Примена машинског учења у статичкој верификацији софтвера

Семинарски рад у оквиру курса
Методологија стручног и научног рада
Математички факултет

Лазар Ранковић, Ања Букуров, Војислав Станоквић
mi13268@alas.matf.bg.ac.rs, mi13043@alas.matf.bg.ac.rs,
mi13240@alas.matf.bg.ac.rs

Абстракт

Испитивање исправности програма представља значајну област рачунарства. Експанзија научних као и софтверских решења може имати како позитнивних тако и негативних последица. С тога је потребно поседовати механизам којим се може испитати исправност програмског кода. Међутим, због неодлучивости Халтинг проблема и других ограничења, јављају се различити проблеми приликом испитивања исправности софтвера. Технике машинског учења проналазе све већу примену у различитим областима рачунарства, и примећено је да се неки од проблема статичке верификације могу ефикасно решити применом ових техника. Овај рад ће дати преглед одређених проблема статичке верификације, као и механизме машинског учења који су постигли значајна побољшања.

Садржај

1 Увод

Рачунарски системи имају све значајнију примену у областима индустрије где грешке проузроковане софтвером могу имати озбиљне последице. Програми се користе за аутоматско навођење возила, у нуклеарним електранама, медицини и другим областима. Ове грешке могу довести до пада авиона (ако се ради о аутоматском навођењу возила), експлозију нуклеарних реактора итд. Због тога је потребно имати механизме који формално доказују исправност софтвера и тиме одстрањују могућност грешака. Област верификације софрвера се бави анализом програмских решења и испитивањем исправности програмског кода. Последњих година су достигнути велики напреци у развоју алата за верификацију, али постоје многи проблеми и ограничења у процесу анализе софтвера која се тешко превазилазе. Анализа програмског кода се заснива на математичким моделима и формалним техникама се испитују својства ових модела. Међутим, у неким ситуацијама је коришћење стриктних математичких механизама тешко или чак немогуће за аутоматизацију, што овај процес чини још компликованијим. Са друге стране област машинског учења проналази примену и постиже добре резултате у различитим аспектима рачунарства, у које спада и верификација софтвера [17, 3]. У овом раду ће бити описане неке технике верификације софтвера као и преглед механизама машинског учења који могу побољшати сам процес испитивања исправности софтвера.

2 Верификација софтвера

Верификација софтвера је дисциплина развоја софтвера која за циљ има проверу да ли програм задовољава све унапред задате захтеве. Ти захтеви су представљени спецификацијом свих жељених особина програма и дефинишу се пре процеса верификације. Највећа примена верификације софтвера је у оптимизацији кода и провери исправности.

Потребно је направити разлику између тоталне и делимичне исправности. Испитивање тоталне исправности захтева да се за све могуће улазе покаже заустављање програма. Доказ заустављања у општем случају није могуће извести [15]. Такође, испитивање нетривијалних семантичких својстава је неодлучив проблем [12]. Према томе, у рачунарству је довољно испитати делимичну исправност софтвера, тј. довољно је показати да ће резултат извршавања програма бити валидна вредност.

Два приступа при верификацији софтвера су *динамичка верификација* и *статичка верификација* [6, 7].

- **Динамичка верификација**
  Овај вид верификације се врши у току извршавања програма и то најчешће скупом унапред припремљених тестова који морају бити испуњени. Очигледно је да због неисцрпне варијације могућих улаза, овај вид тестирања нема за циљ валидацију програма, већ је циљ динамичке верификације проналажење грешка на неком нетривијалном скупу тестова.
- **Статичка верификација**
  Статичка верификација софтвера подразумева анализу софтвера без његовог извршавања, тачније анализу кода применом неке од техника које ће бити описане у наставку. Анализу кода може обављати човек, а може се и аутоматизовати. Аутоматизација

подразумева описивање (па чак и превођење) кода језиком изабране математичке теорије.

3 Технике статичке верификације

Претходним поглављем су дефинисани основни појмови верификације софтвера. Предочено је да је немогуће у општем случају испитати заустављање програма и анализирати нетривијална семантичка својства. Ово поглавље ће дати увид у аутоматизоване технике статичке анализе [6].

Апстрактна интерпретација (енг. Abstract Interpretation) је теорија семантичке апроксимације чија је идеја да изгради нову семантику над програмским језиком тако да се конкретан програм увек завршава [16]. Тако се анализа програма врши над апстрактном семантиком да би се добила апроксимација над целом семантиком. Коришћење апстрактне интерпретације се омогућава помоћу две функције: функције која пресликава конкретне вредности у апстрактне вредности и функције која слика апстрактне вредности у конкретне вредности. Користећи овај математички оквир је релативно лако показати да ако се програм завршава у новој семантици програм ће бити коректан и у стварној семантици.

Симболичка анализа (енг. Symbolic Analysis) је метод статичке анализе који анализира програмске вредности који могу да се мењају. Овај метод има за циљ да изведе математички модел који прецизно описује израчунавање, заправо може се посматрати као нека врста компајлера који преводи програм у симболичке изразе. Квалитет алгебарских система као што су: Axiom, Derive, Macsyma, Maple, Mathematica, MuPAD и Reduce је веома битан јер квалитет овог начина анализе у великој мери зависи од паметних алгебарских упрошћавања [16].

Проверавање ограничених модела (енг. Bounded model checking) је техника верификације која се највише користи у верификација логичких кола. Потребно је програм описати исказним формулама након чега се проверава њихова задовољивост. Добијена формула се потом негира и покушава се доказивање њене незадовољивости. Уколико се покаже да је формула незадовољива, следи да је полазна формула таутологија и да је програм исправан. У случају да се покаже да је негирана формула задовољива, онда полазна формула није таутологија, а добијена валуација представља контрапример који може представљати добру основу за даље дебаговање [4, 16].

4 Машинско учење

У претходним поглављима је описана статичка верификација софтвера. Показана је важност те области и изложене су технике верификације. Ово поглавље ће приближити област машинског учења и описаће главне аспекте ове дисциплине.

4.1 Основе машинског учења

Дефиниција 1. *"За програм кажемо да учи из искуства E кроз обављање задатка T са мером квалитета P, ако повећањем искуства E расте мера P за обављен задатак T."*

— Tom M. Mitchell *[10]*

Машинско учење се може посматрати као област рачунарства која се бави анализом алгоритама који генерализују. Са практичног аспекта, генерализација може значити уопштавање закона над датим подацима.

Три најзначајније подобласти машинског учења су: *надгледано учење*, *ненадгледано учење* и *учење условљавањем*. Подаци из којих алгоритми машинског учења уче, могу бити обележени, необележени и могу се генерисати у фази учења. Оваква природа података је основ за разликовање три наведене подобласти [10].

Међу многим проблемима над којима су често примењивани алгоритми машинског учења, истактнути су проблем регресије и проблем класификације. Под проблемом класификације се подразумева испитивање датог објекта и одређивање класе којој он припада на основу његових својстава (атрибута). Типичан пример класификације је одређивање порекла тумора на основу његове величине. Проблем регресије представља предвиђање понашања непрекидне променљиве. Пример регресије је предикција цене стамбеног објекта на основу његове величине, броја соба и разних других релевантних карактеристика.

Пре примене машинског учења потребно је проучити проблем који се решава, уочити његове специфичности и припремити и анализирати податке из којих ће алгоритми учити. Након детаљне анализе, врши се одабир одговарајућег математичког модела. Изабрани модел се даље тренира над подацима. Тренинг се понавља довољан број пута при чему у свакој итерацији модел евалуира тј. мери се грешка коју тај модел прави. Након сваког мерења, у зависности од алгоритма, врши се корекција модела у циљу минимизације грешке.

4.2 Технике машинског учења

О општој слици примене алгоритама машинског учења је било више речи у претходном поглављу. Како је проблем класификације централни проблем над којим се примењује машинско учење у статичкој верификацији, биће представљена два алгоритма која решавају тај проблем. Зарад потпуности, биће описан и један алгоритам решавања регресионих проблема.

Линеарна регресија

Као што је речено у уводном делу поглавља 4, регресиони проблем представља предвиђање циљне променљиве непознате инстанце на основу осталих њених атрибута. Нека је y_i циљна променљива, а $\vec{x} = (x_1, x_2, ..., x_n)$ вектор атрибута. У примеру предикције вредности куће то могу бити број соба, квадратура куће итд. Инстанца из скупа података ће онда бити $(\vec{x}_i, y_i)$. Модел линеарне регресије, параметризован вектором w, који описује законитост је следећи:

$$h(\vec{x}_i) = w^T \cdot \vec{x}_i \tag{1}$$

Грешка коју модел прави је потребно представити погодним избором функције грешке $L(w)$. Чест избор ове функције је средњеквадратна

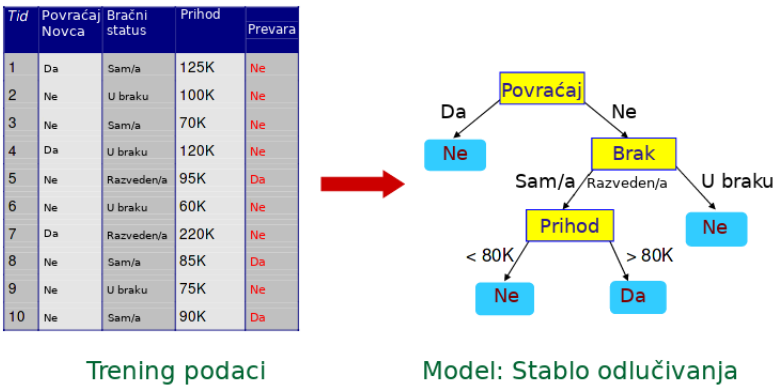
грешка коју модел прави над свим инстанцама из тренинг скупа.

$$L(w) = \frac{1}{N}\sum_{i=1}^{N}(h(\vec{x}_i) - y_i)^2 \tag{2}$$

Тренинг се врши тако што се одређеном оптимизационом техником минимизује функција грешке по параметрима w.

Стабла одлучивања

Стабла одлучивања представљају један од основних метода класификације. Употребу стабала одлучивања оправдава њихова висока интерпретабилност [13]. У листовима стабла одлучивања се налазе вредности циљне променљиве, односно у случају класификације, класе којима инстанца може припасти. Унутрашњи чворови стабла представљају атрибуте по којима се врши подела. Када су ти атрибути категоричког типа, потомци датог чвора су добијени из свих могућих вредности које тај категорички атрибут може имати. У случају да је атрибут некатегоричког типа, најчешће се врши подела могућих вредности на дисјунктне интервале тако да свако дете тог чвора одговара једном од интервала. Слика 1 приказује једно могуће стабло одлучивања добијено на основу података.



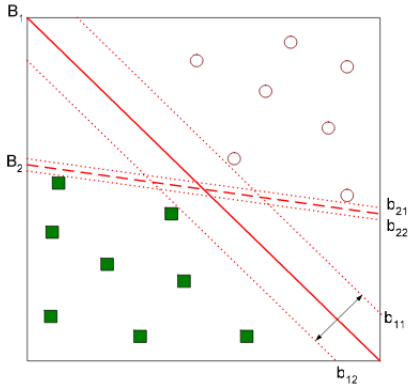
Слика 1: Стабло одлучивања

Метода потпорних вектора

Проблем класификације се може разматрати у следећем контексту. Инстанце које се класификују су представљене тачкама у неком високодимензионалном простору. Бинарни класификатор је хиперраван која дели простор на два дела, тако да се у једном делу простора нађу све инстанце које припадају једној класи, а у другом делу ће се наћи оне које припадају другој класи. Раздвајајућих хиперравни може бити више, па је зато потребно одабрати хиперраван која боље описује поделу међу подацима [11].

Маргина класификације је најмање растојање између тачака које се налазе у различитим потпросторима и бинарног класификатора.

Слика 2 приказује хиперравни B_1 и B_2. Прва хипер̈раван боље раздваја податке. Маргина (b_{11}, b_{12}) је значајно већа од маргине (b_{21}, b_{22}) и то је оно што први класификатор чини знатно бољим.



Слика 2: Приказ различитих хиперравни

Максимизацијом маргине се добија класификатор који боље описује поделу. Зарад конвенције, проблем максимизације се своди на проблем минимизације, те се добија следећи оптимизациони проблем:

$$\min_{w,w_0} \frac{\|w\|^2}{2} \tag{3}$$

Линеарна регресија, стабла одлучивања и метод потпорних вектора су приказани у овом поглављу због својих примена у проблемима статичке верификације. Поглавље 5 се бави овим применама. Следеће поглавље ће представити релевантне проблеме верификације и даће основ за примену ових метода.

5 Одабрани проблеми статичке верификације

До сада смо видели стандардне проблеме и технике статичке верификације и машинског учења. У овом поглављу ћемо издвојити проблеме верификације на које су, применама алгоритама машинског учења постигнути значајнији резултати.

Статичка верификација мора бити у стању да разликује позитивна стања програма од негативних. Негативна су она која доводе програм до грешке. *Интерполантама* (енг. interpolants) називамо предикате који раздвајају позитивна од негативних стања. У статичкој верификацији се коришћењем оваквих интерполанти гради даљи доказ. Показано је да се ове интерполанте могу интерпретирати као бинарни класификатори. Проблем који се овде јавља је генерисање интерполанти, тј. проналажење одговарајућег класификатора [14]. У делу 5.1 су детаљније описани приступи коришћени у [14] и [8].

Поред интерполанти, могуће је препознати нетривијална својства програма која даље резултују грешком. Грађењем *класификатора нетачне инваријанте* (енг. False Invariant Classifier) је могуће рангирати својства програма по томе колику вероватноћу за грешком та својства проузрокују. Одређивање нетривијалног својства датог програма је у

општем случају неодлучив проблем [1, 15]. Део 5.2 приказује примену стабала одлучивања као помоћ при рангирању својстава програма који могу довести до грешке.

Код апстрактне интерпретације је остварив баланс између прецизности изгенерисане инваријанте и скалабилности система за верификацију. Овај баланс је последица детаљне анализе апстрактног синтаксног стабла. Одабир инваријанте је тежак проблем и показано је да се може утврдити тестирањем [8, 14].

5.1 Проналажење интерполанти

Примена машинског учења у проналажењу интерполанти огледа се у добијању модела који представља саму интерполанту. У делу 5.1.1 изложене су основе из [14] базиране на методу потпорних вектора, док је у делу 5.1.2 изложен приступ из рада [8] базиран на стаблима одлучивања. Експериментални резултати показали су да су приступи базирани на машинском учењу упоредиви са традиционалним техникама статичке верификације софтвера.

5.1.1 Проналажење интерполанти користећи метод потпорних вектора

Нека су A и B формуле у теорији линеарне аритметике [9].

$$\phi ::= w^T x + d \geq 0 \mid true \mid false \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \tag{4}$$

При чему је $\vec{w} = (w_1, ..., w_n)^T \in R^n$ вектор константи у простору R^n; $\vec{x} = (x_1, ..., x_n)^T$ вектор променљивих из простора R^n.

Дефиниција 2. *Интерполанта за пар формула (A, B) тако да* $A \wedge B \equiv \bot$ *је формула* I *која задовољава* $A \Rightarrow I, I \wedge B \equiv \bot$ *при чему формула* I *садржи само променљиве које се јављају у формулама* A *и* B.

На коду који следи приказан је програмски код коришћен у [14] као илустрација. Функција непознат број пута инкрементира променљиве x и y, потом их декрементира све док променљива x не постане 0. Коначно, уколико је $y \neq 0$ онда програм одлази у стање грешке. Приметимо да је инваријанта $x = y$ довољна да се докаже да програм никад неће доћи у стање грешке.

```
funkcija primer()
{
1:      int x = 0;
2:      int y = 0;
3:      while (e)
4:      { x++; y++; }
5:      while (x != 0)
6:      { x--; y--; }
7:      if (y != 0)
8:          greska();
}
```

Претпоставимо да се функција `primer()` извршила на следећи начин по линијама кода: $(1, 2, 3, 4, 3, 5, 6, 5, 7, 8)$. Приметимо да наведен ток програма води у стање грешке. Поделимо ток на два скупа, A и B и пронађимо интерполанте.

Скуп A садржи вредности x и y које се добијају након извршавања линија 1, 2, 3 и 4. У скупу B се налазе оне вредности x и y које би се добиле уколико би програм извршио линије 5, 6, 7 и 8 чиме би програм дошао у завршно стање.

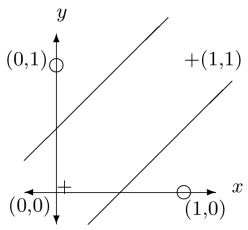
Имамо да $A \wedge B \equiv \bot$ при чему важи:

$$A \equiv x_1 = 0 \wedge y_1 = 0 \wedge ite(e,\ x = x_1 + 1 \wedge y = y_1 + 1,\ x = x_1 \wedge y = y_1)$$
$$B \equiv ite(x = 0,\ x_2 = x \wedge y_2 = y,\ x_2 = x - 1 \wedge y_2 = y - 1) \wedge x_2 = 0 \wedge \neg(y_2 = 0)$$

При чему *ite* означава наредбу `if-then-else`.

A представља скуп достижних стања док B представља скуп стања која воде у стање грешке. Интерполанта је доказ да су скупови A и B дисјунктни и изражава се користећи заједничке променљиве из скупова A и B. Затим, помоћу доказивача теорема се рачунају вредности за (x, y) које задовољавају формуле A и B [14].

Добијене вредности представљају скуп инстанци над којим се може тренирати класификациони модел (попут логистичке регресије или потпорних вектора). Позитивне инстанце представљају вредности променљивих које задовољавају формулу A и аналогно, негативне инстанце представљају вредности променљивих које задовољавају формулу B.



Слика 3: Класификација у тражењу интерполанти

Слика 3 приказује вредности променљивих (x, y) за A као плусеве (тачке $(0, 0)$ и $(1, 1)$) и B као кружиће (тачке $(1, 0)$ и $(0, 1)$). Приказани модел је добијен коришћењем метода потпорних вектора. Резултујуће праве одговарају једначинама:

$$e_1 : 2y = 2x + 1$$
$$e_2 : 2y = 2x - 1$$

Интерполанта која се одавде може извести је

$$2y \leq 2x + 1 \wedge 2y \geq 2x - 1$$

Овај предикат представља инваријанту чијим доказивањем се показује да програм не може доћи у стање грешке. Једноставнија интерполанта $x = y$ се може добити транслирањем добијених правих што ближе позитивним истанцама, докле год се одржава сепарабилност позитивних и негативних инстанци.

Табела 1 приказује резултате из [14] на неким од познатих примера. Интерполанте које су означене са *исто* су интерполанте које су добијене користећи решавач OpenSMT [2] у истом раду.

Табела 1: Добијене интерполанте на неким од познатијих тест примера у области.

Датотека	Време (с)	Интерполанта
f1a	0.022	((y = 1 | x <= 0) & x = 1) | (y = 0 & (y = 1 | x <= 0))
ex1	0.021	xa + 2*ya >= 0 | xa + 2*ya >= 5 | xa + 2*ya >= 5
f2	0.20	y <= 3*x | y <= 3*x + 1 | y <= 3*x + 1
nec1	није доступно	Није пронађена од стране алгоритма
nec2	0.018	x < y (исто)
nec3	0.016	y <= 9 (исто)
nec4	0.021	(x = y | y = 0) | (y = x) | (y = x)
nec5	0.018	s >= 0 (исто)
pldi08	0.017	y > x
fse06	0.017	y + x >= 0 & y >= 0 & y >= 0 & y >= 0

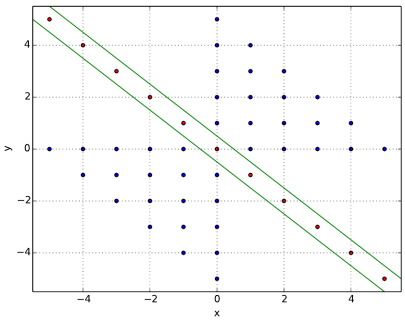
5.1.2 Проналажење интерполанти користећи стабла одлучивања

Интерполанте се могу извести и другим методима машинског учења. Рад [8] илуструје приступ који користи стабла одлучивања. За програмски код се генеришу позитивне и негативне инстанце над којима се гради стабло одлучивања користећи похлепни алгоритам. Правила добијена у стаблу се трансформишу у формулу која се потом проверава да ли је инваријанта користећи SMT решавач.

Резултати су показали да једноставни похлепни алгоритам који гради стабло даје и једноставне формуле за интерполанте. Стабло је лако научило комплексне бинарне инваријанте као једноставне конјункције.

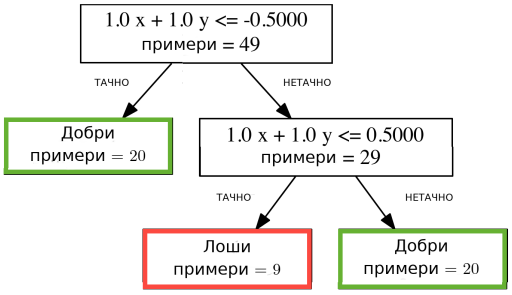
Слика 4 приказује пример програма и његова стања која се могу добити на основу покретања самог програма. Добра стања можемо добити пратећи претпоставке (линија 2), бележећи ток променљивих и провером да ли је испуњен услов $x \neq 0$ са линије 12. Лоша стања можемо добити игноришући услов са линије 2. На пример, тачка $(-2, -2)$ тачка $(-4, -4)$ представљају лоша стања.

```
1  var x, y: Int;
2  assume x = 0 ∧ y ≠ 0;
3
4  while (y ≠ 0) {
5    if (y < 0) {
6      x := x − 1; y := y + 1;
7    } else {
8      x := x + 1; y := y − 1;
9    }
10 }
11
12 assert x ≠ 0;
```



Слика 4: Пример програма. Лева страна приказује код, десна страна садржи добра стања (плаве тачке) и лоша стања (црвене тачке).

Слика 5 приказује стабло добијено применом алгоритма описаног у [8]. Добијени алгоритам је сложености $O(mn \log(n))$, где је m број атрибута а n број инстанци. Детаљни резултати изложени су у раду, а сам метод се показао задовољавајуће и упоредиво у односу на традиционалне методе статичке верификације.



Слика 5: Стабло одлучивања добијено за пример са слике 4.

5.2 Грађење класификатора нетачне инваријанте

Грађење класификатора нетачне инваријанте [1] представља приступ којим се користећи класификатор рангирају својства програма по њиховој вероватноћи да проузрокују грешку. Програмеру се приказује листа пронађених програмских својстава која треба да скрати простор претраге у потрази за грешком. У раду су коришћене две технике класификације машинског учења, метода потпорних вектора и метода стабла одлучивања. Као помоћ при проналажењу својстава програма, користи се Daikon динамички детектор инваријанти [5].

Daikon детектује својства специфичних делова програма попут улаза и излаза из процедура. За скаларне промењиве x, y и z, и константе a, b и c, неки примери својстава су:

- једнакост са константом ($x = a$);
- провера опсега ($a \leq x \leq b$);
- линеарне везе ($z = ax + by + c$);
- уређење ($x \leq y$);
- функције ($y = fn(x)$).

За секвенцијалне промењиве (низови, листе) нека својства су:

- минимум и максимум;
- лексикографско уређење;
- својства која важе за све елементе;
- припадност ($x \in y$).

Daikon може пронаћи и импликације попут *Ако је* $p \neq null$ *онда* $p.vrednost > x$ и дисјункције попут $p.vrednost > c \vee p.levo \in T$.

Добијена својства је потребно превести у векторе како би се омогућила примена алгоритама машинског учења. Један пример кодирања добијених својстава је приказан на табели 2. У раду је кодирање изведено у векторе димензије 388 јер се показало да су коришћени алгоритми успевали да игноришу ирелевантне атрибуте те висока димензионалност није сметала.

У даљем тексту је приказан програмски код који се користи за илустрацију у [1], а у табели 3 листа програмских својстава који могу бити потенцијални проблеми. Добијена листа програмских својстава представља класификоване инстанце које потенцијално откривају грешку.

Табела 2: Пример кодирања програмских својстава у вектор. Број променљивих је означен са #v

Својства	Једначина				Тип променљиве			#v	Резултат
	$\leq$	$=$	$\neq$	$\subseteq$	int	double	niz		
out[1] $\leq$ in[1]	1	0	0	0	0	1	0	2	19
$\forall$ i: in[i] $\leq$ 100	1	0	0	0	0	1	0	1	16
in[0] = out[0]	0	1	0	0	0	1	0	2	15
out.dužina = in.dužina	0	1	0	0	1	0	0	2	13
in $\subseteq$ out	0	0	0	1	0	0	1	2	12
out $\subseteq$ in	0	0	0	1	0	0	1	2	12
in $\neq$ null	0	0	1	0	0	0	1	1	10
out $\neq$ null	0	0	1	0	0	0	1	1	10
Тежине модела	7	3	2	1	4	6	5	3	

Табела 3: Листа понуђених програмских својстава

Својства	Открива грешку?
out[1] $\leq$ in[1]	Да
$\forall i$: in[i] $\leq$ 100	Не
in[0] = out[0]	Да
out.duzina = in.duzina	Не
in $\subseteq$ out	Не
out $\subseteq$ in	Не
in $\neq$ null	Не
out $\neq$ null	Не

```
// Vraca sortiranu kopiju argumenta
double[] bubble_sort(double[] in) {
    double[] out = kopiraj_niz(in);
    for (int x = out.duzina - 1; x >= 1; x--)
    // donja granica treba da bude 0, ne 1
    for (int y = 1; y < x; y++)
        if (out[y] > out[y+1])
            razmeni(out[y], out[y+1])
    return out;
}
```

6 Закључак

Радови приказани у делу 5 показали су да област машинског учења може пронаћи примену у области статичке верификације софтвера. Добијени резултати су били барем упоредиви са другим приступима, а у неким случајевима и доста бољи. Неки од проблема који се јављају при употреби алгоритама машинског учења јесу неинтерпретабилност добијеног модела и неегзактна предвиђања које модел врши. Проблем интерпретабилности је превазиђен стаблима одлучивања [8, 14] која су позната да дају интерпретабилне моделе, док је проблем неегзактног предвиђања ублажен у раду [1] где се као резултат даје листа програмских

својстава које човек анализира. Уколико је неко својство погрешно класификовано, неће проузроковати велику грешку.

Литература

[1] Yuriy Brun and Michael D. Ernst. Finding latent code errors via machine learning over program executions. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 480–490, Washington, DC, USA, 2004. IEEE Computer Society.

[2] Roberto Bruttomesso, Edgar Pek, Natasha Sharygina, and Aliaksei Tsitovich. The opensmt solver. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'10, pages 150–153, Berlin, Heidelberg, 2010. Springer-Verlag.

[3] Christopher M. Brown Dana H. Ballard. *Computer vision*. Prenice-Hall, Inc., 1982.

[4] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 27(7):1165–1178, July 2008.

[5] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 213–224, New York, NY, USA, 1999. ACM.

[6] Milena Vujošević Janičić. *Automatsko generisanje i proveravanje uslova ispravnosti programa*. PhD thesis, Matematički fakultet, Univerzitet u Beogradu, 2013.

[7] Milena Vujošević Janičić. Regresiona verifikacija softvera korišćenjem sistema lav. *InfoM*, 13(49):14–20, 2014.

[8] Siddharth Krishna, Christian Puhrsch, and Thomas Wies. Learning invariants using decision trees. *CoRR*, abs/1501.04725, 2015.

[9] Daniel Kroening and Ofer Strichman. *Linear Arithmetic*, pages 111–147. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[10] Tom M. Mitchell. *Machine Learning*, volume 1. McGraw Hill, 1997.

[11] John Shawe-Taylor Nello Christianini. *An introduction to support vector machines*, volume 1. Cambridge university press, 2000.

[12] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[13] David Landgrebe S. Rasoul Safavian. A survey of decision tree classifier methodology, 1991.

[14] Rahul Sharma, Aditya V. Nori, and Alex Aiken. *Interpolants as Classifiers*, pages 71–87. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[15] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.

[16] Wolfgang Wögerer. A survey of static program analysis techniques, 2005.

[17] Ren Wu, Shengen Yan, Yi Shan, Qingqing Dang, and Gang Sun. Deep image: Scaling up image recognition. *CoRR*, abs/1501.02876, 2015. Withdrawn.