TDK dolgozat

Viszkok Tamás

Sérülékenység előrejelzés JavaScript programokban folyamat metrikák segítségével

Készítette:

Viszkok Tamás

II. évf. programtervező informatikus MSc

Témavezetők:

***Dr. Ferenc Rudolf**, tanszékvezető egyetemi docens*

***Dr. Hegedűs Péter**, tudományos munkatárs*

Szegedi Tudományegyetem

Természettudományi és Informatikai Kar

Szoftverfejlesztés Tanszék

2020

Tartalomjegyzék

1. fejezet

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.

Szerencsére két igen fontos jelenség együttállása segít a probléma kezelésében. Egyrészt korábbi kutatások szerint [7] 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.

Egy korábbi kutatás [2] során létrehozott JavaScript függvények statikus metrikáit tartalmazó adatbázist használtam kiindulópontként. A kutatók ezen adatbázis használatával próbálták JavaScript nyelven írt szoftverek sérülékenységét függvényszinten előre jelezni statikus kódmetrikák segítségével. Én ezt az adatbázist bővítettem tovább az úgynevezett folyamat metri-

kákkal.

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 [6, 9], 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, 84.8%-os F-measure értékkel (95.8% precision, 76.0% recall).

1.1. Dolgozat felépítése

A következő fejezetben ismertetem, hogy milyen igények vezettek a módszer kifejlesztéséhez. Majd a 3. 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 4. fejezetben a modellek betanításához összegyűjtött adatbázis előállításáról, illetve az általam használt metrikákkal való kiegészítéséről lesz szó. Majd az 5. fejezetben a kapott modellek eredményeit hasonlítom össze. Végezetül a 6. fejezetben összegzem az elért eredményeket és néhány jövőbeli fejlesztési ötletet vázolok fel.

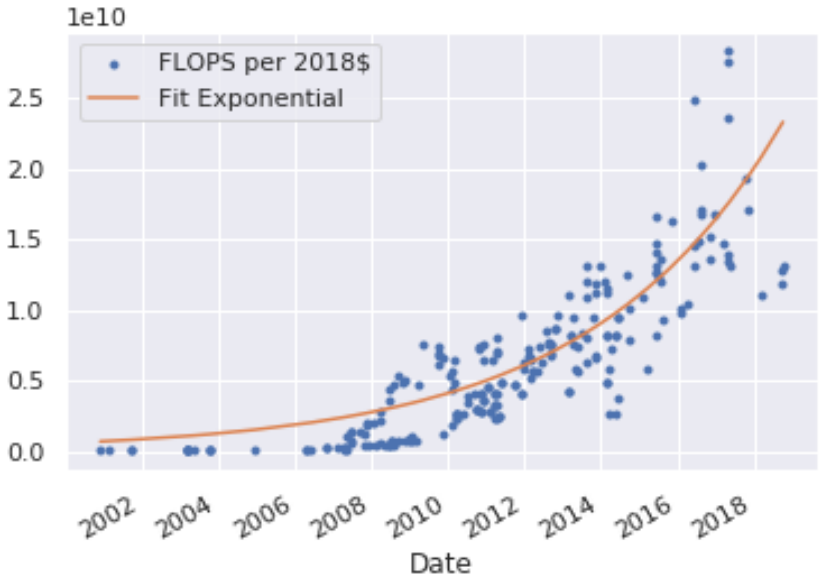
2. fejezet

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.



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

2.1. A mesterséges intelligencia térhódítása

Szerencsére a mesterséges intelligencia virágkorát éljük, a tanulóalgoritmusok használatához szükséges nagy számítási kapacitású eszközök egyre elérhetőbb áron szerezhetők be (lásd 2.1 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. Ezzel szemben a sérülékenység, mint olyan, sok esetben nem tekinthető a hagyományos értelemben vett szoftverhibának, így ezek a módszerek módosítás nélkül nem alkalmazhatóak megfelelő hatékonysággal a sérülékenységek ellen [8, 10]. A megoldás egy új modell betanítása.

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

3. fejezet

Kapcsolódó munkák

A mesterséges intelligencia használatával történő hibadetektálás életképességét korábbi kutatások már bizonyították [1, 5]. 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.

3.1. 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.

3.2. Eszköz a folyamat metrikák kinyeréséhez

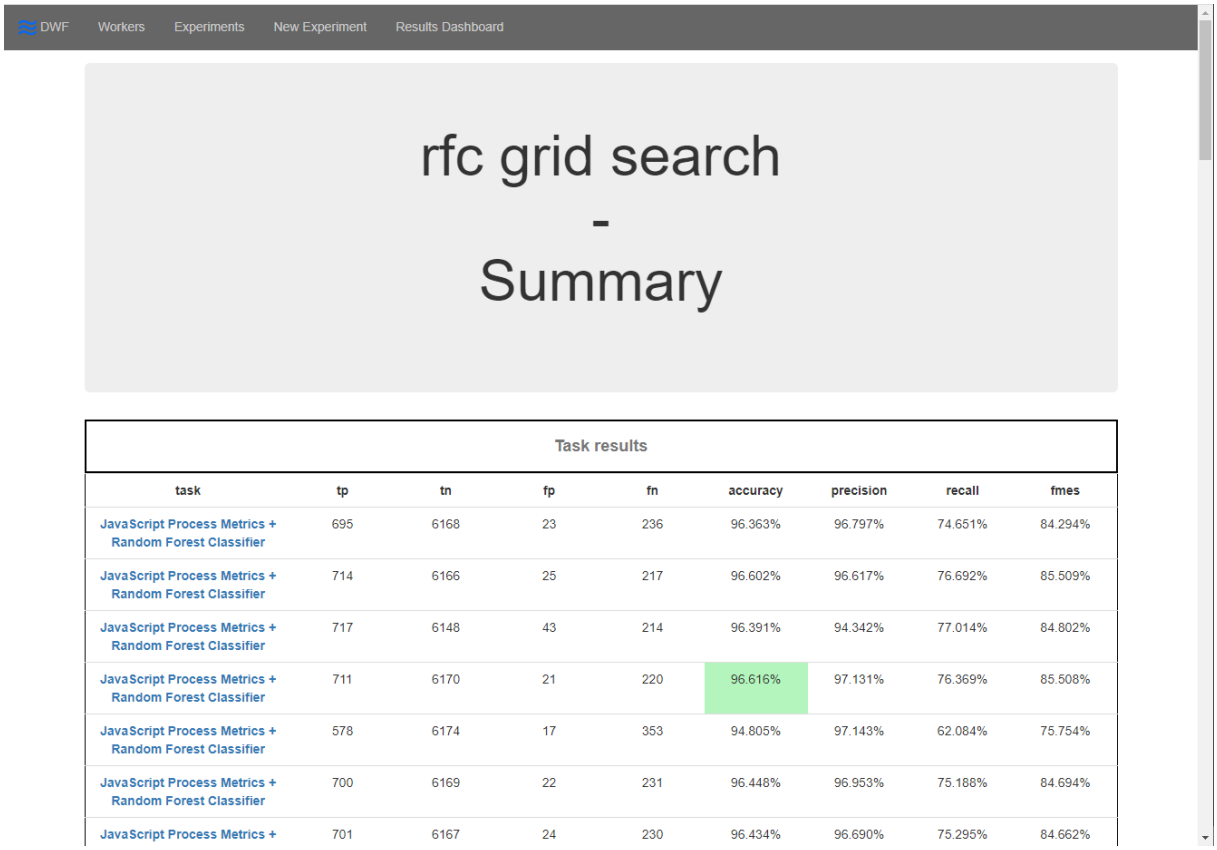
A bevezető részben említett, folyamat metrikák használhatóságáról szóló kutatások [6, 9] eredményei arra engednek következtetni, hogy az adatbázisunkat kibővítve ezen metrikákkal akár nagyságrendekkel jobb eredmények is elérhetőek lennének. Az imént említett adatbázis kibővítéséhez *Gyimesi Péter* [4] 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.

3.3. 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 közel 120 tanítást jelent. Minden ilyen tanulási fázist 5 különböző újramintavételezési módszer használatával futtattam, ami összesen közel 600 tanulási folyamatot jelent. Ezt a közel 600 tanítást kézzel elindítani, majd a kapott eredményeket szintén kézzel összegyűjteni és összehasonlítani elég kényelmetlen és hosszadalmas művelet lenne. Szerencsére a Deep-Water Framework[2] [3] 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 3.1 ábra).

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

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



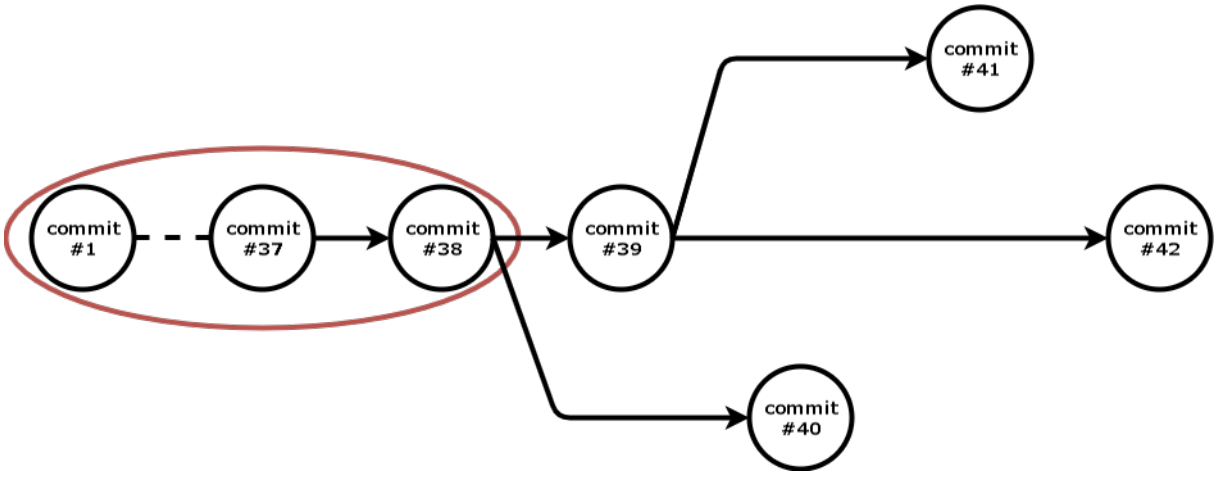
3.1. ábra. Deep Water Framework - Summary nézet

4. 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.



4.1. 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 megegyeznek (lásd 4.1 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.

4.1. Forráskód elemző keretrendszer használata

Az elemzéshez a QualityGate[1] rendszerét használtam, mely rendszer megvalósítja *Gyimesi Péter* [4] 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 8,038 függvény metrikáit gyűjtöttem össze.

Az összesen hat gépen való futtatás megkönnyítésére egy Python szkriptet írtam, amiből

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

Példa 4.1. 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)
```

a 4.1 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 között egy git clone parancsból, pár futást ellenőrző szkript és a docker agent elindításából állnak.

4.2. 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 dolgot 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 le-

kérdeznem 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.

5. fejezet

Eredmények

10 különböző modellt tanítottam be az immáron kibővített, statikus és folyamat metrikákat is tartalmazó adathalmazon. Először is a ZeroR osztályozót, mely egy alsó korlátot jelent a többi osztályozó számára, ugyanis a ZeroR mindenféle logikát nélkülözve, mindössze a leggyakoribb osztályt választja címkézéskor. Ez a modell 86.928% accuracy értéket produkált, azaz a későbbiek során felesleges foglalkozni bármilyen ennél gyengébben teljesítő algoritmussal.

Ezután a kísérletet a default paraméterezésekkel elért eredmények összehasonlításával folytattam, hogy kapjak egy hozzávetőleges képet a modellek egymáshoz viszonyított predikciós erejéről. Ahogy az 5.1. táblázatban látható, a legjobb összesített eredményt, azaz F-measure értéket a Random Forest osztályozó érte el. A legrosszabb eredményt a Naive Bayes érte el, aminek sikerült még a ZeroR eredményeit is alulmúlnia.

Egy másik figyelemre méltó eredmény a fals pozitívak oszlopa, ahol a legjobb eredményt az SVM osztályozó érte el, majd azt követi a k-NN. Annak ellenére, hogy a k-NN és az SVM F-measure értékei elmaradnak a Random Forest által elért eredménytől, ez a két osztályozó

5.1. táblázat. Eredmények default paraméterekkel.

classifier	TP	TN	FP	FN	accuracy	precision	recall	F-measure
Random Forest	699	7054	24	261	**96.5**%	96.7%	72.8%	**83.1**%
Decision Tree	**723**	7006	72	**237**	96.2%	90.9%	**75.3**%	82.4%
Customized DNN	685	7027	51	275	95.9%	93.1%	71.4%	80.8%
Standard DNN	665	7037	41	295	95.8%	94.2%	69.3%	79.8%
k-NN	613	7059	19	347	95.5%	**97.0**%	63.9%	77.0%
SVM	548	**7060**	**18**	412	94.7%	96.8%	57.1%	71.8%
LogReg	332	7007	71	628	91.3%	82.4%	34.6%	48.7%
LinReg	274	7051	27	686	91.1%	91.0%	28.5%	43.5%
Naive Bayes	115	6779	299	845	85.8%	27.8%	12.0%	16.7%

rendelkezik a legmagasabb precision értékkel. Ebből következik, hogy ezek az osztályozók nagyobb eséllyel hunynak szemet bizonyos sérülékenységek felett, cserébe a téves riasztások száma jelentősen kisebb. Felhasználás szempontjából talán ez a legfontosabb szempont, ugyanis a téves riasztások bizalmatlanságot alakíthatnak ki a felhasználókban, aminek következtében akár fel is hagyhatnak a sérülékenység előrejelző szoftverünk használatával.

5.1. Az újramintavételezés hatásai

A Deep-Water framework lehetőséget nyújt az úgy nevezett újramintavételezés használatára. Az újramintavételezésnek két paramétere van. Az egyik az iránya (fel, vagy le), a másik pedig a mennyisége. Felfelé történő újramintavételezés során a kisebb elemszámú osztály adott elemeit megduplázzuk. Lefelé történő esetben pedig a nagyobb számosságú osztály elemei közül törlünk néhányat. A mennyiséggel pedig megadható ezen duplázások, avagy törlések száma. A mennyiség egy százalékos érték, a kisebb számosságú osztály elemszámának adott százalékát jelenti.

Először is a Random Forest osztályozón teszteltem a különböző újramintavételezés hatásait, mivel ez az algoritmus hozta a legjobb eredményt a default paraméteres teszten. Megfigyeltem, hogy a lefele irányuló esetben, azaz a nagyobb elemszámú osztályból való törléssel való osztályok közti eloszlás optimalizálással nagyobb recall értéket sikerült elérni, azaz a sérülékeny függvények nagyobb százalékban kerültek felismerésre. Sajnos ez jóval magasabb fals pozitív jelzéssel járt együtt, azaz a modell nagyobb eséllyel jelzett sérülékenynek olyan függvényeket, melyek nem voltak hibásak, ami elég zavaró lehet felhasználói szemmel. A többi mérőszám tekintetében a legjobb eredmények újramintavételezés nélkül születtek, így a továbbiakban ezt a beállítást használtam.

5.2. Az eredmények kiértékelése

Végül grid-search segítségével osztályozónként külön-külön próbáltam megtalálni a lehető legjobb paraméterezéseket. Ahogy az 5.2. táblázatban megfigyelhető, a legelső kísérletben, default paraméterek használatával futtatott tanulások eredménye nem áll túl távol a végső eredményektől. A statikus metrikákhoz képest mért javulást az F-measure oszlopában zárójelbe írt értékek jelzik.

5.2. táblázat. Legjobb F-measure értékek osztályozónként.

classifier	TP	TN	FP	FN	accuracy	precision	recall	F-measure
Random Forest	**730**	**7046**	**32**	**230**	**96.7%**	**95.8%**	**76.0%**	**84.8% (+11%)**
Decision Tree	723	7006	72	237	96.2%	90.9%	75.3%	82.4% (+10.4%)
k-NN	684	7041	37	276	96.1%	94.9%	71.3%	81.4%(+5.4%)
Standard DNN	687	7019	59	273	95.9%	92.1%	71.6%	80.5% (+9.5%)
Customized DNN	678	7025	53	282	95.8%	92.8%	70.6%	80.2% (+9.2%)
SVM	692	6966	112	268	95.3%	86.1%	72.1%	78.5% (+6.5%)
LogReg	496	6906	172	464	92.1%	74.3%	51.7%	60.9% (+4%)
LinReg	570	6592	486	390	89.1%	54.0%	59.4%	56.6% (+1.6%)

A legjobb eredmények vastag betűtípussal kiemelve láthatóak. A döntési fa alapú modellek messze a legjobb eredményt érték el, ahogy az ilyen relatíve kis méretű adathalmazon várható is volt. Ahogy látható, a Random Forest osztályozó esetében a folyamat metrikák bevezetése 11%-os javulást jelentett, ami ezen a területen óriási előrehaladást jelent, akár már a gyakorlatban is alkalmazható modell létrehozását teszi lehetővé. Emellett a tanulóhalmaz kis méretének ellenére a neuronháló alapú modellek is meglepően jól teljesítettek. A Naive Bayes osztályozó szintén nem érte el a ZeroR által produkált alsó határértéket, így azt meg sem említem a táblázatban. Ami figyelemre méltó, hogy a k-NN, az SVM és a Random Forest osztáloyzók kimagasló eredményt értek el precision tekintetében is, annak ellenére, hogy a táblázatban szereplő eredmények F-measure szerinti sorba rendezéssel születtek. Ez azért fontos, mivel attól még, hogy végezetül a precision értéket szeretnénk maximalizálni, a modellek összesített teljesítménye is fontos szempont, ugyanis hiába nem produkál a modellünk szinte egy téves riasztást sem, ha közben rengeteg sérülékenység felett szemet huny, ami hamis biztonságérzetet adhat a használójának.

Az 5.3. táblázatban csökkenő sorrendben láthatóak az osztályozónkénti legjobb precision értékek. Az oszloponkénti legjobb eredmények itt is félkövér betűtípussal kiemelve figyelhetőek meg. A korábban említett döntési fa alapú modellek újfent a legjobbak közt végeztek, míg a legjobb eredményt a k-NN osztályozó érte el. Ahogy azt korábban is fejtegettem, a legjobb módja a felhasználói élmény növelésének a precision érték maximalizálása, viszont az F-measure értéke, azaz a modell összesített teljesítménye szintén egy fontos szempont. Például a Döntési Fa osztályozó hiába ért el magasabb precision értéket a Random Forestnél, F-measure értékben jelentősen elmarad attól, ami jól látszik a fals negatív jelzések számán, ami közel másfélszerese a Random Forestének. Ezen kísérlet során a k-NN, SVM és Random Forest osztályozók egyaránt 95% precision és 70% F-measure érték felett teljesítettek, így ez a három

5.3. táblázat. Best precision values per classifier.

classifier	TP	TN	FP	FN	accuracy	precision	recall	F-measure
k-NN	565	**7071**	**7**	395	95.0%	**98.8%**	58.9%	73.8%
Decision Tree	429	7068	10	531	93.3%	97.7%	44.7%	61.3%
Random Forest	**599**	7063	15	**361**	**95.3%**	97.6%	**62.4%**	**76.1%**
SVM	548	7060	18	412	94.7%	96.8%	57.1%	71.8%
Customized DNN	551	7048	30	409	94.5%	94.8%	57.4%	71.5%
Standard DNN	572	7045	33	388	94.8%	94.6%	59.6%	73.1%
LogReg	221	7058	20	739	90.6%	91.7%	23.0%	36.8%
LinReg	274	7051	27	686	91.1%	91.0%	28.5%	43.5%

algoritmus az amire a későbbiek során érdemes lehet nagyobb hangsúlyt fektetni.

A sérülékeny JavaScript függvények előrejelzésében legjobban teljesítő algoritmus F-measure tekintetében a Random Forest volt 0.85-ös F-measure értékkel (0.96 precision és 0.76 recall) ami egy nagyon jelentős javulást jelent a pusztán statikus kódmetrikák történő modell építéshez képest, amivel a korábbi kutatások során 0.76-os F-measure értéket sikerült csak elérni (0.91 precision és 0.66 recall értékkel).

A legjobb precision értéket (0.99) a k-NN érte el (statikus metrikákkal ez az érték 0.95 volt), míg a legjobb recall értéket (0.90) az SVM érte el (statikus metrikákkal ez 0.80 volt). Összességében a feladatra leginkább alkalmas modellek a Random Forest, k-NN és az SVM voltak, míg a regresszió alapúak és a Naive Bayes egyaránt sokkal rosszabbul teljesítettek.

6. 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. Végezetül a folyamat metrikák használatának előnyeit mutattam be különböző tanuló algoritmusokat futtatva, kiemelve az algoritmusok által produkált eredmények közti lényegi különbségeket.

Jövőbeli terveim között szerepel a módszer és az elért eredmények publikálása. 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 a korábbi statikus metrikákat tartalmazó adatbázis bővítését, majd a bővített adatbázissal betanítottam különböző mesterséges intelligencia modelleket, aminek segítségével sikerült megalkotni egy olyan új sérülékenység előrejelző technikát, amely a korábbi statikus metrikák használatához képest óriási előrelépést jelent a kiberbiztonság területén.

Köszönetnyílvánítás

Szeretnénk köszönetet mondani munkatársamnak Aladics Tamásnak, aki hozzájárulásával segítette a DWF keretrendszernek a megvalósulását, valamint Gyimesi Péternek aki segítségével megkönnyítette a folyamat metrikák előállítását. Továbbá, szeretnénk köszönetet mondani témavezetőimnek Dr. Hegedűs Péternek és Dr. Ferenc Rudolfnak, akik szakmai támogatásukkal segítették ennek a projektnek a sikerét.

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] Rudolf Ferenc, Tamás Viszkok, Tamás Aladics, Judit Jász, and Péter Hegedűs. Deep-water framework: The swiss army knife of humans working with machine learning models. *SoftwareX*, 12:100551, 2020.

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

[5] 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.

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

[7] 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.

[8] Patrick Morrison, Kim Herzig, Brendan Murphy, and Laurie Williams. Challenges with applying vulnerability prediction models. 04 2015.

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

[10] Yonghee Shin and Laurie Williams. Can traditional fault prediction models be used for vulnerability prediction? *Empirical Software Engineering*, 18, 02 2011.