UNIVERZA V MARIBORU

FAKULTETA ZA ELEKTROTEHNIKO,

RAČUNALNIŠTVO IN INFORMATIKO

Mihael Polanec

Zaznavanje in napovedovanje prisotnosti napak v izvorni kodi s pomočjo metrik programske opreme in strojnega učenja

Magistrsko delo

Maribor, oktober 2018

Zaznavanje in napovedovanje prisotnosti napak v izvorni kodi s pomočjo metrik programske opreme in strojnega učenja

Magistrsko delo

Študent:	Mihael Polanec
Študijski program:	študijski program 2. stopnje Računalništvo in informacijske tehnologije
Mentor:	red. prof. dr. Peter Kokol, univ. dipl. inž. el.
Lektorica:	Helena Molek

ZAHVALA

Zahvaljujem se mentorju red. prof. dr. Petru Kokolu
za strokovno pomoč pri izdelavi magistrskega dela.

Zahvaljujem se vsem, ki so me med študijem na
kakršen koli način podpirali in me spodbujali.

Posebna zahvala staršem, ki so mi skozi vsa leta
stali ob strani in omogočili nemoten študij, in Petri
za vse pozitivne besede, ki mi jih je namenila med ustvarjanjem tega dela.

Zaznavanje in napovedovanje prisotnosti napak v izvorni kodi s pomočjo metrik programske opreme in strojnega učenja

Ključne besede: metrike programske opreme, strojno učenje, napake programske opreme

UDK: 004.4'2/'6:004.5(043.2)

Povzetek

V magistrski nalogi smo spoznali različne tipe metrik za merjenje karakteristik izvorne kode in algoritme strojnega učenja. Obe področji smo združili v aplikaciji, s katero smo testirali natančnost napovedovanja prisotnosti napak v izvorni kodi z različnimi algoritmi strojnega učenja. Aplikacija je razvita v Javi s pomočjo knjižnice WEKA 3.8. S pridobljenimi rezultati smo pokazali, da bi nekatere pristope lahko uporabili za napovedovanje napak v izvorni kodi.

Fault presence detection and prediction in the source code using software metrics and machine learning

Key words: software metrics, machine learning, software faults

UDK: 004.4'2/'6:004.5(043.2)

Abstract

In this master thesis we studied various types of metrics for measuring source code characteristic and machine learning algorithms. We combined the two fields in an application to test the accuracy of fault presence detection with various machine learning algorithms. The application was developed in Java using the WEKA 3.8 library. Using the obtained results, we have shown that some approaches could be used to predict errors in the source code.

KAZALO

KAZALO SLIK

KAZALO GRAFOV

KAZALO TABEL

UPORABLJENE KRATICE

COCOMO – (angl. Constructive Cost Model) konstruktivni model ocenjevanja stroškov

LOC – (angl. Lines of Code) število vrstic kode

OO – objektno orientirano

FP (FPM) – (angl. Function points) funkcijske točke

CC – (angl. Cyclomatic Complexity) ciklomatična kompleksnost

ECC – (angl. Excended Cyclomatic Complexity) razširjena ciklomatična kompleksnost

L&K – Lorenz in Kiddove

MOOSE – (angl. metrics for object-oriented engineering) metrike za objektno orientiran inženiring

EMOOSE – (angl. metrics for object-oriented engineering) razširjene metrike za objektno orientiran inženiring

MOOD – (angl. metrics for object-oriented design) metrike za načrtovanje objektno orientiranih sistemov

GQM – (angl. goal question metric) metrika cilj/vprašanje

QMOOD – (angl. Quality Model for Object-Oriented Design) kvalitetni model za načrt objektno orientiranega sistema

SVN – (angl. Subversion) sistem za upravljanje z izvorno kodo

CFG – (angl. Control flow graph) graf nadzora pretoka

C_p – (angl. structure complexity) strukturna kompleksnost

C&K – Chidamber and Kemerer

DIT – (angl. Depth of Inheritance Tree) globina v drevesu dedovanja

NOC – (angl. Number of Children) število otrok razreda

CBO – (angl. Coupling between Objects) sklopljenost med objekti

RFC – (angl. Response for class) odziv razreda

WMC – (angl. Weighted methods per Class) število obteženih metod razreda

LCOM – (angl. Lack of Cohesion in Methods) pomanjkanje kohezije v metodah

CCM – (angl. Class Coupling Metric) sklopljenost razredov

OXM – (angl. Operating Complexity Metrics) kompleksnost delovanja

RM – (angl. Reuse Metric) metrika pouporabnosti

Ca – (angl. Afferent couplings) število razredov, ki dostopajo do opazovanega razreda

Ce – (angl. Efferent couplings) število razredov, do katerih ta razred dostopa

NPM – (angl. Number of Public Methods) število javnih metod

DAM – (angl. Data Access Metric) metrika dostopnosti podatkov

MOA – (angl. Measure of Aggregation) mera združevanj

MFA – (angl. Measure of Functional Abstraction) mera funkcionalne abstrakcije

CAM – (angl. Cohesion Among Methods of Class) kohezija med metodami

IC – (angl. Inheritance Coupling) sklopljenost dedovanja

CBM – (angl. Coupling Between Methods) sklopljenost med metodami

AMC – (angl. Average Method Complexity) povprečna kompleksnost metod

k-NN – (angl. k-nearest neighbors) algoritem k - najbližjih sosedov

NM – (angl. neural network) nevronska mreža

SVM – (angl. support vector machine) metode podpornih vektorjev

SMO – (angl. Sequential Minimal Optimization) zaporedna minimalna optimizacija

MLP – (angl. Multilayer perceptron) večplastni perceptron

NB – (angl. Naive Bayes) naivni Bayes

TP – (angl. true positives) pravilno klasificirani pozitivni primeri

TN – (angl. true negatives) pravilno klasificirani negativni primeri

FP – (angl. true negatives) napačno klasificirani pozitivni primeri

FN – (angl. true negatives) napačno klasificirani negativni primeri

CSV – (angl. Comma-separated values) podatki, ločeni z vejico

ARFF – (angl. Attribute-Relation File Format) format, ki ga uporablja knjižnica WEKA

1 UVOD

Živimo v času, v katerem smo priča skokovitemu povečanju uporabe in razvoja programske opreme na praktično vseh področjih našega življenja. Če kot primer navedemo samo aplikacije za operacijski sistem Android, pri tem upoštevamo samo aplikacije dostopne preko uradne trgovine Google Play (prej Android Market), ugotovimo, da se je število od decembra 2009, ko je bilo naloženih šestnajst tisoč aplikacij, do decembra 2017 povečalo na tri milijone in pol [1].

Pri razvoju programske opreme stremimo k temu, da bi le ta delovala brezhibno, zanesljivo in seveda brez napak. Vsaka napaka lahko prinese veliko škode in posledično izgubo denarja [2]. Vendar kot vse, kar ustvari človek, tudi programska oprema vsebuje napake. Da bi napako odkrili in popravili še pred izidom programske opreme, se razvijalci le te poslužujejo različnih pristopov, kot so [3]:

- testiranje enot (angl. unit testing),
- pregled kode (angl. code review),
- statična analiza kode (angl. static analysis),
- testiranje po metodi bele ali črne škatle
- in ostalih.

Cilj tega magistrskega dela je bil spoznati in preučiti metrike programske opreme, algoritme strojnega učenja, nato pa ti dve področji združiti v aplikaciji in določiti, kateri klasifikator je na osnovi vrednosti metrik najprimernejši za zaznavo napak v programski kodi.

Omejili smo se na programski jezik Java, natančneje na odprtokodne projekte. Podatke o le teh pa smo pridobili iz dveh repozitorijev. Pri metodah strojnega učenja smo se omejili na

metode nadzorovanega učenja, kar pomeni, da imajo vsi primerki razredov znan podatek o prisotnosti napake.

V okviru magistrske naloge smo tako razvili aplikacijo, s katero smo preizkušali različne algoritme strojnega učenja. Z njihovo pomočjo smo določili, kateri se najbolje obnesejo za reševanje našega problema.

V magistrski nalogi si tako zastavljamo sledeča raziskovalna vprašanja:

- **RV1**: Katera metoda strojnega učenja (oz. kateri klasifikator) je najprimernejši za zaznavanje prisotnosti napak v razredu?
- **RV2**: Ali je pristop napovedovanja napak v razredih na osnovi klasičnih metrik programske opreme primeren za uporabo v praksi?
- **RV3**: Ali je pristop napovedovanja napak na osnovi metrik sprememb primeren za uporabo v praksi?
- **RV4**: Ali je pristop napovedovanja napak na osnovi kombinacije klasičnih metrik programske opreme in metrik sprememb primeren za uporabo v praksi?
- **RV5**: Ali se bo natančnost klasifikacije povečala, če učimo klasifikator na zaporednih verzijah programske opreme?

V uvodnem poglavju smo predstavili temo magistrske naloge, cilje, omejitve in raziskovalna vprašanja. Temu sledita poglavji, kjer smo opisali dve glavni temi te naloge, metrike programske opreme in strojno učenje. V četrtem poglavju smo predstavili sorodna dela. Sledi predstavitev praktične analize naloge, kjer smo na kratko opisali postopek pridobivanja podatkov, uporabljene tehnologije, podali definicije poskusov in opisali implementacijo. V šestem poglavju smo predstavili rezultate, pridobljene s praktično analizo. V zaključku smo podali odgovore na zastavljena raziskovalna vprašanja in možnosti za nadgradnjo te naloge.

2 METRIKE PROGRAMSKE OPREME

Metrike igrajo ključno vlogo v vsaki inženirski disciplini, vendar se programski inženiring pogosto ne uvršča med klasične inženirske discipline iz več razlogov [4]. V splošnem velja, če je neka programska oprema videna kot nekaj, kar nam zagotavlja neko zahtevano funkcionalnost, bo samo nekaj ljudi, če sploh koga, zanimalo, kako je programska oprema zgrajena. Še več, razumevanje in uporaba metrik programske opreme se pogosto zdi kot prezahtevna dejavnost, ki se jo priporoča zgolj ustrezno usposobljenim strokovnjakom [4] [5].

V tem poglavju smo predstavili nekaj splošnih informacij o metrikah programske opreme. Nekaj izbranih smo opisali podrobneje.

2.1 Splošno o metrikah programske opreme

Metrike lahko razvrstimo v več skupin. Vir [6] obravnava delitev metrik v dve skupini: metrike produkta (angl. product metrics) in metrike procesa (angl. process metrics). Vir [7] metrike deli v tri skupine, metrikam procesa in produkta doda metrike projekta (angl. project metrics). V [8] je že omenjenim trem skupinam dodana še skupina hibridnih metrik (angl. hybrid metrics). Podrobnejša delitev metrik, povzeta po [8], je prikazana na sliki 2.1.

Metrike produkta opisujejo karakteristike, kot so: velikost, kompleksnost, lastnosti načrtovanja (diagrami načrtovanja), zmogljivost, raven kakovosti, dokumenti specifikacij, seznam izvorne kode in izvršljivo programsko opremo [6] [7] [8]. Uporabljajo se lahko za izboljšanje procesa razvoja programske opreme in vzdrževanja. Primeri uporabe vključujejo tudi učinkovitost odstranjevanja napak med razvojem, vzorec preizkušanja napake in odzivni čas potreben za odpravo napake [7].

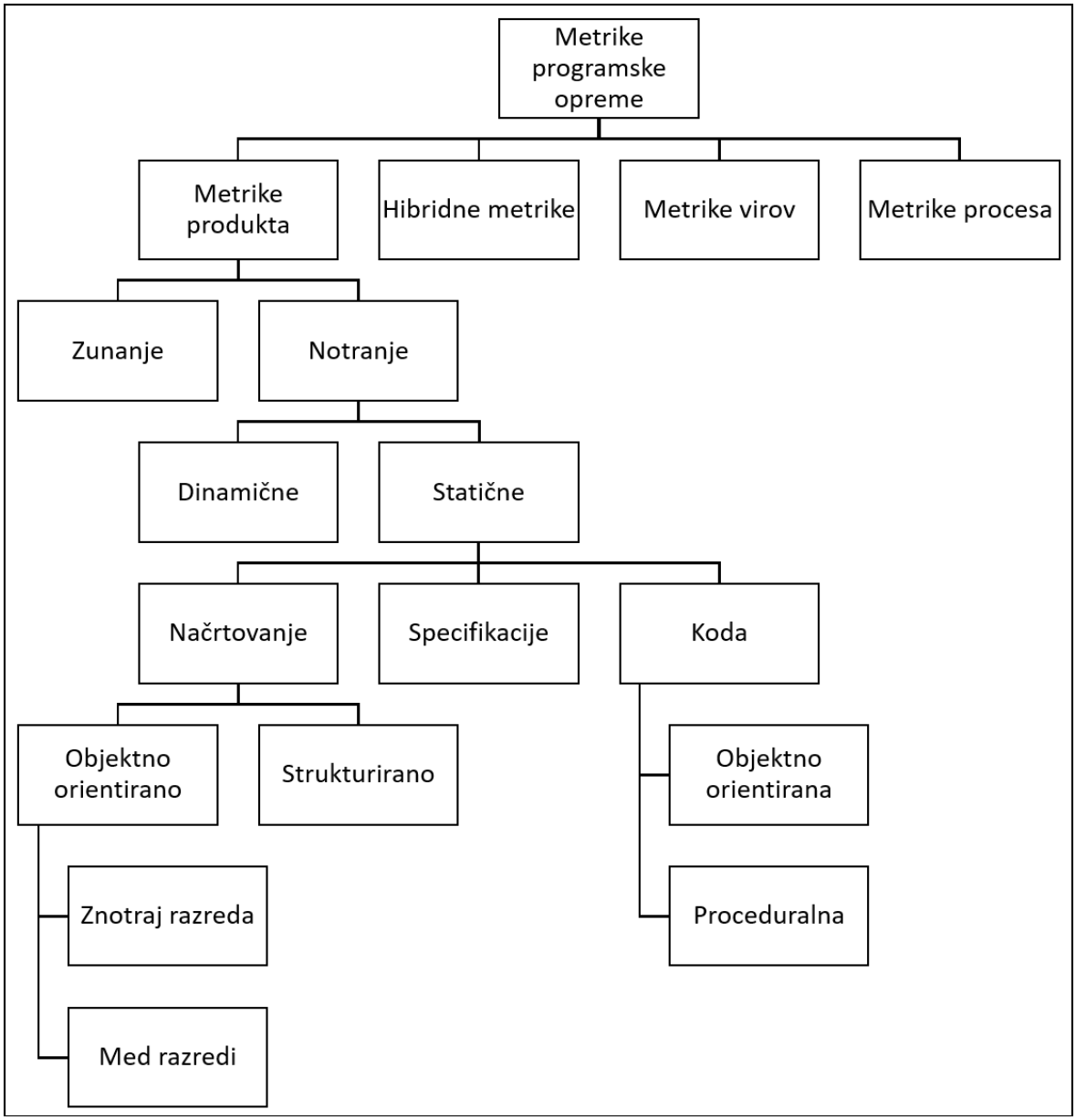
Metrike procesa se uporabljajo za merjenje atributov, ki so povezani z življenjskim ciklom razvoja programske opreme. Najpomembnejši atributi procesa so cena, čas in vložen trud. Te atribute lahko ocenimo s pomočjo Boehmovega konstruktivnega modela ocenjevanja stroškov COCOMO(angl. constructive cost model) [8].

Projektne metrike (v [8] se uporablja izraz metrike virov (angl. resource metrics)) opisujejo lastnosti projekta (karakteristike razpoložljivih virov) in njegovo izvedbo. To so: število razvijalcev in njihova znanja, stroški, urniki dela, produktivnost, zanesljivost in zmogljivost strojne opreme [7] [8].

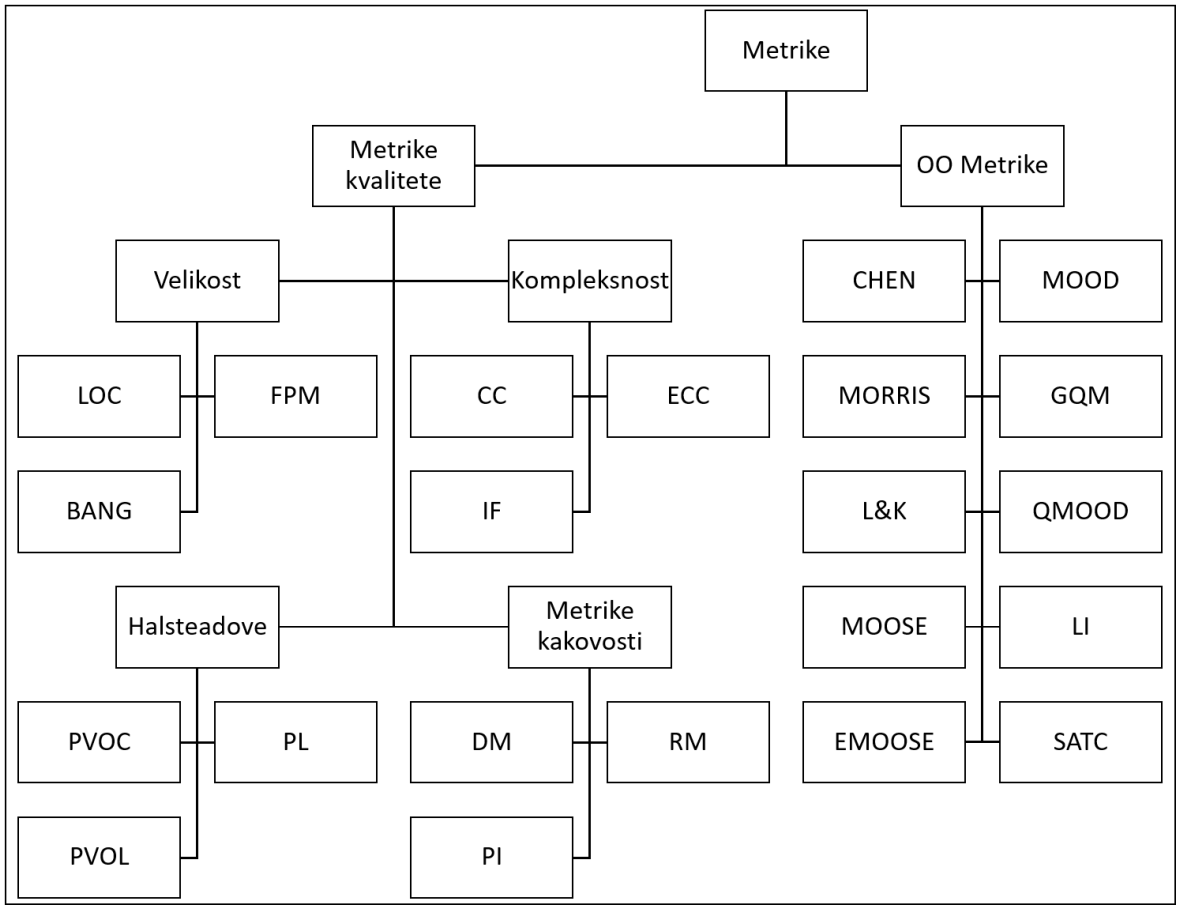
Hibridne metrike so kombinacije produktnih in procesnih metrik. Primera hibridnih metrik sta cena glede, na funkcijsko točko in čas, potreben za dokončanje projekta na LOC [8].

Vseh metrik ne moremo preprosto postaviti v zgolj eno kategorijo. Primer metrike, ki sodi v dve kategoriji, je kakovost med razvojem projekta. Uvrstimo jo lahko med procesne in projektne metrike [7].

Metrike kvalitete programske opreme so podmnožica metrik programske opreme, ki se na programsko opremo osredotočajo z vidika kvalitete produkta, procesa in projekta. V splošnem so metrike kvalitete tesneje povezane z metrikami procesa in produkta. Vsekakor pa imajo parametri, kot so število razvijalcev in njihov nabor znanja, urnik, velikost projekta in struktura organizacije, vpliv na kvaliteto končnega produkta [7]. Metrike, ki jih lahko uporabimo za ugotavljanje kakovosti programske opreme, so prikazane na sliki 2.2. Opisi metrik in kratic, uporabljenih na sliki 2.2, se nahajajo v nadaljevanju in seznamu kratic.



Slika 2.1: Hierarhija metrik programske opreme [8]



Slika 2.2: Hierahija delitve metrik [9] [10]

Kot lahko opazimo, se lahko za merjenje kvalitete programske opreme uporabi več vrst metrik. Ena izmed njih so metrike povezane z velikostjo. Te metrike nam pomagajo količinsko opredeliti velikost programske opreme. Za merjenje velikosti programske opreme se uporabljajo trije tipi metrik [9] [10] [11]:

- LOC – najstarejša metrika, uporabljena za izmero velikosti,
- funkcijske točke – z njo izmerimo trud, potreben za razvoj programske opreme in
- Bang, metriko je definiral Dc. Marco, kot funkcijsko metriko, izračuna se lahko iz določenih algoritmov in podatkovnih tipov iz nabora specifikacij, kot rezultat vrne pa število vseh funkcionalnosti.

Kot prvi je leta 1976 Mc. Cabe predstavil metrike kompleksnosti programske opreme. Kot pri metrikah, povezanih z velikostjo, tudi tukaj obravnavamo tri tipe metrik [9] [10]:

- ciklomatična kompleksnost (angl. cyclomatic complexity) – CC, katerecglavni cilj je določiti indeks testabilnosti in vzdrževanja za nek modul,
- razširjena ciklomatična kompleksnost (angl. extended cyclomatic complexity)– ECC in
- tok informacij (angl. information flow).

Naslednja večja skupina metrik za ugotavljanje kvalitete programske opreme so Halsteadove metrike. S temi metrikami ugotavljamo [9] [10]:

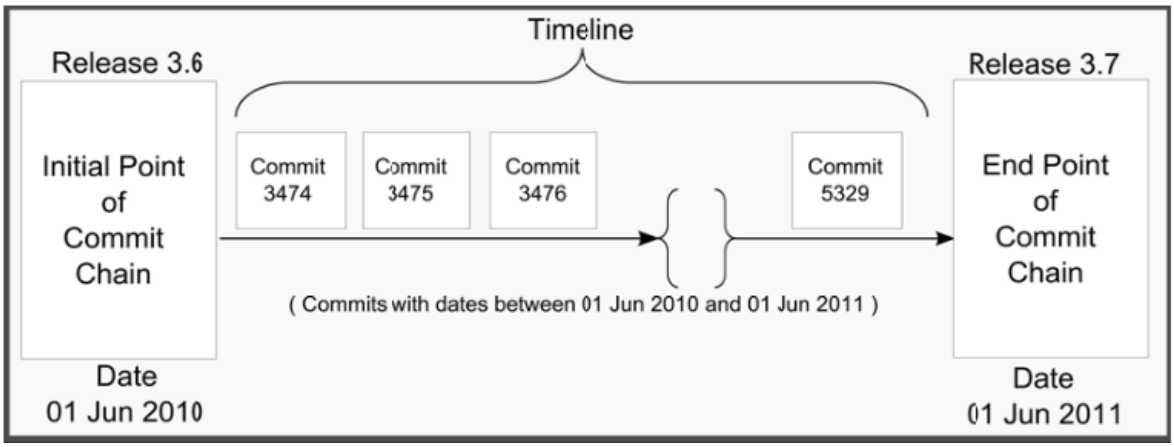
- »besedišče« (angl. program vocabulary),
- dolžino in
- prostornino programa.

Kvaliteto programske opreme lahko merimo tudi z objektno orientiranimi metrikami. Te uporabnikom pomagajo pri razumevanju objektnega modela (povezanosti razredov), predvidevanju napak, testiranju in vzdrževanju programske opreme. Objektno orientirane metrike lahko razdelimo v sledeče skupine [4] [10] [12]:

- Chenove metrike,
- Morrisove metrike,
- Lorenz in Kiddove metrike – L&K,
- metrike za objektno orientiran inženiring - MOOSE (angl. metrics for object-oriented engineering),
- razširjene metrike za objektno orientiran inženiring – EMOOSE (angl. extended metrics for object-oriented software engineering),
- metrike za načrtovanje objektno orientiranih sistemov – MOOD (angl. metrics for object-oriented design),
- metrika cilj/vprašanje – GQM (angl. goal question metric),
- kvalitetni model za načrt objektno orientiranega sistema – QMOOD (angl. Quality Model for Object-Oriented Design),

- Lijeve metrike in
- metrike SATC za objektno orientirane metrike.

Metrike sprememb (angl. change metrics) se za razliko od metrik izvorne kode in CK metrik ne ukvarjajo z vsebino datoteke z izvorno kodo, ampak s podrobnostmi sprememb, ki so nastale v kodi skozi čas. Metrike sprememb se za posamezen projekt pridobijo iz sistema za upravljanje z izvorno kodo, kot sta na primer Git in SVN (subversion). Ena izmed glavnih prednosti uporabe metrik sprememb je v tem, da so neodvisne od programskega jezika. Metrike sprememb se po navadi izračunajo med dvema zaporednima večjima izdajama programske opreme in se izračunavajo za vsako datoteko posebej [13]. Primer pridobivanja podatkov za izračun metrik sprememb je prikazan na sliki 2.3.



Slika 2.3: Pridobivanje podatkov za izračun metrik sprememb [13]

2.2 Definicije metrik programske opreme

V tem podpoglavju podajamo podrobnejše definicije izbranih metrik programske opreme. Kot prvo predstavimo metriko **LOC**, ki je zagotovo ena izmed naj bolj prepoznavnih in najstarejših metrik programske opreme. Že iz imena lahko razberemo, da se uporablja za štetje vrstic v izvorni kodi programa. Metrika se navadno uporablja za napovedovanje, koliko truda bo potrebno vložiti za razvoj programa, oceno produktivnosti programerja in pomaga pri oceni stroškov vzdrževanja, ko je program že napisan [5] [7] [11] [14].
Metrika LOC ima kar nekaj težav [5]:

- nikoli ni bilo nekega standarda, ki bi natančno opredelil metriko za vse jezike;
- Izvorna koda je lahko rezultat generatorja programov, razpredelnic, pouporabljenih modulov neznane velikosti in dedovanja, kjer število vrstic kode nima pravšnjega pomena;
- Metrika LOC se paradoksalno »pomika nazaj«, na višjem nivoju je programski jezik, kar pomeni, da višjenivojski jeziki, ki imajo večjo izrazno moč, delujejo manj produktivno, kot primitivnejši, nižjenivojski jeziki.

Kljub temu, da se metrika LOC uporablja praktično od začetka, se nikoli ni natančno standardiziralo koncepta vrstice programske (izvorne) kode [5].

Kaj šteti? Fizične ali logične vrstice? To je odvisno od uporabljene variacije, saj to lahko privede do velikega odstopanja v velikosti programa. Fizičen zaključek vrstice pomeni, da se vrstica prelomi v novo vrstico oz. da smo na tipkovnici pritisnili tipko Enter. Logično vrstico pa nam predstavlja vrstica, ki jo zaključuje formalno ločilo, npr. podpičje, dvopičje, pika, … V nekaterih jezikih, kot na primer Basic, ki dovoljuje več programskih stavkov v eni fizični vrstici, nas lahko štetje logičnih vrstic privede do 500% več vrstic, kot pa če štejemo samo fizične vrstice. Na drugi strani pa so jeziki, kot je COBOL, ki uporabljajo pogojne stavke, zajete v več fizičnih vrstic, kar lahko povzroči, da s štetjem fizičnih vrstic dobimo 200% večjo velikost programa, kot s štetjem logičnih vrstic [5].

Za lažje razumevanje koncepta logične in fizične vrstice predstavljamo dva primera, napisana v jeziku C. V prvem na sliki 2.4 imamo eno fizično vrstico, dve logični vrstici (stavka for in printf) in en komentar. Na sliki 2.5 je prikazan drugi primer, kjer imamo štiri fizične vrstice, dve logični vrstici in en komentar [15].

```
for (i = 0; i < 100; i++) printf("zdravo"); /* Koliko vrstic kode vidiš? */
```

Slika 2.4: Koncept fizične in logične vrstice – primer 1 [15]

```
/* Koliko vrstic kode vidiš? */
for (i = 0; i < 100; i++)
{
    printf("zdravo");
}
```

Slika 2.5: Koncept fizične in logične vrstice – primer 2 [15]

Naslednje negotovo področje je, katere izmed več možnih tipov vrstic štejemo. Večina proceduralnih jezikov ima pet različnih tipov programskih stavkov [5]:

- izvršljive vrstice (angl. executable lines), uporabljene za različne akcije, npr. seštevanje,
- definicije tipov podatkov (angl. data definitions), uporabljene za identifikacijo tipov,
- komentarji, uporabljeni, da dodajo informacijo, k izvorni kodi,
- prazne vrstice (angl. blank lines), uporabljene, da ločijo sekcije vizualno in
- mrtva koda (angl. dead code), koda, puščena na mestih, kjer se ne more izvršiti, npr. za return stavkom.

Pogosta težava je prav tako štetje kode, ki se je ponovno uporabila. Vsak profesionalni programer bo rutinsko kopiral in pouporabil okoli 20 do 30% kode v aplikaciji do konca razvoja programa v proceduralnem jeziku, kot so npr. C, COBOL ali FORTRAN. V objektno orientiranih jezikih, kot so Java, C++ in Objective C, se količina po pouporabe poveča do 50% zaradi možnosti dedovanja, ki je bistvena v objektno orientiranih jezikih [5].

Kot zadnjo večjo težavo metrike LOC izpostavljamo problem, ko so aplikacije napisane v več programskih jezikih. Takrat je rezultat te metrike težko preprosto interpretirati [5].

Ciklomatična kompleksnost (angl. cyclomatic complexity) – CC je metrika programske opreme, uporabljena za nakazovanje kompleksnosti programa. Metriko je leta 1976 razvil Thomas J. McCabe. Kompleksnost se izračuna iz grafa nadzora pretoka (angl. control flow

graph) CFG. Vozlišča v grafu predstavljajo nedeljive skupine ukazov v programu. Usmerjene povezave med vozlišči kažejo, da se naslednji ukaz izvede neposredno za prvim ukazom [7] [9] [12] [16].

CC nekega dela izvorne kode je število linearno neodvisnih poti znotraj tega dela. Na primer, če imamo izvorno kodo, katera na vsebuje odločitvenih stavkov, potem bo kompleksnost takšnega programa 1, saj obstaja samo ena pot skozi program. V primeru, da se v programu nahaja en *if* stavek, imamo dve poti skozi program, eno če se *if* stavek ovrednoti kot *true*, drugo pa, če se ovrednoti kot *false*, dobimo vrednost kompleksnosti 2. Splošna formula za izračun CC je podana v 2.1 [7] [16].

Če vzamemo za primer levi graf (A) iz slike 2.6, kjer imamo enostaven program, ki recimo vsebuje dva *if* stavka. Po štetju robov, vozlišč in nepovezanih delov ugotovimo, e = 8, n = 7 in p = 1, kar nam, če vstavimo vrednosti v 2.1, da rezultat 3. Obstaja pa tudi alternativni izračun, če končno točko na grafu povežemo z začetno. Na sliki 2.6 je to prikazano na desnem grafu (B). V tem primeru lahko zadnji člen v 2.1 (*2p*) nadomestimo s *p* [7] [16].

$$M = V(G) = e - n + 2p \tag{2.1}$$

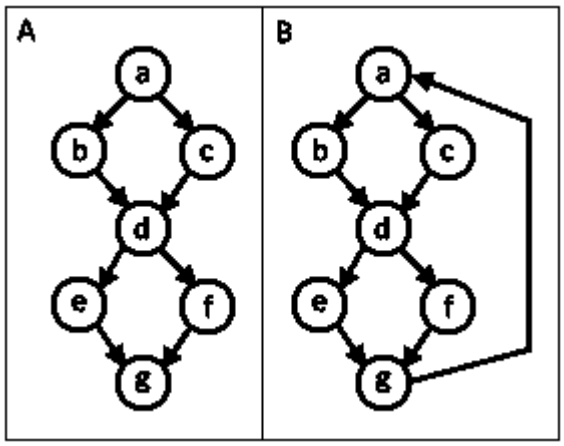
Tukaj je:

V(G) – ciklomatično število grafa G,

e – število robov,

n – število vozlišč in

p – število nepovezanih komponent.



Slika 2.6: Primer CFG [7]

Če povzamemo, lahko M izračunamo kot število binarnih odločitev v programu plus 1. Če katera izmed odločitev ni binarna, je obravnavana kot več binarnih odločitev. V splošnem velja, da če imama odločitev *n* možnosti, potem velja, da jih štejemo kot *n-1* binarnih odločitev. Pogoji v zankah se štejejo kot ena binarna odločitev [7] [16].

Metrika CC je aditivna, kar pomeni, da je kompleksnost grafov, ki jih razumemo kot skupino, enaka vsoti kompleksnosti posameznih grafov. McGabe priporoča, da vrednost CC v posameznem modulu (funkciji) ne preseže 10, saj s tem otežimo vzdrževanje in testiranje [7].

Halstead je razlikoval znanost programske opreme (angl. software science) od raunalniške znanosti (angl. computer science). Njegova predpostavka je, da se vsaka programerska naloga sestoji iz izbiranja in urejanja končnega števila programskih žetonov, ki so osnovne sintaktične enote programskega jezik in jih lahko prevajalnik razloči. Primeri žetonov so ključne besede v programskem jeziku, npr.: *public, static, void, int, double,…* Program je zbirka žetonov, ki jih lahko razvrstimo v dve skupini, med operatorja in operande [7] [12] [16].

Halstead je leta 1977 definiral primitivne mere programskega inženirstva [6] [7] [9] [12]:

- n_1 – število unikatnih operatorjev v programu,
- n_2 – število unikatnih operandov v programu,
- N_1 – število pojavitev operatorjev in
- N_2 – število pojavitev operandov.

Na osnovi teh mer je razvil sistem enačb, s katerimi je možno izraziti besedišče (2.2), dolžino (2.3, 2.4), prostornino programa (2.5, 2.6) in stopnjo le tega (mera kompleksnost) (2.7, 2.8), težavnost (2.9, 2.10), vložen trud (2.11) in predvideno število napak v programu (2.12) [7]. Enačbe podajamo v nadaljevanju [7] [9] [12].

$$n = n_1 + n_2 \tag{2.2}$$

$$N = N_1 + N_2 \tag{2.3}$$

$$N = n_1 \log_2(n_1) + n_2 \log_2(n_2) \tag{2.4}$$

$$V = N \log_2(n) \tag{2.5}$$

$$V = N \log_2(n_1 + n_2) \tag{2.6}$$

$$L = \frac{V^*}{V} \tag{2.7}$$

Tukaj je:

V^* – minimalna prostornina programa [7].

$$L = \frac{2}{n_1} \times \frac{n_2}{N_2} \tag{2.8}$$

$$D = \frac{V}{V^*} \tag{2.9}$$

$$D = \frac{n_1}{2} \times \frac{N_2}{n_2} \tag{2.10}$$

$$E = \frac{V}{L} \tag{2.11}$$

$$B = \frac{V}{S^*} \tag{2.12}$$

Tukaj je:

S^* – povprečno število odločitev med napakami, uporabi se konstanta 3000 [7].

LOC, Halsteadove metrike, McCabe – ova ciklomatična kompleksnost in ostale metrike, ki merijo kompleksnost, obravnavajo vsak modul kot ločeno entiteto. **Strukturne metrike** (angl. structure metrics) poskušajo obravnavati povezave med posameznimi moduli. Najpogostejši metriki strukture načrta (angl. design structure metrics) sta metriki »fan-in« in »fan-out«. Ti dve metriki temeljita na ideji sklopljenosti (angl. coupling) [7]:

- fan-in: število modulov, ki kličejo nek modul,
- fan-out: število modulov, ki so poklicani s strani nekega modula.

V splošnem velja, da so moduli, ki imajo veliko vrednost fan-in, relativno majhni in preprosti ter se nahajajo na nižjih arhitekturnih slojih. Ravno obratno, veliki in kompleksni moduli, imajo majhno vrednost fan-in. Iz tega lahko sklepamo, če ima nek modul oz. komponenta sistema veliki vrednosti fan-in in fan-out, je to lahko znak slabega načrtovanja programske opreme [7].

Henry in Kafura v [17] definirata strukturno kompleksnost (angl. structure complexity) - C_P kot 2.13.

$$C_P = (\text{fan-in} \times \text{fan-out})^2 \tag{2.13}$$

Ob poskusu združitve kompleksnosti modula in kompleksnosti strukture Henry definira hibridno obliko metrike C_P, kot 2.14 [7].

$$HC_P = C_{ip} \times C_P \tag{2.14}$$

Tukaj je:

C_{ip} – notranja kompleksnost procedure *p*, ki jo lahko izmerimo s poljubno metriko kompleksnosti, npr. McCabe-ovo metriko kompleksnosti [7].

Na osnovi različnih pristopov k meram kompleksnosti strukture in modula so drugi avtorji kasneje razvili model kompleksnosti sistema (angl. system complexity model) kot 2.15. Relativna kompleksnost sistema se nato določi kot 2.16. Strukturna kompleksnost se nadaljnje lahko določi tudi kot 2.17 in skupna podatkovna kompleksnost kot 2.18 (podatkovna kompleksnost za en modul se določi kot število vhodno/izhodnih spremenljivk modula ulomljeno z vrednostjo fan-out plus 1). Če poenostavimo, kompleksnost sistema je vsota strukturnih kompleksnosti in podatkovne kompleksnosti [7].

$$C_t = S_t + D_t \tag{2.15}$$

Tukaj je:

C_t – kompleksnost sistema,

S_t – strukturna kompleksnost in

D_t – podatkovna kompleksnost.

$$C = \frac{C_t}{n} \tag{2.16}$$

Tukaj je:

n – število modulov v sistemu.

$$S = \frac{\sum f^2(i)}{n} \tag{2.17}$$

Tukaj je:

S – strukturna kompleksnost,

f(i) – vrednost fan-out i-tega modula in

n – število modulov v sistemu.

$$D = \frac{\sum \frac{V(i)}{f(i)+1}}{n} \tag{2.18}$$

Tukaj je:

D – podatkovna kompleksnost (med moduli),

V(i) – vhodno/izhodne spremenljivke i-tega modula,

f(i) – vrednost fan-out i-tega modula in

n – število modulov v sistemu.

Objektno orientiran pristop pri razvoju programske opreme prinaša veliko prednosti, kot so pouporabnost, razčlemba problema v manjše, lažje razumljive enote (objekte) in lažje spremembe v prihodnosti. Da bi razvijalci sledili dobrim praksam in pisali zanesljivo kodo potrebujejo dobre smernice. Eden izmed načinov, kako slediti dobrim praksam, je uporaba **objektno orientiranih metrik** [12].

V literaturi lahko zasledimo veliko metrik za merjenje kakovosti razvoja objektno orientiranih aplikacij. Nekatere skupine teh metrik so prikazane na sliki 2.2.

Chidamber in Kemerer sta definirala zbirko metrik, t.i. »CK metric suite« s šestimi metrikami [18]. V [9] jih lahko zasledimo pod kratico MOOSE (Metrics for Object-Oriented Software Engineering). S pomočjo metrik v zbirki lahko dobimo vpogled, ali razvijalci sledijo principom objektno orientiranega razvoja [18].

Metrika **utežene metode razreda** – WMC (angl. Weighted Methods per Class) meri kompleksnost razreda. Večja kot je vrednost WMC, kompleksnejši je razred. Definicija pravi: imamo razred C1, z metodami M_1, M_2,… M_n definiranimi v razredu in njihovimi kompleksnostmi c_1, c_2, … c_n, izračunamo WMC kot 2.19. Kompleksnosti lahko določimo s poljubno metriko, npr. z metriko CC [9] [12] [18].

$$WMC = \sum_{i=1}^{n} c_i \tag{2.19}$$

Tukaj je:

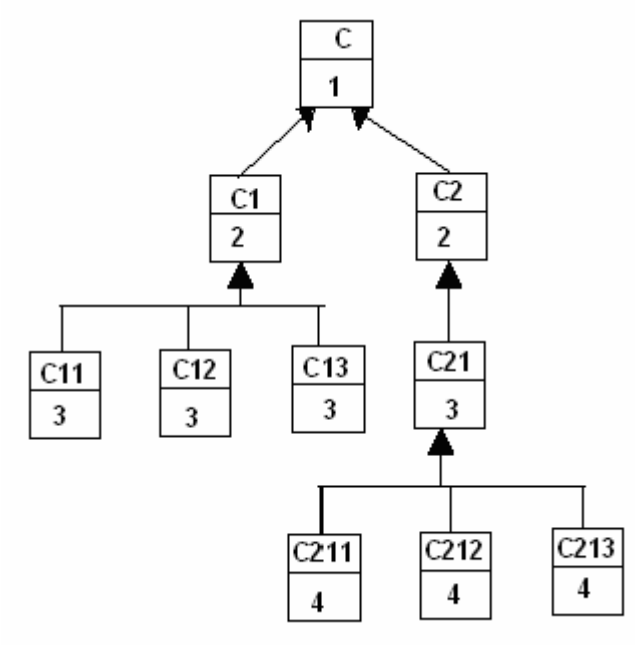
n – število metod in

c_i – kompleksnost metode.

WMC lahko uporabimo, da s pomočjo števila metod in kompleksnosti le teh napovemo čas, potreben za razvoj, in vložen trud za vzdrževanje razreda. Več kot je metod v razredu, večji je potencialni vpliv na podrazrede, saj le ti podedujejo vse metode razreda. V primeru, ko ima razred veliko metod, je najverjetneje prilagojen za to aplikacijo, kar zmanjša oz. omeji možnosti za njegovo pouporabo.

Globina v drevesu dedovanja – DIT (angl. Depth of Inheritance Tree) je metrika, ki nam pove razdaljo med korenom drevesa (razred, ki je najvišje v hierarhiji) in trenutnim razredom. Večja vrednost DIT nam nakazuje večjo možnost pouporabe metod, prav tako pa to pomeni, da je težje predvideti obnašanje razreda [9] [12] [18]. Na sliki 2.7 so prikazane vrednosti DIT za razrede v enostavni hierarhiji.

Število otrok razreda – NOC (angl. Number of Children), vrednost te metrike kaže število neposrednih otrok nekega razreda v hierarhiji razredov. Če gledamo primer na sliki 2.7, lahko opazimo, da je vrednost NOC za razred C enaka 2, saj ima dva neposredna otroka (C1 in C2). Vrednost NOC nakazuje nivo pouporabe v aplikaciji, se pravi, večja kot je vrednost NOC, večja je pouporabnost [9] [12] [18].



Slika 2.7: Vrednosti DIT za razrede na različni globini v drevesu dedovanja [18]

Sklopljenost med objekti – CBO (angl. Coupling between Objects) je metrika, katere rezultat je enak številu razredov, ki so povezani s tem razredom. Razred je sklopljen (povezan) z drugim razredom, če uporablja metode ali atribute drugega razreda. Večja vrednost CBO nakazuje, da se bo pouporabnost tega razreda zmanjšala. Tako lahko sklepamo, da si želimo čim nižje vrednosti metrike CBO [9] [12] [18].

Odziv razreda – RFC (Response for class) predstavlja število vseh metod, ki jih je možno izvesti v odziv sporočilu, prejetemu s strani tega razreda. Definicijo RFC podaja 2.20. Velika vrednost RFC pomeni, da je stopnja kompleksnosti v razredu visoka, kar posledično prinese oteženo razumevanje delovanja razreda in razhroščevanje [12] [18].

$$RFC = |RS|$$
$$RFC = \{M\} \cup all\ i\ \{R_i\} \tag{2.20}$$

Tukaj je:

{R_i} – množica metod, klicanih od metode i in

{M} – množica vseh metod v razredu.

Pomanjkanje kohezije v metodah - LCOM (angl. Lack of Cohesion in Methods) meri stopnjo prisotnosti kohezije, kar nakazuje, kako dobro je sistem načrtovan oz. kako kompleksen je nek razred. LCOM je število parov metod, katerih podobnost je nič, minus število parov metod, katerih podobnost ni nič. Več kot je podobnih metod, bolj je razred povezan (večja kohezija). Nizka vrednost LCOM povečuje kompleksnost, kar pomeni povečano možnost pojavitve napake v procesu razvoja [12] [18].

Med metrike za merjenje objektno orientiranega pristopa sodijo še Chenove, Morrisove, Lorenz in Kiddove – L&K, razširjene metrike za objektno orientiran razvoj EMOOSE (angl. Extended Metrics for Object-Oriented Software Engineering), metrike za objektno orientirano načrtovanje – MOOD (angl. Metrics For Object-Oriented Design), kvalitetni model za objektno orientirano načrtovanje – QMOOD (angl. Quality Model for Object-Oriented Design) in Lijeve metrike [9].

Chen je predlagal nabor metrik, v katerih so upoštevane značilnosti objektno orientiranih jezikov. V tem naboru so metrike [9]:

- sklopljenost razredov - CCM (angl. Class Coupling Metric),
- kompleksnost delovanja – OXM (angl. Operating Complexity Metrics),
- metrika pouporabnosti – RM (angl. Reuse Metric) in še nekaj drugih.

Morrisove metrike so metrike za objektno orientirane sisteme, kjer definirajo sistem, definiran z drevesno strukturo. Kompleksnost sistema se določi z globino drevesa, ki se meri s številom vozlišč (poddreves) le tega. Večje število vozlišč predstavlja kompleksnejši sistem [9].

L&K metrike so združene v štiri skupine: velikost, dedovanje, notranje in zunanje metrike. Definirala sta metrike število prepisanih operacij – NOO (angl. Number of Operations Overriden by a subclass), število dodanih operacij – NOA (angl. Number of Operations

Added by a subclass), povprečna velikost operacije OS (average Operation Size) in druge [9].

V zadnjem sklopu predstavimo še nekaj **metrik sprememb**. Kot osnovo za izračun metrik sprememb si vedno izberemo dve točki v »seznamu« sprememb, (npr. za začetno točko vzamemo spremembo ob izdaji verzije 1.0 in končno točko spremembo ob izdaji verzije 2.0) opravljenih v izvorni kodi projekta. Shematsko imamo to prikazano na sliki 2.3. Metrike se izračunavajo za eno datoteko [13].

Prva metrika so **shranjene spremembe** (anlg. Commits), rezultat te metrike nam pove število opravljenih sprememb v datoteki. Velika vrednost metrike nakazuje večjo verjetnost prisotnosti napake v datoteki [13].

Metrika **dodanih vrstic** (angl. Add) nam pove, koliko vrstic je bilo dodanih v datoteko. Več dodanih vrstic pomeni večjo verjetnost prisotnosti napake [13].

Izbrisane vrstice (angl. Delete) -vrednost te metrike nam predstavlja število vrstic izbrisanih iz datoteke. Več kot je bilo izbrisanih vrstic, večja je verjetnost, da so v njej prisotne napake [13].

Avtorji (angl. Authors) je metrika, ki nam pove, koliko različnih avtorjev je prispevalo k spremembam v datoteki. Predvidevamo, da več različnih avtorjev predstavlja večjo verjetnost prisotnosti napake [13].

Shranjene spremembe 60 (angl. Commits 60) nam podobno kot metrika shranjene spremembe pove število opravljenih sprememb v datoteki, vendar se v tem primeru štejejo samo spremembe opravljene v obdobju zadnjih 60 dni. Tako lahko sklepamo, če je datoteka spremenjena pogosto tik pred izidom aplikacije, da obstaja večja verjetnost napak po izidu aplikacije [13].

Zadnja sprememba (angl. Last Commit) nam predstavlja število dni od konca opazovanega obdobja (izid aplikacije) pa do zadnje opravljene spremembe na datoteki. Manjša vrednost metrike predstavlja večjo verjetnost, da se bo napaka pojavila po izidu aplikacije [13].

Napake v razvoju (angl. In Development Bugs) – ta podatek nam predstavlja vse spremembe, označene z »bug-fix«. Velika vrednost pomeni, da obstaja večja verjetnost pojavitve napake v datoteki [13].

Entropija (angl. Entropy) uporablja Shannonovo[1] definicijo entropije za ocenitev porazdelitve sprememb. Če razdelimo opazovano obdobje na N enakih delov, bi imela enakomerna porazdelitev približno enako število sprememb v vsakem izmed N delov. Entropija se izračuna za vsako datoteko po 2.21. Če za opazovano obdobje vzamemo eno leto, potem lahko uporabimo za N vrednot 12 [13].

$$Entropy = -\sum_{i=1}^{N}\left(\frac{C_i}{C_{total}}\right) \times log(\frac{C_i}{C_{total}}) \tag{2.21}$$

Tukaj je:

N – število obdobij,

C_i – število sprememb v *i*-tem obdobju in

C_{total} – število vseh sprememb.

Povprečno obdobje sprememb – MPC (angl. Mean Period of Change) se uporablja za ocenitev, v katerem obdobju so skoncentrirane spremembe. Izračunamo jo po 2.22 [13].

[1] Claude Elwood Shannon – ameriški matematik in inženir.

$$MPC = \sum_{i=1}^{N} i \times (\frac{C_i}{C_{total}}) \quad (2.22)$$

Tukaj je:

N – število obdobij,

C_i – število sprememb v *i*-tem obdobju in

C_{total} – število vseh sprememb.

Največja sprememba (angl. Maximum Change Set) nam pove največje število datotek, ki so bile spremenjene skupaj z opazovano datoteko. Izračuna se kot 2.23 [13].

Povprečna sprememba (angl. Average Change Set) nam pove, koliko datotek se je povprečno spremenilo skupaj s trenutno datoteko. Povprečno spremembo izračunamo kot 2.24 [13].

$$MAXCHANGESET = \max_{c \in C} n(c) \quad (2.23)$$

Tukaj je:

C – podmnožica vseh sprememb (samo spremembe, v katerih je spremenjena ta datoteka) in

n(c) – število datotek, ki so bile spremenjene v spremembi (commitu) *c*.

$$AVGCHANGESET = \frac{1}{|C|} \sum_{i=1}^{N} n(c_i) \quad (2.24)$$

Tukaj je:

C – podmnožica vseh sprememb (samo spremembe, v katerih je spremenjena ta datoteka),

|C| - število vseh sprememb datoteke in

$n(c_i)$ – število datotek, ki so bile spremenjene v spremembi (commitu) c_i.

3 STROJNO UČENJE

Svet je preplavljen s podatki. Količina le teh iz dneva v dan narašča in temu ni videti konca. Prisotnost računalnikov na vsakem koraku našega življenja, ki omogočajo enostavno shranjevanje podatkov, poceni diski in shranjevanje na spletu (v oblaku), nam ponuja enostavno rešitev, kaj storiti z njimi, ali obdržati vse. Vsaka naša odločitev, bodisi na spletu ali v trgovini, se beleži. Medtem ko se količina podatkov neizprosno veča, se proporcionalno manjša delež podatkov, ki jih ljudje razumemo. V tej gori podatkov, ki jih ljudje sami po sebi ne razumemo, se skrivajo informacije, potencialno uporabne informacije, ki jih redko odkrijemo sami, zato se poslužujemo metod strojnega učenja (angl. machine learning) [19].

Strojno učenje je področje računalništva, ki pogosto uporablja statistične metode, ki dajejo računalniškemu sistemu zmožnost učenja s podatki, ne da bi bili predhodno sprogramirani za neko specifično področje. Z drugimi besedami strojno učenje se ukvarja z metodami, ki uporabljajo izkušnje za izboljšanje zmogljivosti [20].

Tom Michael Mitchell v [21] podaja formalno definicijo algoritmov strojnega učenja: »Računalniški program se naj uči iz izkušenj E glede na določen nabor opravil T in merila uspešnosti P. Zmogljivost programa v nalogah T(merjena s P) se izboljša z izkušnjami E.« [21]

V splošnem to pomeni, da moramo, če želimo imeti dobro definiran problem, identificirati te tri karakteristike: razred opravil, mero zmogljivosti in vir izkušenj [21]. V tabeli 3.1 podajamo nekaj primerov definicij problemov, na sliki 3.1 pa je prikazan načrt sistema strojnega učenja za učenje dame. ***Performance System*** je modul, ki reši nalogo (odigra igro) na podlagi funkcije, ki se je je do sedaj naučil sistem. ***Critic*** dobi na vhodu zgodovino iger in na izhodu poda nabor učnih vzorcev. ***Generalizer*** vzame učne vzorce in generira hipotezo

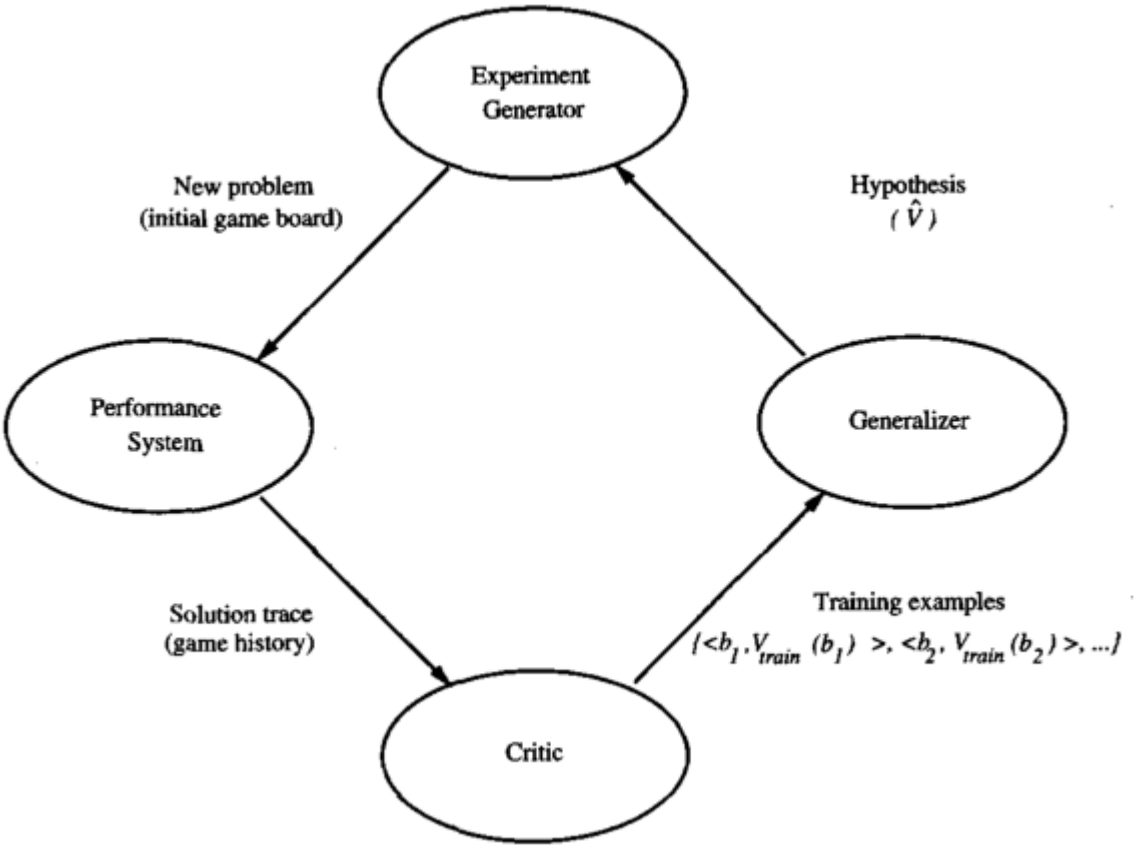
oz. funkcijo, ki pokriva učne vzorce. ***Experiment generator*** dobi na vhodu trenutno hipotezo (naučeno funkcijo) in na izhodu poda nov problem (novo stanje igre) [21].

Tabela 3.1: Primeri definicij strojnega učenja [21]

Problem	**Opravila (T)**	**Mera zmogljivosti (P)**	**Izkušnje (E)**
Program, ki se uči igrati damo	Igranje dame	Odstotek zmag proti nasprotniku	Igranje trening iger sam s seboj
Vožnja avtonomnega vozila	Vožnja po avtocesti z uporabo senzorjev vida (kamera, …)	Prevožena razdalja, preden pride do napake (napako oceni človeški opazovalec)	Zaporedje slik in ukazov zaviranja, pospeševanja ter zavijanja, zajetih med opazovanjem vožnje človeka
Razpoznava besedila	Razpoznavanje in klasifikacija besed v slikah	Odstotek pravilno prepoznanih besed	Baza s slikami besed, ki so že pravilno klasificirane

Načrtovalec sistema strojnega učenja se mora odločiti na podlagi kakšnih izkušenj se bo program (učenec) učil. Tip podatkov, ki je na voljo programu za učenje, ima lahko ključno vlogo pri uspehu oz. neuspehu učenca. Eden izmed ključnih atributov je, ali imajo podatki v učni množici neposreden ali posreden vpliv, glede na odločitve, ki jih naredi sistem. Vzemimo za primer igro dama, kjer se sistem lahko nauči iz neposrednih primerov v učni množici. To so vzorci, sestavljeni iz posameznih stanj v igri, in pravilne poteze, ki sledijo. Lahko pa so podatki posredni, kjer se nahajajo zaporedja potez in končni rezultat igrane igre (zmaga ali poraz). Drugi pomembni atribut je stopnja, do katere lahko učenec kontrolira zaporedje učnih vzorcev. Ta se lahko zanaša na učitelja, da izbere vzorce, ki več doprinesejo k učenju in poda pravilno potezo zanj. Druga možnost pa je, da učenec poda stanja, v katerih se zmede, in vpraša učitelja za pravilno potezo. Tretji atribut učne množice

pa predstavlja porazdelitev vzorcev, nad katerimi merimo zmogljivost sistema. V splošnem je učenje najbolj zanesljivo, kadar je porazdelitev učnih vzorcev podobna prihodnjim testnim primerkom [21].



Slika 3.1: Načrt sistema za učenje dame [21]

Mnogo drugih disciplin ima velik vpliv na strojno učenje. V nadaljevanju podajamo nekaj le teh in njihov vpliv [21]:

- umetna inteligenca – učenje simboličnih predstavitev konceptov, uporaba predhodnega znanja skupaj z učnimi podatki za nadzorovano učenje;
- teorija izračuna računske kompleksnosti – teoretične omejitve različnih problemov učenja, merjene z težavnostjo izračuna (angl. computational effort), številom učnih vzorcev, napak itd., potrebnih za učenje;
- teorija nadzora – procedure, ki se učijo nadzirati proces, da lahko optimizirajo definirane cilje in predvidijo naslednje stanje procesa;

- teorija informacij – meri entropijo in vsebino informacij;
- statistika – karakterizacija napak, ki nastopijo ob ocenitvi natančnosti hipotez, uporaba raznih statističnih testov in intervali zaupanja;
- ostale discipline: filozofija, psihologija in nevrobiologija.

3.1 Metode strojnega učenja

Metode strojnega učenja delimo glede na način uporabe naučenega znanja [22]:

- klasifikacija (uvrščanje),
- regresija,
- učenje asociacij in logičnih relacij,
- učenje sistemov (diferencialnih) enačb,
- spodbujevalno učenje (učenje z ojačitvijo) in
- razvrščanje.

Ena najpogostejših uporab strojnega učenja je **klasifikacija** (angl. classification). Naloga klasifikatorja je za objekt (problem), opisan z množico atributov (značilk, lastnosti), določiti, kateremu izmed možnih razredov pripada. Atributi so neodvisne zvezne ali diskretne spremenljivke, razred pa je odvisna diskretna spremenljivka, ki ji določimo vrednost (razred), glede na vrednosti neodvisnih spremenljivk [22].

Primer klasifikacijskega problema je postavljanje medicinske diagnoze. Vsakega pacienta opisujejo zvezni atributi, kot so višina, teža, starost, krvni pritisk, telesna temperatura itd., diskretni atributi, kot so spol, barva kože itd. Naloga klasifikatorja je postaviti diagnozo oz. določiti enega izmed možnih razredov, npr. pacient je zdrav, prehlajen, ima gripo ali angino [19] [22].

Klasifikator mora imeti na nek način predstavljeno diskretno funkcijo, ki preslika nabor atributov v razred. Funkcija je lahko podana vnaprej ali pa je naučena iz podatkov (rešenih primerov iz preteklosti). Če vzamemo prejšnji primer, kjer smo postavljali diagnozo

pacientu, potrebujemo opise pacientov skupaj z njihovimi diagnozami. Naloga učnega algoritma je torej ta, da iz množice opisov pacientov z znanimi diagnozami zgradi pravilo, ki ga lahko uporabimo za diagnosticiranje novih pacientov [22].

Klasifikatorje ločujemo glede na način predstavitve njihove funkcije. Najbolj znani klasifikatorji so [19] [22]:

- odločitvena drevesa (angl. decision tree),
- naivni Bayesov klasifikator (angl. naive Bayes),
- klasifikator z najbližjimi sosedi – k-NN (angl. k-nearest neighbors),
- logistična regresija (angl. logistic regression),
- klasifikator po metodi podpornih vektorjev – SVM (angl. support vector machine) in
- usmerjene umetne nevronske mreže (angl. artificial neural network).

Naslednja pogosto uporabljena metoda je **regresija**. Naloga regresijskega prediktorja je za objekt določiti vrednost odvisne spremenljivke, ki pa je, za razliko od klasifikacijskega razreda, zvezna. Če smo pri klasifikaciji pacientu poskušali določiti diagnozo, želimo pri regresiji pacientu določiti indeks težavnosti obolenja [22].

Podobno kot pri klasifikaciji mora tudi regresijski prediktor imeti na nek način predstavljeno zvezno funkcijo, ki preslika prostor atributov v napovedano vrednost. Učni algoritem ima torej nalogo, da iz množice opisov primerkov z že znanimi vrednostmi odvisne spremenljivke izračuna zvezno funkcijo, ki jo lahko uporabimo za določanje vrednosti novih primerov [22].

Tako kot klasifikatorje tudi regresijske prediktorje ločimo glede na način predstavitve regresijske funkcije. Najpogostejši regresorji so [22]:

- regresijska drevesa,
- linearna regresija,
- lokalno utežena regresija,
- regresija po metodi podpornih vektorjev in

- usmerjene umetne nevronske mreže.

Nenadzorovano učenje (angl. unsupervised learning) ima drugačno nalogo kot klasifikacija. Podani so opisi primerov (vsak primer opišem z atributi), njihovih razredov pa ne poznamo. Naloga učnega algoritma je objekte razvrstiti v razrede. Poleg besede razvrščanje se včasih v literaturi zasledita tudi izraza grupiranje in grozdenje (angl. clustering). Razvrščanje se uporablja pri analizi naravnih in tehnoloških procesov, ekonomskih trendov in pri preverjanju odvisnosti podatkov itd. [22]

Število želenih razredov je lahko podano vnaprej kot predznanje ali pa mora primerno število razredov določiti sam učni algoritem. Učni algoritem mora določiti relativno majhno število koherentnih[2] razredov (skupin primerov), ki so si med seboj čimbolj podobni. Podobnost med primeri je odvisna od izbrane metrike, ki je odločilnega pomena za rezultate razvrščanja [22].

Algoritmi nenadzorovanega učenja običajno uporabljajo enega izmed treh osnovnih pristopov [22]:

- od spodaj navzgor: algoritem na začetku obravnava vsak primerek kot svoj razred, nato iterativno združuje najbolj podobne razrede med seboj, dokler ne ostaneta samo še dva razreda, na koncu uporabnik ali pa sam sistem izbere najustreznejše število razredov;
- od zgoraj navzdol: na začetku vsi objekti pripadajo enemu razredu, algoritem iterativno razbija razrede na podrazrede, dokler ne doseže željenega števila razredov;
- algoritem na začetku izbere *n* primerov kot nosilce razredov, kjer je *n* željeno število razredov, zatem se vsi primerki glede na podobnost razvrstijo v enega izmed *n* razredov. Glede na lastnosti posameznih razredov se izbere novih *n* predstavnikov razredov (en razred lahko da več kot enega predstavnika ali pa tudi nobenega). Postopek se ponavlja tako dolgo, dokler razredi niso dovolj koherentni.

[2] koherenca – medsebojna povezanost, odvisnost

Spodbujevano učenje ali učenje z ojačitvijo (angl. reinforcement learning) se ukvarja s problemom, kako nekega avtonomnega agenta v danem okolju naučiti optimalnih akcij za dosego zastavljenih ciljev. Agent dobiva iz okolice podatke o trenutnem stanju in z akcijami vpliva na to, da se stanje okolja spreminja. Poleg opisa stanja lahko dobi agent tudi nagrado ali kazen, ki pa ni nujno posledica akcije, ampak zaporedje več akcij. Povratna informacija v obliki kazni ali nagrade je zakasnjena, kar privede do oteženega in počasnejšega učenja. Agent (učenec) se lahko posluži bolj konzervativnega pristopa in se zanaša na do sedaj naučeno, ali pa raziskuje in uporablja redkeje uporabljene akcije, da si pridobi nove izkušnje v novih, neznanih situacijah [22].

Naloga učenca je sestaviti optimalno strategijo, ki maksimizira uteženo vsoto vseh nagrad oz. minimizira uteženo vsoto vseh kazni. Običajno si želimo čim prej priti do cilja, zato je utežitev nagrad ponavadi taka, da so najpomembnejše takojšnje nagrade. Najpogostejši pristop k spodbujevanemu učenju je iterativno ažuriranje tabele ocen možnih akcij iz danega stanja [22].

S spodbujevanim učenjem se rešujejo problemi [22]:

- kontrole dinamičnih sistemov (upravljanje strojev),
- razni optimizacijski problemi in
- igranje iger (šah, dama, tarok itd).

3.2 K – najbližjih sosedov

Pri algoritmu k – najbližjih sosedov (na kratko *k*-NN) hranimo vse učne primere nespremenjene. Ko želimo napovedati razred r_x novemu primeru u_x, poiščemo med učni primeri *k* najbližjih (najbolj podobnih) primerov in pri klasifikaciji napovemo razred, ki mu pripada največ izmed *k* najbližjih sosedov. Pri regresiji pa napovemo povprečno vrednost razreda vseh *k* najbližjih sosedov. Parameter *k* običajno nastavimo na neko liho število. Če v učnih podatkih ni napak, potem se bo najbolje obnesel algoritem 1-NN. Če pa so v učnih primerih napake, pa lahko s povečevanjem parametra *k* zmanjšamo verjetnost, da je vseh

k učnih primerov napačnih. Po drugi strani pa z večanjem števila *k* povečujemo možnost, da h klasifikaciji prispevajo tudi učni primeri, ki niso dovolj podobni novemu primeru [22].

Algoritem *k*-NN je občutljiv na izbrano metriko pri računanju razdalj med novim primerom in učnimi primeri. Največkrat se uporablja evklidska razdalja. Vsi zvezni atributi se normalizirajo na interval [0, 1], za diskretne atribute pa je razdalja med različnima vrednostnima 1, med enakima pa 0 [22].

Bolj robustna varianta algoritma *k*-NN uporablja uteževanje vpliva učnih primerov na napoved. Vpliv se uteži z razdaljo učnega primera do novega primera. Razdalja lahko vpliva linearno, kvadratično, eksponentno itd. Če uporabljamo uteževanje učnih primerov, potem dejansko ni potrebno omejevati števila najbližjih sosedov na *k*, ampak lahko na napoved vplivajo vsi učni primeri, saj bo vpliv zelo oddaljenih učnih primerov zanemarljiv [22].

3.3 Odločitvena drevesa

Odločitvena drevesa so tipičen predstavnik strojnega učenja. Objekti so v učni množici predstavljeni kot par ([$atribut_1$, $atribut_2$, …, $atribut_N$], *odločitev*). Izbrani atributi v vektorju bi naj na najboljši možen način opisovali posamezen objekt. Odločitev je tisti atribut, ki je znan pri objektih v učni množici, ne pa tudi pri objektih, kjer bomo s pomočjo odločitvenega drevesa sprejemali odločitve. Običajno gre pri odločitvi za lastnost, ki se je ne da izmeriti, lahko pa izmera te lastnosti predstavlja velik strošek ali pa veliko časovno zahtevnost [23].

Odločitveno drevo je sestavljeno iz notranjih vozlišč, ki ustrezajo atributom vej, ki ustrezajo podmnožicam vrednosti atributov in listov, ki ustrezajo razredom [22] [23]. Pri tem so pogoji (pari atribut – podmnožica vrednosti), ki jih srečamo na poti, konjunktivno[3] povezani [22].

[3] Konjunkcija je operacija med izjavami. Konjunkcija izjav A in B je $A \wedge B$, ki je pravilna, samo če sta oba operanda pravilna (predstavlja torej logično operacijo in).

Odločitvena drevesa uporabljamo pri klasifikaciji, kjer z njimi poskušamo [19] [22] [23]:

- napovedati dogodek v prihodnosti,
- poiskati alternativne možnosti za dosego cilja, ki bodo:
    - skrajšale čas,
    - zmanjšale stroške,
    - omogočile doseganje željenih rezultatov.

Odločitveno drevo zgradimo s pomočjo učne množice, ki jo sestavljajo učni objekti. Razredi se medsebojno izključujejo, kar pomeni, da lahko učni objekt pripada samo enemu razredu. Prav tako ne sme več objektov, ki so opisani z enakim vektorjem atributov, imeti različne odločitve. Atributi so lahko diskretni ali numerični. Numerične atribute pred začetkom gradnje odločitvenega drevesa preslikamo v diskretno obliko. Način preslikave numeričnih atributov v diskretno obliko lahko odločilno vpliva na uspešnost zgrajenega odločitvenega drevesa [23].

Postopek generiranja odločitvenega drevesa imenujemo tudi indukcija odločitvenega drevesa. Pri generiranju začnemo s praznim drevesom in celotno množico učnih objektov. V vsakem koraku s pomočjo hevristične funkcije izberemo atribut, ki na poti do trenutnega vozlišča še ni bil uporabljen. Učno množico delimo na podlagi možnih vrednosti atributa z uporabo strategije deli in vladaj [23]. Postopek ponavljamo tako dolgo, dokler ni zadoščeno pogojem za končanje gradnje odločitvenega drevesa, ki so [22] [23]:

- vsi učni objekti v določenem vozlišču pripadajo istemu razredu;
- odstotek objektov, ki pripadajo večinskemu razredu v določenem vozlišču, je večji ali enak toleranci pri predhodnem klestenju;
- premalo učnih primerov za zanesljivo nadaljevanje gradnje;
- zmanjkalo je atributov (na poti do določenega vozlišča smo porabili vse atribute, učni objekti v tem vozlišču pa ne pripadajo istemu razredu).

Psevdokod algoritma za gradnjo odločitvenega drevesa je predstavljen na sliki 3.2. S *C* označimo množico učnih objektov in A_i predstavlja oznako za atribut ter $\{A_{i1}, A_{i2}, \ldots, A_{iv}\}$ so vse možne vrednosti tega atributa [23].

1. *Če so vsi objekti v* $\mathbf{C}$ *člani istega razreda ali če je* $\mathbf{C}$ *prazna množica, označimo vozlišče kot list, mu določimo razred in končamo postopek. Če je* $\mathbf{C}$ *prazna množica v list namesto razreda zapišemo 'OdločitevNiMogoča'.*
2. *Naj bo* $\mathbf{A_i}$ *atribut izbran s pomočjo hevristične funkcije v trenutnem vozlišču z vrednostmi* $\{\mathbf{A_{i1}}, \mathbf{A_{i2}}, \ldots, \mathbf{A_{iv}}\}$*. Vozlišče označimo kot notranje in vanj vstavimo atribut* $\mathbf{A_i}$*.*
3. *S pomočjo vrednosti atributa razdelimo* $\mathbf{C}$ *na podmnožice* $\{\mathbf{C_{i1}}, \mathbf{C_{i2}}, \ldots, \mathbf{C_{iv}}\}$*. Vsaka podmnožica* $\mathbf{C_{ij}}$ *vsebuje tiste objekte iz* $\mathbf{C}$*, ki imajo vrednost atributa* $\mathbf{A_i}$ *enako* $\mathbf{A_{ij}}$*.*
4. *Za vsako vrednost atributa in ustrezno podmnožico* $\mathbf{C_{ij}}$ *ustvarimo vozlišče.*
5. *Za vsako ustvarjeno vozlišče ponovimo cel postopek gradnje z* $\mathbf{C_{ij}}$ *in z množico še ne uporabljenih atributov na poti do ustvarjenega vozlišča.*

Slika 3.2: Psevdokod algoritma za gradnjo odločitvenega drevesa [23]

Odločitveno drevo, ki je zgrajeno na podlagi vseh možnih vrednosti, pogosto vsebuje precej nepotrebne strukture (nepotrebnih vej), zato jih navadno pred uporabo oklestimo (angl. prune) [19]. Klestenje je postopek, s katerim ublažimo oz. skoraj v celoti odpravimo posledice prekomernega prilagajanja učnim objektom. Glavna vzroka za prekomerno prilagajanje odločitvenega drevesa učnim objektom sta šum v podatkih in pomanjkanje učnih primerov. Če je učnih objektov premalo, iz njih ne moremo izvleči splošnega znanja. Osnovna ideja je onemogočiti tvorbe oz. omogočiti odstranitev nepomembnih. S tem tudi omejimo velikost odločitvenega drevesa [23].

V splošnem ločimo med dvema vrstama klestenja [19] [22] [23]:

- predhodno in
- naknadno klestenje.

Predhodno klestenje se izvaja med samim postopkom gradnje drevesa. Ne gre za pravo klestenje, ampak za predčasno ustavitev gradnje odločitvenega drevesa. Na ta način lahko preprečimo gradnjo nepomembnih vej. Prednost predhodnega klestenja je možnost natančne določitve stopnje klestenja. Običajno določimo prag v odstotkih, ki pove, kolikšen delež učnih objektov z izidom različnih od večine bomo zanemarili [23].

V primerjavi s predhodnim klestenjem pri naknadnem delamo z že zgrajenim drevesom. Pri tej vrsti veje dejansko klestimo, saj jih nadomestimo z listi [22] [23]. Tudi naknadno klestenje obravnava dva pristopa [23]:

- naknadno klestenje s statično oceno napake in
- naknadno klestenje z zmanjševanjem napake.

Ključno vlogo pri gradnji odločitvenega drevesa ima hevristična funkcija, ki pregleda vse atribute, ki na poti do trenutnega vozlišča še niso bili uporabljenimi in med njimi izbrati takšnega, ki se pri delitvi učne množice najbolj približa idealni delitvi. Idealno delitev nam predstavljajo podmnožice z enakimi odločitvami. Ena izmed enostavnejših hevrističnih funkcij temelji na osnovi funkcije za oceno količine informacij – entropije. Funkcija (3.1) meri količino informacij odvisnosti od števila učnih objektov s posameznimi odločitvami oz. izidi. Hevristična funkcija za ocenitev atributa je predstavljena v 3.2. Atribut, ki ima najnižjo vrednost $E(C,A_i)$, je najprimernejši za nadaljevanje gradnje odločitvenega drevesa [23].

Če predpostavimo, da je učna množica sestavljena iz *n* objektov in *m* atributov, potem ima gradnja drevesa časovno zahtevnost $O(mn\log n)$. Če pa še dodamo operacijo klestenja, dobimo časovno zahtevnost $O(mn\log n) + O(n(\log n)^2)$ [19].

$$l(o_1, o_2, \dots, o_m) = -\frac{o_1}{\sum_{j=1}^{m} o_j} \times \log_m \frac{o_1}{\sum_{j=1}^{m} o_j} - \frac{o_2}{\sum_{j=1}^{m} o_j} \times \log_m \frac{o_2}{\sum_{j=1}^{m} o_j} - \dots - \frac{o_m}{\sum_{j=1}^{m} o_j} \times \log_m \frac{o_m}{\sum_{j=1}^{m} o_j} \quad (3.1)$$

Tukaj je:

$o_1, o_2, \dots, o_m$, – števila učnih objektov z izidi od 1 do m in

m – število možnih razredov.

$$E(C, A_i) = \sum_{j=1}^{v} \frac{\sum_{k=1}^{m} o_{ijk}}{\sum_{k=1}^{m} o_k} \times l(o_{ij1}, o_{ij2}, \dots, o_{ijm}) \quad (3.2)$$

Tukaj je:

C – učna množica,

A_i – atribut.

Prednosti odločitvenih dreves so [24]:

- so enostavna za razumevanje;
- omogočajo dodajanje novih scenarijev;
- lahko jih kombiniramo z drugimi tehnikami.

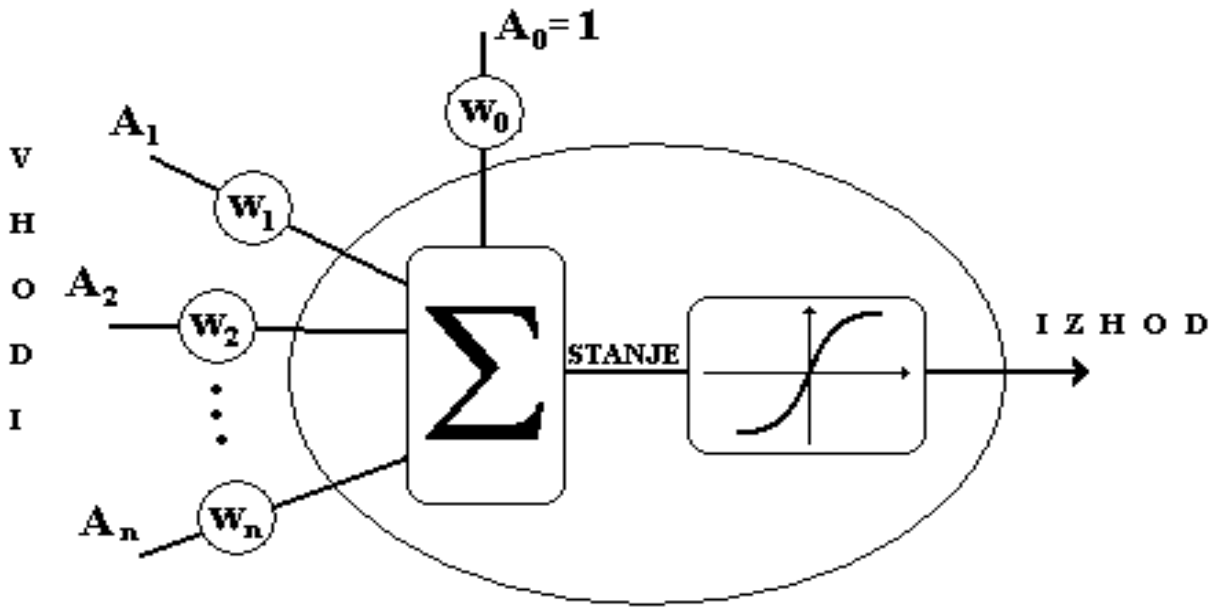
Odločitvena drevesa imajo tudi nekaj slabosti [24]:

- so nestabilna (majhna sprememba v podatkih lahko povzroči veliko spremembo v strukturi optimalnega odločitvenega drevesa),
- izračuni lahko postanejo zelo kompleksni, zlasti če je veliko odločitev med seboj povezanih.

3.4 Nevronske mreže

Začetki nevronskih mrež segajo že v zgodnja 40. leta, ko sta McCulloch in Pitts predstavila matematični model živčne celice-nevrona. Nevron predstavlja osnovni element za gradnjo nevronske mreže. Na sliki 3.3 je predstavljen model nevrona. Oznake od A_0 do A_n nam

predstavljajo vhode v nevron. Vsak vhod je obtežen z utežmi, ki so označene od W_0 do W_n, uteži W_0 pravimo tudi prag [23] [25].



Slika 3.3: Model nevrona [23]

Prva komponenta nevrona (slika 3.3) je vhodna funkcija *f* (3.3), ki predstavlja seštevek produktov vhodov in uteži [22] [23] [25].

Druga komponenta pa je aktivacijska funkcija. Izhod aktivacijske funkcije je lahko diskreten ali analogen. Poznamo naslednje aktivacijske funkcije [22] [23] [25]:

- stopničasta,
- linearna,
- odsekovno linearna,
- Gaussova in
- sigmoidna (npr. logistična, hiperbolični tangens).

$$f = \sum_{i=0}^{n} W_i \times A_i \tag{3.3}$$

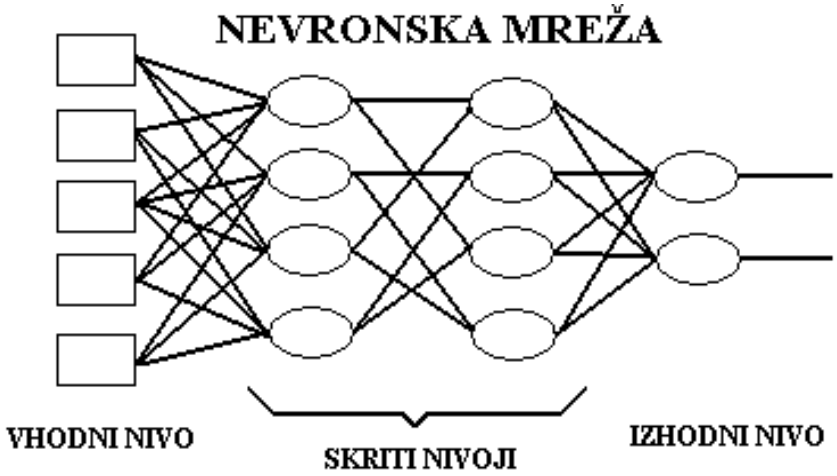
Tukaj je:

n – število vhodov,

A_i – vrednosti vhodov v nevron in

W_i – uteži.

Na sliki 3.4 je prikazana »feed-forward« nevronska mreža z dvema skritima nivojema, pri katerih ni dovoljeno povezati dveh nevronov na istem nivoju, povezati nevron na prejšnji nivo in preskočiti nevron pri povezavi naprej [23] [25].



Slika 3.4: Nevronska mreža [23]

Med klasične predstavnike nevronskih mrež štejemo [23]:

- Backpropagation NM,
- Hopfieldove NM in
- Kohonenove NM.

3.5 Ansambelske metode

»Ansambelsko« sprejemanje odločitev za človeka ni nič novega, saj ga prakticiramo v vsakodnevnem življenju. Vsakdanji primer »ansambelskega« odločanja je nakup nekega dražjega izdelka, npr. računalnika, instrumenta, avtomobila ipd., kjer za nasvet povprašamo prijatelje, sorodnike in prodajalce ter z njihovo pomočjo sprejmemo odločitev o nakupu. [26].

Osnovna ideja ansambelskih metod je, da namesto enega klasifikatorja zgradimo več le teh (imenovanih tudi eksperti), ki na koncu glasujejo za končno odločitev. Želimo si, da so klasifikatorji čim bolj raznoliki, saj želimo izkoristiti specializiranost posameznega klasifikatorja. Cilj ansambelskih metod je torej povečati zaupanje v pravilnost rezultata. Uporaba ansambelskih metod prinese povečano natančnost klasifikacije, vendar pa je te rezultate pogosto težko interpretirati [22] [26]. Znane ansambelske metode so [22] [27]:

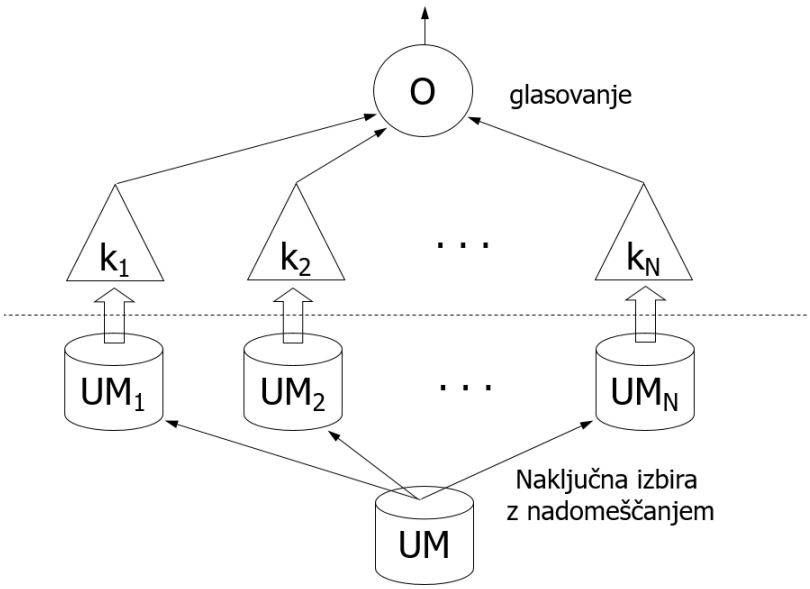
- »bagging«,
- »boosting«,
- »stacking«,
- naključni gozdovi (angl. random forest)…

Izraz »**baging**« izhaja iz izraza »bootstrap agreggating[4]«. Pri baggingu (slika 3.5) generiramo serijo različnih učnih množic. Če ima učna množica *n* primerov, potem vsakič *n* krat naključno izberemo primer iz učne množice z vračanjem. To pomeni, da se isti učni primer v tako generirani učni množici lahko večkrat ponovi, nekaterih primerov iz originalne učne množice pa sploh ne vsebuje. Nad vsako generirano učno množico poženemo učni algoritem. Tako dobimo veliko število potencialno različnih odločitev. Končno rešitev dobimo z glasovanjem - predstavlja jo rešitev z največ glasovi [22] [28].

[4] Bootstrap je metoda razmnoževanja učnih primerov, kadar jih nimamo dovolj za učenje [24].

Bagging se obnese predvsem pri nestabilnih učnih algoritmih z visoko varianco, kot so odločitvena in regresijska drevesa. Je robusten, saj z večanjem števila modelov ne pride do prevelikega prileganja učni množici [22].

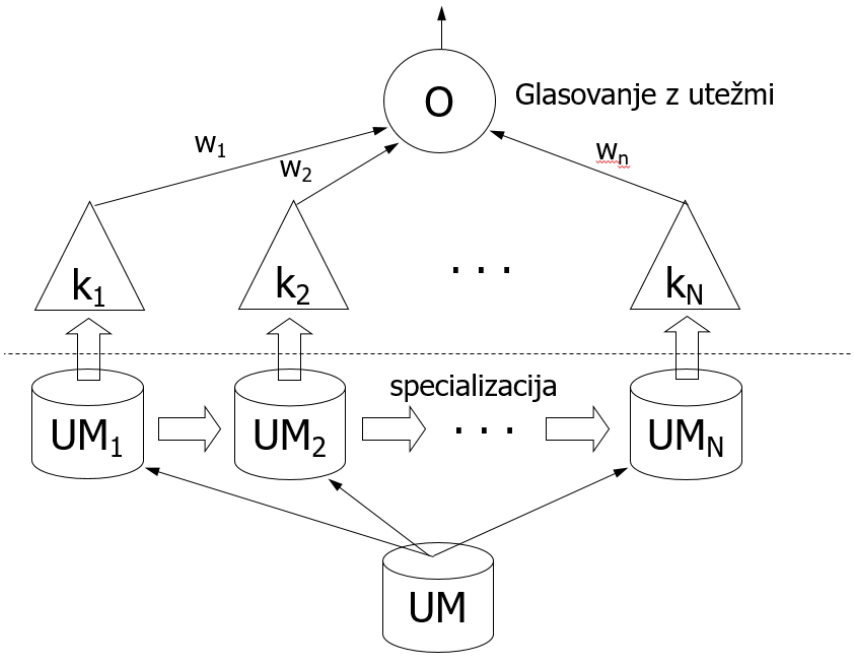
Ideja metode **boosting** (slika 3.6) je uteževanje učnih primerov glede na njihovo težavnost. Metoda predpostavlja, da učni algoritem zna obravnavati utežene učne primere. V primeru, da učni algoritem tega ne zna, uteževanje učnih primerov simuliramo tako, da generiramo učno množico. Učno množico generiramo na podoben način kot pri baggingu, vendar ne uporabimo uniformne distribucije za izbiro posameznih primerov. Primeri z večjo težo imajo večjo verjetnost, da bodo izbrani [22]. Postopek gradnje je iterativen, kar pomeni, da so klasifikatorji, ki jih dodajamo v ansambel, odvisni od rezultatov klasifikacije prejšnjih klasifikatorjev. Vsak nov klasifikator spodbudimo k specializaciji za primerke, ki v prejšnji iteraciji niso bili pravilno klasificirani [28].



Slika 3.5: Bagging [28]

Boosting pogosto dosega večjo natančnost, kot bagging in ga lahko uporabimo tudi na stabilnih učnih algoritmih z majhno varianco, vendar pa včasih pride do prevelikega prileganja učni množici (angl. overfitting) [22].

Metoda **naključnih gozdov** (anl. random forest) je namenjena izboljšanju natančnosti drevesnih algoritmov. Metoda je robustna, saj zmanjša varianco drevesnih algoritmov. Ideja je generirati zaporedje odločitvenih dreves, tako da se pri izbiri najboljšega atributa v vsakem vozlišču naključno izbere relativno majhno število atributov, ki vstopajo v izbor za najboljši atribut. Breiman [29] predlaga ali naključno izbiro $\log(\check{s}teviloAtributov) + 1$ ali pa preprosto popolnoma naključno izbiro atributa v vsakem vozlišču drevesa [22].



Slika 3.6: Boosting [28]

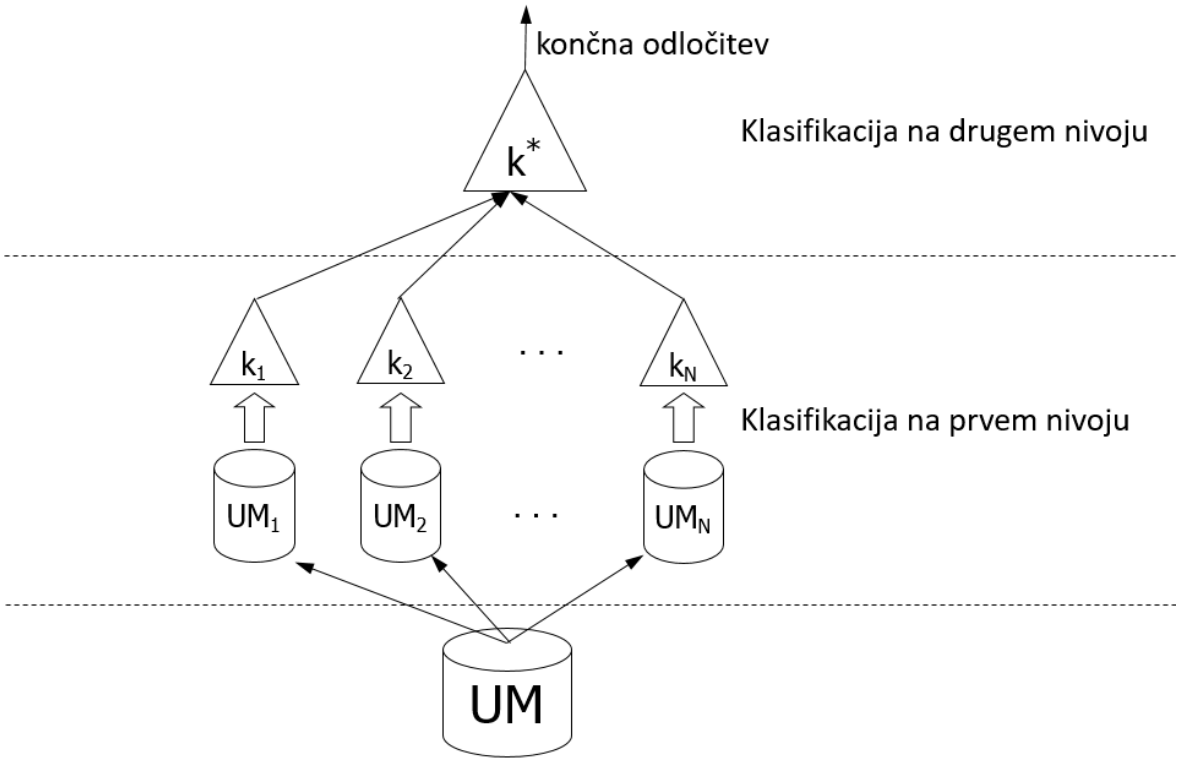
Vsako drevo, ki je zgrajeno na učni množici, se zatem uporabi za klasifikacijo novega primera po metodi glasovanja, kjer ima vsako drevo natanko en glas, ki ga nameni razredu, katerega bi to drevo klasificiralo [22].

Stacking (včasih tudi stacked generalization) je metoda, pri kateri združimo več različnih modelov strojnega učenja. Ta metoda ni tako razširjena kot bagging in boosting. Postopek gradnje je sledeč [27]:

- razdeli učno množico (podobno, kot pri baggingu uporabimo metodo bootstrap);
- na osnovi razdeljene učne množice naučimo klasifikatorje na prvem nivoju;

- izhodi klasifikatorjev prvega nivoja nam predstavljajo vhod v končni klasifikator na drugem nivoju, ki poda končno odločitev.

Opisan postopek je prikazan na sliki 3.7. Stacking je bil uspešno uporabljen pri nalogah nadzorovanega in nenadzorovanega učenja [27].



Slika 3.7: Stacking [27]

3.6 Metode podpornih vektorjev

Metode podpornih vektorjev – SVM (angl. support vector machine) so med najuspešnejšimi metodami za klasifikacijo in regresijo. Večina algoritmov strojnega učenja deluje tako, da iz množice atributov izbere pomembnejše in nad njimi zgradi neko ne preveč zapleteno funkcijo. SVM pa uporabi čim več razpoložljivih atributov, ki niso vsi zelo pomembni in jih z linearno kombinacijo lahko uporabimo za napovedovanje odvisne spremenljivke. Pri metodi SVM je pomembno predvsem, kako kombinirati atribute, saj bo sama metoda z ustrezno kombinacijo izluščila željeno informacijo [22].

Primerne so za učenje na veliki množici učnih primerov, opisanih z velikim številom manj pomembnih atributov. Prav tako metode SVM dosegajo veliko točnost napovedi. Slabost le teh je, da težko interpretiramo naučeno odločitev [22].

Osnovna ideja algoritma SVM je v danem prostoru atributov postaviti optimalno hiperravnino. Če so primeri linearno ločljivi, je v splošnem možnih več hiperravnin, ki ločujejo dva razreda med seboj. Optimalna hiperravnina je tista, ki je enako in najbolj oddaljena od najbližjih primerov iz obeh razredov. Najbližjim primerom hiperravnine pravimo podporni vektorji, razdalji hiperravnine od podpornih vektorjev pa rob (angl. margin). Optimalna hiperravnina je tista, ki ima maksimalni rob [22].

Druga bistvena ideja metode SVM je uporaba implicitne transformacije atributnega prostora v kompleksnejši atributni prostor. Uporabnikom ni potrebno narediti transformacije, saj za to poskrbi metoda SVM. V originalnem prostoru pogosto linearna hiperravnina ne zadošča za sprejemljivo klasifikacijsko točnost. Z nelinearno transformacijo lahko postane prostor primeren za linearno ločitveno hiperravnino [22].

4 SORODNA DELA

V okviru magistrske naloge smo pregledali članke, učbenike, knjige in spletne vire na temo metrik programske opreme in strojnega učenja ter preučili osnovne koncepte obravnavanih tem.

Za iskanje člankov smo večinoma uporabili elektronske vire[5], ki jih za študente in zaposlene ponuja univerza v Mariboru. Članke in nekatere knjige smo našli v digitalnih knjižnicah:

- ScienceDirect[6],
- Springer Link[7],
- IEEE Xplore[8] in
- na straneh splošnih spletnih iskalnikov.

Ob iskanju smo se osredotočili na vire, ki so med naslovi, ključnimi besedami ali povzetki navajali sledeče pojme ter besedne zveze:

- metrike (angl. metrics),
- metrike programske opreme (angl. software metrics),
- metrike kvalitete programske opreme (angl. software quality metrics),
- strojno učenje (angl. machine learning),
- zaznavanje napak (angl. defect prediction),
- razredi (koda) z napako (angl. fault-prone clasess).

V nadaljevanju poglavja nekaj izbranih člankov predstavimo podrobneje.

[5] http://www.ukm.um.si/elektronski-viri
[6] https://www.sciencedirect.com
[7] https://link.springer.com
[8] https://ieeexplore.ieee.org/

4.1 Iskanje novih metrik sprememb za zaznavanje napak v programskih sistemih

V članku *Mining Github for Novel Change Metrics to Predict Buggy Files in Software Systems* so avtorji definirali nekaj metrik sprememb, za katere je moč intuitivno sklepati, da so pomembne za zaznavanje napak. Model za zaznavanje napak so zgradili na osnovi petih različnih verzij projekta Eclipse JDT [13].

Podatke, potrebne za izračun metrik sprememb (opisanih v podpoglavju 2.2), so pridobili iz repozitorija Github. Za pridobivanje vrednosti prvih šestih metrik (shranjene spremembe, število dodanih vrstic, izbrisanih vrstic, avtorjev, sprememb v zadnjih šestdesetih dneh in zadnja sprememba) so se prebrale zaporedno vse spremembe, ki so bile narejene nad izvorno kodo med datumoma D1 in D2 (datuma med dvema izdajama Eclipse JDT) [13]. Za vsako opravljeno spremembo pridobijo podatke o [13]:

- avtorju spremembe,
- datumu in času spremembe,
- številu vseh spremenjenih datotek,
- imenih modificiranih datotek in
- številu vrstic dodanih ali izbrisanih iz spremenjene datoteke.

Ohranijo se samo datoteke, ki so bile uporabljene v obeh izdajah (npr. za pridobivanje metrik za verziji 3.2 in 3.3, če je ena datoteka bila dodana za verzijo 3.3, v 3.2 ni obstajala, potem takšne datoteke ne upoštevamo). Datoteke, ki med obema izdajama niso bile spremenjene, imajo vrednost 0 za vseh prvih šest metrik in se izpustijo [13].

Za izračun vrednosti metrike število odpravljenih napak v razvoju so bile upoštevane samo spremembe, ki so v sporočilu vsebovale besede »bug«, »fix«, ali »fixed« [13].

Pri izračunu entropije so obdobje razdelili na 12 delov (en del predstavlja približno en mesec, saj Eclipse izide na približno eno leto). Za vsako datoteko nato izračunajo, kolikokrat je bila spremenjena v nekem obdobju, temu sledi izračun entropije po 2.21 [13].

Za izračun največje in povprečne spremembe uporabljajo avtorji tri spremenljivke:

- *Max* – hrani vrednost največ spremenjenih datotek, ki so bile spremenjene skupaj z opazovano datoteko v spremembah;
- *Count* – hrani število vseh sprememb, v katerih se pojavi ta datoteka;
- *Sum* – hrani vsoto vseh spremenjenih datotek [13].

Za vsako spremembo je bil postopek sledeč [13]:

- za vsako datoteko, ki jo ta sprememba modificira povečaj *Count* za 1;
- naj bo *N* število vseh datotek v opazovani spremembi;
    - k vrednosti *Sum* za datoteko prištej *N-1* (ker ne štejemo trenutno opazovane datoteke);
    - spremeni *Max* na (*N – 1*), če velja *Max* < (*N – 1*).

Končna vrednost spremenljivke *Max* predstavlja vrednost metrike največje spremembe. Povprečno spremembo izračunamo tako, da *Sum* delimo z *Count* [13].

Za izračun števila napak po izidu so na osnovi poročil o napakah poiskali spremembe v naslednji verziji, ki so odpravljale opisano napako. Pri tem so predpostavili, da so vse napake bile popravljene v roku šestih mesecev. To pomeni, da so vse napake, javljene v prvih šestih mesecih po izidu ene verzije, bile popravljene do naslednjega izida [13].

Za izračun metrik so bile uporabljene izdaje orodja Eclipse od 3.0 do 3.5 [13], Natančne podatke o verziji in datumu izida podaja tabela 4.1.

Tabela 4.1: Izdaje orodja Eclipse [13]

Verzija	**Datum izida**
3.0	25.6.2004
3.1	27.6.2005
3.2	29.6.2006
3.3	25.6.2007
3.4	17.6.2008
3.5	11.6.2009

Avtorji so za izgradnjo modelov uporabljali algoritme naivni bayes (NB), odločitveno drevo (classification and regression tree – CART), logistično regresijo in NB tree, slednji je hibrid

med naivnim bayesom in odločitvenimi drevesi. Za primerjavo so uporabili 10-kratno prečno preverjanje. Podatki so bili razdeljeni na 10 približno enakih podmnožic. Klasifikator je bil naučen na devetih podmnožicah, preostala podmnožica je bila uporabljena za testiranje/validacijo. Vsakič znova so bile izračunane vrednosti števila pravilno klasificiranih pozitivnih primerov - TP (angl. true positives), števila pravilno klasificiranih negativnih primerovi – TN (angl.true negatives), števila napačno klasificiranih pozitivnih primerov – FP (angl. false positives) in števila napačno klasificiranih negativnih primerov – FN(angl. false negatives). Vrednosti za vseh 10 podmnožic so na koncu kombinirali, da so dobili povprečno preciznost, priklic in metriko f [13].

Rezultati so pokazali, da se je za dano nalogo najbolje izkazal algoritem NB tree. V tabeli 4.2 so prikazane vrednosti preciznosti, priklica in mere f za algoritem NB tree v posameznih izdajah orodja Eclipse JDT. Prav tako konsistentnost rezultatov nakazuje, da je izbran nabor metrik dovolj dober za izgradnjo modela za napovedovanje napak [13].

Tabela 4.2: Rezultati napovedovanja algoritma NB tree [13]

	Preciznost	**Priklic**	**F-mera**
Eclipse 3.0 – 3.1	76.4	76.4	76.3
Eclipse 3.1 – 3.2	75.4	76.4	75
Eclipse 3.2 – 3.3	74.6	77.9	73.6
Eclipse 3.3 – 3.4	75.3	76.1	75.3
Eclipse 3.4 – 3.5	73.4	75.4	72.9
Povprečje	75	76.4	74.62
Standardni odklon	1.11	0.91	1.363

4.2 Uporaba pragovnih vrednosti metrik programske opreme za napovedovanje napak v razredih v objektno orientirani programski opremi

Avtorji članka »*Using Software Metrics Thresholds to Predict Fault-Prone Classes in Object-Oriented Software*« so si za cilj študije postavili preučiti in primerjati različne pragovne vrednosti za napovedovanje napak [2].

V prvem koraku so izbrali nabor metrik, ki jih bodo uporabili za napovedovanje napak. Odločili so se za uporabo metrik [2]:

- LOC,
- CBO,
- RFC in
- WMC.

Za tak način so se odločili zato, ker bi naj bil to nabor metrik, ki se najbolje obnese za napovedovanje napak in ker vsaka metrika doprinese drugačen tip informacij [2].

Drugi korak je predstavljal izbor podatkov o metrikah. Odločili so se da pridobijo podatke iz repozitorija PROMISE. V študiji so nato uporabili podatke projektov [2]:

- Apache ANT,
- Apache IVY,
- KC1,
- JEdit in
- Eclipse JDT.

Tretji korak je bil določitev pragovnih vrednosti za vsako metriko z uporabo algoritmov krivulje ROC in Alvesovega rangiranja, saj metoda v izvornem članku ni imela imena, zato so jo poimenovali po avtorju članka [2].

V četrtem koraku so izbrali ustrezen algoritem strojnega učenja, ki bo služil kot osnovna metoda za preverjanje zmogljivosti (natančnosti) pragovnih vrednosti metrik. Izbran je bil algoritem bayesova mreža [2].

V zadnjem koraku je bila izvedena še primerjava zmogljivosti pragovnih vrednosti in modela strojnega učenja – bayesove mreže. Za izračun zmogljivosti so uporabljali merili lažna pozitivna stopnja – FPR (angl. false positive rate) in lažna negativna stopnja – FNR (angl.

false negative rate). Te vrednosti se da enostavno izračunati iz konfuzijske matrike (angl. confusion matrix) [2].

Pragovne vrednosti, določeno po metodi ROC krivulj in Alvesovega rangiranja za projekt Apache ANT, so predstavljene v tabeli 4.3. V vrstici z oznako Alves30 so predstavljene vrednosti na 30% porazdelitve, v Alves70 pa na 70% le te [2].

Tabela 4.3: Pragovne vrednosti za projekt Apache ant [2]

	Metrika			
Metoda	LOC	CBO	RFC	WMC
ROC	336	9	46	15
Alves30	327	7	40	17
Alves70	1031	18	103	61

Za vsako metodo so izvedli klasifikacijo 4 krat. Učna množica je bila vsakič pripravljena malo drugače. V prvi iteraciji so bili razredi klasificirani tako, da so vsebovali napako, če je vsaj ena izmed metrik presegla prag (predstavljeno v tabeli 4.3 in originalnem članku). V drugi iteraciji je bil razred klasificiran tako, da je vseboval napako, če sta vsaj dve metriki presegli vrednost praga. Klasifikacijo so izvedli z orodjem Weka in z uporabo 10-kratnega prečnega preverjanja [2].

Ugotovili so da, v primeru, ko je bil prag določen z ROC krivuljo, obstaja obratno sorazmerje med FPR in FNR. Če je bilo povečano število metrik, ki so morale preseči vrednost praga, se je FPR zmanjšal, medtem ko se je FNR povečal. Za ta pojav obstaja logična razlaga, saj s povečanjem števila metrik, ki morajo preseči prag, da je razred klasificiran, kot razred z napako, bo več razredov klasificiranih kot takih, ki ne vsebujejo napake [2].

V primeru uporabe metode določanja praga z Alvesovim rangiranjem so zavrgli rezultate za določanje praga z metodo na 70% porazdelitve, saj je dala izjemno slabe rezultate. Obdržali so tako samo rezultate, kjer je bil prag določen na 30% porazdelitve. V slednjem primeru so ugotovili, da daje klasifikacija dobre rezultate v primeru, ko morata dve metriki preseči vrednost praga, da je razred klasificiran kot razred z napako [2].

4.3 Uporaba metrik programske opreme in ansambelskih metod za zaznavanje napak

V članku »*Using Source Code Metrics and Ensemble Methods for Fault Proneness Prediction*« so si avtorji za cilj zastavili razvoj modela za napovedovanje napak z uporabo metrik programske opreme in ansambelskih metod [30].

Za eksperimentalni del so pridobili podatke o metrikah iz repozitorija PROMISE, uporabili so podatke o 45 projektih. Kot vhod v model so med drugim izbrali metrike: DIT, WMC, RFC,CBO, LCOM, LOC, CC, … [30]

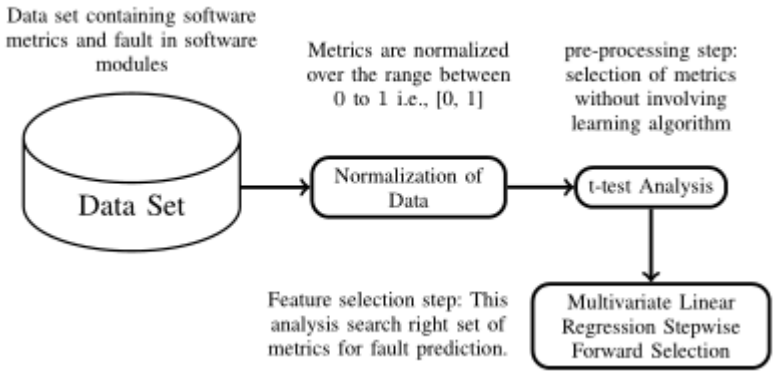
Na sliki 4.1 je prikazan postopek ogrodja za izbiro metrik za učenje modelov za napovedovanje napak. Prvi korak je normalizacija podatkov, temu sledi analiza z uporabo statističnega testa »t-test«, kjer se izberejo metrike za uporabo v modelu. V zadnjem koraku pa se z uporabo linearne regresije izberejo metrike, ki ne vsebujejo redundantnih informacij, Te metrike nato uporabijo za učenje modelov [30].

Za klasifikacijo so avtorji izbrali heterogene ansambelske metode (vsi klasifikacijski modeli so različni) [30]. V ansamblu so bili uporabljeni sledeči klasifikacijski modeli [30]:

- logistična regresija,
- umetna nevronska mreža in
- tri različice »Radial Basis Function« nevronske mreže.

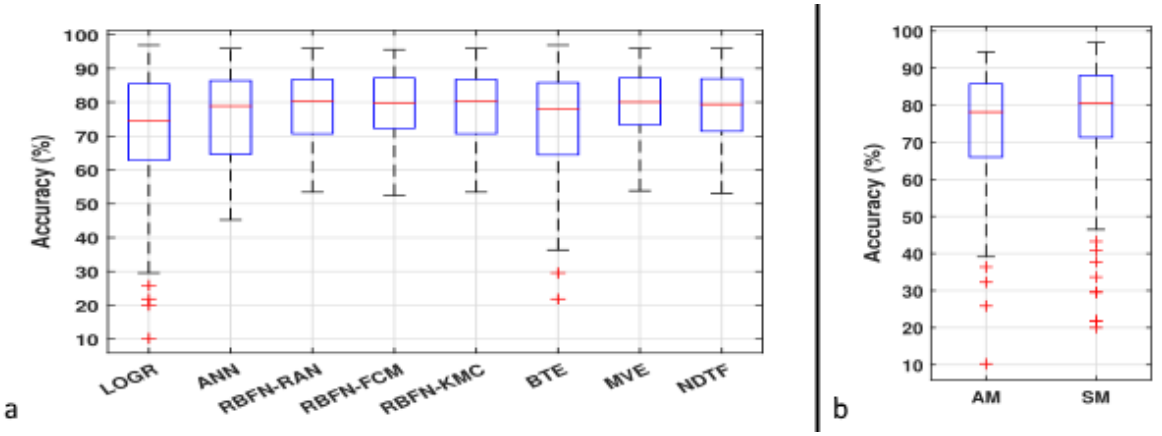
Za glasovanje o končni odločitvi so uporabljali naslednja pravila [30]:

- najboljši model iz učne množice,
- večinsko glasovanje in
- gozd odločitvenih dreves - DTE(angl. decision tree forest).



Slika 4.1: Postopek izbire metrik [30]

Avtorji so skozi eksperimente sklepali, da z izbiro samo določenih metrik iz celotnega nabora lahko izboljšajo klasifikacijo in zgradijo boljši model. Natančnost za izbran nabor je prikazana na sliki 4.2b. AM označuje celoten nabor metrik, SM pa izbran nabor metrik. Iz slike 4.2a pa je moč razbrati, da ansambelske metode izboljšajo natančnost klasifikacije v primerjavi s samostojnimi modeli, med ansambelskimi metodami pa je najnatančnejša metoda večinskega glasovanja [30].



Slika 4.2: Natančnost klasifikacije [30]

4.4 Ostalo

V tem podpoglavju še na kratko opišemo članke, ki so bili pregledani, vendar jih nismo podrobneje opisovali.

V članku »*Defect prediction with bad smells in code*« so avtorji skušali podati odgovor na vprašanje, ali lahko z metrikami za zaznavanje slabega vonja v kodi, pridobljenimi z orodjem »Microsof CodeAnalysis tool«, dodane k osnovnemu naboru metrik, izboljšajo napovedovanje napak v industrijskih projektih. Ugotovili so, da so se rezultati v razširjenem naboru izboljšali za 0,0091 [31].

Članek »*Software Metrics for Fault Prediction Using Machine Learning Approaches*« se osredotoči na primerjavo različnih del, ki obravnavajo temo napovedovanje napak z uporabo metrik programske opreme. Med seboj primerjajo 13 del, ki temeljijo na podatkih, pridobljenih iz repozitorija PROMISE [32].

Avtorji članka »Change Bursts as Defect Predictors« so v svoji raziskavi dokazovali, da lahko z zaznavanjem pogostih sprememb opravljenih nad izvorno kodo napovemo, katera komponenta vsebuje napako. Ugotovili so, da je natančnost klasifikacije 91,1%, kar nakazuje, da lahko število pogostih sprememb služi za napovedovanje komponent z napako [33].

5 PRAKTIČNA ANALIZA

V tem poglavju predstavimo tehnologije, ki smo jih uporabili za izvedbo praktičnega dela magistrske naloge, repozitorije iz katerih smo črpali podatke, opišemo implementacijo aplikacije in zasnovo poskusov.

5.1 Pridobivanje podatkov

Za izvedbo praktičnega dela magistrske naloge smo v prvi vrsti potrebovali ustrezne podatke o razredih. Za vsak javanski razred smo potrebovali vrednosti metrik in podatek, ali razred vsebuje napako. Podatke bi sicer lahko pridobili tako, da bi vzeli nek projekt in ga analizirali, vendar smo med pregledovanjem literature zasledili, da se kar nekaj člankov sklicuje na repozitorij PROMISE, zato smo se odločili, da ga uporabimo. Med iskanjem omenjenega repozitorija smo ugotovili, da le ta obstaja pod novim imenom tera-PROMISE in predstavlja nadgradnjo starega repozitorija.

Cilj repozitorija tera-PROMISE je zagotoviti preverljive in ponovljive modele, ki jih je mogoče uporabiti za raziskave na področju računalništva. Omogoča enostavno deljenje pridobljenih podatkov med raziskovalci in strokovnjaki. Vsakdo lahko shrani nov nabor podatkov ali dostopa do že obstoječega modela, ki ga je delil nekdo drug. Na ta način lahko raziskovalci primerjajo svoje rezultate in rezultate drugih ter tako lažje pridejo do zaključkov [34] [35].

V repozitoriju tera-PROMISE bi lahko za napovedovanje napak v izvorni kodi uporabili podatke, pridobljene iz odprtokodnih in akademskih projektov, vendar smo se omejili samo na odprtokodne projekte.

Poskus smo tako lahko izvajali na petinštiridesetih verzijah trinajstih odprtokodnih projektov. V poskus so bili vključeni sledeči projekti [35]:

- Apache Ant je orodje za avtomatizacijo gradnje programske opreme;
- Apache Camel je odprtokodno ogrodje za sporočilno usmerjeno posredniško programsko opremo;
- Apache Forrest je ogrodje, ki pretvori vhodne podatke iz različnih virov v enotno predstavitev v enem ali večih izhodnih formatih;
- Apache Ivy je orodje, ki upravlja odvisnosti med programskimi moduli in se osredotoča na fleksibilnost in preprostost;
- JEdit je urejevalnik besedila, namenjen programerjem, podpira označevanje sintakse in samodejnega zamikanja besedila za več kot 200 programskih jezikov;
- Apache Log4j je knjižnjica za beleženje izpisov dnevnikov na različne lokalne in oddaljene lokacije ter omogoča filtriranje dogodkov v času izvajanja;
- Apache Lucene je orodje, ki zagotavlja tehnologijo indeksiranja in iskanja po besedilih ter omogoča tudi preverjanje črkovanja, poudarjanje zadetkov in napredne analize;
- Apache POI je zbirka knjižnic za branje in pisanje v formatih, ki jih uporabljajo programi paketa Microsoft Office;
- Apache Synapse je enostaven, lahek in visoko zmogljiv komunikacijski vmesnik med povezanimi aplikacijami;
- Apache Tomcat je komponenta spletnega strežnika, ki skrbi za interakcijo s komponentami »Java serverlet«;
- Apache Velocity je orodje za generiranje izgleda spletnih strani in omogoča ločitev Javanske kode iz spletnih strani, kar posledično pomeni lažje vzdrževanje spletnih strani skozi čas in je alternativa tehnologijama Java Server Pages (JSP) in PHP;
- Xalan-Java je orodje, ki je sposobno obdelati in pretvoriti XML dokumente v dokumente tipa HTML, navadno besedilo ali druge tipe dokumentov, ki temeljijo na jeziku XML;
- Xerces je XML parser, ki podpira XML shemo 1.0, DOM 2 in SAX verzijo 2.

V pridobljenem repozitoriju je vsak razred predstavljen s 24 atributi:

- ime projekta, kateremu razred pripada,
- verzija projekta,
- ime razreda (vključno z imenskim prostorom),
- dvajset metrik programske opreme in
- število napak v razredu.

Vrednosti metrik, ki opisujejo posamezen razred, so bile pridobljene s pomočjo orodja Ckjm [35]. Nekaj uporabljenih metrik je že podrobneje opisanih v drugem poglavju, zato te tukaj samo naštejemo:

- WMC,
- DIT,
- NOC,
- CBO,
- RFC,
- LCOM,
- LCOM3 (izpeljanka LCOM),
- št. razredov, ki dostopajo do razreda – Ca (angl. Afferent couplings),
- št. razredov, do katerih ta razred dostopa – Ce (angl. Efferent couplings),
- število javnih metod - NPM (angl. Number of Public Methods),
- LOC,
- metrika dostopnosti podatkov – DAM (angl. Data Access Metric),
- mera združevanja – MOA (angl. Measure of Aggregation),
- mera funkcionalne abstrakcije – MFA (angl. Measure of Functional Abstraction),
- kohezija med metodami – CAM (angl. Cohesion Among Methods of Class),
- sklopljenost dedovanja – IC (angl. Inheritance Coupling),
- sklopljenost med metodami – CBM (angl. Coupling Between Methods),
- povprečna kompleksnost metod – AMC (angl. Average Method Complexity),
- CC (uporabljata se maksimalna vrednost CC med vsemi metodami in pa srednja vrednost).

V tabelah 5.1 in 5.2 je prikazanih več podrobnosti posameznih projektov. Iz repozitorija tera-PROMISE smo tako pridobili podatke za skupno 17129 razredov, od tega jih 5813 vsebuje vsaj eno napako. Izraženo v odstotkih – 33,94% razredov vsebuje napako.

Tabela 5.1: Podrobosti projektov tera-PROMISE – 1. del

Projekt	**Verzija**	**Število razredov**	**Število razredov z napako**	**Število razredov z napako (%)**
Apache Ant	1.3	187	20	10,7
Apache Ant	1.4	265	40	15,09
Apache Ant	1.5	401	32	7,98
Apache Ant	1.6	523	92	17,59
Apache Ant	1.7	745	166	22,28
Apache Camel	1.0	339	13	3,83
Apache Camel	1.2	608	216	35,53
Apache Camel	1.4	872	145	16,63
Apache Camel	1.6	965	188	19,48
Apache Forrest	0.6	6	1	16,67
Apache Forrest	0.7	29	5	17,24
Apache Forrest	0.8	32	2	6,25
Apache Ivy	1.1	111	63	56,76
Apache Ivy	1.4	241	16	6,64
Apache Ivy	2.0	352	40	11,36
Apache Log4j	1.0	135	34	25,19
Apache Log4j	1.1	109	37	33,94
Apache Log4j	1.2	205	189	92,2
Apache Lucene	2.0	195	91	46,67
Apache Lucene	2.2	247	144	58,3
Apache Lucene	2.4	340	203	59,71
Apache POI	1.5	237	141	59,49
Apache POI	2.0	314	37	11,78
Apache POI	2.5	385	248	64,42
Apache POI	3.0	442	281	63,57
Apache Synapse	1.0	157	16	10,19
Apache Synapse	1.1	222	60	27,03
Apache Synapse	1.2	256	86	33,59
Apache Tomcat	1.0	858	77	8,97
Apache Velocity	1.4	196	147	75
Apache Velocity	1.5	214	142	66,36
Apache Velocity	1.6	229	78	34,06

Tabela 5.2: Podrobosti projektov tera-PROMISE – 2. del

Projekt	**Verzija**	**Število razredov**	**Število razredov z napako**	**Število razredov z napako (%)**
JEdit	3.2	272	90	33,09
JEdit	4.0	306	75	24,51
JEdit	4.1	312	79	25,32
JEdit	4.2	367	48	13,08
JEdit	4.3	492	11	2,24
Xalan-Java	2.5	803	387	48,19
Xalan-Java	2.6	885	411	46,44
Xalan-Java	2.4	723	110	15,21
Xalan-Java	2.7	909	898	98,79
Xerces	1.0	162	77	47,53
Xerces	1.2	440	71	16,14
Xerces	1.3	453	69	15,23
Xerces	1.4	588	437	74,32

Ker smo v poskus želeli vključiti tudi metrike sprememb, smo poiskali še drug repozitorij – Bug prediction dataset. Repozitorij je zasnovan tako, da je možno zgraditi modele za napovedovanje napak na nivoju razredov. Repozitorij vsebuje podatke petih programskih sistemov:

- Eclipse JDT Core – jedro integriranega razvojnega okolja za javo,
- Eclipse PDE UI – podaja nabor orodij za razvoj vtičnikov za Eclipse ,
- Equinox Framework – ogrodje, ki omogoča razvijalcem, da razvijejo aplikacijo kot nabor paketov z uporabo skupne infrastrukture storitev,
- Apache Lucene in
- Mylyn – podsistem orodja Eclipse, namenjen za upravljanje opravil.

Za vsak sistem (vsak razred v sistemu) je možno pridobiti [36]:

- vrednosti 15 metrik sprememb, izračunanih iz sistema za upravljanje z izvorno kodo,
- vrednosti 17 metrik programske opreme in
- kategorizirano število napak, ugotovljenih po izidu programske opreme.

Nabor metrik sprememb vključuje:

- število verzij - numberOfVersions,
- število popravkov - numberOfFixes ,
- število sprememb razreda - numberOfRefactorings,
- število avtorjev – numberOfAuthors,
- število dodanih vrstic – linesAdded in druge.

Nabor klasičnih metrik programske opreme se razlikuje od nabora pri repozitoriju tera-PROMISE in vsebuje sledeče metrike:

- CBO,
- DIT,
- fan-in,
- fan-out,
- LCOM,
- NOC,
- število atributov,
- število podedovanih atributov,
- LOC,
- število metod,
- število podedovanih metod,
- število privatnih atributov,
- število privatnih metod,
- število javnih atributov,
- število javnih metod – NPM,
- RFC in
- WMC.

Iz repozitorija smo si prenesli in uporabili podatke samo za eno verzijo posameznega sistema ne pa vseh možnih verzij. V tabeli 5.3 so prikazane podrobnosti pridobljenih

projektov. Skupaj smo iz repozitorija pridobili podatke za 5371 razredov, od katerih jih 853 (15,88%) vsebuje napako.

Tabela 5.3: Podrobnosti projektov iz repozitorija Bug prediction dataset

Projekt	**Število razredov**	**Število razredov z napako**	**Število razredov z napako (%)**
Eclipse JDT Core	997	206	20,66
Eclipse PDE UI	1497	209	13,96
Equinox Framework	324	129	39,81
Apache Lucene	691	64	9,26
Mylyn	1862	245	13,16

Podatki, ki smo jih pridobili iz obeh repozitorijev, so na voljo v formatu CSV (angl. Comma-separated values). Za lažjo uporabo v kombinaciji s knjižnico Weka smo pridobljene datoteke pretvorili v format, ki ga knjižnica najbolje pozna. ARFF (angl. Attribute-Relation File Format). Podatek o številu napak v razredu je bil pretvorjen v dva razreda. Razred označen z »0« je predstavljal razrede brez napak, razred označen z »1« pa razrede, ki vsebujejo vsaj eno napako. Odstranjeni so bili vsi atributi, ki ne doprinesejo k zaznavanju prisotnosti napak v razredu (ime, verzija).

5.2 Uporabljene tehnologije

Za implementacijo aplikacije smo uporabili programskih jezik Java, natančneje verzijo 8. Java je splošno namenski objektno orientiran programski jezik. Razvit je bil z mislijo, da napišemo kodo enkrat in jo poganjamo na vseh platformah (»write once, run anywhere«) [37].

Aplikacijo smo razvijali v orodju IntelliJ IDEA (Community Edition). IntelliJ IDEA je integrirano razvojno okolje za Javo. Razvija ga podjetje JetBrains. Napisan je v Javi. Prosto dostopna različica (Community Edition) je na voljo pod licenco Apache 2, zanjo imamo mogoč vpogled v izvorno kodo na repozitoriju GitHub [38].

Za učenje algoritmov strojnega učenja smo uporabili knjižnico WEKA (Waikato Environment for Knowledge Analysis), verzijo 3.8. WEKA ponuja implementacijo različnih algoritmov strojnega učenja, ki jih lahko apliciramo na različne nabore podatkov. Zraven vsega pa ponuja pester nabor orodij za transformacijo podatkov, kot so algoritmi za diskretizacijo in vzorčenje [19] [39].

WEKA vsebuje metode za glavna opravila podatkovnega rudarjenja: regresijo, klasifikacijo, razvrščanje, rudarjenje pravil in izbiro atributov. Vsi algoritmi na vhodu prejmejo podatke v obliki ene relacijske tabele, pridobljene iz datoteke ali generirane iz poizvedbe na podatkovni bazi [19] [39]. Tabela je navadno shranjena v ARFF datoteki, primer vsebine le te je viden na sliki 5.1, kjer je prikazan del datoteke, ki smo jo uporabili ob izvedbi poskusa. Datoteka je sestavljena iz dveh delov. Prvi del se imenuje glava, ki vsebuje podatek o imenu podatkovne zbirke in seznam vseh atributov ter njihovih tipov. Drugi podatkovni del pa vsebuje vse primerke, ki so na voljo v zbirki [19] [39].

5.3 Definicije poskusov

Za izvedbo poskusov smo najprej določili klasifikatorje. Odločili smo se, da uporabimo:

- C4.5,
- zaporedna minimalna optimizacija – SMO (angl. Sequential Minimal Optimization),
- backpropagation NM in
- naključni gozd (angl. Random Forest).

V knjižnici WEKA je algoritem za grajenje odločitvenih dreves – C4.5 poimenovan J48. Uporabljamo dve konfiguraciji algoritma. V prvi drevo klestimo s pragom zaupanja 0,25. V drugi konfiguraciji pa odločitvenega drevesa ne klestimo.

Algoritem SMO je klasifikator po metodi podpornih vektorjev. Implementacija v knjižnici WEKA zamenja vse manjkajoče vrednosti in pretvori nominalne atribute v binarne. Prav tako izvede normalizacijo vseh atributov.

```
@RELATION ant-1.3

@ATTRIBUTE wmc     NUMERIC
@ATTRIBUTE dit     NUMERIC
@ATTRIBUTE noc     NUMERIC
@ATTRIBUTE cbo     NUMERIC
@ATTRIBUTE rfc     NUMERIC
@ATTRIBUTE lcom    NUMERIC
@ATTRIBUTE ca      NUMERIC
@ATTRIBUTE ce      NUMERIC
@ATTRIBUTE npm     NUMERIC
@ATTRIBUTE lcom3   NUMERIC
@ATTRIBUTE loc     NUMERIC
@ATTRIBUTE dam     NUMERIC
@ATTRIBUTE moa     NUMERIC
@ATTRIBUTE mfa     NUMERIC
@ATTRIBUTE cam     NUMERIC
@ATTRIBUTE ic      NUMERIC
@ATTRIBUTE cbm     NUMERIC
@ATTRIBUTE amc     NUMERIC
@ATTRIBUTE max_cc  NUMERIC
@ATTRIBUTE avg_cc  NUMERIC
% 0 - razred ne vsebuje napake
% 1 - razred vsebuje napako
@ATTRIBUTE bug     {0, 1}

@DATA
6,1,0,4,20,1,1,4,6,0.76,161,1.0,1,0.0,0.33333334,0,0,25.0,9,2.6666667,0
10,1,0,9,49,1,0,9,8,0.9206349,382,1.0,0,0.0,0.8333333,0,0,35.1,4,1.9,0
2,1,0,2,4,0,1,2,1,0.5,20,1.0,1,0.0,0.625,0,0,8.0,1,0.5,0
…
```

Slika 5.1: Primer ARFF datoteke

Tretji uporabljen klasifikator najdemo pod imenom MultilayerPerceptron – MLP, ki uporablja algoritem backpropagation za klasifikacijo primerkov. Uporabili smo tri različne konfiguracije. Podrobnosti konfiguracij predstavlja tabela 5.4. V zadnjem stolpcu je zapisano koliko vozlišč je na vsakem skritem nivoju. V prvi vrstici predstavlja *a* vrednost, ki jo izračunamo kot $(\check{s}tevilo\_atributov + \check{s}tevilo\_razredov)/2$.

Tabela 5.4: Konfiguracije klasifikatorja MLP

Konfiguracija	**Hitrost učenja**	**Število epoh**	**Število skritih nivojev**	**Število vozlišč**
1	0,3	500	1	*a*
2	0,05	1000	2	10, 2
3	0,1	3000	4	20, 15, 10, 5

Kot zadnji klasifikator pa uporabljamo metodo naključnih gozdov. Vsako drevo je zgrajeno iz vseh elementov v učni množici. Klasifikator ima vgrajeno možnost, da se izvaja na več procesorskih jedrih.

V sklopu magistrske naloge smo se na osnovi pridobljenih podatkov in pregledane literature odločili za izvedbo petih poskusov. Vsak poskus je imel v osnovi sledeče korake:

- za vsak klasifikator,
    - ponovi 30 krat,
        - izvedi učenje klasifikatorja,
        - izvedi validacijo,
        - shrani rezultate.

V prvem poskusu, označili smo ga s **P1**, smo preverjali natančnost klasifikacije na podlagi primerkov iz repozitorija tera-PROMISE. Spomnimo, da je v tem repozitoriju vsak primerek razreda opisan z 20 metrikami programske opreme, en atribut pa predstavlja prisotnost napake v razredu. Za validacijo naučenega klasifikatorja smo uporabili 10-kratno križno validacijo (angl. 10-fold cross validation), prav tako smo za učenje klasifikatorja iz celotne množice 17129 primerkov naključno izbrali 1000 primerkov.

Drugi poskus, označen s **P2**, je v osnovi enak kot prvi. Razlika je le v tem, da smo tukaj merili natančnost klasifikacije na repozitoriju Bug prediction dataset, in sicer samo za metrike programske opreme. Razlika v obeh repozitorijih je v naboru metrik in razmerju med pozitivnimi ter negativnimi vzorci. Vsak razred je tako opisan s sedemnajstimi različnimi metrikami programske opreme.

V tretjem poskusu, označenem s **P3**, kjer smo merili natančnost klasifikacije, če uporabljamo za napovedovanje prisotnosti napak samo metrike sprememb (uporabljamo repozitorij Bug prediction dataset). Način validacije je enak kot v **P1** in **P2**. Vsak razred je opisan s petnajstimi metrikami sprememb.

V četrtem poskusu, označenem s **P4**, smo merili natančnost klasifikacije z združenimi metrikami programske opreme in metrikami sprememb, vsak razred je tako opisovalo 15 metrik sprememb in 17 metrik programske opreme.

V tabeli 5.5 podajamo natančnejše specifikacije prvih štirih poskusov. V predzadnjem stolpcu je predstavljeno število vseh primerkov v učni množici, v zadnjem pa koliko naključnih primerkov je bilo uporabljenih za učenje. Za izbiro 1000 naključnih primerkov smo se odločili iz dveh razlogov. Prvi razlog je bil čas učenja. Pri učenju klasifikatorjev z vsemi primerki v učni množici (v primeru repozitorija tera-PROMISE) je največ časa za učenje potreboval klasifikator MLP (tretja konfiguracija), in sicer približno eno uro in 44 minut. Ko smo klasifikatorje učili na 1000 naključnih primerkih, pa se je čas učenja skrajšal na sedem minut in pol. Drugi, pomembnejši, razlog pa je bil ta, da se je natančnost klasifikacije spremenila minimalno. Povprečna natančnost klasifikacije vseh klasifikatorjev in učenja nad celotno množico je znašala 63,9%. Pri učenju s 1000 naključnimi primerki pa je bila povprečna natančnost 63,2%.

Tabela 5.5: Podrobnosti poskusov P1 - P4

Oznaka	**Repozitorij**	**Tip metrik**	**Validacija**	**Vseh prim.**	**Izbranih prim.**
P1	tera-PROMISE	klasične	10 – kratna križna	17129	1000
P2	Bug prediction dataset	klasične	10 – kratna križna	5371	1000
P3	Bug prediction dataset	metrike sprememb	10 – kratna križna	5371	1000
P4	Bug prediction dataset	klasične + metrike sprememb	10 – kratna križna	5371	1000

V zadnjem poskusu, označenem s **P5**, pa smo preverjali natančnost klasifikacije na zaporednih verzijah projekta. Na primer, če je imel projekt štiri verzije, smo posamezni klasifikator najprej učili na verziji ena in preverjali natančnost klasifikacije na verziji dve. Nato smo klasifikator učili na verziji ena in dve ter preverili natančnost na verziji tri. Na koncu smo še učili klasifikator na verzijah ena, dve in tri ter preverili natančnost na verziji štiri.

5.4 Opis implementacije

V podpoglavju 5.1 smo zapisali, da smo vse podatke na začetku pretvorili iz formata CSV v format ARFF. Pretvorbe datotek nismo implementirali v Javi, ampak smo v ta namen spisali nekaj preprostih PowerShell skript. Preneseni podatki so bili urejeni hierarhično v mapah na disku. Na prvem nivoju sta dve mapi (ena za vsak repozitorij). V teh dveh mapah so se nahajale podmape, poimenovane po imenu projekta. Vsaka mapa s projektom pa je vsebovala CSV datoteke s podatki za različne verzije programa. PowerShell skripte so zraven pretvorbe poskrbele, da so se v ARFF datoteko shranili samo atributi, ki smo jih potrebovali za klasifikacijo, prav tako pa so pretvorile podatek o številu napak iz vrednosti med 0 in *n*, kjer *n* predstavlja število napak v razredu, v vrednosti 0 in 1. Vrednost 0 je bila v primeru, ko je bilo število napak 0, sicer smo v datoteko shranili vrednost 1.

V poskusih uporabljamo dva načina validacije, prvi je 10–kratna križna validacija, ki jo uporabljamo v poskusih od P1 do P4. Postopek teh poskusov v Javi z uporabo knjižnice WEKA izvajamo na način, kot je prikazano na sliki 5.2. Na začetku vsakega poskusa smo izmed vseh možnih vzorcev izbrali 1000 naključnih na ta način:

- iz celotne množice odstrani podvojene vrednosti;
- izračunaj razmerje med pozitivnimi in negativnimi vzorci in
- izberi 1000 naključnih vzorcev na način, da ohranimo izvorno razmerje pozitivnih in negativnih vzorcev.

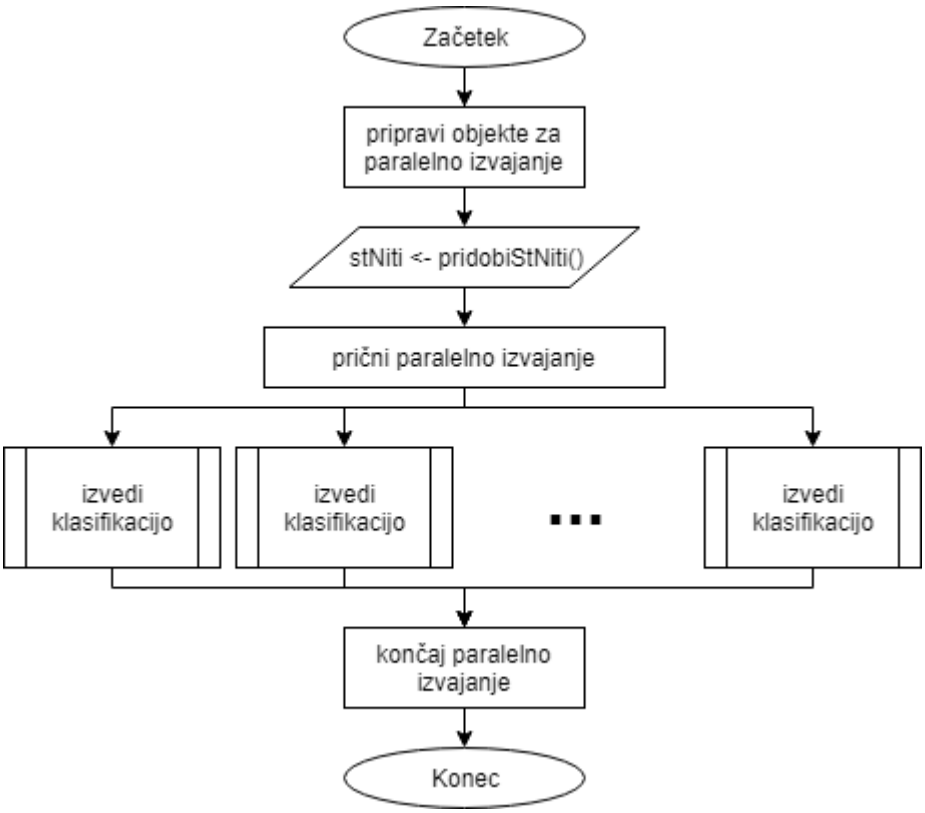
Poskus P5 pa izvedemo tako, da najprej preberemo vse projekte in njihove verzije. Nato za vsak projekt v učno množico dajemo podatke za prejšnje verzije projekta. Trenutno verzijo projekta pa uporabimo za ocenjevanje natančnosti. S programerskega vidika se ocenjevanje natančnosti veliko ne spremeni. V izvorni kodi, prikazani na sliki 5.2, lahko kodo iz zadnje notranje zanke prestavimo izven zanke, saj ne izvajamo več 10–kratne križne validacije, podatke zamenjamo za testno in učno množico, kot je opisano na začetku tega odstavka.

```
// preberi ARFF datoteko in izberi 1000 naključnih vzorcev
Instances data = FileUtils.takeRandomSamples(...);
// nastavi indeks razreda (za kateri razred napovedujemo vrednosti)
data.setClassIndex(data.numAttributes() - 1);
// da lahko ponovimo eksperiment uporabljamo vedno isto seme
// v generatorju naključnih števil
int currentSeed = baseSeed;
// vsak poskus ponovimo 30 krat
for (int i = 0; i < 30; i++) {
    ++currentSeed;
    Random rand = new Random(currentSeed);
    Instances randData = new Instances(data);
    // premešamo podatke
    randData.randomize(rand);
    // izvedemo stratifikacijo
    randData.stratify(10);

    // za vsak klasifikator izvedi
    for (Classifier cls : Classifiers.getClassifiers()) {
        // ustvarimo objekt za validacijo modelov strojnega učenja
        Evaluation eval = new Evaluation(randData);
        // izvedi 10-kratno križno validacijo
        for (int n = 0; n < 10; n++) {
            // razdeli naključnih 1000 vzorcev v učno
            Instances train = randData.trainCV(10, n);
            // in testno množico
            Instances test = randData.testCV(10, n);

            Classifier clsCopy = AbstractClassifier.makeCopy(cls);
            // nauči klasifikator
            clsCopy.buildClassifier(train);
            // oceni naučen klasifikator
            eval.evaluateModel(clsCopy, test);
        }
        // shrani rezultate v datoteko
        FileUtils.writeToFile(eval);
    }
}
```

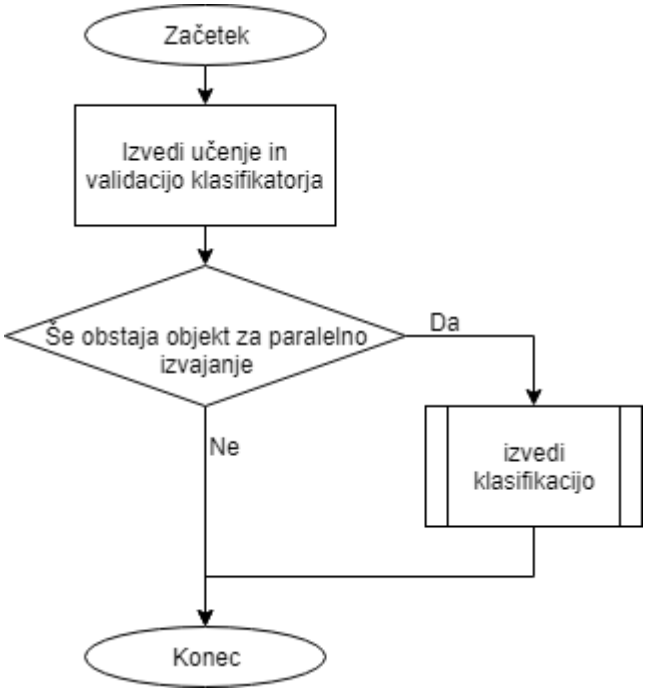
Slika 5.2: Koda izvedbe poskusov P1 – P4 v Javi

Da bi pospešili izvajanje poskusov, smo vsako klasifikacijo »zapakirali« v objekte, ki implementirajo vmesnik Runnable, kar nam omogoča, da eno ponovitev klasifikacije izvedemo v svoji niti. Postopek izvajanja klasifikacije na več nitih opisuje diagram poteka na sliki 5.3. Na začetku vedno pripravimo objekte za paralelno izvajanje. Nato pridobimo podatek, koliko niti lahko hkrati izvajamo na procesorju, in ustvarimo toliko niti. Znotraj niti izvedemo postopek, prikazan na sliki 5.4, kjer najprej izvedemo učenje in validacijo klasifikatorja (postopek je prikazan na sliki 5.2). Nato preverimo, če še obstaja objekt za paralelno izvajanje. Če obstaja, ustvarimo novo nit in ponovimo korake na sliki 5.4. Če bi izvajali učenje in validacijo vsakega klasifikatorja zaporedno in postopek ponovili 30-krat,

bi za to porabili 5 ur 26 minut in 43 sekund. Z vzporednim izvajanjem pa smo čas skrajšali na 1 uro 52 minut in 15 sekund, kar je približno 2,9-krat hitreje. Poskuse smo izvajali na računalniku s procesorjem Intel i7-6700HQ, ki ima osnovno frekvenco 2,6GHz, in omogoča izvajanje z osmimi nitmi naenkrat. Vendar smo izvajanje poskusov omejili tako, da hkrati izvajamo samo štiri niti.



Slika 5.3: Diagram poteka priprave vzporednega izvajanja



Slika 5.4: Diagram poteka izvajanja posamezne niti

6 REZULTATI

V magistrski nalogi smo preverjali natančnost določanja prisotnosti napake v izvorni kodi različnih konfiguracij klasifikatorjev v več situacijah (samo klasične metrike programske opreme, metrike sprememb, klasične metrike in metrike sprememb ter klasične metrike nad zaporednimi verzijami), podrobneje opisanih v prejšnjem poglavju.

V vseh poskusih smo vsako konfiguracijo pognali (učili in merili natančnost klasifikacije) 30 krat. Za vsako meritev smo hranili naslednje podatke: naziv klasifikatorja, zaporedno številko meritve, začetno seme v generatorju naključnih števil, število pravilno napovedanih razredov z napako, število napačno napovedanih razredov z napako, število pravilno napovedanih razredov brez napake, število napačno napovedanih razredov brez napake, skupno število pravilno napovedanih rezultatov, skupno število napačno napovedanih rezultatov, natančnost klasifikacije in čas izvajanja ene meritve. Na sliki 6.1 je prikazan primer začetka shranjene datoteke s podatki. Zaradi velike količine podatkov in lažje berljivosti podajamo samo izbrane povprečne vrednosti tridesetih meritev.

	A	B	C	D	E	F	G	H	I	J	K
1	Classifier	Meritev	Seed	TP	FP	TN	FN	Pravilno	Napacno	Natancnost	Cas izvajanja (s)
2	weka.classifiers.trees.J48 -C 0.25 -M 2	1	5	187	177	441	195	628	372	0,628	0,687606815
3	weka.classifiers.trees.RandomForest -	23	27	150	127	491	232	641	359	0,641	3,519371092
4	weka.classifiers.functions.MultilayerP	25	29	123	95	523	259	646	354	0,646	18,91153606
5	weka.classifiers.functions.SMO -C 1.0	3	7	32	25	593	350	625	375	0,625	0,701136516
6	weka.classifiers.functions.SMO -C 1.0	22	26	32	21	597	350	629	371	0,629	0,324840809
7	weka.classifiers.functions.SMO -C 1.0	23	27	33	21	597	349	630	370	0,63	0,300902402
8	weka.classifiers.trees.RandomForest -	25	29	159	116	502	223	661	339	0,661	2,990595541

Slika 6.1: Primer datoteke s podatki o poskusu

Vrednosti v tabelah so urejene glede na povprečno natančnost klasifikacije. Vse oznake, uporabljene v tabelah tega poglavja, so podane v tabeli 6.1.

Tabela 6.1: Legenda uporabljenih oznak

Oznaka	**Pomen**
C4.5-1	Prva konfiguracija odločitvenega drevesa C4.5 (s klestenjem)
C4.5-2	Druga konfiguracija odločitvenega drevesa C4.5 (brez klestenja)
SMO	Klasifikator SMO
RF	Klasifikator naključni gozd
MLP1	Prva konfiguracija klasifikatorja MLP
MLP2	Druga konfiguracija klasifikatorja MLP
MLP3	Tretja konfiguracija klasifikatorja MLP
TP	Povprečna vrednost pravilno napovedanih razredov z napako
FP	Povprečna vrednost napačno napovedanih razredov z napako
TN	Povprečna vrednost pravilno napovedanih razredov brez napake
FN	Povprečna vrednost napačno napovedanih razredov brez napake
NAT	Povprečna natančnost tridesetih meritev (v odstotkih)
STDDEV	Standardni odklon natančnosti
ČAS	Povprečen čas izvajanja ene ponovitve meritve v sekundah

6.1 Poskus P1

V poskusu P1 smo preverjali natančnost podanih konfiguracij klasifikatorjev nad podatki v repozitoriju tera-PROMISE. Razmerje med negativnimi in pozitivnimi primerki je 66:34 (66% primerkov ne vsebuje napake, 34% jo). Rezultati poskusa so pokazali, da se je najbolje odrezal klasifikator MLP2, povprečna natančnost je bila 64,98%. Najslabše se je odrezal algoritem C4.5-2. Bolj problematičen je klasifikator MLP3, ki nikoli ne zazna napake v razredu. To se zgodi zaradi premajhne količine podatkov. Podrobnosti podajamo v tabeli 6.2. Na grafu 6.1 še prikažemo primerjavo najmanjše in največje natančnosti klasifikatorjev s povprečno natančnostjo.

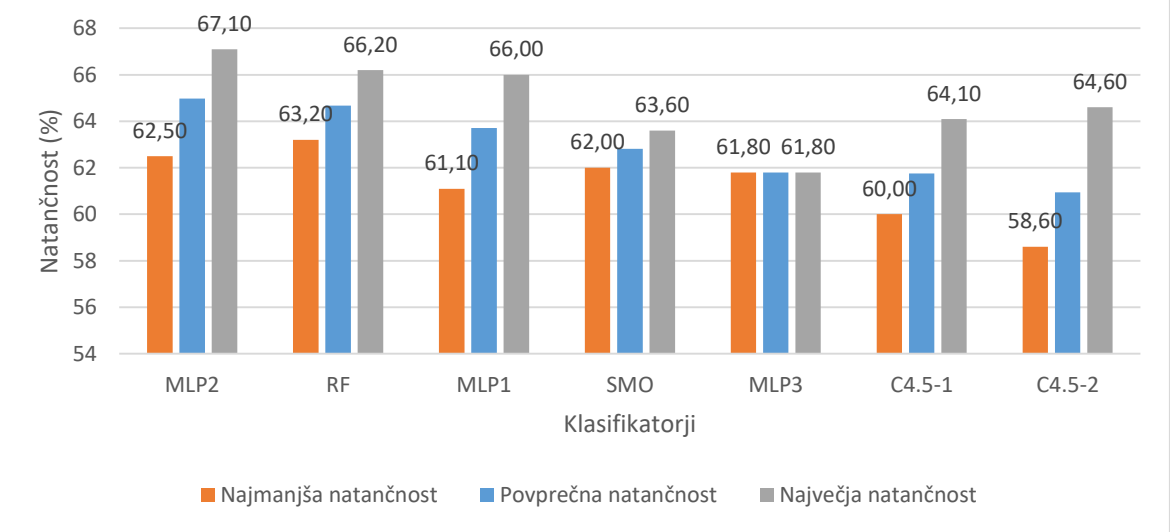
6.2 Poskus P2

Pri poskusu P2 smo preverjali natančnost klasifikacije za isti tip problema. Imamo razred, opisan z metrikami programske opreme, in želimo določiti, ali vsebuje napako. Razlika s prejšnjim poskusom je v tem, da uporabljamo podatke iz drugega repozitorija (Bug prediction dataset) in vsak primerek je opisan z drugačnim naborom metrik programske

opreme. Med vsemi primerki je 84,12% takšnih, ki ne vsebujejo napake, in 15,88% takšnih, ki vsebujejo napako.

Tabela 6.2: Rezultati poskusa P1

Klasifikator	TP	FP	TN	FN	NAT(%)	STDDEV	ČAS(s)
MLP2	151,30	119,53	498,47	230,70	64,98	1,01	50,19
RF	153,83	125,10	492,90	228,17	64,67	0,76	2,78
MLP1	135,63	116,57	501,43	246,37	63,71	1,19	21,09
SMO	33,37	23,23	594,77	348,63	62,81	0,31	0,32
MLP3	0,00	0,00	618,00	382,00	61,80	0,00	578,72
C4.5-1	186,07	186,57	431,43	195,93	61,75	1,06	0,19
C4.5-2	191,40	199,93	418,07	190,60	60,95	1,56	0,15



Graf 6.1: Natančnosti klasifikatorjev v poskusu P1

Ponovno se je za najboljši klasifikator izkazal MLP2 s povprečno natančnostjo 84,3%. Prav tako pa je klasifikator MLP3 proglasil vsak testni primerek kot negativen (ne vsebuje napake). Opazimo lahko, da je bilo odstopanje v meritvah manjše kot pri poskusu P1. Podrobnosti meritev so prikazane v tabeli 6.3, primerjava povprečne natančnosti z najmanjšo in največjo pa je prikazana na grafu 6.2.

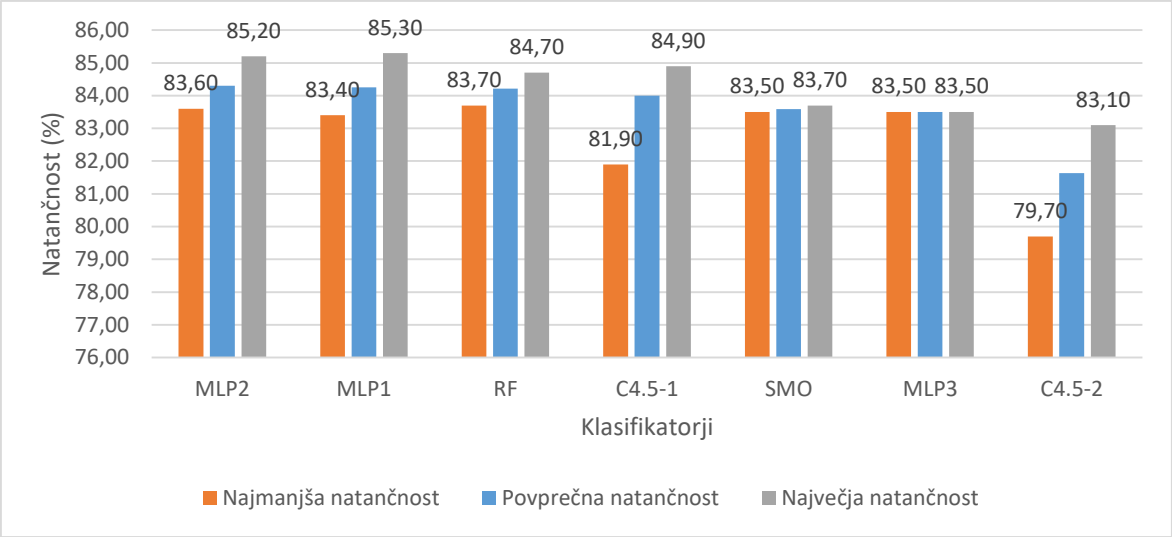
Tabela 6.3: Rezultati poskusa P2

Klasifikator	TP	FP	TN	FN	NAT(%)	STDDEV	ČAS(s)
MLP2	26,33	18,30	816,70	138,67	84,30	0,38	35,86
MLP1	25,03	17,53	817,47	139,97	84,25	0,42	14,55
RF	30,87	23,73	811,27	134,13	84,21	0,29	0,79
C4.5-1	46,47	41,50	793,50	118,53	84,00	0,76	0,15
SMO	1,90	1,00	834,00	163,10	83,59	0,05	0,21
MLP3	0,00	0,00	835,00	165,00	83,50	0,00	501,02
C4.5-2	51,27	69,90	765,10	113,73	81,64	1,07	0,11

6.3 Poskus P3

V poskusu smo preverjali natančnost klasifikacije, ko imamo razred opisan z metrikami sprememb. Uporabljen je bil enak repozitorij kot v poskusu P2, zato je razmerje med pozitivnimi in negativnimi vzorci enako.

Tudi tukaj se je za najboljši klasifikator izkazal MLP2 s povprečno natančnostjo 84,6%. Opazimo lahko, da se je praktično podvojilo število pravilno napovedanih pozitivnih primerkov v primerjavi s poskusom P2. Kot v prejšnjih dveh poskusih je tudi v tem klasifikator MLP3 vse primerke označil, da ne vsebujejo napake. Podrobnosti meritev poskusa P3 so prikazane v tabeli 6.4 in na grafu 6.3.



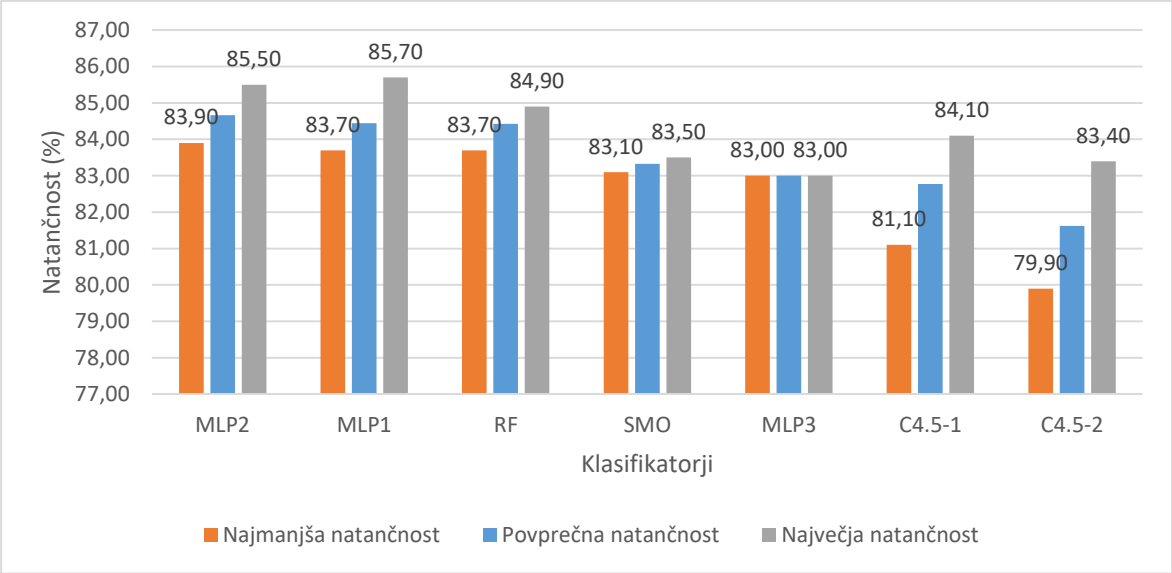
Graf 6.2: Primerjava natančnosti klasifikatorjev v poskusu P2

Tabela 6.4: Rezultati poskusa P3

Klasifikator	TP	FP	TN	FN	NAT(%)	STDDEV	ČAS(s)
MLP2	50,00	33,37	796,63	120,00	84,66	0,39	32,26
MLP1	46,03	31,60	798,40	123,97	84,44	0,50	11,68
RF	44,93	30,63	799,37	125,07	84,43	0,28	0,73
SMO	3,67	0,40	829,60	166,33	83,33	0,12	0,19
MLP3	0,00	0,00	830,00	170,00	83,00	0,00	2345,16
C4.5-1	46,77	49,03	780,97	123,23	82,77	0,68	0,12
C4.5-2	50,23	64,00	766,00	119,77	81,62	1,12	0,09

6.4 Poskus P4

Razred lahko opišemo s kombinacijo metrik programske opreme in metrik sprememb. Kot v poskusih P2 in P3 tudi tukaj uporabljamo podatke iz repozitorija Bug prediction dataset. V naboru vseh podatkov je tako 84,12% takšnih, ki vsebujejo napako, in 15,88%, ki je ne.



Graf 6.3: Primerjava natančnosti klasifikatorjev v poskusu P3

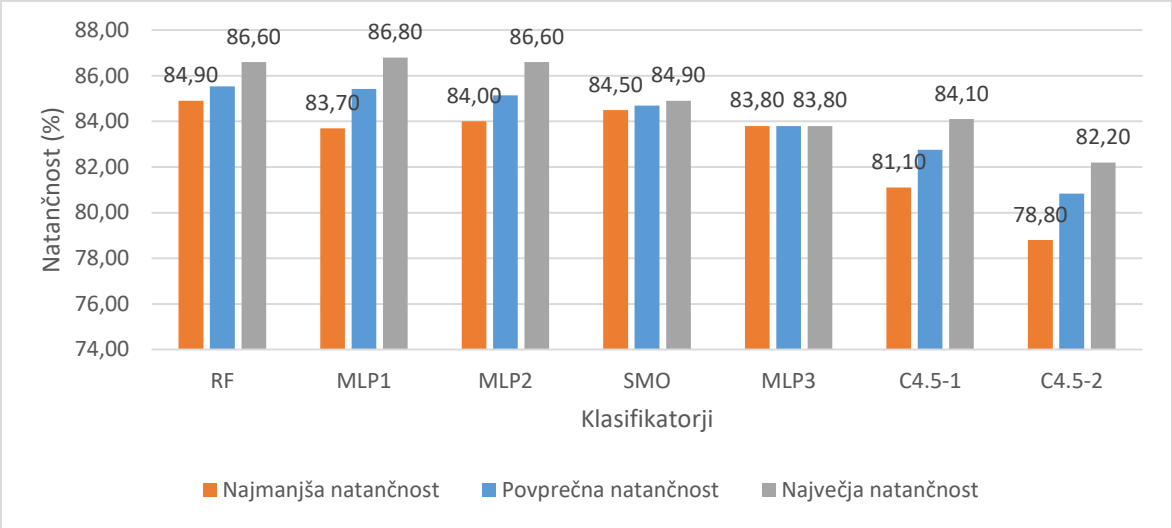
V tem poskusu se je najbolje odrezal klasifikator RF, povprečna natančnost je bila 85,54%. Opazimo lahko, da se je povprečna natančnost klasifikacije v primerjavi s poskusoma P2 in P3 izboljšala. Klasifikator MLP3 ponovno ni znal razpoznati razreda z napako. Podrobni rezultati poskusa P4 so prikazani v tabeli 6.5 in na grafu 6.4.

Tabela 6.5: Rezultati poskusa P4

Klasifikator	**TP**	**FP**	**TN**	**FN**	**NAT(%)**	**STDDEV**	**ČAS(s)**
RF	39,87	22,47	815,53	122,13	85,54	0,37	0,95
MLP1	51,67	35,37	802,63	110,33	85,43	0,82	43,50
MLP2	51,63	38,23	799,77	110,37	85,14	0,70	50,40
SMO	9,97	0,97	837,03	152,03	84,70	0,09	0,31
MLP3	0,00	0,00	838,00	162,00	83,80	0,00	615,42
C4.5-1	53,73	64,13	773,87	108,27	82,76	0,78	0,29
C4.5-2	61,27	90,87	747,13	100,73	80,84	1,74	0,26

6.5 Poskus P5

Ta poskus se od prejšnjih razlikuje po tem, da merimo natančnost klasifikacije na zaporednih verzijah programske opreme. Poskus smo lahko izvedli na osnovi podatkov, pridobljenih iz repozitorija tera-PROMISE.



Graf 6.4: Primerjava natančnosti klasifikatorjev v poskusu P4

Če ima projekt tri verzije, jih označimo z 1.1, 1.2 in 1.3. Potem smo vsak klasifikator v prvi fazi učili na podatkih verzije 1.1, njegovo natančnost pa testirali na verziji 1.2. Postopek smo ponovili 30-krat in uporabili povprečno natančnost. V drugi fazi smo klasifikatorje učili na podatkih verzij 1.1 in 1.2, natančnost pa testirali na verziji 1.3. Če je bilo verzij več, smo seveda postopek ponovili večkrat. Za učenje smo uporabili vse možne primerke.

Prvi projekt, na katerem smo merili natančnost klasifikacije, je bil Apache Ant. Na voljo smo imeli podatke petih zaporednih verzij projekta, in sicer 1.3, 1.4, 1.5, 1.6 in 1.7. Podrobnosti podatkov, s katerimi smo razpolagali, so prikazane v tabeli 6.6. V stolpcu »učenje« so podane verzije projekta, na katerih smo učili klasifikator (učna množica). V stolpcu »testiranje« se nahaja verzija, s katero smo testirali natančnost klasifikacije. Stolpca »brez napak« in »z napako« vsebujeta podatke o tem, koliko razredov v učni množici (ne)vsebuje napake, zadnja dva stolpca pa predstavljata to vrednost v odstotkih.

Tabela 6.6: Podatki za projekt Apache Ant

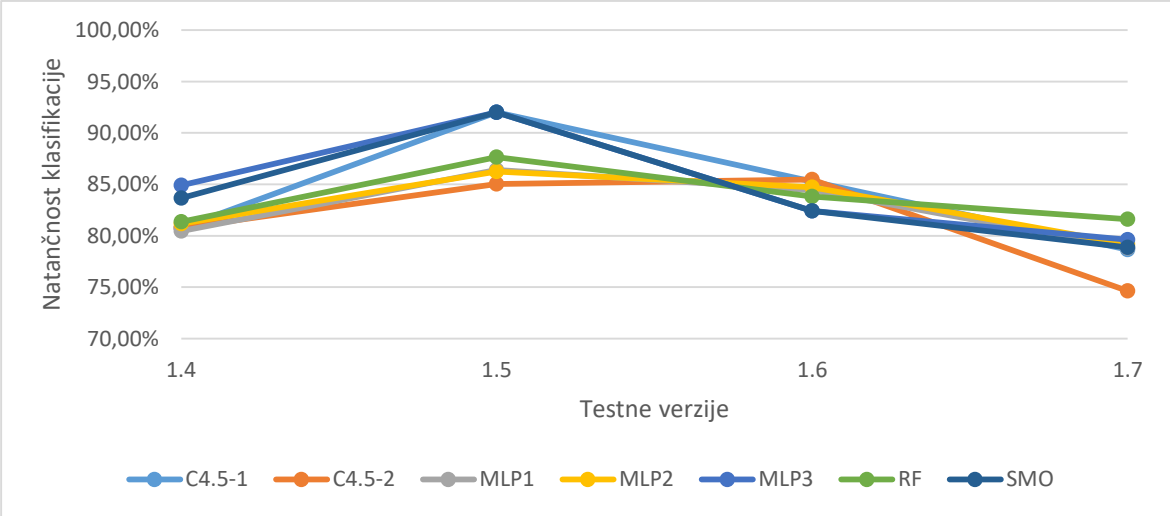
Učenje	Testiranje	Brez napak	Z napako	Brez napak(%)	Z napako(%)
1.3	1.4	160	20	88,89%	11,11%
1.3, 1.4	1.5	308	59	83,92%	16,08%
1.3, 1.4, 1.5	1.6	586	91	86,56%	13,44%
1.3, 1.4, 1.5, 1.6	1.7	873	183	82,67%	17,33%

Podrobnosti rezultatov so prikazane v tabeli 6.7. Imena stolpcev v prvi vrstici povejo verzijo projekta, na kateri smo testirali natančnost klasifikatorja. Prikazane vrednosti so povprečje tridesetih meritev. Takšno notacijo uporabljamo v tabelah, ki sledijo.

Ob pogledu na podatke lahko opazimo, da ne moremo preprosto določiti najboljšega klasifikatorja, saj so se na različnih testnih verzijah projekta menjavali. Ugotovimo lahko, da je povprečno gledano primerke najbolje klasificiral klasifikator MLP3, sledi mu SMO in na tretjem mestu je C4.5-1. Do največjih odstopanj tako pride pri verziji 1.5, ko kar trije klasifikatorji presežejo natančnost 92%, ostali pa se gibljejo okoli 86%. Natančnejši pogled na spreminjanje natančnosti klasifikacije ponuja graf 6.5.

Tabela 6.7: Rezultati meritev poskusa P5 za projekt Apache Ant

Klasifikator	**1.4**	**1.5**	**1.6**	**1.7**
C4.5-1	80,75%	92,02%	85,28%	78,66%
C4.5-2	80,75%	85,04%	85,47%	74,63%
MLP1	80,44%	86,42%	84,45%	78,81%
MLP2	81,18%	86,25%	84,72%	79,17%
MLP3	84,91%	92,02%	82,41%	79,63%
RF	81,36%	87,66%	83,83%	81,60%
SMO	83,66%	92,02%	82,41%	78,88%



Graf 6.5: Natančnosti klasifikacije za projekt Apache Ant

Naslednji projekt, je bil Apache Camel, za katerega smo imeli na voljo podatke o štirih verzijah, in sicer 1.0, 1.2, 1.4 in 1.6. Podrobnosti podatkov podajamo v tabeli 6.8.

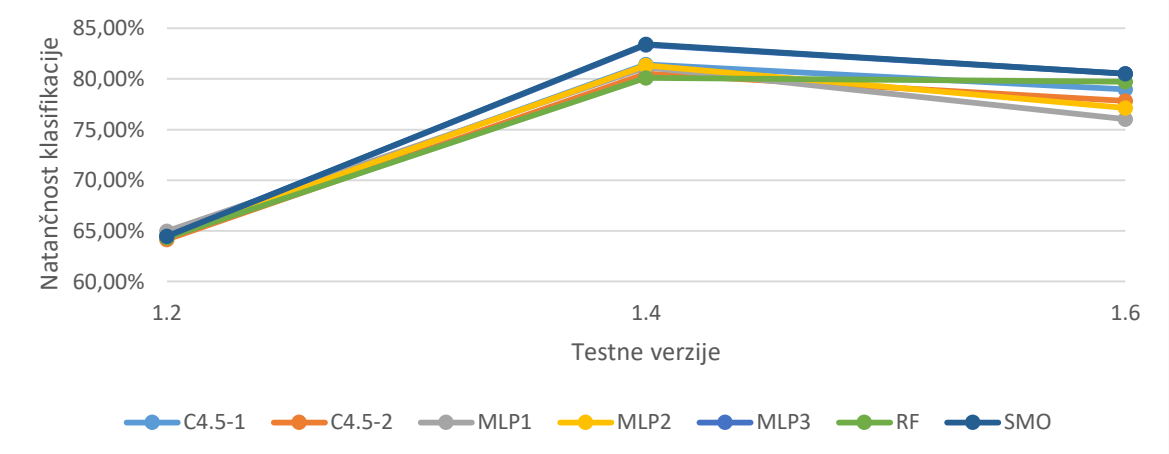
Tabela 6.8: Podatki za projekt Apache Camel

Učenje	Testiranje	Brez napak	Z napako	Brez napak(%)	Z napako(%)
1.0	1.2	314	13	96,02%	3,98%
1.0, 1.2	1.4	611	222	73,35%	26,65%
1.0, 1.2, 1.4	1.6	1141	356	76,22%	23,78%

Rezultate P5 podajamo v tabeli 6.9. Opazimo lahko, da so algoritmi podobno slabo klasificirali na verziji 1.2, ko je bila v učni množici velika večina primerkov brez napake, natančnost pa je bila testirana na verziji, ki je imela veliko napak. Najbolje sta se odrezala klasifikatorja MLP3 in SMO. Spreminjanje natančnosti klasifikacije je prikazano na grafu 6.6.

Tabela 6.9: Rezultati meritev poskusa P5 za projekt Apache Camel

Klasifikator	1.2	1.4	1.6
C4.5-1	64,47%	81,42%	78,96%
C4.5-2	64,14%	80,50%	77,82%
MLP1	64,98%	81,14%	76,03%
MLP2	64,47%	81,35%	77,14%
MLP3	64,47%	83,37%	80,52%
RF	64,35%	80,09%	79,71%
SMO	64,47%	83,41%	80,53%



Graf 6.6: Natančnosti klasifikacije za projekt Apache Camel

Za projekt Apache Forrest smo imeli na razpolago podatke o treh verzijah, 0.6, 0.7 in 0.8. Podrobnosti se nahajajo v tabeli 6.10. Opazimo lahko, da gre za majhen projekt v zgodnji fazi razvoja.

Tabela 6.10: Podatki za projekt Apache Forrest

Učenje	**Testiranje**	**Brez napak**	**Z napako**	**Brez napak(%)**	**Z napako(%)**
0.6	0.7	5	1	83,33%	16,67%
0.6, 0.7	0.8	27	6	81,82%	18,18%

Rezultate P5 podajamo v tabeli 6.11, kjer lahko opazimo, da so klasifikatorji, kljub majhnemu številu primerkov v učnih množicah, natančno napovedovali (ne)prisotnost napake. Najbolje so se odrezali algoritmi MLP3, C4.5-1 in C4.5-2.

Tabela 6.11: Rezultati meritev poskusa P5 za projekt Apache Forrest

Klasifikator	**0.7**	**0.8**
C4.5-1	82,76%	93,75%
C4.5-2	82,76%	93,75%
MLP1	62,07%	84,38%
MLP2	82,76%	91,25%
MLP3	82,76%	93,75%
RF	73,33%	90,31%
SMO	65,52%	93,75%

Tudi za projekt Apache Ivy smo imeli na razpolago podatke o treh verzijah, 1.1, 1.4 in 2.0. Za te verzije podajamo podrobnosti v tabeli 6.12. Tudi tukaj opazimo, da gre za relativno majhen projekt.

Tabela 6.12: Podatki za projekt Apache Ivy

Učenje	**Testiranje**	**Brez napak**	**Z napako**	**Brez napak(%)**	**Z napako(%)**
1.1	1.4	46	63	42,20%	57,80%
1.1, 1.4	2.0	257	79	76,49%	23,51%

V rezultatih, podanih v tabeli 6.13, lahko opazimo, da so klasifikatorji relativno slabo napovedovali prisotnost napake na verziji 1.4. Najbolj bode v oči podatek, da je klasifikator MLP3 klasificiral podatke z zgolj 6,64% natančnostjo. So pa klasifikatorji precej bolje napovedali prisotnost napake na verziji 2.0.

Tabela 6.13: Rezultati meritev poskusa P5 za projekt Apache Ivy

Klasifikator	**1.4**	**2.0**
C4.5-1	52,28%	83,24%
C4.5-2	56,02%	80,68%
MLP1	58,06%	82,52%
MLP2	60,07%	83,58%
MLP3	6,64%	88,64%
RF	52,90%	86,98%
SMO	54,07%	88,35%

Za naslednji projekt, jEdit, smo imeli na razpolago podatke za pet verzij, in sicer 3.2, 4.0, 4.1, 4.2 in 4.3. Podrobnosti podajamo v tabeli 6.14.

Tabela 6.14: Podatki za projekt jEdit

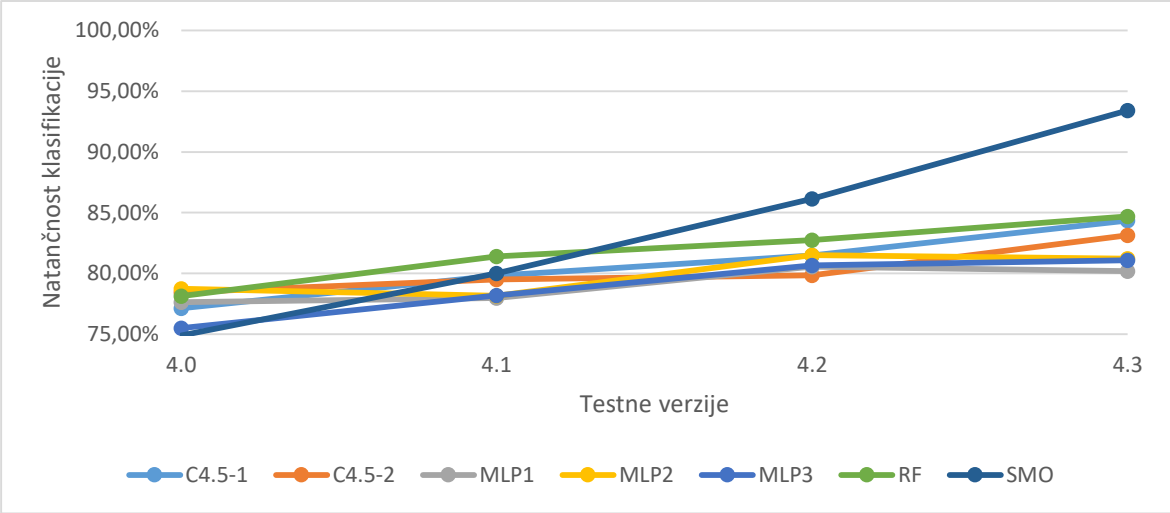
Učenje	**Testiranje**	**Brez napak**	**Z napako**	**Brez napak(%)**	**Z napako(%)**
3.2	4.0	178	90	66,42%	33,58%
3.2, 4.0	4.1	345	165	67,65%	32,35%
3.2, 4.0, 4.1	4.2	497	243	67,16%	32,84%
3.2, 4.0, 4.1, 4.2	4.3	752	291	72,10%	27,90%

V rezultatih, podanih v tabeli 6.15, lahko opazimo, da se je natančnost klasifikacije skozi testne verzije lepo stopnjevala, to je razvidno tudi na grafu 6.7. Daleč največjo natančnost klasifikacije je bilo moč opaziti pri klasifikatorju SMO, in sicer 93,41%, sledila sta mu klasifikatorja RM in C4.5-1.

Za projekt Apache Log4j smo imeli na razpolago podatke za tri verzije, in sicer 1.0, 1.1 in 1.2. Podrobnosti projekta so podane v tabeli 6.16.

Tabela 6.15: Rezultati meritev poskusa P5 za projekt jEdit

Klasifikator	4.0	4.1	4.2	4.3
C4.5-1	77,12%	79,81%	81,47%	84,35%
C4.5-2	78,43%	79,49%	79,84%	83,13%
MLP1	77,61%	77,97%	80,55%	80,19%
MLP2	78,73%	78,16%	81,49%	81,21%
MLP3	75,49%	78,17%	80,64%	81,08%
RF	78,14%	81,40%	82,75%	84,69%
SMO	74,89%	79,99%	86,15%	93,41%



Graf 6.7: Natančnosti klasifikacije za projekt jEdit

Tabela 6.16: Podatki za projekt Apache Log4j

Učenje	Testiranje	Brez napak	Z napako	Brez napak(%)	Z napako(%)
1.0	1.1	101	34	74,81%	25,19%
1.0, 1.1	1.2	128	70	64,65%	35,35%

V rezultatih je bilo moč opaziti, da je bila natančnost klasifikacije za verzijo 1.1 relativno visoka, medtem ko je pri verziji 1.2 močno upadla. Pri algoritmu MLP3 je natančnost klasifikacije na verziji 2.0 znašala zgolj 7,8%. Skupno gledano so klasifikacijo najbolje opravili klasifikatorji MLP2, MLP1 in SMO. Podrobnejši rezultati so podani v tabeli 6.17.

Tabela 6.17: Rezultati meritev poskusa P5 za projekt Apache Log4j

Klasifikator	**1.1**	**1.2**
C4.5-1	77,06%	24,88%
C4.5-2	78,90%	26,83%
MLP1	78,78%	36,70%
MLP2	80,92%	41,92%
MLP3	66,06%	7,80%
RF	80,89%	31,38%
SMO	80,55%	32,68%

Tudi za projekt Apache Lucene smo imeli na razpolago podatke treh verzij, 2.0, 2.2 in 2.4. Podrobnosti podajamo v tabeli 6.18. Opazimo lahko, da sta obe učni množici precej v ravnovesju, saj vsebujeta, v primerjavi z ostalimi, precej podobno število razredov z napako in brez nje.

Tabela 6.18: Podatki za projekt Apache Lucene

Učenje	**Testiranje**	**Brez napak**	**Z napako**	**Brez napak(%)**	**Z napako(%)**
2.0	2.2	102	91	52,85%	47,15%
2.0, 2.2	2.4	170	230	42,50%	57,50%

V rezultatih, podanih v tabeli 6.19, opazimo, da so klasifikatorji v primeru obeh verzij približno enako uspešno napovedali prisotnost napake v razredu. Samo klasifikator MLP3 je na verziji 2.2 odstopal z 42,25% natančnostjo. Najbolje so se odrezali klasifikatorji C4.5-1, SMO in C4.5-2.

Tabela 6.19: Rezultati meritev poskusa P5 za projekt Apache Lucene

Klasifikator	**2.2**	**2.4**
C4.5-1	59,92%	64,12%
C4.5-2	59,51%	64,12%
MLP1	59,60%	60,73%
MLP2	59,77%	61,82%
MLP3	42,25%	59,71%
RF	61,79%	59,32%
SMO	58,65%	65,00%

Za projekt Apache POI smo imeli na razpolago podatke štirih verzij, 1.5, 2.0, 2.5 in 3.0. Kot pri prejšnjemu projektu smo imeli tudi pri tem (razvidno iz tabele 6.20) precej uravnovešene učne množice.

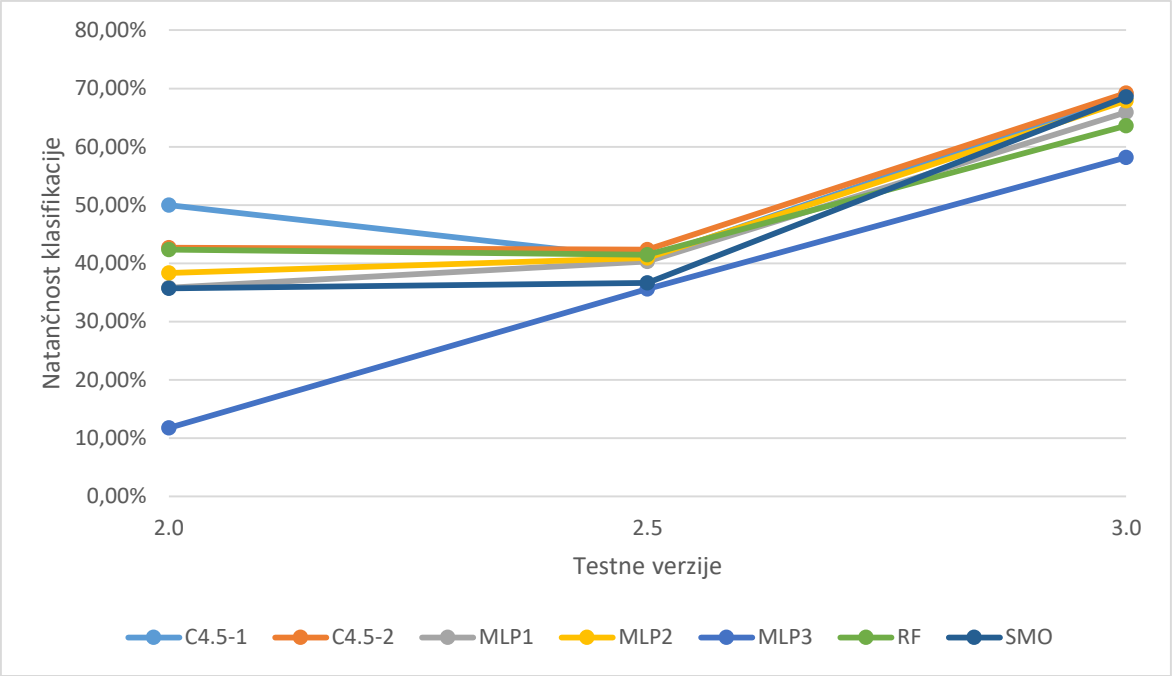
Tabela 6.20: Podatki za projekt Apache POI

Učenje	**Testiranje**	**Brez napak**	**Z napako**	**Brez napak(%)**	**Z napako(%)**
1.5	2.0	89	130	40,64%	59,36%
1.5, 2.0	2.5	272	164	62,39%	37,61%
1.5, 2.0, 2.5	3.0	328	355	48,02%	51,98%

V rezultatih, podrobneje prikazanih v tabeli 6.21, je razvidno, da je natančnost klasifikacije za verziji 2.0 in 2.5 slaba in je v veliki večini krepko pod 50%, se pa natančnost klasifikacije občutno izboljša v primeru, ko klasifikatorje učimo na verzijah 1.5, 2.0 in 2.5, kar je razvidno tudi iz grafa 6.8. Najbolje so se odrezali klasifikatorji C4.5-1, C4.5-2 in RF.

Tabela 6.21: Rezultati meritev poskusa P5 za projekt Apache POI

Klasifikator	**2.0**	**2.5**	**3.0**
C4.5-1	50,00%	40,78%	69,00%
C4.5-2	42,68%	42,34%	69,23%
MLP1	35,83%	40,33%	65,90%
MLP2	38,34%	40,94%	67,87%
MLP3	11,78%	35,58%	58,14%
RF	42,35%	41,46%	63,62%
SMO	35,69%	36,63%	68,55%



Graf 6.8: Natančnosti klasifikacije za projekt Apache POI

V repozitoriju tera-PROMISE so za projekt Apache Synapse na voljo podatki o treh verzijah, in sicer 1.0, 1.1 in 1.2, podrobnosti le teh so prikazane v tabeli 6.22.

Tabela 6.22: Podatki za projekt Apache Synapse

Učenje	**Testiranje**	**Brez napak**	**Z napako**	**Brez napak(%)**	**Z napako(%)**
1.0	1.1	137	16	89,54%	10,46%
1.0, 1.1	1.2	268	76	77,91%	22,09%

Natančnost klasifikacije je bila v primeru obeh testnih verzij precej podobna. V primeru verzije 1.1 je znašala približno 73,5%, v primeru verzije 1.2 pa se je natančnost malenkostno zmanjšala na približno 69%. Najbolje so se odrezali klasifikatorji C4.5-1, C4.5-2 in SMO.

Tudi za projekt Apache Velocity so na razpolago podatki o treh verzijah projekta, 1.4, 1.5 in 1.6. Podrobnosti o podatkih se nahajajo v tabeli 6.24.

Tabela 6.23: Rezultati meritev poskusa P5 za projekt Apache Synapse

Klasifikator	**1.1**	**1.2**
C4.5-1	73,42%	72,27%
C4.5-2	73,42%	71,48%
MLP1	73,53%	69,35%
MLP2	74,53%	69,54%
MLP3	72,97%	66,41%
RF	73,80%	69,91%
SMO	72,97%	70,85%

Tabela 6.24: Podatki za projekt Apache Velocity

Učenje	**Testiranje**	**Brez napak**	**Z napako**	**Brez napak(%)**	**Z napako(%)**
1.4	1.5	47	132	26,26%	73,74%
1.4, 1.5	1.6	110	262	29,57%	70,43%

Rezultati natančnosti klasifikacije so bili v primeru učenja klasifikatorjev na verziji 1.4 in testiranju na 1.5 solidni, saj se je natančnost gibala okoli 63%. Ko pa smo učili klasifikatorje na verzijah 1.4 in 1.5, natančnost klasifikacije pa testirali na verziji 1.6, se je le ta spustila precej pod 50%. Podrobnejši rezultati so podani v tabeli 6.25.

Tabela 6.25: Rezultati meritev poskusa P5 za projekt Apache Velocity

Klasifikator	**1.5**	**1.6**
C4.5-1	63,55%	48,91%
C4.5-2	61,68%	52,40%
MLP1	61,14%	42,55%
MLP2	63,93%	45,62%
MLP3	66,36%	34,06%
RF	65,73%	54,93%
SMO	63,64%	35,66%

Naslednji projekt, kjer smo merili natančnost klasifikacije na zaporednih verzijah projekta, je bil Xalan-Java, za katerega so bili na voljo podatki štirih zaporednih verzij, 2.4, 2.5, 2.6 in 2.7. Podrobnosti so prikazane v tabeli 6.26.

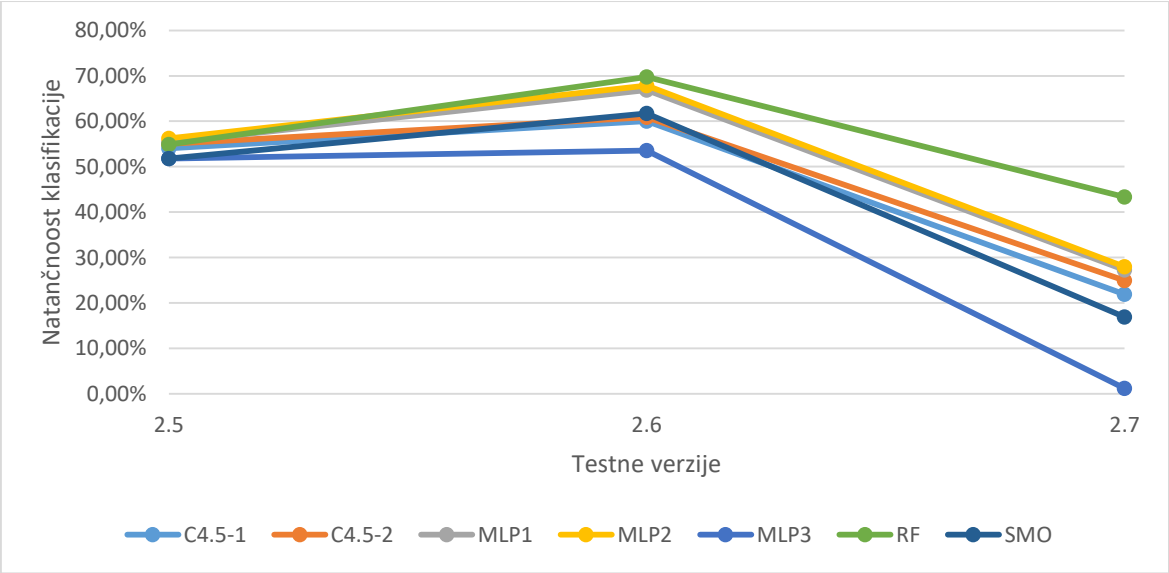
Tabela 6.26: Podatki za projekt Xalan-Java

Učenje	Testiranje	Brez napak	Z napako	Brez napak(%)	Z napako(%)
2.4	2.5	584	110	84,15%	15,85%
2.4, 2.5	2.6	834	475	63,71%	36,29%
2.4, 2.5, 2.6	2.7	1003	727	57,98%	42,02%

Natančnost klasifikacije je ob testiranju natančnosti klasifikatorjev na verziji 2.5 znašala približno 54%, nato se je ob testiranju natančnosti pri verziji projekta 2.6 dvignila na približno 63%. Ob testiranju na verziji 2.7 pa močno padla na povprečno 23%. Podrobnosti meritev so razvidne iz tabele 6.27 in grafa 6.9.

Tabela 6.27: Rezultati meritev poskusa P5 za projekt Xalan-Java

Klasifikator	2.5	2.6	2.7
C4.5-1	54,05%	60,00%	21,89%
C4.5-2	55,04%	60,90%	24,97%
MLP1	55,33%	66,86%	27,28%
MLP2	56,22%	67,86%	27,97%
MLP3	51,81%	53,56%	1,21%
RF	54,91%	69,75%	43,34%
SMO	51,81%	61,72%	16,89%



Graf 6.9: Natančnosti klasifikacije za projekt Xalan-Java

Zadnji projekt, na katerem smo merili natančnost klasifikacije, je bil Xerces. Za ta projekt smo imeli na razpolago podatke štirih verzij, in sicer 1.0, 1.2, 1.3 in 1.4, kar je razvidno iz tabele 6.28.

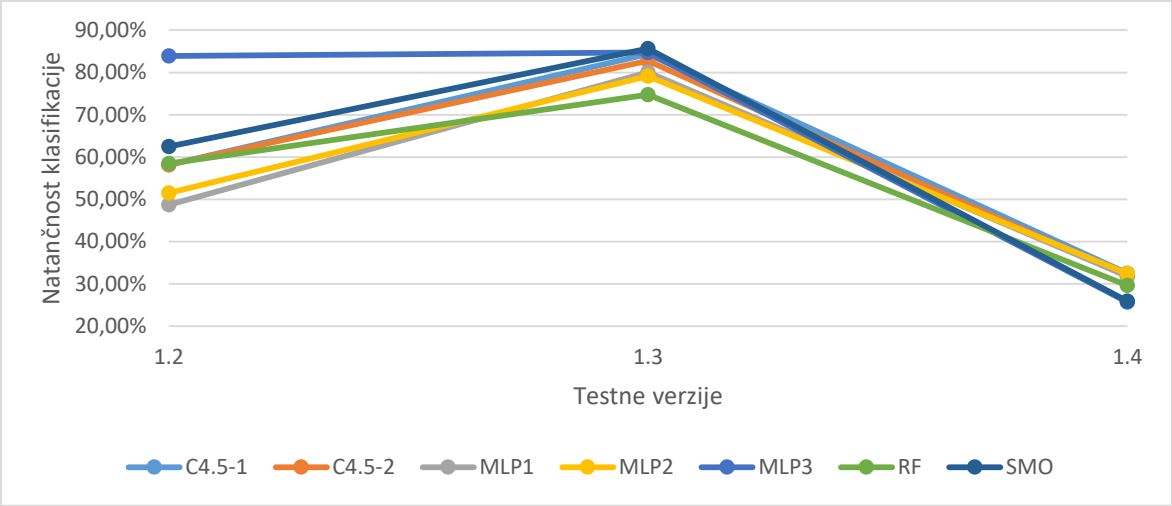
Tabela 6.28: Podatki za projekt Xerces

Učenje	**Testiranje**	**Brez napak**	**Z napako**	**Brez napak(%)**	**Z napako(%)**
1.0	1.2	83	67	55,33%	44,67%
1.0, 1.2	1.3	364	122	74,90%	25,10%
1.0, 1.2, 1.3	1.4	450	188	70,53%	29,47%

Tudi v tem primeru smo zaznali, da se je natančnost klasifikacije izboljšala, ko smo testirali natančnost na verziji 1.3, v primerjavi z verzijo 1.2. Temu vzponu pa je sledil strm padec ob testiranju natančnosti na verziji projekta 1.4, kjer se je povprečna natančnost klasifikacije spustila pod 30%. To je razvidno tudi na grafu 6.10. Podrobnosti meritev so predstavljene v tabeli 6.29.

Tabela 6.29: Rezultati meritev poskusa P5 za projekt Xerces

Klasifikator	**1.2**	**1.3**	**1.4**
C4.5-1	58,18%	84,33%	32,48%
C4.5-2	58,18%	82,78%	31,80%
MLP1	48,67%	80,07%	31,82%
MLP2	51,52%	79,15%	32,45%
MLP3	83,86%	84,77%	25,68%
RF	58,47%	74,75%	29,64%
SMO	62,45%	85,65%	25,91%



Graf 6.10: Natančnosti klasifikacije za projekt Xerces

7 ZAKLJUČEK

V magistrski nalogi smo se spoznali z različnimi tipi metrik za merjenje kakovosti programske opreme in različnimi algoritmi strojnega učenja. Obe temi smo preučili in opisali. Prav tako smo podrobno preučili sorodna dela in nekaj zanimivejših podrobneje predstavili. Seznanili smo se tudi z knjižnico WEKA 3.8 in njenimi zmožnostmi za uporabo v lastnih aplikacijah. Novo pridobljeno znanje nam je služilo kot izhodišče za izdelavo lastne aplikacije.

Izdelana aplikacija je obsegala izvedbo klasifikacije nad več podatkovnimi datotekami, pridobljenimi iz repozitorijev tera-PROMISE in Bug prediction dataset. Aplikacija je rezultat vsake meritve shranjevala v CSV datoteko, ki smo jo potem lahko enostavno uporabili za analizo rezultatov. Prav tako nam je omogočala sočasno izvedbo več meritev, kar je zelo pospešilo izvedbo poskusov.

V magistrskem delu smo skušali odgovoriti na pet raziskovalnih vprašanj. Odgovore nanje poskušamo podati v nadaljevanju.

RV1: Katera metoda strojnega učenja (oz. kateri klasifikator) je najprimernejši za zaznavanje prisotnosti napak v razredu?
Odgovor: V šestem poglavju smo podrobno predstavili vse rezultate. Iz rezultatov poskusov P1-P4 (v teh smo izvajali klasifikacijo nad vsemi možnimi primerki) lahko ugotovimo, da se je najbolje izkazal klasifikator MLP2. Vseeno pa je treba poudariti, da se je natančnost gibala v območju, kjer smo presekali učno množico, se pravi, če je v učni množici 66% razredov brez napak, potem se je povprečna natančnost gibala okoli 66%, ±5%.

RV2: Ali je pristop napovedovanja napak v razredih na osnovi klasičnih metrik programske opreme primeren za uporabo v praksi?

Odgovor: Natančnost klasifikacije za klasične metrike smo merili na dveh različnih repozitorijih. Repozitorij tera-PROMISE je vseboval 66% razredov brez napak, natančnost se je gibala okoli 63%, zato bi takšen model težko uporabili v praksi. Repozitorij Bug prediction dataset pa je vseboval 84% primerkov razredov brez napake. Natančnost se je gibala v območju 83,5%. Iz tega podatka lahko sklepamo, da bi bilo možno nekatere modele uporabiti za napovedovanje napak tudi v praksi. Potrebno pa upoštevati dejstvo, da je klasifikator MLP3 popolnoma vse primerke (v obeh repozitorijih) označil, da ne vsebujejo napake. Čeprav je njegova natančnost klasifikacije relativno velika (83,5%), takšnega modela ne moremo uporabiti v praksi. Na tem mestu dodajmo še informacijo, da do pojava, da MLP3 označi vse razrede, da ne vsebujejo napak, pride zaradi dejstva, da uporabljamo »samo« 1000 naključnih vzorcev iz celotnega repozitorija. Če smo nabor povečali, se je tudi klasifikator MLP3 obnesel bolje.

RV3: Ali je pristop napovedovanja napak na osnovi metrik sprememb primeren za uporabo v praksi?

Odgovor: Za odgovor na to vprašanje moramo natančneje pogledati rezultate v poskusu P3. Poskus smo izvajali na osnovi podatkov iz repozitorija Bug prediction dataset, torej 84% razredov ni vsebovalo napake. Iz rezultatov lahko sklepamo, kot pri vprašanju **RV2**, da bi bilo možno nekatere modele uporabiti za napovedovanje napak. Iz samih rezultatov pa ni mogoče določiti, kateri tip metrik je bolje uporabiti (klasične ali metrike sprememb) za napovedovanje prisotnosti napak v razredih.

RV4: Ali je pristop napovedovanja napak na osnovi kombinacije klasičnih metrik programske opreme in metrik sprememb primeren za uporabo v praksi?

Odgovor: Tudi ta pristop smo poskusili na podatkih iz repozitorija Bug prediction dataset v poskusu P4. Iz pridobljenih rezultatov lahko sklepamo, da je nekatere modele možno uporabiti za napovedovanje napak v razredih. Prav tako je moč opaziti, da se je povprečna

natančnost klasifikacije, v primerjavi z napovedovanjem napak na samo enem tipu metrik, izboljšala za slab odstotek.

RV5: Ali se bo natančnost klasifikacije povečala, če učimo klasifikator na zaporednih verzijah programske opreme?
Odgovor: Za odgovor na to vprašanje smo izvedli poskus na osnovi podatkov iz repozitorija tera-PROMISE. Pričakovali smo, da se bo natančnost klasifikacije izboljšala, če bomo klasifikator učili na podatkih prejšnjih verzij programa in ga testirali na trenutno opazovani, vendar pa tega rezultati žal niso potrdili. V nekaterih projektih, primer jEdit, se je natančnost klasifikacije skozi čas izboljšala, pri ostalih pa se je večinoma celo poslabšala.

Raziskavo oz. modele strojnega učenja bi lahko v prihodnje poskusili izboljšati z iskanjem optimalnih parametrov z genetskim algoritmom. Prav tako bi lahko, bodisi z neko statistično metodo ali nekim genetskim algoritmom, poskusili določiti, katere metrike največ doprinesejo k sami klasifikaciji prisotnosti napake in bi v učenju algoritmov uporabili samo te metrike.

LITERATURA

[1] „Number of available applications in the Google Play Store from December 2009 to December 2017,“ 2018. [Elektronski]. Available: https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/. [Poskus dostopa 31. 3. 2018].

[2] A. Boucher in M. Badri, „Using Software Metrics Thresholds to Predict Fault-Prone Classes in Object-Oriented Software,“ v *4th Intl Conf on Applied Computing and Information Technology*, Las Vegas, 2016.

[3] T. Moharič, „Testiranje programske opreme,“ 2015. [Elektronski]. Available: https://dk.um.si/IzpisGradiva.php?lang=slv&id=55287. [Poskus dostopa 1. 4. 2018].

[4] M. Lanza in R. Marinescu, Object-Oriented Metrics in Practice, Berlin, Heidelberg: Springer, 2006.

[5] C. Jones, Applied software measurement : global analysis of productivity and quality, New York: McGraw-Hill, 2008.

[6] „Software metrics and measurement,“ Wikimedia Foundation, [Elektronski]. Available: https://en.wikiversity.org/wiki/Software_metrics_and_measurement#Overview. [Poskus dostopa 30. 4. 2018].

[7] S. H. Kan, Metrics and Models in Software Quality Engineering, 2nd Edition, Addison-Wesley Professional, 2002.

[8] B. Upadhyaya in S. K. Misra, „A Survey on Formulation of Faulty Class Detection System for Object,“ *IJISET - International Journal of Innovative Science, Engineering & Technology,* Izv. Vol. 2 Issue 2, pp. 516-519, 2015.

[9] A. Sharma in S. K. Dubej, „Comparison of Software Quality Metrics for Object-Oriented System,“ *International Journal of Computer Science & Management Studies,* Izvo. %1 od %2Special Issue of Vol. 12, June 2012, pp. 12 - 24, 2012.

[10] M. Akram, S. Mandala in M. R, „A Literature Review on Various Software Metrics,“ *International Journal for Research in Applied Science & Engineering,* Izvo. %1 od %211, November 2016, pp. 529 - 535, 2016.

[11] S. D. Conte, H. E. Dunsmore, V. Y. Shen in W. M. Zage, „A Software Metrics Survey,“ Department of Computer Sciences Purdue University, Indiana, 1987.

[12] S. M. Jamali, „Object Oriented Metrics (A Survey Approach),“ Department of Computer Engineering Sharif University of Technology, Tehran, 2006.

[13] K. Muthukumaran, A. Choudhary in N. L. Bhanu Murthy, „Mining Github for Novel Change Metrics to Predict Buggy Files in Software Systems,“ *International Conference on Computational Intelligence & Networks,* pp. 15 - 20, 2015.

[14] N. Fenton in M. Neil, „Software metrics: successes, failures and new directions,“ *The Journal of Systems and Software,* Izv. 47, pp. 149-157, 1999.

[15] „Source lines of code,“ [Elektronski]. Available: https://en.wikipedia.org/wiki/Source_lines_of_code. [Poskus dostopa 4. 5. 2018].

[16] M. Kreimeyer in U. Lindermann, Complexity Metrics in Engineering Design, Berlin: Springer, 2011.

[17] D. Kafura in S. Henry, „Software structure metrics based on information flow,“ *IEEE Transactions on Software Engineering Volume SE-7,* Izv. 5, pp. 510 - 518, 1981.

[18] M. Sarker in J. Börstler, An overview of Object Oriented Design Metrics, Umeå: Department of Computer Science, Umeå University, Sweden, 2005.

[19] I. Witten in E. Frank, Data mining: Practical machine learning tools and techniques, second edition, The Morgan Kaufmann Series in Data Management Systems, Elsevier Science, 2005.

[20] A. L. Samuel, „Some Studies in Machine Learning Using the Game of Checkers,“ *IBM Journal,* pp. 211-229, 1959.

[21] T. M. Mitchell, Machine Learning, McGraw-Hill Science/Engineering/Math, 1997.

[22] I. Kononenko, Strojno učenje, Ljubljana: Fakulteta za računalništvo in informatiko, 2005.

[23] P. Kokol, Š. Hleb Babič, V. Podgorelec in M. Zorman, Inteligentni sistemi, Maribor: Fakulteta za elektrotehniko, računalništvo in informatiko, 2000.

[24] N. Nayab in J. Scheid, „A Review of Decision Tree Disadvantages," 2. 9. 2011. [Elektronski]. Available: https://www.brighthubpm.com/project-planning/106005-disadvantages-to-using-decision-trees/. [Poskus dostopa 28. 5. 2018].

[25] M. T. Hagan, H. B. Demuth, M. H. Beale in O. De Jesús, *Neural Network Design, 2nd edition,* 2014.

[26] C. Zhang, M. Yunqian in R. Polikar, Ensemble Machine Learning, New York: Springer Science+Business Media, 2012.

[27] R. Polikar, „Ensemble learning," 2009. [Elektronski]. Available: http://www.scholarpedia.org/article/Ensemble_learning. [Poskus dostopa 1. 6. 2018].

[28] Z.-H. Zhou, *Ensemble Methods: Foundations and Algorithms,* 2012.

[29] L. Breiman, „Random Forests," *Machine Learning,* Izv. 45, pp. 5-32, 2001.

[30] L. Kumar, S. Rath in A. Sureka, „Using Source Code Metrics and Ensemble Methods for Fault Proneness Prediction," 2017.

[31] J. Hryszko, L. Madeyski, M. Dąbrowska in P. Konopka, „Defect prediction with bad smells in code," *Software Engineering: Improving Practice through Research,* pp. 163-176, 2016.

[32] Meiliana, K. Syaeful , L. H. Spits Warnars Harco , G. F. Lumban, E. Abdurachman in B. Soewito, „Software Metrics for Fault Prediction Using Machine Learning Approaches," 2017.

[33] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig in B. Murphy, „Change Bursts as Defect Predictors," IEEE, San Jose, CA, 2010.

[34] „tera-PROMISE,“ [Elektronski]. Available: http://openscience.us/repo/about/. [Poskus dostopa 1. 7. 2018].

[35] M. Jureczko in L. Madeyski, „Towards Identifying Software Project Clusters with Regard to Defect Prediction,“ v *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, Timişoara, 2010.

[36] M. D'Ambros, M. Lanza in R. Robbes, „An Extensive Comparison of Bug Prediction Approaches,“ *In Proceedings of MSR 2010 (7th IEEE Working Conference on Mining Software Repositories),* pp. 31 - 41, 2010.

[37] H. Schildt, Java: A Beginner's Guide, Sixth Edition, Mc Graw Hill Education, 2014.

[38] „IntelliJ FAQ,“ JetBrains, [Elektronski]. Available: http://www.jetbrains.org/display/IJOS/FAQ. [Poskus dostopa 24. 7. 2018].

[39] F. Eibe, M. A. Hall in I. H. Witten, The WEKA Workbench. Online Appendix for "Data Mining: Practical Machine Learning Tools and Techniques", Morgan Kaufmann, 2016.