UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Nikola Z. Vidič

PRIMENA MAŠINSKOG UČENJA U VERIFIKACIJI SOFTVERA

master rad

Beograd, 2018.

Mentor:

dr Milena Vujošević Janičić, docent
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

dr Mladen Nikolić, docent
Univerzitet u Beogradu, Matematički fakultet

dr Filip Marić, vanredni profesor
Univerzitet u Beogradu, Matematički fakultet

Datum odbrane: _____

Naslov master rada: Primena mašinskog učenja u verifikaciji softvera

Rezime:

Ključne reči:

Sadržaj

Glava 1

Uvod

Glava 2

Mašinsko učenje

Sama ideja *mašinskog učenja (eng. machine learning)* javlja se još četrdesetih godina dvadesetog veka u radovima Alana Tjuringa (Alan Turing). Razvoj mašinskog učenja vođen je željom da se razume i oponaša ljudski potencijal za učenje. Pedesetih godina mašinsko učenje se razvija zajedno sa pojmom perceptrona, pretka neuronskih mreža. Razvoj mašinskog učenja u formi neuronskih mreža nastavlja se sve do devedesetih godina. Početkom dvehiljaditih dešava se proboj na polju razvoja *veštačke inteligencije (eng. artificial inteligence)* i mnogi problemi za koje se smatralo da će jos dugo ostati nerešeni bivaju rešeni, velikim delom zahvaljujući mašinskom učenju. Uopšteno, „mašinsko učenje predstavlja disciplinu koja se bavi konstrukcijom sistema koji se prilagođavaju i popravljaju svoje performanse sa povećanjem iskustva, oličenog u količini relevantnih podataka [**?**]". Mašinsko učenje proučava indukcijski način zaključivanja tj. generalizaciju (uopštavanje ka univerzalnim zaključcima). Primene mašinskog učenja su brojne: prepoznavanje različitih oblika na slikama (na primer, lica i tumora), autonomno upravljanje vozilima (na primer, automobilima ili letelicama), igranje igara na tabli kao što je šah, klasifikacija teksta, prepoznavanje cifara i mnoge druge [18].

Mašinsko učenje za rešavanje problema koristi različite metode. Ove metode se prema prirodi problema učenja svrstavaju u jednu od tri grupe: *nadgledano učenje* (eng. *supervised learning*), *nenadgledano učenje* (eng. *unsupervised learning*) ili *učenje potkrepljivanjem* (eng. *reinforcement learning*).

2.1 Nadgledano učenje

Nadgledano učenje se odlikuje ulaznim podacima tj. podacima na osnovu kojih se uči i izlaznim podacima tj. podacima koje je potrebno naučiti. Naziv je posledica sličnosti postupka nadgledanog učenja i učenja u kome profesor zada učeniku zadatke i nakon što ih učenik reši, dâ učeniku odgovore radi poređenja rezultata. Algoritmi mašinskog učenja se opisuju kao učenje ciljne funkcije f koja najbolje opisuje vezu između ulaznih promenljivih x i izlazne promenljive y, odnosno $y = f(x)$. Naučena veza (ciljna funkcija f) se kasnije koristi za buduća predviđanja izlaza y na osnovu novih vrednosti ulaza x. Najčešće je ulaz predstavljen vektorom vrednosti promenljivih koje se nazivaju *atributima* (eng. *features*), a izlaz kao jedna promenljiva koja se zove *ciljna promenljiva* (eng. *target variable*). Kako se u današnje vreme raspolaže ogromnim količinama podataka merenim gigabajtima i terabajtima, neophodno je pronaći metode koje automatski pronalaze veze između promenljivih. Veze tj. izgrađene ciljne funkcije se nazivaju *modelima mašinskog učenja*. Postoji veliki broj modela i ne opisuju svi podjednako dobro veze među podacima. Od kvalitetnog modela se očekuje da vrši dobru generalizaciju tj. da prilikom predviđanja retko greši [18, 13].

Osnovne vrste nadgledanog učenja su *klasifikacija* i *regresija*. „Klasifikacija je problem predviđanja kategoričke ciljne promenljive [18]". Vrednosti kategoričke promenljive pripadaju nekom konačnom skupu, pri čemu ne postoji uređenje među tim vrednostima. „Regresija je problem predviđanja neprekidne ciljne promenljive [18]". Neprekidne promenljive uzimaju vrednosti iz neograničenog skupa vrednosti.

U nastavku će detaljnije biti opisani modeli klasifikacije zasnovani na algoritmima k najbližih suseda, slučajnim šumama i logističkoj regresiji.

2.1.1 Algoritam k najbližih suseda

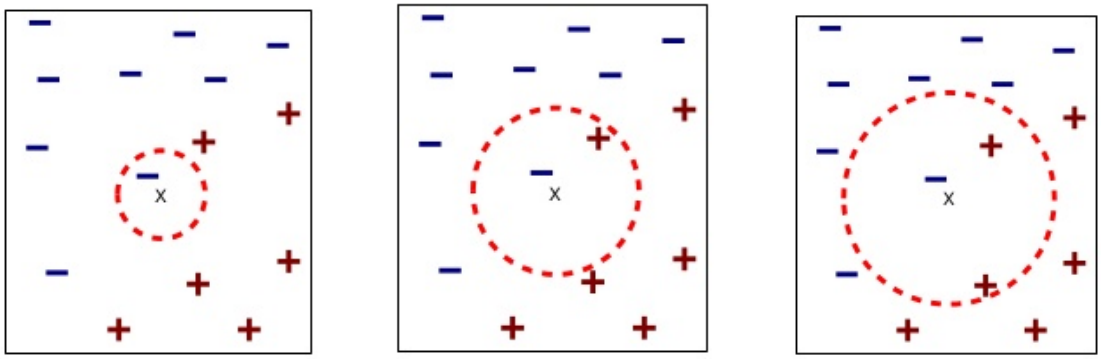
Klasifikatori zasnovani na algoritmu k najbližih suseda spadaju u klasifikatore zasnovane na instancama. Klasifikatori zasnovani na instancama spadaju u neparametarske modele. Takvi modeli ne mogu se opisati konačnim skupom parametara i oni moraju da čuvaju skup podataka za trening na osnovu koga se vrši klasifikacija novih instanci. Dakle, kod modela zasnovanih na instancama ne postoji eksplicitan model već je model sadržan u skupu trening instanci. Sva izračunavanja vrše se u fazi predviđanja. Ovi klasifikatori svoju predikciju zasnivaju na lokalnim podacima

pa su stoga podložni greškama zbog postojanja šuma. Zavisni su od izbora mere bliskosti i adekvatnog preprocesiranja podataka. Ako vrednosti atributa imaju skale različitih vrednosti, jedan atribut može više uticati na ishod klasifikacije a da to nije opravdano. Ovo se može rešiti svođenjem atributa na istu skalu standardizacijom. Pored svoje jednostavnosti, ovi modeli nalaze široku primenu [18, 23].

Objekti podaci koje koristi metod najbližih suseda najčešće su predstavljeni kao tačke u d-dimenzionom prostoru, gde je d broj atributa objekata. Pretpostavlja se da postoji rastojanje među objektima kao i metrika koja definiše to rastojanje. Ideja vodilja primene metoda najbližih suseda je izreka : „Ako hoda kao patka, priča kao patka i izgleda kao patka, onda je najverovatnije patka" [18, 23].

2.1.1.1 Dodeljivanje klase objektu

Prilikom klasifikacije nepoznatog objekta prvo se među poznatim objektima pronađe njegovih k najbližih suseda na osnovu izabrane mere bliskosti (često je u upotrebi *Euklidsko rastojanje*). Algoritam novom objektu pridružuje klasu koja se *najčešće javlja među njegovim susedima* (eng. *majority voting*). U slučaju nerešenog ishoda, klasa nepoznatog objekta se dobija slučajnim izborom iz skupa najzastupljenijih klasa. Da bi se izbegli nerešeni ishodi, česta je praksa da se za k uzima neparan broj [23, 18].



(a) k=1 (b) k=2 (c) k=3

Slika 2.1: Susedstva i dodeljivanje klasa novoj instanci za k=1, 2 i 3

Na slici 2.1 prikazana su susedstva instance koja se nalazi u sredini kruga kada algoritam k najbližih suseda uzima vrednosti 1, 2 i 3 za k. Instanci će biti dodeljena klasa na osnovu klasa njenih najbližih suseda. Na slici 2.1a klasa najbližeg suseda je (-) pa će i klasa instance biti (-). U slučaju 2.1c dva suseda su klase (+), a jedan

klase (-) pa će klasa instance biti (+). Slika 2.1b ilustruje situaciju podjednake zastupljenosti klasa (jedan (+) i jedan (-)). U ovakvim situacijama, instanci se dodeljuje klasa slučajnim izborom [23].

Za rezultat klasifikacije jako je bitan izbor parametra k. U slučaju male vrednosti parametra k može doći do preprilagođavanja. Do grešaka dolazi jer je mali broj suseda uključen u razmatranje. Često se može javiti greška zbog prisustva šuma. Sa druge strane, velika vrednost parametra k vodi ka potprilagođavanju. U tom slučaju do greške može doći jer se razmatraju klase onih objekata koji nisu u neposrednom susedstvu [18, 23].

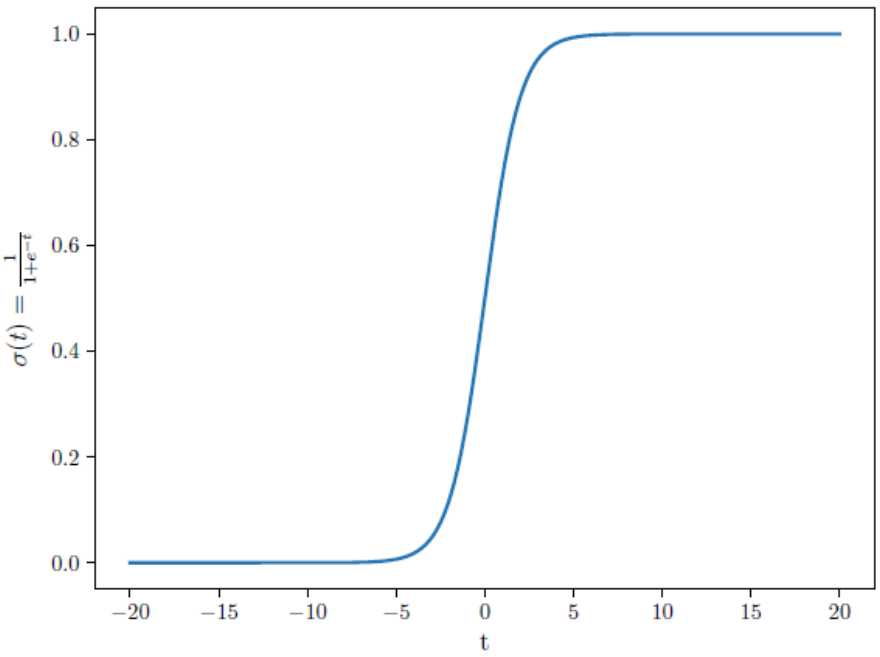
2.1.2 Logistička regresija

Logistička regresija predstavlja probabilistički model klasifikacije. Koristi se i za probleme binarne i za probleme višeklasne klasifikacije. Kod probabilističkih modela je potrebno definisati raspodelu verovatnoće koju kasnije treba oceniti na osnovu podataka. Da bi ocena raspodele bila računski izvodljiva potrebno je uvesti pretpostavke o raspodeli podataka i međusobnoj zavisnosti promenljivih. Pogrešne pretpostavke o podacima, njihovoj raspodeli i zavisnosti, vode ka lošijim rezultatima predviđanja. U daljem tekstu detaljnije je opisana binarna logistička regresija [18].

Logistička regresija pretpostavlja međusobnu nezavisnost vrednosti ciljne promenljive y i *Bernulijevu raspodelu* ciljne promenljive y za date vrednosti atributa x. To znači da postoji parametar μ iz intervala [0,1] takav da je $p(y{=}1|x) = \mu$, a $p(y{=}0|x) = 1\text{-}\mu$. Ovako definisan model logističke regresije nije kompletan. Potrebno je definisati zavisnost parametra μ od vrednosti atributa x. Da bi se parametar μ uklopio u definiciju verovatnoće, njegove vrednosti moraju biti u intervalu [0,1]. Poželjno bi bilo koristiti model linearne regresije zbog njegove jednostavnosti i lake interpretabilnosti, ali na prvi pogled ovo nije moguće jer vrednosti linearnog modela pripadaju intervalu $[-\infty, \infty]$. Međutim, moguće je koristiti model linearne regresije ako se njegove vrednosti transformišu nekom nenegativnom, monotonom, neprekidnom i diferencijabilnom funkcijom u interval [0,1]. U tu svrhu koristi se *sigmoidna funkcija* σ (moguća je upotreba i nekih drugih funkcija):

$$\sigma(t) = \frac{1}{1 + e^{(-t)}}$$

Sigmoidna (logistička) funkcija, čiji je grafik prikazan na slici 2.2, uzima proizvoljan realan broj i dodeljuje mu vrednost iz intervala (0,1).



Slika 2.2: Sigmoidna funkcija

Transformacijom vrednosti linearnog modela, definisana je i zavisnost parametra Bernulijeve raspodele μ od vrednosti atributa x. Slično modelu linearne regresije, forma modela logističke regresije je jednačina:

$$p_w(y = 1|x) = \sigma(w \cdot x)$$

Izlaz modela je linearna kombinacija njegovog ulaza i koeficijenata. Koeficijenti se dobijaju iz podataka prilikom samog treniranja modela i oni predstavljaju reprezentaciju modela u memoriji [18, 14].

Treniranje modela linearne regresije odgovara oceni njegovih parametara i zasniva se na **principu maksimalne verodostojnosti**. Kao što smo videli, da bi se precizirao statistički model potrebno je uključiti neke parametre. Da bi dobijeni rezultat imao smisla, izbor vrednosti parametara je jako bitan. Jedan od principa izbora ovih vrednosti, princip maksimalne verodostojnosti, je neprihvatanje onih vrednosti parametara za koje su posmatrani podaci malo verovatni. Na taj način biće prihvaćene one vrednosti parametara za koje su posmatrani podaci visoko verovatni [18].

2.1.3 Slučajne šume

Grupni metodi (eng. ensemble methods) su tehnike koje za cilj imaju poboljšanje tačnosti klasifikacije koje se postiže kombinovanjem predviđanja većeg broja klasifikatora. Ovi metodi konstruišu veći broj *baznih klasifikatora* (eng. *base classifiers*) i kombinovanjem njihovih rezultata vrše svoje predviđanje. Rezultat predviđanja je srednja vrednost u slučaju regresije ili najzastupljenija vrednost u slučaju klasifikacije. Rezultati dobijeni na ovaj način su bolji od rezultata dobijenih korišćenjem samo jednog klasifikatora.
Da bi grupni klasifikatori zaista davali bolje rezultate od pojedinačnih klasifikatora, potrebno je da budu ispunjena dva uslova [23]:

1) Bazni klasifikatori treba da budu međusobno nezavisni. U tom slučaju konačna predikcija će biti pogrešna samo ako više od polovine baznih klasifikatora pogreši u predikciji. Potpuna nezavisnost klasifikatora je teško ostvariva, ali se u praksi pokazalo da nije neophodna da bi se ostvarili bolji rezultati.

2) Bazni klasifikatori treba da budu bolji od slučajnog klasifikatora

Osnovna ideja grupnih metoda je kreiranje velikog broja klasifikatora na osnovu trening podataka i neki vid agregacije njihovih rezultata u slučaju nepoznate instance. Grupa klasifikatora može se konstruisati na neki od sledećih načina [23]:

Manipulacijom trening podataka Ponovnim izborom, u skladu sa izabranom raspodelom, iz originalnog skupa trening podataka kreira se veći broj novih skupova podataka. Raspodela utiče na verovatnoću da će podatak biti izabran za novi trening skup. Zatim se za svaki novi trening skup kreira model klasifikacije izabranim algoritmom (npr. stablo odlučivanja). Dva grupna metoda koja rade na ovaj način su upakivanje i podsticanje.

Upakivanje (eng. *bagging*) je tehnika koja iznova vrši izbor uzoraka sa zamenom na osnovu uniformne raspodele verovatnoća. Kako sve instance imaju jednaku verovatnoću da budu izabrane, ova tehnika je manje podložna greškama usled preprilagođavanja. Dobijeni skup uzoraka je iste veličine kao i originalni skup. Kako se izbor uzoraka vrši sa zamenom neki podaci će biti izabrani više puta a neki nijednom. Na dobijenim

skupovima se treniraju klasifikatori. Nakon što se kreiraju svi klasifikatori moguće je klasifikovati nepoznate instance. Novoj instanci će biti pridružena najzastupljenija klasa.

Podsticanje (eng. *boosting*) je iterativna procedura koja postepeno menja raspodelu trening podataka i na taj način favorizuje one podatke koji su teži za klasifikaciju. Svakoj trening instanci je pridružen koeficijent težine koji se može promeniti po završetku iteracije. Težine se mogu koristiti bilo kao raspodela pri kreiranju novih skupova podataka bilo za treniranje pristrasnih modela.

Manipulacijom ulaznih atributa Novi skupovi trening podataka nastaju kao podskupovi originalnih podataka, slučajnim izborom podataka početnog skupa ili analizom domena. Ovaj pristup daje dobre rezultate u slučaju redundantnih atributa. Slučajne šume su primer metoda koji manipuliše ulaznim atributima i koristi klasifikatore zasnovane na stablima odlučivanja u svojoj osnovi.

Manipulacijom algoritma učenja Moguće je menjati i sam algoritam učenja. Ovaj način nalazi praktičnu primenu ubacivanjem faktora slučajnosti prilikom treniranja skupa stabala odlučivanja. Umesto izbora najboljeg atributa za podelu u čvoru, moguće je slučajnim izborom odabrati jedan od nekoliko najboljih atibuta podele.

Manipulacijom oznaka klasa Ako je broj klasa dovoljno, veliki višeklasni problem je moguće transformisati u binarni problem podelom oznaka klasa, na slučajan način, u dva disjunktna skupa A_0 i A_1. Trening podacima čije klase pripadaju skupu A_0 pridružuje se klasa 0, a podacima čije klase pripadaju skupu A_1 klasa 1. Podaci sa novim oznakama klasa se koriste za treniranje baznog klasifikatora. Grupa klasifikatora se dobija višestrukim ponavljanjem kreiranja skupova A_0 i A_1 i treniranja klasifikatora nad dobijenim skupovima. Dodeljivanje klase nepoznatoj instanci vrši se tako što svaki bazni klasifikator C_i vrši predviđanje. Ako je predviđena klasa 0, sve klase skupa A_0 dobijaju glas. Analogno, ako je predviđena klasa 1, sve klase skupa A_1 dobijaju glas. Glasovi se prebrojavaju i rezultujuća klasa će biti ona sa najviše glasova.

Slučajne šume (eng. *Random forests*) pripadaju klasi grupnih metoda. Kao bazni klasifikatori se koriste stabla odlučivanja koja se konstruišu nad skupom nezavisnih

slučajnih vektora koji nastaju metodom manipulacije trening podataka. *Upakivanje sa korišćenjem stabala odlučivanja* (eng. *Bagging using decision trees*) je specijalan slučaj slučajnih šuma u kom se slučajnost dodaje u proces pravljenja modela (do sada je postojala u trening podacima) tako što se novi skupovi podataka kreiraju metodom slučajnog izbora sa zamenom od elemenata polaznog skupa. Dodavanje slučajnosti smanjuje korelaciju između stabala, a samim tim i grešku generalizacije cele grupe. Upakivanje popravlja grešku generalizacije smanjujuću varijansu baznih klasifikatora. Pored toga, upakivanje je manje podložno i greškama usled preprilagođavanja, koje je velika mana stabala odlučivanja, zato što nijedna trening instanca nema prednost prilikom izbora, već sve imaju podjednaku verovatnoću da će biti izabrane [23].

Svako stablo odlučivanja za podelu u čvoru koristi odgovarajući slučajni vektor podataka koji se dobija na jedan od sledećih načina [23]:

1) Slučajnim izborom se određuje F atributa koji će se koristiti za podelu u svakom od čvorova (umesto da se za podelu razmatraju svi atributi) nakon čega se stablo konstruiše do kraja bez odsecanja. Ako postoji potreba za većim faktorom slučajnosti moguće je koristiti upakivanje za kreiranje trening skupova. Izbor broja atributa F utiče na snagu i korelaciju modela slučajnih šuma. Za male vrednosti F stabla šume su slabo korelisana, dok veliko F omogućava jače modele zbog korišćenja većeg broja atributa. Često se kao ravnoteža bira $F = log_2 d + 1$, gde je d broj ulaznih atributa. Ovaj pristup vodi manjem vremenu izvršavanja algoritma jer se prililkom podele u čvoru ne razmatraju svi atributi.

2) Ako početnih atributa d ima malo, teško je pronaći nezavisni skup slučajnih atributa koji se koriste pri izgradnji modela. Ovo se može prevazići proširenjem skupa atributa njihovom linearnom kombinacijom. Na nivou svakog čvora novi atribut nastaje tako što se metodom slučajnog izbora izabere L atributa iz polaznog skupa, nakon čega se ti atributi linearno kombinuju sa koeficijentima dobijenim iz uniformne raspodele na intervalu [-1,1]. Svaki čvor će dobiti F novih atributa, a najbolji od njih će biti korišćen za podelu čvora.

3) Mogući pristup podeli unutar čvora je da se umesto najboljeg atributa podele na slučajan način izabere jedan od F najboljih atributa. Stabla dobijena na ovaj način mogu imati veći stepen korelacije u slučaju nedovoljno velikog parametra

F. Pored toga, ovaj pristup ne daje uštedu u vremenu izvršavanja kao prethodna dva pristupa jer je potrebno ispitati svaki atribut podele u svakom čvoru stabla.

2.2 Mašinsko učenje u Python-u

„Python je interpretirani, objektno-orijentisani viši programski jezik sa dinamičkom semantikom. Pythonova jednostavna sintaksa se lako uči i naglašava čitljivost koda koja za posledicu ima nisku cenu održavanja." Python podržava veliki broj paketa i modula što vodi ponovnoj upotrebljivosti i modularnosti koda. Nepostojanje faze kompilacije ubrzava čitav proces pisanja i testiranja koda. Debagovanje je prilično jednostavno, greške proizvode izuzetke i ako se oni ne uhvate interpreter ispisuje *stanje steka* (eng. *stack trace*). Štaviše, često je najjednostavniji način debagovanja ubacivanje print naredbi [9].

Autor Pythona je Holanđanin Gido Van Rosum (hol. Guido Van Rossum). Ideju o Pythonu Gido dobija krajem 1980-ih kada je radio kao programer na Državnom institutu za matematiku i informatiku CWI (Centrum voor Wiskunde en Informatica) u jeziku ABC kojim je inspirisan. Implementaciju započinje decembra 1989-te, a prva verzija jezika (verzija 0.9.0) puštena je 1991-ve. Dalji razvoj Pythona se nastavlja. Januara 1994-te izlazi verzija 1.0, unapređena verzija 2.0 izlazi u oktobru 2000-te, a verzija 3.0 u decembru 2008-me [2, 1].

2.2.1 SciPy ekosistem

SciPy je ekosistem slobodnog softvera za matematiku, nauku i inženjerstvo. Njegovu osnovu čini programski jezik Python i grupa paketa od kojih su za mašinsko učenje najznačajniji [6]:

1) **NumPy** je osnovni paket za naučna izračunavanja u Pythonu. Ovaj paket definiše numeričke nizove, matrične tipove, kao i osnovne operacije nad njima. Glavni objekat paketa je homogeni višedimenzioni niz koji je predstavljen kao tabela elemenata (najčešće brojeva), pri čemu su svi istog tipa i indeksirani torkom pozitivnih celih brojeva. Dimenzije niza se nazivaju *osama* (eng. *axis*). Klasa nizova NumPy paketa se naziva *ndarray*.

- Tačka prostora npr. [1, 2, 1] sa svoje tri koordinate ima jednu osu (dimenziju). Osa se sastoji od 3 elementa pa se kaže da je dužine 3.
- Dvodimenzioni niz, npr.

$$[[1., 0., 0.],$$

$$[0., 1., 2.]]$$

  ima dve ose. Prva osa ima 2 elementa, a druga 3 [8, 7].

2) **Pandas** je paket koji obezbeđuje strukture podataka za jednostavan i intuitivan rad sa podacima. Predstavlja nadgradnju NumPy biblioteke i napravljen je tako da se lako integriše u razna okruženja koja se koriste za naučna izračunavanja. Pogodan je za podatke koji su smešteni u tabelama, uređene i neuređene vremenske serije, proizvoljne matrice i sve oblike podataka dobijenih na osnovu statističkih posmatranja. Pandas uvodi dve strukture podataka, Series za jednodimenzione podatke i DataFrame za dvodimenzione [4].

3) **Matplotlib** je biblioteka koja služi za kreiranje različitih vrsta dijagrama. Većinu dijagrama moguće je kreirati u svega nekoliko linija koda, a sami dijagrami se mogu prilagoditi potrebama korisnika izborom stilova linija (pune, isprekidane,...), svojstava fontova, svojstava koordinatnih osa, itd [3].

Scikit-learn je modul koji implementira veliki broj algoritama mašinskog učenja. Pored algoritama za rešavanje problema klasifikacije, regresije i klasterovanja ovaj modul podržava i razne tehnike preprocesiranja podataka, kao i tehnike ocenjivanja modela. Zasniva se na bibliotekama NumPy, SciPy i matplotlib [5].

Glava 3

Verifikacija softvera

Softver se nalazi svuda oko nas. U svakodnevnom životu, ljudi se sve više oslanjaju na sisteme zasnovane na informacionim i komunikacionim tehnologijama, pri čemu ti sistemi postaju sve složeniji i sve zastupljeniji. Ovi sistemi nalaze primenu u velikom broju oblasti: od berze, preko telefonskih i internet tehnologija pa sve do medicinskih sistema. Osim što se od njih očekuju dobre performanse u smislu vremena odziva i izračunavanja, jedan od ključnih aspekata kvaliteta sistema je odsustvo grešaka. Međutim, softver je pisan od strane ljudi pa stoga ne može biti savršen i podložan je greškama [11, 21].

Neke greške mogu biti pogubne po proizvođače. Greška pri deljenju brojeva u pokretnom zarezu na Intelovom Pentium II čipu (Intel Pentium Floating-Point Division Bug) ranih 1990-ih izazvala je gubitak od oko 475 miliona dolara da bi se čip zamenio i narušila je reputaciju Intela kao proizvođača pouzdanih čipova. Softverska greška u sistemu za upravljanje prtljagom prouzrokovala je odlaganje otvaranja aerodroma u Denveru za 9 meseci, što je dovelo do gubitka od 1,1 milion dolara po danu. Prestanak rada sistema za rezervaciju karata neke aviokompanije na svega 24 časa bi doveo kompaniju do bankrota zbog propuštenih porudžbina. NASA-ina kosmička letelica (NASA Mars Polar Lander) je usled nedovoljno testiranja 1999-te nestala (tačnije slupala se) prilikom pokušaja sletanja. Nekoliko stotina milijardi dolara potrošeno je za ispravljanje baga Y2K (The Y2K (Year 2000) Bug [10]) [11, 21].

Neke druge greške imaju daleko ozbiljnije posledice i njihova cena se ogleda u ljudskim životima. Softveri se koriste za upravljanje sistemima kao što su hemijske

i nuklearne elektrane, sistemima za upravljanje saobraćaja, sistemima za odbranu od olujnih talasa (eng. storm surge barriers), a greške ovih sistema mogu imati katastrofalne posledice. Greška u protivraketnom sistemu (Patriot Missile Defense System) 1991-ve dovela je do smrti 28 vojnika. Greška u mašini za terapiju radijacijom Therac-25 prouzrokovala je smrt šest pacijenata u periodu između 1985-te i 1987-me zbog prevelike izlošenosti radijaciji. [11, 21]

3.1 Specifikacija

Provera ispravnosti nekog softvera, odnosno njegovog ponašanja, nije moguća bez poznavanja željenog ponašanja datog softvera. Sa ciljem definisanja željenog ponašanja uvodi se pojam specifikacije. *Specifikacija* predstavlja dogovor razvojnog tima oko toga šta softver treba da radi. Rezultat ispitivanja zahteva klijenta su podaci koji opisuju šta klijent očekuje od programa. Ovi podaci ne opisuju proizvod koji treba napraviti. Opis proizvoda je dat specifikacijom. Za definisanje izgleda i ponašanja proizvoda specifikacija koristi informacije dobijene od klijenta kao i obavezne zahteve koje program mora da sadrži a koje klijent nije naveo. Izgled specifikacije može da varira. Kompanije koje razvijaju proizvode u oblastima medicine, vazduhoplovstva, kao i za vladine organizacije koriste striktniji oblik specifikacije koji podrazumeva veliki broj provera. Rezultat je iscrpna i temeljna specifikacija koja se ne može menjati, osim pod ekstremnim okolnostima i svaki član razvojnog tima tačno zna šta treba da napravi. Sa druge strane, kompanije koje razvijaju programe čija je primena u manje rizičnim oblastima primenjuju opuštenij vid specifikacije, pišu je „na salveti" ako je uopšte i napišu. Prednost ovakvog pristupa pisanju specifikacije ogleda se u fleksibilnosti, ali je veliki nedostatak potencijalna neusklađenost oko ponašanja programa kao i to što se sve do završetka projekta ne zna kako će njegova finalna verzija izgledati. [19, 21]

3.2 Verifikacija i validacija

Verifikacija i validacija su pojmovi koji se često međusobno prepliću a imaju različito značenje. *Verifikacija* (eng. software verification) je postupak potvrđivanja da softver zadovoljava specifikaciju. *Validacija* (eng. software validation) je postupak potvrđivanja da softver zadovoljava potrebe korisnika. Razlika između ova dva slična pojma se može razjasniti na primeru problema sa Habl teleskopom

(eng. Hubble telescope). Habl je reflektujući teleskop koji koristi veliko ogledalo da uveća objekte ka kojima je uperen. Pravljenje ogledala je zahtevalo veliku tačnost i preciznost. Kako je ogledalo pravljeno da se koristi u svemiru, jedini način da se testira je bio pažljivim merenjem svih atributa ogledala i njihovim poređenjem sa specifikacijom. Testovi su prošli uspešno i teleskop je lansiran u zemljinu orbitu aprila 1990-te. Ubrzo nakon što je teleskop počeo sa radom otkriveno je da su slike koje vraća mutne kao posledica greške u proizvodnji ogledala. Iako je ogledalo zadovoljavalo specifikaciju, specifikacija je bila pogrešna što je dovelo do ogledala koje ima visoku preciznost a nisku tačnost. Testiranje je u ovom slučaju potvrdilo da ogledalo zadovoljava specifikaciju tj. verifikacija je bila uspešna. Sa druge stranje testiranje nije potvrdilo da ogledalo zadovoljava zahteve korisnika i validacija, da je rađena, ne bi prošla. Cena ove greške je bila u novoj svemirskoj misiji 1993-će da bi se dodalo korektivno sočivo [21].

Uopšteno govoreći, program ne zadovoljava specifikaciju ako sadrži greške koje narušavaju funkcionalne i nefunkcionalne željene osobine programa. Funkcionalne osobine definišu koji su to željeni izlazi za zadate ulaze, a u nefunkcionalne osobine softvera spadaju vreme odziva, performanse i efikasnost. Bitna klasa grešaka koje mogu da naruše funkcionalne osobine softvera je klasa grešaka koje utiču na ispunjavanje bezbednosnih zahteva, npr. dovode do kraha programa ili utiču na performanse i efikasnost. Neki od primera su pokušaj deljenja nulom, pokušaj dereferenciranja NULL pokazivača, pokušaj čitanja sadržaja van granica rezervisane memorije i prisustvo kružnih blokada. Ove greške se mogu javiti nevezano od namene programa [19].

Osnovne vrste verifikacije softvera su dinamička i statička [19]. Ova dva pojma je jednostavno objasniti na primeru kupovine polovnog automobila. Gledanje ispod haube, traženje tragova rđe, traženje ulubljenja, traženje ogrebotina na farbi, naprsnuća stakla, šutiranje guma su tehnike statičkog testiranja. Paljenje motora automobila i vožnja koja obuhvata dodavanje gasa i kočenje su tehnike dinamičkog testiranja [21].

3.3 Dinamička verifikacija softvera

Dinamička verifikacija softvera je provera njegove ispravnosti u toku izvršavanja. Provera ispravnosti se najčešće vrši testiranjem. Samim testiranjem se ne može do-

kazati ispravnost programa. Ono što testiranje može da uradi je da pokaže prisustvo grešaka, ne i njihovo odsustvo. Zbog toga je svrha testiranja otkrivanje grešaka koje program sadrži. [19].

Testiranje je proces u kojem se program pokreće sa reprezentativnim skupom ulaznih podataka, nakon čega se rezultati porede sa očekivanim. Strategija testiranja služi za određivanje reprezentativnih ulaza pri čemu reprezentativni skup treba da zadovoljava sledeće uslove: (1) ima visok potencijal otkrivanja greške, (2) relativno je male veličine, (3) vodi do visokog poverenja u softver nakog što softver uspešno prođe sve test primere. Dva osnovna izvora testova su programski kôd i specifikacija programa. U zavisnosti od izvora testova postoje tri strategije testiranja [19]:

Strategije testiranja crne kutije (eng. black-box testing) su strategije generisanja testova na osnovu specifikacije programa. Poznate su pod nazivima testiranje vođeno podacima ili testiranje vođeno ulazom i izlazom. Ideja je da se program posmatra kao crna kutija, sve se zasniva na zahtevima i specifikaciji programa, bez ikakvih briga o njegovoj strukturi i unutrašnjem ponašanju. Fokus strategije je na pronalaženju uslova koji dovode do ponašanja programa koje je suprotno ponašanju navedenom u specifikaciji. Test primeri se izvode samo na osnovu specifikacije softvera. U okviru ovih strategija izdvajaju se testiranje funkcionalnih i nefunkcionalnih osobina softvera, kao i regresiono testiranje [19, 20, 16].

Da bi se ovom strategijom pronašle sve greške koje postoje u programu potrebno je testirati svaki mogući ulaz u program.
Za testiranje programa koji rešava problem trouglova (za date celobrojne vrednosti stranica trougla program daje odgovor na pitanje da li je trougao jednakokraki, jednakostranični ili nejednakostranični [22]) potrebno je testirati sve validne i nevalidne trouglove. Skup celobrojnih vrednosti je sam po sebi ogroman, uz nevalidne ulaze broj ulaza koje treba proveriti se drastično povećava. Iscrpno testiranje velikih programa predstavlja jos veći problem. Da bi se testirala ispravnost kompajlera potrebno je kao ulaz koristiti ne samo svaki validni program (kojih ima beskonačno), već i svaki nevalidni program (kojih takođe ima beskonačno). Tek tada se sa sigurnošću može tvrditi da kompajler neće uspešno prevesti neispravan program.
Stvari se dodatno komplikuju kod programa koji koriste neki oblik memorije

poput aplikacija koje rade sa bazama podataka. U slučaju sistema za rezervaciju avionskih karata izvršavanje jedne transakcije (na primer, provera slobodnih mesta čitanjem iz baze, rezervacija karte upisom u bazu) zavisi od toga šta se desilo u prethodnim transakcijama. Stoga je potrebno proveriti ne samo sve moguće validne i nevalidne transakcije već i sve moguće sekvence transakcija [20].

Prethodni primeri pokazuju da je iscrpno testiranje nemoguće zbog beskonačnog skupa test primera. Da bi se otkrili svi nedostaci softvera, osoba koja vrši testiranje mora da isproba sve moguće kombinacije ulaza. Pošto je prostor mogućih ulaza programa veliki, bira se njegov podskup. Veliki nedostatak ove strategije je i što osoba koja testira softver ne može da zna koliki je procenat softvera testiran. Takođe, postoji opasnost da neke putanje izvršavanja nikad ne podlegnu testiranju. Sa druge strane, ovakav pristup testiranja pomaže testeru da izabere konačne podskupove ulaza (test primera) na osnovu kojih će se pronaći što veći broj grešaka [20, 16].

Strategije testiranja bele kutije (eng. white-box testing) su strategije kod kojih se testiranje zasniva na poznavanju unutrašnjih putanja, strukture i implementacije koda. Informacije dobijene na osnovu gledanja koda programa koriste se da bi se donela odluka o tome šta testirati i kako pristupiti testiranju i sami testovi se generišu na osnovu programskog koda. Nazivaju se i strategijama strukturnog testiranja (eng. structural testing) zato što se testovi kreiraju i izvršavaju na osnovu posmatranja strukture koda. Najčešći primer ovakvog testiranja su jedinični testovi (testovi jedinica koda). Ideja ove strategije je da se napravi paralela sa strategijom crne kutije pri čemu se test primeri dobijaju iz strukture softvera. Osnovne strategije strukturnog testiranja zasnivaju se na kriterijumu pokrivenosti koda (eng. code coverage). Pokrivenost se može računati prema broju izvršenih putanja kroz program, broju instrukcija, broju grana nakon svake instrukcije grananja ili kombinacijom prethodnih kriterijuma. Smatra se da je pomenuta paralela sa strategijom crne kutije iscrpno testiranje putanja kroz program) [16, 21, 19, 20].

Slično strategijama crne kutije i ovde postoje nedostaci: (1) broj jedinstvenih logičkih putanja kroz program može biti ogroman i (2) čak iako se testiraju sve putanje moguće je da program sadrži greške. Ovo se može desiti iz jednog od tri razloga: (1) iscrpno testiranje putanja ne daje garanciju da program

zadovoljava specifikaciju (pri pisanju programa koji uređuje neke vrednosti u rastućem poretku moguće je napraviti grešku i uređenje vršiti u opadajućem poretku), (2) moguće je da program ne radi kako treba zato što mu nedostaje neka putanja (a testiranje ne može da detektuje nedostajuće putanje) i (3) greške osetljive na podatke (eng. data sensitive errors) mogu ostati neotkrivene (na primer, kod poređenja da li je razlika dva broja manja od neke vrednosti, računanje $a - b < c$ će sadržati greške zato što zapravo treba posmatrati apsolutnu vrednost razlike). Još jedan nedostatak je što osoba koja testira program, da bi isti razumela, mora da poseduje znanje iz oblasti programiranja. Prednost ove strategije je što se identifikuju i testiraju sve putanje kroz softver [19, 20, 16].

Strategije testiranja sive kutije (eng. gray-box testing) predstavljaju mešavinu prethodnih strategija u kojoj se na softver i dalje gleda kao na crnu kutiju s tim što se „zaviri" u kôd programa tj. koristi se i specifikacija i programski kôd. Razlika u odnosu na strategije testiranja bele kutije je ta da se gleda jednostavniji, skript kôd, koji se ne gleda toliko detaljno [19, 21].

3.4 Statička verifikacija softvera

Statička verifikacija softvera je provera njegove ispravnosti na osnovu koda, bez njegovog izvršavanja. Analiza koda može biti od strane ljudi ili automatizovana.

Neodlučivost halting problema (problem ispitivanja ispravnosti programa) predstavlja ograničenje za automatizaciju provere ispravnosti programa. Posledica neodlučivosti je da se osobine polaznog sistema mogu verno opisati neodlučivim teorijama, ali proces dokazivanja nije moguće automatizovati baš zbog neodlučivosti teorija. Rešenje ovog problema je u korišćenju interaktivnih dokazivača teorema (eng. interactive theorem provers) kao što su Isabelle i Coq. Korišćenjem ovih sistema moguće je pouzdano interaktivno dokazati ispravnost programa. Proverom samog dokaza dodatno se obezbeđuje najviši stepen pouzdanosti i sigurnosti u dokaz ispravnosti. Problem sa ovim sistemima je što zahtevaju ljudski rad zbog čega se ne koriste za čitave programe, već za njihove delove ili kompleksne osobine [19]. Potpuna automatizacija procesa verifikacije moguća je ako se polazni sistem opiše nekom odlučivom teorijom, ali samo pod pretpostavkom neograničenog vremena. Neograničeno vreme, kao i bilo koji drugi neograničeni resurs u praksi nisu mogući.

Svaki konačan opis u okviru odlučive teorije je, u opštem slučaju, samo aproksimacija ponašanja polaznog sistema. Posledice aproksimacija su neprecizna analiza jer opis polaznog sistema nije veran i analiza koju za neke programe nije moguće završiti zbog nedostatka resursa ili u konačnom vremenu. Kvalitet analite tj. dobijanje preciznijih rezultata u konačnom vremenu i sa konačnim resursima, može se poboljšati dodavanjem obeležja (eng. annotations). Obeležjima se dodaju informacije koje je teško ili nemoguće automatski izvesti. Takođe se dodaju i informacije koje je moguće automatski izvesti ali se dodavanjem tih informacija štedi na vremenu potrebnom za analizu. Obeležjima se najčešće zadaju invarijante petlji, kao i preduslovi i postuslovi funkcija [19]. Potpuno automatizovani pristupi verifikaciji programa su simboličko izvršavanje, apstraktna interpretacija i proveravanje modela.

3.4.1 Simboličko izvršavanje

Simboličko izvršavanje (eng. symbolic execution) predstavlja analizu programa u kojoj se umesto konkretnih prate simboličke vrednosti promenljivih. Ovakvom analizom se kreiraju simbolički izrazi koji opisuju određene putanje kroz program što omogućava istovremenu proveru svih ulaza koji prate istu putanju. Simboličko izvršavanje istovremeno ispituje veliki broj putanja izvršavanja bez potrebe za konkretnim ulaznim parametrima. Umesto toga, vrši se apstrakcija ulaznih vrednosti simbolima nakon čega se koriste rešavači ograničenja (eng. constraint solvers) da bi se proizvele konkretne vrednosti ulaza koje dovode do grešaka. Simboličko izvršavanje nastaje sredinom 1970-ih sa ciljem otkrivanja da li se pomoću softvera mogu narušiti određena svojstva programa. Neki od interesantnih aspekata bi bili nepostojanje deljenja nulom i nepostojanje pokušaja dereferenciranja NULL pokazivača. Za razliku od konkretnog izvršavanja gde se program pokreće sa konkretnim vrednostima ulaza i ima jednu putanju, simboličkim izvršavanjem je moguće istovremeno izvršavanje većeg broja potencijalnih putanja (koje nastaju u zavisnosti od vrednosti ulaza). Zbog toga je ključna ideja omogućiti programu da koristi simboličke vrednosti [19, 12].

Ako simboličko izvršavanje pokaže da neka putanja zadovoljava uslove ispravnosti programa tada svi ulazi koji prolaze putanju zadovoljavaju uslove programa čime se znatno dobija na efikasnosti provere u odnosu na tehnike dinamičke analize. Problem sa ovim pristupom je broj mogućih putanja koji je često prevelik da bi

se u potpunosti pretražio. Zbog toga simboličko izvršavanje svoju primenu češće nalazi u pronalaženju grešaka nego u verifikaciji programa. Veoma važan faktor koji utiče na brzinu otkrivanja greške u programu je redosled ispitivanja putanja. Ako se greška javlja u putanji koja se ispituje prva vreme otkrivanja greške je minimalno. U slučaju da se greška javlja u putanji koja se poslednja analizira vreme otkrivanja greške će biti veliko, a moguće je i da se greška uopšte ne pronađe. Ako postoji veliki broj putanja kroz program može doći do isteka vremena predviđenog za analizu pre nego što se greška otkrije [19].

3.4.2 Apstraktna interpretacija

„*Apstraktna interpretacija* (eng. abstract interpretation) tehnika je aproksimacije formalne semantike programa, tj. matematičkog modela ponašanja programa [19]". „Apstraktna interpretacija obuhvata procenu ponašanja programa na nekom apstraktnom domenu, sa ciljem dobijanja približnog rešenja [17]". Apstraktni domen je aproksimacija reprezentacije skupova konkretnih vrednosti, a za slikanje konkretnih vrednosti u vrednosti apstraktnog domena koristi se funkcija apstrakcije [17]. Semantika programa je opisana domenom D_c i relacijama nad domenom pri čemu se relacije mogu menjati tokom izvršavanja naredbi programa. Provera da li neko svojstvo važi nad domenom D_c može biti problem u slučaju velikih programa jer veliki programi znače velike domene i veliki broj mogućih putanja kroz program. Dodatan problem predstavlja neodlučivost koja se može javiti u raznim kontekstima. Kao potencijalno rešenje nameće se aproksimacija domena D_c apstraktnim domenom D_a. Apstraktni domen ne može biti precizan kao konkretni domen, ali može dati informacije o nekim svojstvima domena. Sa druge strane, apstrakcijom domena se mogu izgubiti važne informacije. Primer jedne apstrakcije bila bi apstrakcija beskonačnog domena celih brojeva koji se menja skupom znakova brojeva +, -, 0 . Ovakva apstrakcija bi verno prikazala znak rezultata množenja dva cela broja ako su poznati samo znaci operanada. Ako je informacija od značaja znak rezultata sabiranja dva cela broja, pri čemu su poznati samo znaci operanada, ovakva apstrakcija se u nekim slučajevima pokazuje kao nedovoljna (znak rezultata sabiranja brojeva -3 i 5 je (+), a znak rezultata sabiranja brojeva -7 i 5 (-)) [19].

3.4.3 Proveravanje modela

Proveravanje modela (eng. model checking) je metod verifikacije u kojem se sistem, hardverski ili softverski, koji je potrebno verifikovati opisuje konačnim aparatom (stanjima i tranzicijama između stanja), a specifikacija se zadaje u terminima temporalne logike (specifikacija je opisana kao logička formula). U temporalnoj logici je moguće razmatrati uređenje događaja u toku vremena. Na primer, moguće je da trenutno netačno svojstvo postane tačno u nekom trenutku u budućnosti. Stanje modela čine vrednosti promenljivih i stanja steka i hipa, a tranzicije opisuju kako program prelazi iz jednog stanja u drugo stanje. Metodi zasnovani na proveravanju modela ispituju stanja programa do kojih se može doći. Nakon što je sistem opisan, stanja automata se obilaze sa ciljem dokazivanja uslova specifikacije. Proveravanje modela metodom grube sile istražuje sva moguća stanja sistema na sistematičan način, slično programu za igranje šaha. Na ovaj način je moguće dokazati da sistem zadovoljava neko svojstvo. Međutim, sa trenutno raspoloživim procesorima i memorijama ispitivanje najvećih mogućih prostora stanja predstavlja izazov. Najbolji programi za proveravanje modela mogu da rade sa prostorima stanja reda 10^8 do 10^9, a korišćenjem pametnih algoritama i specijalnih struktura za konkretne probleme prostor stanja može drastično da se uveća (10^{20} do 10^{476}!). Metod proverava da li sva relevantna stanja sistema zadovoljavaju željena svojstva. Ako je prostor stanja konačan metod će se izvršiti u konačnom vremenu. U slučaju neuspeha tj. ako se pronađe stanje koje narušava svojstvo koje se razmatra, generiše se kontraprimer koji definiše putanju izvršavanja koja vodi od inicijalnog stanja sistema do stanja kojim je svojstvo koje se proverava narušeno. Mana proveravanja modela je eksplozivan porast stanja tj. broj mogućih stanja eksponencijalno raste sa porastom broja promenljivih (komponenti sistema). Zbog toga se koristi metod proveravanja ograničenih modela [19, 17, 15, 11].

Proveravanje ograničenih modela (eng. bounded model checking) je formalna tehnika verifikacije koja se mahom koristi u industriji poluprovodnika. Proveravanje ograničenih modela vrši brzu pretragu prostora stanja i za određene probleme daje znatno poboljšanje performansi izračunavanja u odnosu na metod proveravanja modela. Uspeh tehnike zasnovan je na velikim mogućnostima iskaznih SAT rešavača (eng. propositional SAT solvers). Ovaj metod se zasniva na ograničavanju dužine putanje stanja koja se proverava. Ako se za zadatu vrednost dužine ne pronađe greška moguće je povećati vrednost tako da se provera nastavlja dok se ne pronađe

greška, model ne postane prevelik za analizu ili vrednost dužine ne dostigne gornju granicu dužine mogućih putanja čime je model verifikovan [19, 17, 15].

Glava 4

Zaključak

Bibliografija

[1] A brief timeline of python. https://python-history.blogspot.com/2009/01/brief-timeline-of-python.html.

[2] History of python. https://www.python-course.eu/python3_history_and_philosophy.php.

[3] Matplotlib.org. https://matplotlib.org/.

[4] pandas: powerful Python data analysis toolkit. https://pandas.pydata.org/pandas-docs/stable/.

[5] scikit-learn: Machine Learning in Python. http://scikit-learn.org/stable/index.html.

[6] Scipy.org. https://www.scipy.org/.

[7] Scipy.org: Quickstart tutorial. https://docs.scipy.org/doc/numpy/user/quickstart.html.

[8] Scipy.org: Scientific computing tools for python. https://www.scipy.org/about.html.

[9] What is python? executive summary. https://www.python.org/doc/essays/blurb/.

[10] Y2k bug. https://www.nationalgeographic.org/encyclopedia/Y2K-bug/.

[11] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press Cambridge, Massachusetts London, England, 2008.

[12] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3), 2018.

[13] Jason Brownlee. How machine learning algorithms work (they learn a mapping of input to output). `https://machinelearningmastery.com/how-machine-learning-algorithms-work/`.

[14] Jason Brownlee. Logistic regression for machine learning. `https://machinelearningmastery.com/logistic-regression-for-machine-learning/`.

[15] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, Jul 2001.

[16] Lee Copeland. *A Practitioner's Guide to Software Test Design*. Artech House, 2004.

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

[18] Mladen Nikolić i Anđelka Zečević. Mašinsko učenje. 2018.

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

[20] Glenford J. Myers. *The Art of Software Testing*. John Wiley & Sons, Inc., Hoboken, New Jersey, 2 edition, 2004.

[21] Ron Patton. *Software Testing*. Sams Publishing, 800 E. 96th St., Indianapolis, Indiana, 46240 USA, 2001.

[22] Prof. Dr. Holger Schlingloff. Software testing I. Humboldt-Universität zu Berlin and Fraunhofer Institute of Computer Architectureand Software Technology, November 2006.

[23] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining*. Addison-Wesley Longman Publishing Co., Inc., 2005.

Biografija autora