZeroR	86.928%	0.000%	0.000%	0.000%

kattintva tehetjük meg.

Esetemben a 10 algoritmus versenyzett egymással a felület által ajánlott alapértelmezett paramétereket felhasználva. Az eredmények az 5.1 és az 5.2 táblázatban láthatóan alakultak. Oszloponként az elért legjobb értékeket vastagított betűvel emeltem ki a könnyebb olvashatóság kedvéért. Az alapbeállításokkal a Random Forest osztályozó teljesített a legjobban F-measure tekintetében.

Most, hogy a keretrendszer használatát bemutattam nincs más hátra, mint elvégezni pár tanítást, a lehető legjobb paraméterek megtalálására. Ezekről a tanításokról szól a következő fejezet.

6. fejezet

További kutatási eredmények

Az első próbálkozás eredményeit tekintve a döntési fa alapú módszerek tűnnek a legígéretesebbnek, viszont a paraméterek módosításával ez még változhat. Az egyes paraméterek módosításainak eredményét mutatja be ez a fejezet. Először a Resample Type legjobb értékét és irányát határoztam meg, majd pedig az egyes tanulók legjobb paramétereit.

6.1. táblázat. Resample Type F-measure szerint rendezve.

task	type	amount	precision	fmes	parameters
Random Forest	up	50	93.082%	**85.747%**	max-depth:15 ǀ n-estimators:50
Random Forest	none	-	97.131%	85.508%	max-depth 15 ǀ n-estimators 50
SVM	up	50	86.228%	80.023%	C 2.6 ǀ gamma 0.02 ǀ kernel rbf
SVM	none	-	96.660%	68.333%	C 1.0 ǀ degree 2 ǀ kernel poly
K-NN	none	-	93.605%	82.593%	n_neighbors 6 ǀ weights distance
K-NN	none	-	**97.800%**	75.953%	n_neighbors 32
Decision Tree	none	-	94.024%	84.086%	criterion entropy ǀ max-depth 8

6.1. Legjobb Resample Type érték meghatározása

A Resample Type értékeinek beállításához véletlenszerűen választottam pár osztályozót. F-measure szerint rendezve a felfelé történő újramintavételezés tűnt a legjobbnak, viszont Precision-t tekintve sokkal jobban teljesítettek az osztályozók Resample Type megadása nélkül. Mivel a cél a sérülékenységek előrejelzése, így a jobb Precision sokkal nagyobb értékkel bír, mint az összesített teljesítménye az algoritmusoknak, ugyanis a precision értéke annál nagyobb, minél nagyobb százalékban valóban sérülékenyek a sérülékenynek jelzett függvények, azaz minél kevesebb függvényt jelez tévesen sérülékenynek. A 6.1

táblázatban algoritmusonként láthatóak a legjobb F-measure, illetve Precisiont elérő beállítások. Precision szerint rendezve az újramintavételezés nélküli tanulások érték el a legjobb eredményeket, illetve néhány esetben még F-measure szerint is. Ahogy a táblázatból kivehető, a legjobb F-measure értéket Random Forest algoritmussal lehet elérni, míg a legjobb Precision-t a K legközelebbi szomszéd algoritmus produkálta. A következő fejezetben a dolgozat témáját foglalom össze röviden.

7. fejezet

Konklúzió és jövőbeli tervek

Először egy általános körképet mutattam a vállalati partnerek esetén felmerülő problémákról, illetve egy új meglátást aminek segítségével hatékonyan lehetne kezelni ezeket a helyzeteket.

Ezek után a folyamat metrikák kinyeréséhez elérhető eszköz működését és használatát ecseteltem részletesebben. Majd az általam fejlesztett keretrendszer bővítésének lépéseit részleteztem.

Később a megvalósított modul használatát, illetve maga a keretrendszer alapvető képességeit és a felépítését mutattam be. Majd a következőkben részletesen, példákon keresztül a megvalósított módszeremet működését mutattam be. Végezetül a rendszer teljesítményét értékeltem ki különböző tanuló algoritmusokat futtatva, kiemelve az algoritmusok által produkált eredmények közti lényegi különbségeket.

Jövőbeli terveim között szerepel a módszer és az elért eredmények publikálása, illetve TDK dolgozatom bemutatása a szeptemberben megrendezésre kerülő konferencián. Ezen felül számos további fejlesztési lehetőség kínálkozik a rendszer továbbfejlesztésére. Újabb metrikák kialakítását lehet kivitelezni, a jelenlegi rendszer hibáit korrigálni. Valamint további tanulóalgoritmusokat lehet kipróbálni, esetleg tovább finomítani a korábbi paraméterbeállításokon.

Összegzésképpen, a korábbi fejezetekben bemutattam egy olyan új sérülékenység előrejelző technikát, amely a korábbi statikus metrikák használatához képest óriási előrelépést jelent a tudomány ezen ágazatában. Az DWF keretrendszer úgy lett megtervezve, hogy könnyedén integrálhatók legyenek a legkülönfélébb előfeldolgozó, avagy Feature Assembling algoritmusok, illetve számos egyéb szolgáltatást is biztosít, mint például az

eredmények összehasonlítása, illetve vizualizációja. A rendszer megadja azt a lehetőséget, hogy bizonyos megszorítások mellett a felhasználók használni tudjanak különböző egyedi algoritmusokat is a kutatásaik során, miközben a megszokott funkciókat használatát is élvezhetik.

Irodalomjegyzék

[1] S. Delphine Immaculate, M. Farida Begam, and M. Floramary. Software bug prediction using supervised machine learning algorithms. In *2019 International Conference on Data Science and Communication (IconDSC)*, pages 1–7, 2019.

[2] Rudolf Ferenc, Péter Hegedüs, Péter Gyimesi, Gabor Antal, Dénes Bán, and Tibor Gyimothy. Challenging machine learning algorithms in predicting vulnerable javascript functions. pages 8–14, 05 2019.

[3] Péter Gyimesi. Automatic calculation of process metrics and their bug prediction capabilities. *Acta Cybernetica*, 23:537–559, 01 2017.

[4] Awni Hammouri, Mustafa Hammad, Mohammad Alnabhan, and Fatima Alsarayrah. Software bug prediction using machine learning approach. *International Journal of Advanced Computer Science and Applications*, 9, 01 2018.

[5] Marian Jureczko and Lech Madeyski. A review of process metrics in defect prediction studies. *Methods of Applied Computer Science*, 5:133–145, 01 2011.

[6] Nancy Mead, Julia Allen, Mark Ardis, Thomas Hilburn, Andrew Kornecki, Rick Linger, and James McDonald. Software assurance curriculum project volume 1: Master of software assurance reference curriculum. page 154, 08 2010.

[7] Foyzur Rahman and Premkumar Devanbu. How, and why, process metrics are better. pages 432–441, 05 2013.

Köszönetnyilvánítás

Ez a diplomamunka a SETIT projekt (2018-1.2.1-NKP-2018-00004)[1] keretében készült. Szeretnénk külön köszönetet mondani munkatársamnak Aladics Tamásnak és Gyimesi Péternek, akik hozzájárulásukkal lehetővé tették a DWF keretrendszernek a megvalósulását. Továbbá, szeretnénk köszönetet mondani témavezetőimnek Dr. Hegedűs Péternek és Dr. Ferenc Rudolfnak, akik szakmai támogatásukkal segítették ennek a projektnek a sikerét.

[1] A 2018-1.2.1-NKP-2018-00004 számú projekt a Nemzeti Kutatási és Innovációs Alapból biztosított támogatással, a "Nemzeti Kiválósági Program: 2018-1.2.1-NKP" pályázati program finanszírozásában valósult meg.