PANEVROPSKI APEIRON UNIVERZITET
Banja Luka

Redovne studije
Studijski program „Inženjering informacionih tehnologija“

Smjer
„Programiranje i softversko inženjerstvo“

„PRIMJENA TEHNIKA MAŠINSKOG UČENJA U OTKRIVANJU DEFEKATA SOFTVERA“

(DIPLOMSKI RAD)

Mentor: Prof. dr Željko Stanković

Student: Sanja Savić
Broj indeksa: 9-21/FITIIT1-pro-240

Banja Luka, jul 2025.

SADRŽAJ

1. UVOD 5
2. OSNOVE TESTIRANJA SOFTVERA 7
2.1 Definicija i vrste softverskih defekata 7
2.2 Proces testiranja softvera 8
2.2.1 Životni ciklus defekta (Bug Life Cycle) 11
2.2.2 Praktičan primjer procesa testiranja 12
2.2.3 Uloga alata u procesu testiranja 14
2.3 Manualno ili automatsko testiranje 16
2.4 Tradicionalni alati i metode za otkrivanje defekata 17
3. MAŠINSKO UČENJE U FUNKCIJI TESTIRANJA 20
3.1 Kategorije mašinskog učenja 20
3.2 Primjena mašinskog učenja 21
3.2.1 Primjena u proizvodnim procesima 22
3.2.2 Složeni senzorski sistemi i nedestruktivna ispitivanja 22
3.2.3 Primjena u softverskom inženjeringu 23
3.3 Prednosti, izazovi i buduće perspektive mašinskog učenja 24
3.3.1 Prednosti primjene mašinskog učenja 24
3.3.2 Ograničenja primjene mašinskog učenja 25
3.3.3 Budućnost primjene mašinskog učenja 26
3.4 Uporedna analiza algoritama: Random Forest, SVM i Neural Networks 27
3.4.1 Random Forest 27
3.4.2 Support Vector Machines (SVM) 28
3.4.3 Neuronske mreže (Neural Networks) 29
4. INTEGRACIJA MAŠINSKOG UČENJA I PYTHON-a 31
4.1 Upotreba Python-a za mašinsko učenje 31
4.2 Najvažnije biblioteke 32
4.3 Primjeri jednostavnih ML zadataka u Pythonu 33
Primjer 1: Klasifikacija – Prepoznavanje vrste cvijeta pomoću Iris skupa podataka 33
Primjer 2: Regresija – Predviđanje cijene kuće na osnovu kvadrature 34
Primjer 3: Klasterovanje – Grupisanje podataka pomoću KMeans algoritma 36

4.4 Prednosti Pythona u razvoju i testiranju softvera 37
5. PRAKTIČNA IMPLEMENTACIJA MODELA U PYTHON-u 40
5.1 Opis i priprema skupa podataka 41
5.1.1 Opis skupa podataka CM1 41
5.1.2 Priprema podataka za analizu 42
5.2 Treniranje i evaluacija modela 44
5.2.2 Implementacija i treniranje modela 44
5.2.3 Testiranje modela 44
5.2.4 Evaluacija performansi 45
5.2.5 Interpretacija rezultata 45
5.2.6 Prednosti i ograničenja pristupa 46
5.3 Prikaz rezultata 46
6. ZAKLJUČAK 49
LITERATURA I TABELA SLIKA 51

SPISAK SKRAĆENICA:

ISTQB (International Software Testing Qualifications Board - Međunarodni odbor za testiranje softvera)

CI (Continuous Integration) **-** kontinuirana integracija

CD (Continuous Delivery**) -** kontinuirana isporuka

UX (User Experience) – korisničko iskustvo

PDV- Porez na dodatu vrijednost

ML (Machine Learning) – mašinsko učenje

NDT (Non-Destructive Testing) ne destruktivne metode ispitivanja

IoT (Internet of Things) – Internet stvari

DNN (Deep Neural Networks) - duboke neuronske mreže

CNN (Convolutional Neural Networks) - konvolucione neuronske mreže

SVM (Support Vector Machines) - mašine potpornog vektora

RNN (Recurrent Neural Networks) - rekurentne neuronske mreže

1. UVOD

U savremenom softverskom inženjeringu, detekcija defekata predstavlja jedan od najvažnijih koraka u procesu razvoja i održavanja softverskih sistema. Softverski defekti, koji se javljaju tokom različitih faza razvoja, uključuju greške u dizajnu, implementaciji, testiranju ili čak održavanju softvera. Iako su softverski defekti gotovo neizbježni u složenim projektima, njihova pravovremena identifikacija i ispravljanje su od suštinskog značaja za obezbjeđivanje visokog kvaliteta i pouzdanosti softverskih proizvoda. Neprepoznavanje ili zanemarivanje defekata može dovesti do ozbiljnih problema kao što su kvarovi u radu sistema, gubitak podataka, sigurnosni propusti, kao i do smanjenja zadovoljstva korisnika, što sve zajedno može imati značajne ekonomske i reputacione posljedice.

Tradicionalni pristupi testiranju softvera uglavnom se oslanjaju na manuelnu analizu i izvođenje testova, kao i na upotrebu statičkih i dinamičkih alata za pronalaženje grešaka. Međutim, kako se složenost softverskih sistema povećava, a rokovi za isporuku sve više skraćuju, ovi pristupi pokazuju svoje limite. Manuelno testiranje je često skupo, vremenski zahtjevno i podložno ljudskim greškama, dok postojeći alati ne mogu uvijek da detektuju skrivene ili kompleksne defekte, naročito kada se radi o velikim bazama koda i brojnoj grupi softverskih modula. Sve to dovodi do potrebe za razvojem novih, automatizovanih i inteligentnijih metoda koje mogu da analiziraju velike količine podataka, uče iz prošlih grešaka i predviđaju potencijalne probleme prije nego što se pojave u produkcionom okruženju.

U ovom kontekstu, mašinsko učenje se pojavljuje kao perspektivna tehnologija sa velikim potencijalom za unapređenje procesa testiranja softvera. Mašinsko učenje je oblast vještačke inteligencije koja se bavi razvojem algoritama i modela koji mogu da prepoznaju obrasce u podacima i na osnovu toga donose odluke ili pravila bez potrebe za eksplicitnim programiranjem svakog koraka. Za razliku od tradicionalnog programiranja gdje se precizno definišu pravila ponašanja, mašinsko učenje omogućava računarima da sami “uče” iz dostupnih podataka i da se prilagođavaju novim situacijama na osnovu stečenog iskustva. Ovo je naročito korisno u oblasti softverskog inženjeringa, gdje kompleksnost i dinamičnost sistema zahtijevaju fleksibilne i samostalne pristupe za identifikaciju i klasifikaciju defekata.

Primjena mašinskog učenja u testiranju softvera obuhvata različite aspekte, uključujući automatsku detekciju defektnih modula, predikciju mjesta u kodu gdje je najverovatnije da će se pojaviti greške, analizu uzroka i posljedica defekata, kao i optimizaciju rasporeda testova kako bi se maksimizirala efikasnost testiranja. Ovakvi sistemi omogućavaju da se umanji ljudski faktor, smanji vrijeme potrebno za pronalaženje grešaka, i poveća pouzdanost softverskih proizvoda. Implementacija modela mašinskog u ovom domenu može da rezultira značajnim uštedama resursa i unapređenjem ukupnog kvaliteta softvera, što je od velikog značaja za kompanije koje razvijaju kritične softverske sisteme, kao i za krajnje korisnike.

Jedan od ključnih faktora koji je doveo do brze popularizacije mašinskog učenja u oblasti testiranja softvera jeste dostupnost moćnih programskih jezika i alata. Među njima, Python zauzima posebno mjesto zbog svoje jednostavnosti, fleksibilnosti i bogatog ekosistema biblioteka namjenjenih za obradu podataka i implementaciju modela mašinskog učenja. Biblioteke poput pandas za rad sa podacima, scikit-learn za implementaciju različitih algoritama mašinskog učenja, Keras i TensorFlow za rad sa neuronskim mrežama, kao i matplotlib i seaborn za vizualizaciju podataka, predstavljaju snažne alate koji omogućavaju istraživačima i inženjerima da brzo prototipiziraju i testiraju modele. Korišćenjem Pythona moguće je efikasno integrisati mašinsko učenje u postojeće procese razvoja softvera, čime se postiže automatizacija i povećava produktivnost.

Pored toga, Pythonova zajednica i veliki broj dostupnih resursa i tutorijala olakšavaju usvajanje i primjenu novih tehnologija, što dodatno podstiče implementaciju rješenja mašinskog učenja u industrijskim i istraživačkim okruženjima. Fleksibilnost ovog jezika omogućava povezivanje sa različitim bazama podataka, alatima za verzionisanje koda, i sistemima za kontinuiranu integraciju, čime se podržava kompletan životni ciklus razvoja softvera sa ugrađenim inteligentnim testiranjem.

Sve navedeno ukazuje da je primjena mašinskog učenja u testiranju softvera ne samo tehnički izvodljiva, već i strateški značajna. U eri digitalizacije i velikih softverskih projekata, automatizovani sistemi koji mogu da unaprijede detekciju defekata, predviđaju probleme i preporučuju poboljšanja, postaju neophodni za održavanje konkurentnosti i kvaliteta proizvoda. Dalji razvoj i istraživanja u ovoj oblasti otvaraju nove mogućnosti za unapređenje sigurnosti, stabilnosti i efikasnosti softverskih sistema širom različitih industrijskih sektora.

Cilj ovog rada jeste da se, kroz teorijsku obradu i praktičnu implementaciju, pokaže na koji način se mašinsko učenje može koristiti za detekciju softverskih defekata. Konkretno, demonstriraće se upotreba Naivnog Bajesovog klasifikatora nad javno dostupnim skupom podataka (NASA CM1), uz analizu dobijenih rezultata i diskusiju o ograničenjima i mogućnostima ovog pristupa.

Značaj rada ogleda se u sve većem zahtjevu za automatizacijom testiranja, posebno u projektima koji zahtijevaju visoku pouzdanost, kao što su ugrađeni sistemi, sistemi za upravljanje transportom, avijacijom, medicinskom opremom i slično. Takođe, rad ima edukativnu vrijednost jer povezuje teorijske koncepte iz oblasti mašinskog učenja sa praktičnom primjenom u realnom okruženju razvoja softvera.

U radu se najprije obrađuju osnovni pojmovi iz oblasti testiranja softvera, s naglaskom na vrste defekata i metode njihove detekcije. Potom se daje pregled mašinskog učenja i njegove primjene u ovoj oblasti, uz uporednu analizu nekoliko poznatih algoritama. U centralnom dijelu rada predstavljena je praktična implementacija modela u programskom jeziku Python, korišćenjem biblioteka kao što su pandas, scikit-learn i druge. Analiza se vrši na stvarnim podacima iz NASA-inog projekta, čime se obezbjeđuje realističan okvir za testiranje tačnosti i korisnosti pristupa. Na kraju rada izloženi su zaključci i preporuke za buduća istraživanja i primjene u industriji.

2. OSNOVE TESTIRANJA SOFTVERA

Testiranje softvera predstavlja sistematičan proces evaluacije softverskog proizvoda sa ciljem identifikacije grešaka, provjere usklađenosti sa zahtjevima i osiguranja kvaliteta. Kao sastavni dio životnog ciklusa razvoja softvera, testiranje omogućava otkrivanje problema u ranim fazama razvoja, čime se značajno smanjuje rizik od ozbiljnih defekata u produkcionom okruženju. Efikasno testiranje doprinosi stabilnosti, pouzdanosti i bezbjednosti softverskog sistema, čime se direktno utiče na zadovoljstvo krajnjih korisnika i reputaciju proizvođača softvera.

2.1 Definicija i vrste softverskih defekata

Softverski defekti predstavljaju centralni izazov u procesu razvoja i održavanja softvera, jer njihova prisutnost može uzrokovati odstupanja u funkcionalnosti, performansama ili bezbjednosti sistema. Defekt se u najširem smislu definiše kao greška u kodu, dizajnu, specifikaciji ili logici koja može dovesti do pogrešnog ponašanja softverske komponente ili cjelokupnog sistema. Važno je naglasiti da defekti ne moraju uvijek odmah prouzrokovati kvar (failure[1]), ali predstavljaju potencijalnu prijetnju kvalitetu softverskog proizvoda. (Pressman, Roger S.; Maxim, Bruce R., 2014)

Identifikacija i klasifikacija defekata od suštinske su važnosti za efikasno upravljanje kvalitetom. U nastavku su predstavljene najčešće vrste softverskih defekata:

- Sintaksne greške – nastaju usred nepravilnog korišćenja sintakse programskog jezika, i obično se detektuju tokom kompajliranja.
- Logičke greške – proizlaze iz pogrešne implementacije algoritama, pri čemu softver funkcioniše, ali ne daje tačne rezultate.
- Greške u dizajnu – nastaju u fazi projektovanja sistema, i obično su posljedica neadekvatnog modeliranja ili pogrešnog tumačenja zahtjeva.
- Greške u interfejsu – javljaju se u komunikaciji između različitih softverskih modula ili između softvera i korisnika.
- Greške u performansama – uključuju probleme kao što su sporo izvršavanje, curenje memorije ili zagušenje sistema.
- Greške u validaciji i verifikaciji – vezane su za nepoštovanje definisanih uslova i provjera koje garantuju ispravno ponašanje sistema.

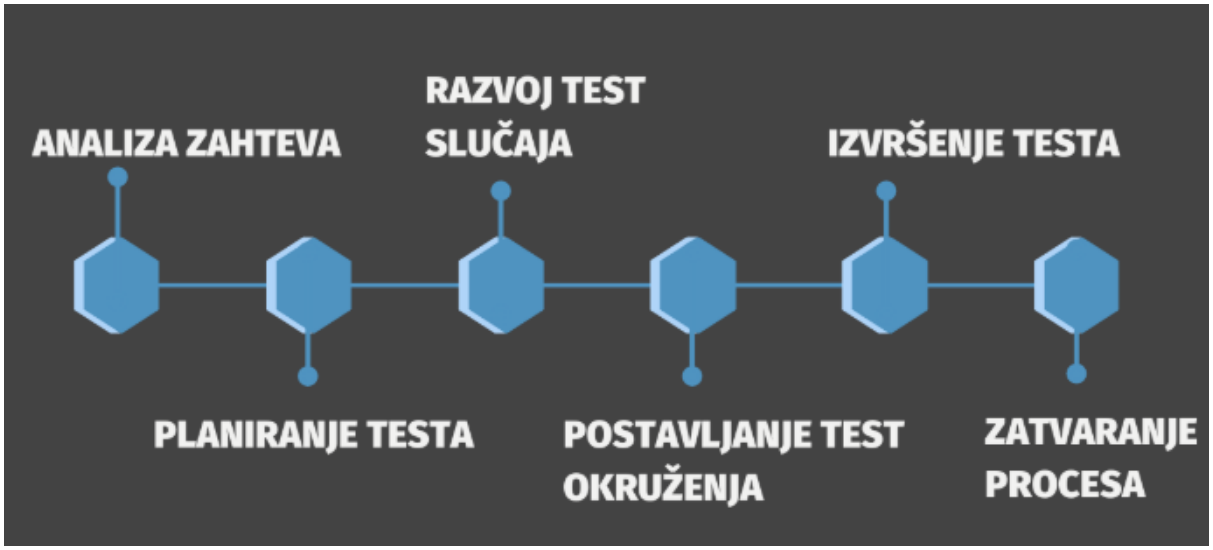
Klasifikacija defekata takođe može biti zasnovana na njihovoj ozbiljnosti (kritične, visoke, srednje, niske), učestalosti ili fazi u kojoj su otkriveni. Ova klasifikacija omogućava prioritetizaciju u procesu otklanjanja defekata i optimizaciju test strategije.

Razumijevanje prirode i uzroka defekata ključno je za izgradnju efikasnog test okruženja, kao i za unapređenje procesa razvoja kroz prevenciju ponavljanja istih grešaka u budućnosti.

[1] Failure (engl.) označava neuspjeh ili kvar u radu softverskog sistema, to jeste odstupanje od očekivanog ponašanja sistema pri izvršavanju.

2.2 Proces testiranja softvera

Proces testiranja softvera je strukturisan skup aktivnosti koje se sprovode kako bi se obezbijedilo da softverski proizvod zadovoljava zahtjeve korisnika, funkcioniše ispravno i da je dovoljno robustan i bezbjedan za upotrebu u predviđenom okruženju. Efikasan proces testiranja nije ad-hoc aktivnost[2], već organizovana procedura koja uključuje planiranje, dizajniranje, implementaciju, izvršavanje i evaluaciju testova kao što je prikazano na slici 1.



Slika 1. Dijagram procesa testiranja softvera (https://www.helloworld.rs/blog/Kako-izgleda-proces-testiranja-softvera/12291)

Prema ISTQB standardima[3], proces testiranja se najčešće organizuje kroz sljedeće osnovne faze:

1. Planiranje testiranja (Test Planning)

Planiranje testiranja predstavlja prvu i jednu od najvažnijih faza u cjelokupnom procesu. U ovoj fazi definišu se jasni ciljevi testiranja, koji su u skladu sa zahtjevima projekta i očekivanjima korisnika. Planiranje obuhvata i precizno određivanje obima test aktivnosti, što podrazumijeva identifikaciju funkcionalnosti i komponenti softvera koje će biti testirane, kao i onih koje nisu uključene u testiranje.

Pored toga, u planu se definišu resursi potrebni za uspješnu realizaciju testiranja, uključujući ljudske kapacitete, potrebne alate, hardverske i softverske uslove, kao i budžet. Posebna pažnja posvećuje se vremenskom okviru, gdje se uspostavljaju rokovi za svaku fazu testiranja, omogućavajući koordinisani rad unutar projektnih timova.

[2] Ad-hoc aktivnost označava privremenu, spontanu ili jednokratnu aktivnost koja se obavlja van uobičajenih planiranih procedura, često kao odgovor na specifičan, neposredan potreban zadatak ili problem.
[3] ISTQB (engl. International Software Testing Qualifications Board - Međunarodni odbor za testiranje softvera) standardi predstavljaju globalno prihvaćene smjernice i najbolje prakse za testiranje softvera, koje definišu terminologiju, procese, tehnike i metodologije u oblasti testiranja, kako bi se osigurala dosljednost, kvalitet i efikasnost u softverskom testiranju.

Jedan od ključnih aspekata planiranja je definisanje kriterijuma za izlaz iz testiranja, koji predstavljaju uslove koje softver mora ispuniti da bi se testiranje smatralo uspješno završenim. Ovi kriterijumi mogu uključivati minimalnu pokrivenost testovima, broj dozvoljenih defekata, ili stabilnost sistema tokom određenog vremenskog perioda.

Takođe, u ovoj fazi identifikuju se potencijalni rizici vezani za kvalitet softvera, kao i strategije za njihovo otklanjanje ili ublažavanje. Upravljanje rizicima omogućava pravovremeno reagovanje i minimiziranje negativnih uticaja na projekat.

Plan testiranja dokumentuje se u obliku zvaničnog dokumenta, test plan, koji služi kao vodič i referenca za sve naredne aktivnosti u procesu testiranja. Njegova detaljna i precizna izrada od ključnog je značaja za uspjeh cjelokupnog projekta.

2. Dizajniranje testova (Test Design)

Nakon planiranja, sljedeća faza obuhvata kreiranje konkretnih test slučajeva i test scenarija. Ovaj proces se zasniva na detaljnoj analizi specifikacija i zahtjeva softverskog sistema. Cilj je da se kroz dobro definisane testove obuhvate svi relevantni aspekti funkcionalnosti, performansi i sigurnosti softvera.

Dizajn testova uključuje izbor ulaznih podataka koji će biti korišćeni za izvršavanje testova, kao i definisanje očekivanih rezultata, što omogućava precizno upoređivanje stvarnog ponašanja sistema sa željenim ishodima. Ovim se direktno obezbjeđuje detekcija odstupanja i potencijalnih defekata.

Test slučajevi mogu biti različitih tipova:

- Funkcionalni testovi provjeravaju da li softver ispravno izvršava definisane funkcionalnosti u skladu sa zahtjevima.
- Nefunkcionalni testovi procjenjuju karakteristike poput performansi, sigurnosti, upotrebljivosti i pouzdanosti.
- Regresioni testovi služe za provjeru da li nove izmjene u kodu nisu negativno uticale na postojeće funkcionalnosti.

Važno je da test slučajevi budu jasni, mjerljivi i ponovljivi, kako bi se mogli koristiti u automatizovanim testovima ili prilikom manuelnog izvršavanja od strane QA (Quality Assurance) tima.

3. Implementacija i priprema testova (Test Implementation and Setup)

U ovoj fazi dolazi do konkretne pripreme i implementacije testova koji su dizajnirani u prethodnoj fazi. To uključuje kreiranje test skripti, naročito kada je riječ o automatizaciji testiranja, kao i podešavanje potrebnog test okruženja.

Test okruženje mora biti adekvatno konfigurisan i izolovan od produkcionog sistema, kako bi rezultati testiranja bili pouzdani i ponovljivi. Pored tehničke konfiguracije, neophodna je i priprema test podataka koji se koriste tokom testiranja, sa ciljem da što vjerodostojnije simuliraju realne uslove rada softvera.

Ova faza podrazumijeva i validaciju alata i sistema koji će se koristiti u toku testiranja, uključujući alate za automatizaciju, praćenje defekata i izvještavanje. Time se obezbjeđuje da sve komponente sistema funkcionišu u skladu sa zahtjevima procesa.

Automatizacija testova u okviru ove faze postaje ključni element, naročito u agilnim i DevOps okruženjima [4] gdje je neophodno brzo i često izvršavanje testova. Kreirane skripte omogućavaju kontinuiranu provjeru koda i smanjuju manuelni rad, povećavajući efikasnost i pouzdanost testiranja.

4. Izvršavanje testova (Test Execution)

Faza izvršavanja testova podrazumijeva realizaciju definisanih test slučajeva u odgovarajućem test okruženju. Testeri ili automatizovani sistemi pokreću testove i prate rezultate njihovog izvršenja. Svaki test se ocjenjuje kao uspješan ili neuspješan na osnovu poređenja stvarnih rezultata sa očekivanim ishodima.

U toku izvršenja, svi uočeni defekti i anomalije se bilježe u sistem za praćenje grešaka. Važno je detaljno dokumentovati ne samo neuspješne testove, već i one koji su blokirani usljed tehničkih problema ili drugih ograničenja.

Pravovremeno i precizno evidentiranje rezultata omogućava brzo reagovanje, analizu uzroka defekata i njihovu korekciju u narednim iteracijama razvoja. Efikasna komunikacija između test tima i razvojnih inženjera je od ključnog značaja za ubrzanje procesa ispravki i ponovnog testiranja.

5. Evaluacija i izvještavanje (Test Reporting and Closure)

Nakon izvršenja svih planiranih testova pristupa se evaluaciji ostvarenih rezultata. Cilj je da se procjeni u kojoj mjeri su ispunjeni ciljevi definisani u fazi planiranja. Analiza obuhvata mjerenje kvaliteta softvera, broj i ozbiljnost otkrivenih defekata, kao i procjenu preostalih rizika.

Na osnovu ovih podataka kreiraju se detaljni izvještaji o testiranju, koji služe kao osnova za donošenje odluka o prihvatanju softvera. Izvještaji treba da budu jasni, transparentni i razumljivi svim zainteresovanim stranama, uključujući menadžment i razvojne timove.

U završnoj fazi se formalno zatvara proces testiranja (Test Closure), što podrazumijeva arhiviranje svih test dokumenata, evaluaciju efikasnosti samog procesa i identifikaciju oblasti za buduće poboljšanje. Povratne informacije iz ove faze predstavljaju vrijedan input za unapređenje kvaliteta testiranja u narednim projektima. (International Software Testing Qualifications Board, 2025)

[4] DevOps okruženja predstavljaju objedinjeni pristup razvoju softvera i upravljanju informacionim tehnologijama koji integriše procese, alate i organizacione kulture u cilju unapređenja brzine, efikasnosti i kvaliteta isporuke softverskih rješenja.

2.2.1 Životni ciklus defekta (Bug Life Cycle)

U procesu upravljanja kvalitetom softvera, ključnu ulogu ima praćenje i efikasno rješavanje defekata. Defekt, koji predstavlja grešku ili neusaglašenost u softverskom proizvodu, od trenutka kada je otkriven pa do konačnog zatvaranja prolazi kroz niz jasno definisanih faza poznatih kao životni ciklus defekta. Razumijevanje ovog ciklusa od suštinskog je značaja za koordinaciju između različitih timova, kao i za optimizaciju procesa ispravljanja grešaka, čime se direktno doprinosi ukupnoj pouzdanosti i kvalitetu softverskog rješenja.

Životni ciklus defekta najčešće se modelira kroz sljedeće faze: New → Assigned → Open → Fixed → Retest → Closed, sa mogućnošću da defekt bude Reopened ukoliko problem nije u potpunosti riješen ili se ponovo pojavi. Ove faze predstavljaju standardni tok kojim defekt prolazi u većini savremenih razvojnih i test okruženja, uključujući alate za praćenje grešaka kao što su JIRA, Bugzilla ili Azure DevOps. (Katalon, 2025)

New (Novi): U ovoj početnoj fazi, defekt je prvi put prijavljen i zabilježen u sistemu za praćenje. Najčešće, test inženjer, QA tim ili krajnji korisnik identifikuje nepravilnost i kreira zapis sa osnovnim informacijama o grešci, uključujući opis problema, uslove pod kojima se javlja i eventualno prijedloge za reprodukciju. Ovaj korak je od kritične važnosti jer precizno i jasno prijavljeni defekti omogućavaju efikasniju dalju obradu.

Assigned (Dodijeljen): Nakon prijave, defekt se dodjeljuje konkretnom članu razvojnog tima, obično programeru ili timu zaduženom za određeni dio softvera. Ova dodjela se vrši na osnovu kompetencija, raspoloživosti i prioriteta zadataka. Cilj ove faze je da se osigura da postoji odgovorna osoba koja je preuzela zadatak rješavanja problema.

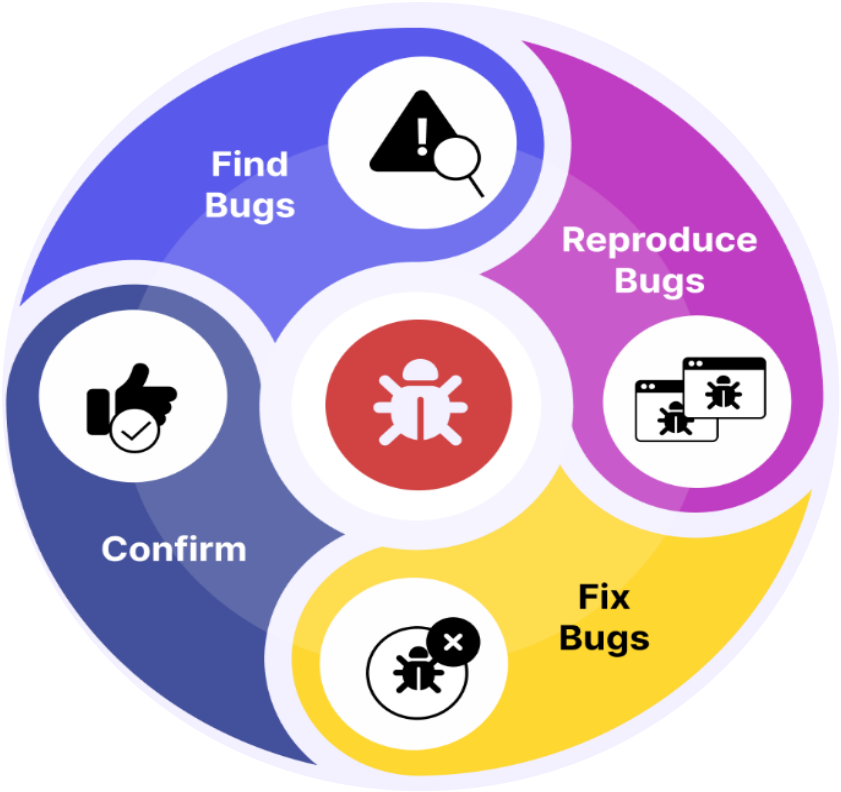
Open (Otvoren): U ovoj fazi programer formalno potvrđuje da je defekt validan i započinje rad na njegovom otklanjanju. Ukoliko se ispostavi da prijavljeni problem nije greška, već možda dio funkcionalnosti ili uslova koji nisu jasno definisani, defekt može biti odbijen ili označen kao nevalidan. Međutim, ukoliko je greška potvrđena, nastavlja se sa koracima neophodnim za popravku.

Fixed (Ispravljen): Nakon što programer implementira korekciju u kodu, defekt se označava kao „ispravljen“. U ovom trenutku, iako je problem navodno riješen, ispravka još nije testirana, te nije poznato da li je greška zaista uklonjena bez dodatnih problema.

Retest (Ponovno testiranje): Test inženjer ili QA tim ponovo izvršava testove koji su prethodno identifikovali defekt, sa ciljem da potvrdi da je ispravka uspješna i da nije došlo do novih neželjenih posljedica. Ova faza je ključna jer osigurava validaciju popravke i doprinosi održavanju stabilnosti softvera.

Closed (Zatvoren): Ukoliko testiranje pokaže da je problem u potpunosti uklonjen i da se ne pojavljuje, defekt se zvanično zatvara. Time se proces rješavanja greške smatra završnim, a defekt više nije predmet daljih aktivnosti.

Reopened (Ponovno otvoren): Ukoliko nakon zatvaranja defekta isti problem ponovo nastane ili ispravka nije adekvatno riješila problem, defekt se može ponovo otvoriti. Ovaj povratak u ciklus omogućava dodatnu analizu i korekciju greške, čime se spriječavaju trajni problemi u softveru.



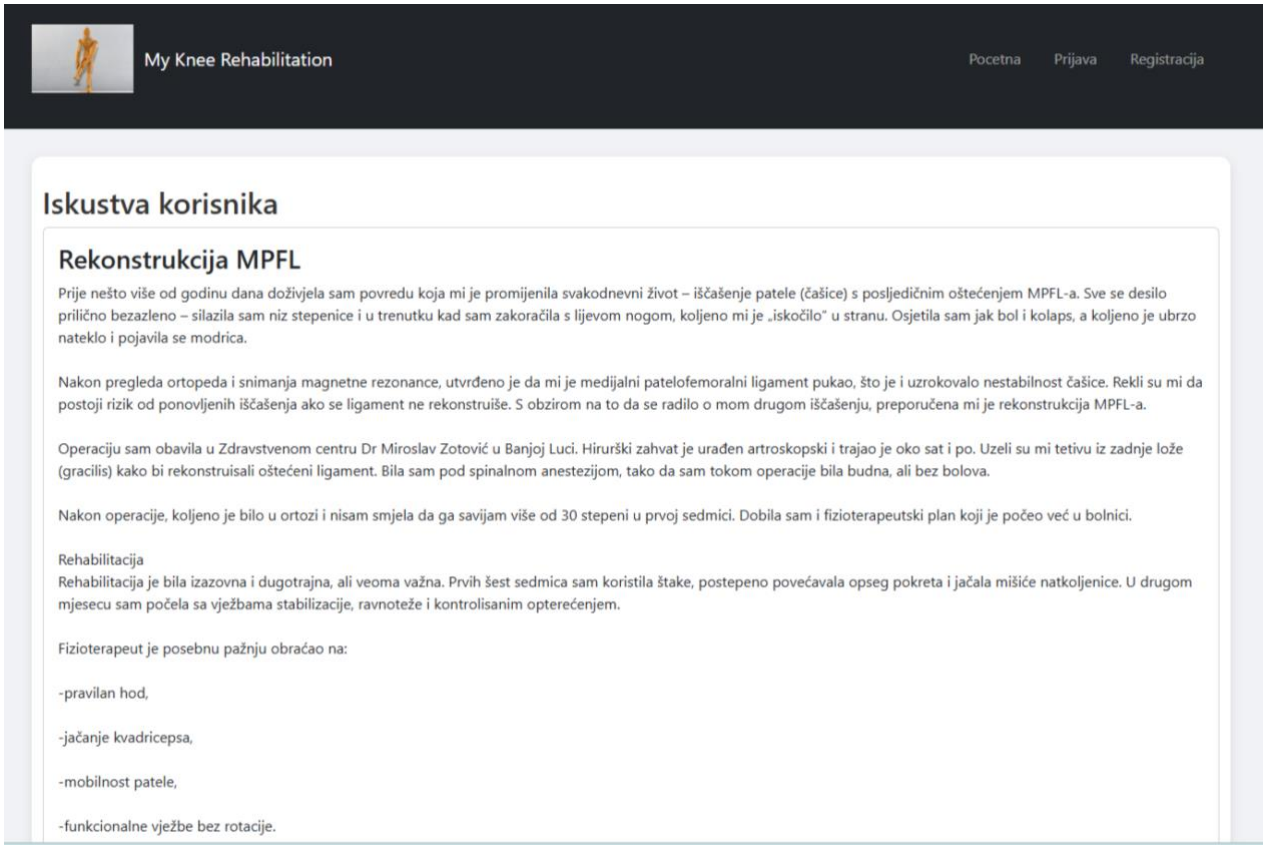
Slika 2. Dijagram životnog ciklusa defekta (https://katalon.com/resources-center/blog/bug-defect-life-cycle)

Važno je napomenuti da u zavisnosti od organizacionih praksi i korišćenih alata, životni ciklus defekta može da sadrži dodatne faze kao što su Deferred (odloženi defekti koji se neće trenutno rješavati), Duplicate (duplikati već prijavljenih grešaka), Rejected (defekti odbijeni kao nevalidni), ili Won't Fix (defekti koji se neće ispravljati zbog različitih razloga poput niskog prioriteta ili rizika).

Pravilno upravljanje životnim ciklusom defekta omogućava transparentnost u radu, bolju komunikaciju među timovima i brže donošenje odluka o prioritetima, što sve zajedno doprinosi efikasnijem razvoju i održavanju softverskih proizvoda visokog kvaliteta.

2.2.2 Praktičan primjer procesa testiranja

U nastavku će biti objašnjen razvoj web platforme na kojoj pacijenti dijele svoja iskustva, kao što je prikazano na slici 3, vezana za operaciju koljena i proces rehabilitacije. Jedan od funkcionalnih zahtjeva je: autor može napisati, izmjeniti i objaviti post sa svojim iskustvom, dok posjetilac može samo da pregleda postove.



Slika 3. Web platforma "My Knee Rehabilitation"

Proces testiranja može izgledati ovako:

1. Planiranje
   Test tim definiše da će testirati funkcionalnosti vezane za korisničke uloge: kreiranje i uređivanje postova od strane autora, prikaz postova posjetiocima, kao i administratorske opcije (na primjer odobravanje sadržaja). Postavlja se strategija regresionog testiranja za svaku novu funkcionalnost koja se implementira.
2. Dizajniranje test slučajeva
   - Test slučaj 1: Autor uspješno kreira novi post i on se prikazuje na početnoj stranici.
   - Test slučaj 2: Posjetilac pokušava da kreira post (što ne bi trebalo biti dozvoljeno).
   - Test slučaj 3: Administrator briše post koji sadrži neprikladan sadržaj.
   - Test slučaj 4: Autor pokušava da izmijeni post drugog autora (što ne bi smijelo biti moguće).
3. Implementacija i priprema
   Kreira se automatizovana skripta pomoću Selenium alata koja simulira radnje autora, administratora i posjetioca. Test podaci uključuju više korisničkih naloga, objavljene i neobjavljene postove, kao i razne scenarije (na primjer pokušaj neovlaštenog pristupa).
4. Izvršavanje
   Prilikom izvršavanja test slučaja 2 otkriven je defekt – posjetilac ipak može pristupiti stranici za kreiranje posta direktnim unosom URL-a u adresnu traku.

5. Evidentiranje i upravljanje defektima
   Defekt se unosi u sistem za praćenje grešaka sa statusom New, dodjeljuje se backend programeru (PHP logika pristupa), i prati se dok se ne otkloni i verifikuje rješenje.
6. Zatvaranje
   Nakon što se potvrdi da posjetilac više ne može neovlašteno pristupiti formi za unos posta, tim izrađuje završni izvještaj. Platforma se smatra stabilnom za dalje faze razvoja i pušta se uživo za korisnike.

2.2.3 Uloga alata u procesu testiranja

Proces testiranja softvera u savremenim razvojnim okruženjima ne može se zamisliti bez upotrebe različitih alata koji značajno unapređuju efikasnost, pouzdanost i brzinu testiranja. Alati za testiranje softvera predstavljaju tehnološku osnovu na kojoj se grade automatizovani, poluautomatski ili manuelni testni procesi, omogućavajući timovima da obave složene zadatke u kraćem vremenskom roku i sa većim stepenom preciznosti. Njihova primjena omogućava smanjenje ljudskih grešaka, bolje upravljanje procesom testiranja, kao i lakšu integraciju sa drugim fazama softverskog razvoja.

Automatizacija testiranja predstavlja jedan od najvažnijih aspekata modernog QA procesa. Automatizovani testovi omogućavaju ponovnu i pouzdanu provjeru funkcionalnosti softvera, smanjujući potrebu za ručnim izvršavanjem test slučajeva i time ubrzavajući čitav proces. Među najpoznatijim i najčešće korišćenim alatima za automatizaciju su:

- Selenium: Otvoreni softver za automatizaciju web aplikacija koji podržava različite programske jezike i omogućava kreiranje složenih test skripti. Selenium je standard u industriji zbog svoje fleksibilnosti i široke zajednice korisnika.
- Cypress: Relativno noviji alat, posebno dizajniran za testiranje modernih web aplikacija. Cypress se odlikuje jednostavnom integracijom i brzim izvršavanjem testova, sa naglaskom na efikasno rukovanje asinhronim operacijama i debugovanje.
- TestComplete: Komercijalni alat koji podržava automatizaciju testova za desktop, web i mobilne aplikacije. TestComplete pruža intuitivan interfejs za kreiranje testova bez potrebe za intenzivnim programiranjem, što ga čini dostupnim širokom spektru QA profesionalaca. (Geeks for Geeks, 2025)

Ovi alati omogućavaju ne samo automatizaciju funkcionalnih testova, već i regresionih testova, performansnih provjera, kao i integracionih testova, čime se osigurava stabilnost i kvalitet softverskih rješenja kroz čitav razvojni ciklus.

Upravljanje test slučajevima (test case management) predstavlja ključni dio organizacije procesa testiranja. Alati iz ove kategorije pomažu u kreiranju, organizovanju, praćenju i dokumentovanju test slučajeva, kao i u evidentiranju rezultata testiranja. Ovo omogućava jasnu transparentnost i kontrolu nad napretkom testiranja, kao i bolju saradnju unutar timova. Najznačajniji alati uključuju:

- TestRail: Popularan alat za upravljanje test slučajevima koji omogućava detaljno praćenje test plana, test suite-ova i pojedinačnih testova. TestRail podržava integracije sa alatima za praćenje defekata i kontinuiranu integraciju, čime se olakšava koordinacija procesa.

- Zephyr: Alat koji se često koristi u agilnim razvojnim okruženjima, poznat po svojoj integraciji sa Jira platformom i podršci za različite vrste testova. Omogućava timovima da lako prate stanje testiranja i brzo reaguju na pronađene probleme.
- qTest: Sveobuhvatna platforma koja omogućava upravljanje test slučajevima, kao i analitiku i izvještavanje o testiranju. qTest je pogodna za veće organizacije koje zahtijevaju skalabilna i prilagodljiva rješenja.

Ovi alati značajno poboljšavaju organizaciju rada i omogućavaju efikasnu kontrolu kvaliteta kroz sistematski pristup testiranju.

Praćenje defekata je nezaobilazan dio procesa testiranja, jer omogućava evidentiranje, klasifikaciju, prioritetizaciju i rješavanje pronalazaka i grešaka. Pravilno upravljanje defektima omogućava timovima da na vrijeme identifikuju kritične probleme, prate njihov status i efikasno komuniciraju između razvojnih i QA timova. Među najpopularnijim alatima za ovu svrhu su:

- JIRA: Jedan od najraširenijih alata za upravljanje projektima i praćenje defekata. JIRA omogućava kreiranje detaljnih zapisa o problemima, dodjeljivanje zadataka i praćenje toka rješavanja problema kroz različite faze. Pored toga, JIRA se lako integriše sa brojnim drugim alatima za razvoj i testiranje.
- Bugzilla: Open source alat za praćenje defekata koji je dugo prisutan na tržištu i poznat po svojoj stabilnosti i jednostavnosti. Bugzilla podržava detaljno praćenje grešaka i bogate mogućnosti prilagođavanja procesa.
- MantisBT: Još jedan popularan open source alat za praćenje defekata, koji je jednostavan za korišćenje i omogućava osnovne funkcionalnosti za evidenciju i rješavanje problema. (Geeks for Geeks, 2025)

Korišćenje ovih alata omogućava da se greške ne „izgube“ tokom procesa, da se smanji vrijeme rješavanja i da se unaprijedi kvalitet softverskog proizvoda.

U savremenim razvojno-operativnim (DevOps) praksama, kontinuirana integracija (CI-Continuous Integration[5]) i kontinuirano isporučivanje (CD - Continuous Delivery[6]) predstavljaju ključne principe za brzu i pouzdanu isporuku softvera. Alati za CI/CD omogućavaju automatsko pokretanje testova nakon svake promjene u kodu, čime se omogućava brza povratna informacija i rano otkrivanje problema. Neki od najznačajnijih alata u ovoj oblasti su:

- Jenkins: Open source alat za automatizaciju procesa buildovanja, testiranja i isporuke softvera. Jenkins podržava veliki broj plugina koji omogućavaju integraciju sa alatima za testiranje, verzionisanje koda i upravljanje projektima.

[5] CI (Continuous Integration), ili kontinuirana integracija, je praksa u softverskom inženjerstvu kojom se kod često integriše u zajednički repozitorij, obično više puta dnevno, a svaki put kada se to desi – automatski se pokreću testovi i validacije.

[6] CD – Continuous Delivery (kontinuirana isporuka) je praksa u razvoju softvera koja nadograđuje CI i omogućava da se softver može automatski isporučiti u produkciju ili testno okruženje u bilo kom trenutku, brzo i pouzdano.

- GitLab CI/CD: Integrisana CI/CD platforma unutar GitLab okruženja, koja omogućava definisanje i izvršavanje pipelines za automatizaciju testiranja i isporuke softvera direktno u okviru repozitorijuma koda.
- Azure DevOps: Microsoftova platforma koja pruža usluge za planiranje, izgradnju, testiranje i isporuku softvera. Azure DevOps je pogodna za timove koji koriste Microsoftove tehnologije i žele integrisano rješenje za CI/CD.

Integracija ovih alata u DevOps procese omogućava da se testiranje ne odvija izolovano, već kao dio kontinuiranog ciklusa razvoja i isporuke softvera. Time se obezbjeđuje brza povratna informacija, smanjuju rizici od grešaka u produkciji i povećava ukupna stabilnost i kvalitet softverskih proizvoda.

Uloga alata u procesu testiranja softvera je nezamjenjiva. Svaka od pomenutih kategorija alata doprinosi pojedinačnim aspektima procesa – od automatizacije testiranja, preko upravljanja test slučajevima, do praćenja defekata i integracije u razvojne cikluse. Efikasna upotreba ovih alata omogućava organizacijama da ubrzaju razvoj, smanje troškove i povećaju pouzdanost svojih softverskih rješenja. Osim tehničkih prednosti, ovi alati poboljšavaju i saradnju među članovima timova, omogućavaju transparentnost procesa i omogućavaju donošenje kvalitetnijih odluka u toku razvoja softvera.

2.3 Manualno ili automatsko testiranje

Odabir između manualnog (ručnog) i automatskog testiranja predstavlja jedno od najvažnijih strateških pitanja u procesu obezbjeđivanja kvaliteta softvera. Oba pristupa imaju svoju ulogu u životnom ciklusu razvoja softverskih sistema, a njihova primjena zavisi od prirode projekta, složenosti funkcionalnosti, zahtjeva za brzinom isporuke i dostupnih resursa.

Manualno testiranje je tradicionalan i još uvijek široko rasprostranjen metod koji se zasniva na interaktivnom radu testera sa aplikacijom. Tester koristi softver na način sličan krajnjem korisniku i provjerava da li aplikacija funkcioniše u skladu sa specifikacijama i očekivanjima. Ovaj pristup je naročito vrijedan u ranim fazama razvoja kada su korisnički interfejs i poslovna logika još u promjeni, kao i kada se testira iskustvo korisnika (UX- User Experience), vizuelna prezentacija ili intuitivnost funkcionalnosti. Njegova najveća prednost je fleksibilnost, jer tester može u realnom vremenu donositi odluke, prilagođavati pristup i otkrivati nepredviđene greške. (Geeks for Geeks, 2025)

Međutim, uprkos svojoj korisnosti, manualno testiranje ima ozbiljna ograničenja. Ono je vremenski zahtjevno, sklono ljudskim greškama i teško skalabilno, posebno kada se radi o regresionom testiranju koje zahtijeva često ponavljanje već poznatih scenarija. Ove slabosti postaju izraženije u agilnom okruženju, gdje su brzina i učestale isporuke ključne.

S druge strane, automatsko testiranje koristi specijalizovane alate i skripte da bi izvršilo testove bez direktne ljudske intervencije. Ovaj pristup je naročito pogodan za repetitivne, stabilne i dobro definisane funkcionalnosti koje se često provjeravaju, kao što su validacije unosa, autentifikacija korisnika, API pozivi ili cjelokupni tokovi kroz aplikaciju. Alati kao što su Selenium, Cypress, JUnit i TestNG omogućavaju brzu i dosljednu provjeru softverskog ponašanja, što značajno ubrzava regresiono testiranje i omogućava integraciju testiranja u DevOps procese.

Uprkos očiglednim prednostima, automatsko testiranje nije univerzalno rješenje. Njegova implementacija zahtjeva inicijalna ulaganja u razvoj i održavanje test skripti, tehničku obuku kadrova i detaljno razumijevanje aplikacije. Takođe, automatski alati imaju ograničenu

sposobnost da procijene vizuelne aspekte korisničkog interfejsa ili otkriju neočekivano ponašanje koje može proći nezapaženo ako nije eksplicitno kodirano u test scenarijima.

Praksa pokazuje da nijedan od ovih pristupa nije dovoljan sam za sebe. Najefikasniji rezultati se postižu kombinacijom oba, to jeste hibridnim pristupom. Manualno testiranje se koristi za otkrivanje kompleksnih ili novih defekata, dok se automatsko testiranje fokusira na učestala i stabilna testiranja. Na taj način se obezbjeđuje i pokrivenost i brzina, a kvalitet softverskog proizvoda značajno se povećava.

Jedan konkretan primjer može ilustrativno prikazati ovu podjelu. Prilikom razvoja web aplikacije za elektronsku prodaju, ručno testiranje se koristi za provjeru korisničkog iskustva pri dodavanju proizvoda u korpu, dok se automatski testovi koriste za svakodnevno provjeravanje procesa logovanja, obračuna PDV-a[7] i slanja potvrde putem emaila. Ova kombinacija omogućava sveobuhvatno testiranje sistema, uz optimalno korišćenje vremena i resursa.

Izbor između manualnog i automatskog testiranja ne bi trebalo da bude isključiv, već strateški osmišljen i usklađen sa ciljevima projekta. Test menadžeri treba da procijene koje dijelove sistema je najisplativije automatizovati, a gdje ljudska pažnja i kreativnost mogu donijeti veću vrijednost.

2.4 Tradicionalni alati i metode za otkrivanje defekata

Otkrivanje defekata u proizvodnom procesu i finalnim proizvodima predstavlja jedan od ključnih segmenata u osiguranju kvaliteta i pouzdanosti. Tradicionalne metode, uprkos razvoju novih tehnologija i automatizaciji, i dalje zauzimaju važno mjesto u industrijskoj praksi, posebno zbog svoje pristupačnosti, jednostavnosti primjene i efektivnosti u ranim fazama kontrole. Ove metode su rezultat dugogodišnjeg razvoja i prakse, a često služe kao prva linija odbrane od defektnih proizvoda koji bi mogli ugroziti sigurnost ili zadovoljstvo korisnika.

Vizuelna inspekcija je najstariji i najosnovniji metod za otkrivanje defekata. U svojoj suštini, ona podrazumijeva neposredan pregled proizvoda ljudskim okom, često uz pomoć dodatnih alata kao što su lampa, lupa ili mikroskop, u zavisnosti od veličine i vrste proizvoda. Ova metoda omogućava identifikaciju različitih vrsta defekata, kao što su površinske ogrebotine, pukotine, nepravilnosti u boji, deformacije ili neusklađenosti u obliku. Njena jednostavnost i direktnost čine je veoma korisnom, ali istovremeno zahtijevaju visok nivo pažnje i stručnosti inspektora.

Efikasnost vizuelne inspekcije zavisi od uslova u kojima se sprovodi. Dobar izvor svjetlosti, odgovarajuća ergonomija radnog mjesta i standardizovani uslovi pregleda značajno poboljšavaju rezultate. Međutim, ova metoda ima i svoja ograničenja — ona može otkriti samo defekte koji su površinski vidljivi, dok se unutrašnje ili mikroskopske nepravilnosti često ne mogu detektovati.

Za određene primjene, vizuelna inspekcija se kombinuje sa drugim tradicionalnim metodama ili se koristi kao prvi korak u višestepenom sistemu kontrole kvaliteta, gdje sumnjivi proizvodi bivaju podvrgnuti detaljnijim analizama.

Pored vizuelnog pregleda, u tradicionalnoj praksi veliki značaj imaju precizna mehanička mjerenja. Korišćenjem instrumenata poput mikrometara, šublera, dubinomera i ostalih

[7] PDV- Porez na dodatu vrijednost

specijalizovanih alata za mjerenje, moguće je otkriti odstupanja u dimenzijama i obliku koje mogu ukazivati na defekte nastale u toku proizvodnje.

Ova vrsta kontrole je posebno važna u industrijama gdje je tačnost dimenzija kritična, poput automobilske industrije, proizvodnje mašina ili elektronike. Čak i minimalna odstupanja mogu dovesti do problema u sklapanju, funkcionalnosti ili trajnosti proizvoda. Na primjer, neprecizno obrađen metalni dio može izazvati preveliko habanje ili kvar tokom rada mašine.

Prednost mehaničkih mjerenja je u njihovoj preciznosti i objektivnosti, jer se zasnivaju na tačnim vrijednostima, za razliku od vizuelne inspekcije koja zavisi od subjektivnog doživljaja inspektora. Ipak, ova metoda je vremenski zahtjevnija i zahtijeva dobro obučeno osoblje, kao i redovno kalibrisanje mjernih instrumenata da bi rezultati bili pouzdani.

Nedeztruktivna ispitivanja predstavljaju skup tradicionalnih tehnika koje omogućavaju detekciju defekata unutar materijala ili na njegovoj površini, bez oštećivanja samog proizvoda. Ove metode su naročito važne u industrijama gdje je očuvanje integriteta proizvoda neophodno, kao što su vazduhoplovstvo, brodogradnja, automobilska industrija i građevinarstvo.

Ultrazvučno ispitivanje koristi zvučne talase visoke frekvencije koji prolaze kroz materijal i odbijaju se od nepravilnosti poput pukotina ili nehomogenosti. Na osnovu vremena povratnog signala i njegovih karakteristika, moguće je precizno locirati i ocjeniti defekt. Ovo ispitivanje je posebno efikasno za otkrivanje unutrašnjih oštećenja koja nisu vidljiva spolja.

Magnetno ispitivanje se koristi za detekciju površinskih i blizu površinskih defekata u feromagnetnim materijalima. Materijal se magnetizuje, a zatim se nanose sitne magnetne čestice koje se skupljaju na mjestima sa nepravilnostima u magnetnom polju, ukazujući na moguće pukotine ili druge defekte.

Penetraciona ispitivanja podrazumijevaju nanošenje specijalnih tečnosti koje ulaze u pore i pukotine na površini, a zatim se uklanjaju višak tečnosti sa površine. Nakon toga se nanosi razvijač koji „izvlači“ penetrant iz defekata, čineći ih vidljivim golim okom. Ova metoda je jednostavna i efikasna za površinske defekte, ali ne može da detektuje unutrašnje oštećenja.

Sve ove metode se tradicionalno primjenjuju samostalno ili u kombinaciji, u zavisnosti od vrste proizvoda i zahtjeva kvaliteta. Iako postoje nove, digitalizovane i automatizovane tehnologije, nedeztruktivna ispitivanja ostaju stub u procesu kontrole jer omogućavaju detaljnu analizu bez rizika od oštećenja proizvoda.

Iako su tradicionalni alati i metode široko primjenjivani i provjereni, oni nisu bez svojih nedostataka. Vizuelna inspekcija i mehanička mjerenja često zavise od ljudskog faktora, što može dovesti do varijabilnosti i grešaka u procjeni. Zavisnost od stručnosti inspektora i subjektivni kriterijumi u nekim slučajevima mogu umanjiti preciznost kontrole.

Takođe, ove metode mogu biti vremenski zahtjevne i manje pogodne za masovnu proizvodnju gdje je potrebna brza i automatizovana kontrola. Nedeztruktivna ispitivanja zahtjevaju specijalizovanu opremu i obučeno osoblje, a u nekim slučajevima i značajne investicije.

Zbog ovih razloga, tradicionalne metode se danas često koriste u kombinaciji sa savremenim tehnologijama poput digitalnih kamera, vještačke inteligencije i automatizovanih sistema za inspekciju. Takve integracije omogućavaju da se iskoriste prednosti klasičnih tehnika, dok se istovremeno umanjuju njihove slabosti.

Tradicionalni alati i metode za otkrivanje defekata predstavljaju neizostavan dio sistema kontrole kvaliteta u industriji. Njihova jednostavnost, dostupnost i efikasnost u prepoznavanju širokog spektra defekata čine ih osnovom za sve ostale, modernije pristupe. Bez obzira na napredak u tehnologiji, ovi metodi i dalje igraju vitalnu ulogu, posebno u ranim fazama proizvodnje i u situacijama gdje je potreban neposredan odgovor na potencijalne probleme.

Razumijevanje njihovih prednosti i ograničenja omogućava bolje planiranje i organizaciju procesa kontrole kvaliteta, kao i efikasnije korišćenje resursa u proizvodnji. U tom kontekstu, tradicionalne metode ne treba posmatrati kao zastarjele, već kao pouzdane i provjerene alate koji, uz adekvatnu primjenu i integraciju sa savremenim tehnologijama, doprinose ukupnom uspjehu proizvodnog procesa.

3. MAŠINSKO UČENJE U FUNKCIJI TESTIRANJA

Mašinsko učenje (engl. Machine Learning – ML[8]), kao izuzetno dinamična oblast unutar šireg domena vještačke inteligencije, predstavlja jednu od ključnih tehnologija savremenog informacionog društva. Njegova primjena sve je zastupljenija u različitim industrijskim sektorima, uključujući proizvodnju, zdravstvo, finansije, ali i specifično – u oblasti testiranja i kontrole kvaliteta proizvoda. Primjenom algoritama mašinskog učenja omogućava se automatizovana analiza velikih količina podataka, prepoznavanje kompleksnih obrazaca i donošenje odluka koje unapređuju efikasnost i tačnost detekcije defekata.

U kontekstu industrijskih testiranja, gdje su preciznost i brzina odlučujući faktori, mašinsko učenje ima potencijal da značajno smanji ljudske greške, ubrza proces inspekcije, te pruži viši stepen konzistentnosti i objektivnosti u ocjeni kvaliteta. Ovo poglavlje pruža sistematičan uvod u osnovne koncepte mašinskog učenja, klasifikaciju metoda, kao i konkretne primjere njegove primjene u optimizaciji testnih procedura i unapređenju kontrole kvaliteta proizvoda.

Mašinsko učenje je disciplina koja omogućava računarima da automatski uče iz podataka i unapređuju svoje performanse kroz iskustvo, bez potrebe za eksplicitnim programiranjem svakog pojedinačnog zadatka. Ključna prednost ML sistema jeste sposobnost da prepoznaju obrasce i odnose u podacima koji često ostaju nevidljivi klasičnim metodama analize, omogućavajući tako preciznije predikcije i bolju klasifikaciju informacija.

Osnovni princip mašinskog učenja zasniva se na korištenju velikih skupova podataka (datasets) za treniranje modela, kako bi oni mogli generalizovati naučene obrasce i efikasno ih primjenjivati na nove, nepoznate ulaze. U industrijskim uslovima, to znači da se mašinski modeli mogu koristiti za otkrivanje grešaka, predikciju kvarova ili automatsku kategorizaciju proizvoda u realnom vremenu, čime se omogućava rana intervencija i sprječavanje potencijalnih problema u kasnijim fazama proizvodnje.

3.1 Kategorije mašinskog učenja

Mašinsko učenje obuhvata različite pristupe koji se, u zavisnosti od načina na koji se podaci koriste tokom procesa treniranja, dijele na tri osnovne kategorije: nadzirano učenje, nenadzirano učenje i učenje pojačanjem. (IBM, 2025)

Nadzirano učenje

Nadzirano učenje (engl. Supervised Learning) podrazumijeva treniranje modela na osnovu ulazno-izlaznih parova, gdje su svi primjeri unaprijed označeni odgovarajućim labelama. Model uči funkciju koja povezuje ulazne karakteristike sa željenim izlazom, s ciljem da tu funkciju uspješno primijeni i na buduće, nepoznate podatke. U kontekstu testiranja, ova metoda se koristi za automatsko prepoznavanje poznatih tipova defekata, bazirano na istorijskim podacima i unaprijed označenim uzorcima.

[8]ML (Machine Learning) – mašinsko učenje

Nenadzirano učenje

Nenadzirano učenje (engl. Unsupervised Learning) se koristi kada trenirajući podaci nemaju eksplicitne oznake. Algoritam u tom slučaju samostalno otkriva skrivene strukture i odnose među podacima, grupiše ih u klastere ili identifikuje anomalije. Ovaj pristup je naročito koristan u slučajevima kada nisu unaprijed poznate sve moguće greške ili kada se želi detektovati nepredviđeno ponašanje proizvoda koje može ukazivati na novi tip defekta.

Učenje pojačanjem

Učenje pojačanjem (engl. Reinforcement Learning) predstavlja oblik adaptivnog učenja kroz interakciju sa okruženjem. Agent postepeno optimizira svoje ponašanje na osnovu nagrada i kazni koje dobija, sa ciljem maksimizacije ukupne koristi. Iako je njegova primjena u testiranju još uvijek ograničena, u budućnosti može igrati značajnu ulogu u optimizaciji složenih proizvodnih procesa, naročito u autonomnim sistemima.

3.2 Primjena mašinskog učenja

Implementacija ML modela u industrijsku praksu testiranja i kontrole kvaliteta donosi višestruke koristi. Prije svega, omogućava automatizaciju inspekcijskih procedura, značajno smanjujući zavisnost od ljudskog faktora, čime se smanjuje broj subjektivnih grešaka i povećava ukupna dosljednost sistema.

Jedna od najefikasnijih primjena je analiza vizuelnih podataka pomoću tehnika dubokog učenja (Deep Learning), koje se baziraju na složenim strukturama vještačkih neuronskih mreža. Ovi sistemi su sposobni da sa visokom preciznošću identifikuju najsitnije defekte, čak i u prisustvu šuma, promjenljivog osvjetljenja ili drugih nepravilnosti koje bi mogle zbuniti klasične algoritme. Takvi modeli već se aktivno primjenjuju u inspekciji elektronskih komponenti, tekstilnoj proizvodnji, automobilskoj industriji i drugim granama u kojima su vizuelni standardi od presudnog značaja.

Pored slike, sve češće se koriste senzorski podaci kao što su vibracije, akustični signali, temperaturni profili ili elektromagnetne emisije, koji omogućavaju neinvazivnu detekciju skrivenih defekata. Korištenjem naprednih ML modela, moguće je analizirati ove kompleksne signale i na osnovu njih pravovremeno detektovati potencijalne kvarove ili predvidjeti potrebu za preventivnim održavanjem. Na taj način se povećava pouzdanost sistema i produžava vijek trajanja opreme, uz smanjenje operativnih troškova.

Otkrivanje defekata predstavlja jedan od najkritičnijih aspekata u okviru procesa kontrole kvaliteta i osiguranja pouzdanosti proizvoda u savremenoj industrijskoj proizvodnji. Tradicionalne metode detekcije, koje se oslanjaju na ručne inspekcije, vizuelna opažanja ili standardne tehničke algoritme, često su vremenski zahtjevne, podložne ljudskim greškama i teško skalabilne. U tom kontekstu, mašinsko učenje nudi napredne mogućnosti automatizacije, povećanja tačnosti i ubrzanja procesa detekcije, uz znatno smanjenje troškova i operativnog rizika.

Mašinsko učenje omogućava modelima da prepoznaju obrasce u velikim skupovima podataka i da klasifikuju ili predviđaju potencijalne defekte bez potrebe za eksplicitnim

programiranjem svake pravila ponašanja. U nastavku se razmatraju konkretne industrijske primjene ML-a u otkrivanju defekata, uključujući tekstilnu, elektronsku, automobilsku i softversku industriju, kao i metode nedestruktivnog testiranja. Ova raznovrsna polja primjene svjedoče o univerzalnosti i prilagodljivosti ML pristupa u detekciji grešaka u realnim, složenim sistemima.

3.2.1 Primjena u proizvodnim procesima

U industriji elektronike, vizuelna inspekcija predstavlja temeljni postupak za detekciju mikroskopskih defekata na štampanim pločama i komponentama. Umjesto da se ova analiza obavlja ručno, što zahtijeva visoku preciznost i dugotrajnu obuku osoblja, savremeni ML sistemi koriste kamere visoke rezolucije u kombinaciji sa konvolucionim neuronskim mrežama kako bi automatski prepoznali pukotine, nepravilno lemljenje, prekomjerno lemljenje, prisustvo stranih čestica ili oštećenja površine.

Takvi modeli su posebno efikasni jer mogu da "nauče" karakteristike defekata na osnovu istorijskih slika i zatim generalizuju znanje na nove primjere. Jedna od ključnih prednosti jeste njihova sposobnost da zadrže visoku tačnost čak i pri promjenama u osvjetljenju, uglovima kamere ili varijacijama u materijalima. Uvođenje ML modela omogućava da se vizuelna inspekcija svede na milisekunde po komponenti, čime se ostvaruje znatna ušteda vremena i poboljšava kapacitet proizvodne linije.

U tekstilnoj industriji, algoritmi mašinskog učenja se koriste za prepoznavanje strukturnih i površinskih nepravilnosti na tkaninama. Korištenjem video nadzora u realnom vremenu, uz integraciju ML modela za detekciju anomalija, mogu se automatski identifikovati rupe, greške u uzorku, promjene u boji, mrlje i deformacije u tkanju. Zahvaljujući dubokim neuronskim mrežama, ovakvi sistemi postižu preciznost koja često nadmašuje sposobnosti ljudskih inspektora, naročito kada je riječ o uočavanju defekata koji su teško vidljivi golim okom ili koji se javljaju u brzim fazama obrade materijala.

3.2.2 Složeni senzorski sistemi i nedestruktivna ispitivanja

Posebno značajna primjena mašinskog učenja ogleda se u analizi multisenzorskih podataka u proizvodnim okruženjima. Ovdje se koristi širok spektar izvora podataka: vibracije, zvuk, temperatura, elektromagnetne karakteristike, pritisak, infracrveno zračenje i drugi parametri koji zajedno pružaju kompleksan uvid u stanje mašine ili materijala.

U okviru tako zvanih nedestruktivnih metoda ispitivanja (NDT[9]), ML algoritmi analiziraju povratne signale sa ultrazvučnih, magnetskih ili termografskih senzora kako bi otkrili skrivene defekte u materijalima bez potrebe za njihovim oštećenjem. Ovi modeli uče na osnovu poznatih primjera (na primjer signala iz materijala sa poznatim oštećenjima) i kasnije klasifikuju signale iz stvarne proizvodnje kao “normalne” ili “anomalne”. Na taj način se omogućava precizno lociranje defekta unutar strukture, što je od ključnog značaja u

[9] NDT (Non-Destructive Testing) ne destruktivne metode ispitivanja su tehnike za ispitivanje materijala, dijelova ili konstrukcija bez njihovog oštećenja ili uništenja.

industrijama poput vazduhoplovstva, energetike i građevinarstva, gdje kvar može imati katastrofalne posljedice.

U automobilskoj industriji, ML se koristi za analizu podataka iz pametnih senzora integrisanih u proizvodnu liniju. Ovi senzori bilježe stanja tokom svakog segmenta sklapanja vozila. Analizom korelacija među tim podacima, modeli mogu otkriti nepravilnosti koje su izvan dometa tradicionalnih tehničkih kontrola, kao što su vibracioni obrasci koji prethode mehaničkom kvaru ili mikroodstupanja u radu motora.

3.2.3 Primjena u softverskom inženjeringu

Primjena mašinskog učenja nije ograničena samo na fizičke proizvode – značajna istraživanja i implementacije se realizuju i u softverskom testiranju, gdje je cilj detekcija grešaka u kodu, optimizacija testnih slučajeva i automatizacija procesa verifikacije i validacije.

ML modeli se mogu trenirati na osnovu prethodnih izvještaja o greškama, logova sa servera ili karakteristika samog koda (na primjer dužina funkcija, broj petlji, upotreba globalnih varijabli). Na osnovu toga, algoritam može identifikovati sekcije softverskog sistema koje su podložnije greškama i time omogućiti prioritizaciju testiranja. Takođe, u kontekstu automatizacije regresionog testiranja, modeli uče koje promjene u kodu su najčešće uzrok novih problema, pa prema tome fokusiraju resurse na najrizičnija mjesta.

Zanimljiv je i razvoj prediktivnih modela koji, analizom istorijskih grešaka, mogu davati preporuke za rješavanje problema ili čak automatski ispravljati jednostavne greške u kodu. Time se ne samo ubrzava razvojni ciklus, već i značajno unapređuje stabilnost krajnjeg softverskog rješenja.

Primjena mašinskog učenja u otkrivanju defekata postaje standard u industrijskoj praksi zbog svoje efikasnosti, skalabilnosti i mogućnosti adaptacije na različite izvore podataka. U poređenju sa tradicionalnim metodama, ML pristupi pružaju bržu detekciju, manju stopu lažnih alarma, kao i sposobnost učenja i prilagođavanja u promjenjivim uslovima proizvodnje.

Ipak, uspješna implementacija ovih sistema zahtijeva pažljiv odabir podataka za treniranje, balansiranje između tačnosti i kompleksnosti modela, kao i kontinualno održavanje sistema kroz ažuriranje podataka i prilagođavanje novim uslovima. Dodatno, postoji potreba za saradnjom između inženjera, stručnjaka za informacione tehnologije i domenskih eksperata kako bi se osigurao kontekstualno relevantan dizajn modela.

U narednim godinama očekuje se sve veća integracija ML tehnologija sa Internetom stvari (IoT[10]) i industrijom 4.0, što će omogućiti potpuni prelazak sa reaktivnih na proaktivne i prediktivne sisteme upravljanja kvalitetom. Kombinacijom ML sa edge computing tehnologijama, moguće je omogućiti obradu podataka i detekciju defekata direktno na proizvodnoj liniji, u realnom vremenu, bez potrebe za centralizovanom obradom. Time će se dodatno ubrzati donošenje odluka i minimizirati operativni zastoji.

[10] IoT (Internet of Things) – Internet stvari

Mašinsko učenje ne samo da unapređuje otkrivanje defekata, već redefiniše sam način na koji razumijemo kvalitet – kao dinamičan, podatkom vođen proces koji se stalno optimizira kroz inteligentne tehnologije.

3.3 Prednosti, izazovi i buduće perspektive mašinskog učenja

Savremeni softverski sistemi postaju sve složeniji, zahtijevajući ne samo pouzdanu funkcionalnost već i visoku prilagodljivost, sigurnost i stabilnost u različitim okruženjima. Testiranje softvera, kao ključna komponenta procesa razvoja, ima zadatak da obezbijedi upravo taj nivo kvaliteta i pouzdanosti. Međutim, tradicionalne metode testiranja često su vremenski zahtjevne, zavisne od ljudskog faktora i manje efikasne kada je u pitanju obrada velikih količina podataka.

Mašinsko učenje (ML) unosi revolucionarne promjene u ovaj domen, omogućavajući ne samo automatizaciju rutinskih zadataka, već i inteligentnu analizu, predviđanje i optimizaciju testnih procesa. U ovom poglavlju analiziraju se glavne prednosti koje mašinsko učenje donosi u testiranju softvera, ali i njegova ograničenja i izazovi u praktičnoj primjeni.

3.3.1 Prednosti primjene mašinskog učenja

Jedna od najznačajnijih prednosti mašinskog učenja u testiranju je mogućnost automatizovanog generisanja i izvršavanja testnih slučajeva. Tradicionalni pristupi, kao što su ručno pisanje testova ili statičko definisanje skripti, često su neefikasni u dinamičnim razvojnim okruženjima. Mašinsko učenje omogućava da se, na osnovu postojećih obrazaca u kodu ili ponašanju aplikacije, automatski generišu testovi koji obuhvataju nove i ranije nezabilježene scenarije.

Konkretno, tehnike kao što su reinforcement learning i model-based testing omogućavaju alatima da „uče“ kako se sistem ponaša u različitim uslovima i u skladu s tim kreiraju relevantne testne skripte. Ovo ne samo da ubrzava proces testiranja već značajno smanjuje prostor za ljudsku grešku.

Primjenom tehnika kao što su nadzorovano učenje (supervised learning), moguće je analizirati istorijske podatke o greškama i kvarovima u softveru, kako bi se identifikovale kritične zone u kodu koje su najsklonije greškama. Na taj način, testiranje se može fokusirati upravo na one komponente koje nose najveći rizik, čime se ostvaruje bolja iskorišćenost resursa i veća pokrivenost testova.

Takvi prediktivni modeli se uspješno koriste u DevOps praksama, gdje se veliki broj promjena u kodu dešava svakodnevno, a potrebno je brzo identifikovati potencijalne izvore problema prije nego što dođu u produkciju.

U kontekstu agilnog razvoja, gdje se softver konstantno mijenja, klasični pristupi testiranju često ne mogu ispratiti sve promjene u realnom vremenu. Mašinsko učenje, međutim, omogućava dinamičko prilagođavanje testova**,** odnosno „pametno“ ažuriranje testnih skripti na osnovu novih funkcionalnosti, refaktorisanja koda ili promjena u konfiguracijama.

Algoritmi kao što su klasterisanje, decision trees, kao i sequence modeling, omogućavaju otkrivanje veza između promjena u kodu i potencijalnih uticaja na stabilnost sistema, čime se osigurava kontinuirana relevantnost testiranja bez potrebe za ručnim revizijama.

Moderni softverski sistemi proizvode ogromne količine logova, telemetry podataka i performansnih metrika, koje je nemoguće efikasno analizirati klasičnim metodama. Mašinsko učenje omogućava automatsku analizu ovih podataka, otkrivanje anomalija, neočekivanih ponašanja ili šablona koji upućuju na potencijalne kvarove.

Na primjer, primjena analize vremenskih nizova (time series analysis) i detekcije anomalija u produkcionim sistemima omogućila je rano upozoravanje na degradaciju performansi ili skrivene bagove, što direktno doprinosi stabilnosti i bezbjednosti sistema.

3.3.2 Ograničenja primjene mašinskog učenja

Kvalitet ML modela direktno zavisi od kvaliteta i količine podataka dostupnih za trening. U domenu testiranja softvera, ovo može biti problematično, naročito u ranim fazama razvoja ili kod novih sistema, gdje ne postoji dovoljno podataka o prethodnim greškama, scenarijima upotrebe ili logovima.

Pored toga, podaci mogu biti neuravnoteženi, odnosno greške se javljaju rijetko, što otežava modelima da nauče relevantne obrasce. U takvim situacijama, potrebno je primijeniti specijalizovane tehnike kao što su sampling, cost-sensitive learning, ili sintetičko generisanje podataka (data augmentation).

Razvijanje, treniranje i održavanje naprednih ML modela zahtijeva značajno stručno znanje, kao i odgovarajuću infrastrukturu. Za manje organizacije ili startape, ovaj aspekt može predstavljati značajnu prepreku. Nedostatak ML inženjera, DevOps/MLOps infrastrukture i potreba za stalnim ažuriranjem modela ograničavaju primjenu ovih tehnika u praksi.

Takođe, nepravilno konfigurisani modeli mogu proizvesti lažne pozitive ili negative, što potencijalno ugrožava integritet testnog procesa.

Jedan od glavnih izazova u primjeni dubokih modela (na primjer neuronskih mreža) jeste nedostatak transparentnosti. U mnogim industrijama, kao što su finansije ili zdravstvo, potrebno je precizno objasniti zašto je sistem donio određenu odluku. Ukoliko ML model ne može da ponudi razumljivo objašnjenje, njegova primjena postaje ograničena.

Iako postoje tehnike kao što su LIME, SHAP i attention mechanisms koje pomažu u razumijevanju uticaja određenih ulaza na izlaz modela, one nisu uvijek dovoljno pouzdane ili jednostavne za krajnje korisnike.

Modeli obučeni na podacima iz jedne domene ili vrste softvera možda neće dobro funkcionisati u drugim okruženjima. Na primjer, model treniran na veb aplikacijama možda neće imati dobru prediktivnu moć kada se primijeni na sisteme ugrađenog softvera ili mobilne aplikacije.

Ovo zahtijeva kontinuirano praćenje performansi, redovno doobučavanje modela i implementaciju ML lifecycle strategija, što dodatno komplikuje proces.

ML sistemi mogu biti podložni manipulacijama i napadima, posebno ako se oslanjaju na ulazne podatke koji mogu biti kompromitovani. Napadi kao što su adversarial examples mogu dovesti do toga da model donese pogrešne zaključke, što u kontekstu testiranja može dovesti do puštanja defektnog softvera u produkciju.

Zbog toga je neophodno implementirati bezbjednosne mehanizme i politike validacije modela kako bi se osigurala pouzdanost čitavog sistema.

Integracija mašinskog učenja u proces testiranja softvera donosi značajne benefite u vidu automatizacije, adaptivnosti, bolje alokacije resursa i dublje analize podataka. Ipak, ovaj napredak dolazi uz određene izazove – potrebu za kvalitetnim podacima, tehničku složenost, pitanja objašnjivosti i potencijalne bezbjednosne rizike.

Najbolji rezultati se postižu kombinovanjem tradicionalnih tehnika testiranja sa mašinskim učenjem, gdje ljudska ekspertiza ostaje ključna u verifikaciji rezultata i prilagođavanju modela specifičnostima projekta. Budućnost testiranja leži u hibridnim pristupima, koji spajaju najbolje iz oba svijeta – mašinsku preciznost i ljudsku intuiciju.

3.3.3 Budućnost primjene mašinskog učenja

Uprkos brojnim prednostima, implementacija mašinskog učenja u procesima testiranja nije bez izazova. Jedan od ključnih problema predstavlja prikupljanje i obrada kvalitetnih podataka. Bez reprezentativnog i balansiranog skupa podataka za treniranje, modeli mogu pokazivati nizak nivo generalizacije, što dovodi do pogrešnih zaključaka u stvarnom okruženju. Poseban izazov predstavlja održavanje aktuelnosti modela u slučaju promjene uslova u proizvodnji ili uvođenja novih tehnologija, što zahtijeva kontinuiranu evaluaciju i retrening modela.

Pored tehničkih prepreka, značajna barijera može biti i potreba za visokoobrazovanim stručnim kadrom, kao i ulaganje u odgovarajuću infrastrukturu (računarska oprema, softverski alati, skladištenje podataka). To posebno pogađa mala i srednja preduzeća koja nemaju kapacitete za uspostavljanje kompleksnih Sistema vještačke inteligencije. Ipak, razvoj intuitivnih softverskih platformi i dostupnost alata otvorenog koda (na primjer TensorFlow, PyTorch, Scikit-learn) omogućavaju sve širu primjenu ML tehnologija i van okvira velikih korporacija.

U budućnosti se očekuje razvoj hibridnih sistema koji kombinuju tradicionalne pristupe sa metodama mašinskog učenja, čime se može postići bolja preciznost, robusnost i fleksibilnost sistema za testiranje. Dodatno, integracija ML algoritama sa Internetom stvari (IoT) i konceptom pametnih fabrika (Smart Manufacturing) otvara nove horizonte u realno vremenskom nadzoru kvaliteta i automatskom donošenju odluka.

Mašinsko učenje predstavlja moćan alat u funkciji savremenog industrijskog testiranja. Njegova sposobnost da analizira velike količine podataka, otkriva obrasce i unaprijedi proces odlučivanja, čini ga nezamjenjivim u kontekstu sve većih zahtjeva za brzinom, preciznošću i pouzdanošću proizvodnih sistema. Iako izazovi ostaju, trend razvoja i digitalne transformacije industrije jasno ukazuje na to da će primjena ML tehnologija postati standard u procesima kontrole kvaliteta i automatizovane inspekcije.

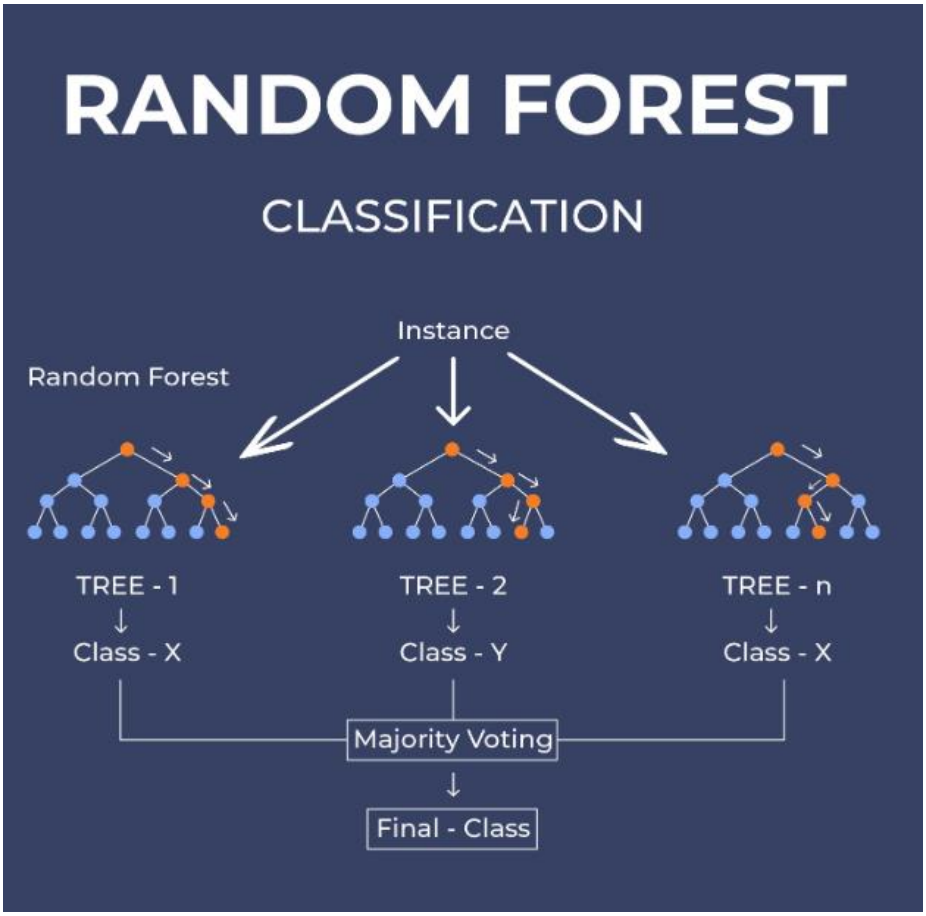
.

3.4 Uporedna analiza algoritama: Random Forest, SVM i Neural Networks

U savremenim aplikacijama mašinskog učenja, naročito u domenu detekcije defekata u proizvodnim i tehničkim sistemima, izbor odgovarajućeg algoritma od ključnog je značaja za postizanje optimalnih performansi. Detekcija defekata zahtijeva visoku preciznost, robusnost i efikasnost u obradi podataka koji često dolaze iz različitih izvora – od strukturiranih numeričkih vrijednosti do visoko nestrukturiranih podataka kao što su slike, zvučni zapisi i vremenski nizovi.Tri najčešće korištena pristupa u takvim aplikacijama su Random Forest, Support Vector Machines (SVM[11]) i neuronske mreže (Neural Networks). Svaki od ovih algoritama ima svoje specifične prednosti, ograničenja i optimalne domene primjene. Ova uporedna analiza nastoji da pruži detaljan uvid u karakteristike, performanse i primjenjivost svakog od ovih algoritama, sa posebnim fokusom na otkrivanje defekata.

3.4.1 Random Forest

Random Forest predstavlja ansambl tehniku koja kombinuje veliki broj stabala odlučivanja kako bi se poboljšala tačnost i stabilnost predikcija. Svako stablo u šumi trenira se na različitom podskupu podataka, uz nasumičan izbor podskupa karakteristika. Konačna odluka se donosi na osnovu većinskog glasanja (klasifikacija) ili srednje vrijednosti (regresija).



[11] SVM (Support Vector Machines) - Mašine potpornog vektora

Slika 4. Random Forest algoritam (https://fr.pinterest.com/pin/848787861035795075/visual-search/?x=16&y=16&w=532&h=532&surfaceType=flashlight)

Jedna od glavnih prednosti Random Forest algoritma jeste njegova sposobnost da se nosi sa visokodimenzionalnim podacima i da efikasno upravlja prisustvom šuma u podacima. Model je naročito robustan u situacijama kada postoji veliki broj ulaznih karakteristika koje nisu sve jednako relevantne. Štaviše, algoritam može da izračuna relativnu važnost svakog atributa, što pomaže inženjerima u razumijevanju faktora koji utiču na pojavu defekata.

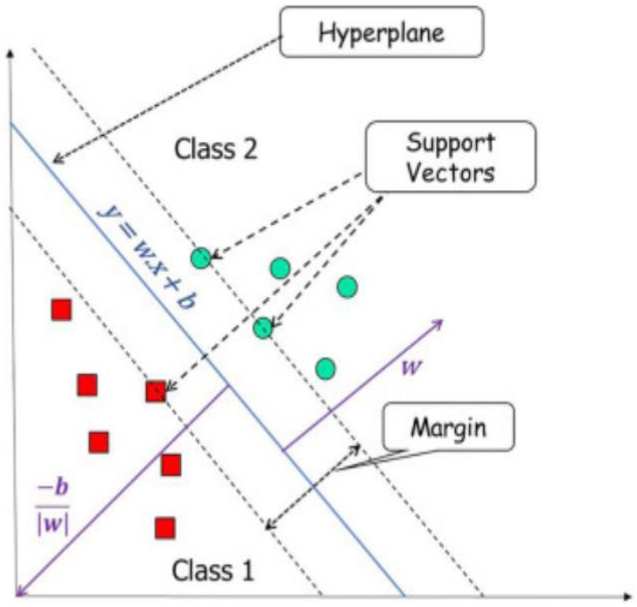
Random Forest algoritmi su često korišćeni u industriji kada se radi sa strukturiranim podacima, poput mjerenja sa senzora, logova sa proizvodnih linija, vremenskih nizova tehničkih parametara i slično. Oni su idealni za brzu primjenu, jer ne zahtijevaju mnogo predobrada podataka, a modeli se mogu trenirati relativno brzo na srednje velikim skupovima.

Iako veoma moćan, Random Forest ima i svoja ograničenja. Sa vrlo velikim datasetovima (na primjer stotinama hiljada uzoraka i hiljadama karakteristika), računska kompleksnost može postati značajna, posebno u pogledu memorijskog zahtjeva. Takođe, iako je otporniji na prenaučenost od pojedinačnih stabala, može postati neinterpretabilan – to jeste korisnicima je teško objasniti na osnovu čega je donesena neka konkretna odluka.

Za analizu vizuelnih podataka (na primjer slike), Random Forest nije optimalan jer zahtijeva prethodnu ekstrakciju karakteristika – u suprotnom ne može efikasno obrađivati piksele kao ulazne podatke. U tim slučajevima, duboke neuronske mreže imaju značajnu prednost.

3.4.2 Support Vector Machines (SVM)

SVM predstavlja klasičan, ali i dalje visoko relevantan algoritam za binarnu klasifikaciju, koji funkcioniše tako što traži optimalnu hiperravan koja razdvaja klase podataka u višedimenzionalnom prostoru. Korišćenjem kernel funkcija (na primjer polinomski, sigmoidni kernel), SVM se može efikasno primijeniti i na nelinearne probleme.



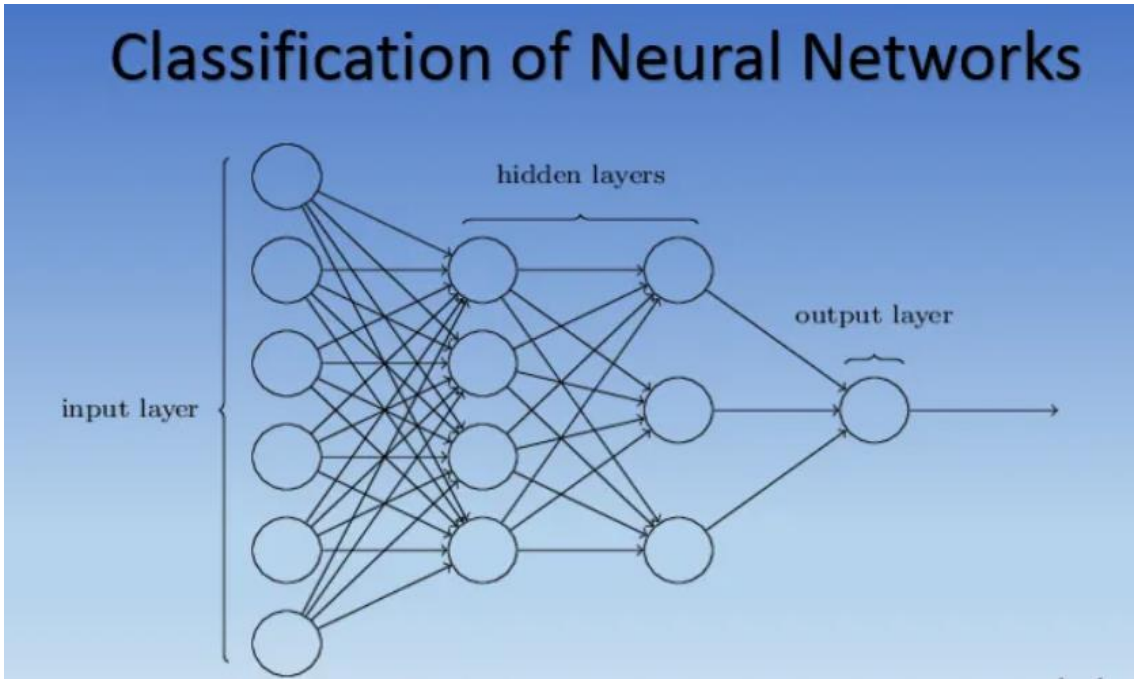
Slika 5. SVM algoritam (https://artpictures.club/autumn-2023.html)

Jedna od ključnih prednosti SVM algoritma jeste njegova sposobnost da postigne visoku preciznost čak i na malim skupovima podataka. U situacijama kada nije dostupno mnogo označenih primjera defekata – što je često slučaj u visoko specijalizovanim industrijama – SVM može da generalizuje bolje od mnogih drugih modela. SVM je takođe robustan na prenaučenost, naročito kada se pravilno izaberu parametri regularizacije. Pored toga, model ima solidnu teorijsku osnovu, što omogućava bolju kontrolu i analizu u eksperimentalnim uslovima.

Međutim, SVM nije bez mana. Prvo, njegova računska složenost brzo raste sa povećanjem broja uzoraka, što može postati problem kod većih datasetova. Takođe, izbor kernel funkcije i podešavanje hiperparametara (C, gamma i tako dalje) može biti složen i zahtijeva ekspertsko znanje i višekratne iteracije. SVM nije optimalan za direktan rad sa nestrukturiranim podacima kao što su slike ili zvučni zapisi, osim ako se prethodno ne sprovede ručna ekstrakcija karakteristika (na primjer HOG, SIFT). Ovo znatno povećava kompleksnost procesa, dok istovremeno smanjuje fleksibilnost modela.

3.4.3 Neuronske mreže (Neural Networks)

Neuronske mreže, a posebno duboke neuronske mreže (DNN[12]) i konvolucione neuronske mreže (CNN[13]), predstavljaju trenutno najmoćnije alate za obradu nestrukturiranih podataka. CNN mreže su naročito efikasne u obradi vizuelnih informacija, jer imaju sposobnost da same uče hijerarhiju karakteristika iz sirovih podataka, bez potrebe za prethodnim inženjeringom.



Slika 6. Dijagram neuronske mreže (https://hr.education-wiki.com/4761295-classification-of-neural-network)

[12] DNN (Deep Neural Networks) - duboke neuronske mreže su klasa vještačkih neuronskih mreža koje se sastoje od više skrivenih slojeva između ulaznog i izlaznog sloja

[13] CNN (Convolutional Neural Networks) - konvolucione neuronske mreže su specijalna vrsta dubokih neuronskih mreža, posebno efikasne u obradi podataka sa prostornom strukturom, kao što su slike ili video.

Najveća prednost neuronskih mreža jeste njihova sposobnost da detektuju složene nelinearne obrasce koji su teško uočljivi čak i stručnjacima. CNN mreže se koriste u automatskoj vizuelnoj inspekciji za otkrivanje mikroskopskih pukotina, oštećenja na površinama, anomalija u teksturi i oblika, što je od suštinske važnosti u elektronici, metalurgiji i tekstilnoj industriji.

Osim CNN-ova, rekurentne neuronske mreže (RNN[14]) i transformeri se koriste za analizu vremenskih nizova i senzorskih podataka. Ovi modeli omogućavaju sofisticirane predikcije bazirane na prethodnim obrascima i kontekstu, čime se ostvaruju rani signali upozorenja na moguće defekte.

Glavna ograničenja neuronskih mreža su njihova računska zahtjevnost i potreba za velikim količinama podataka za treniranje. Bez adekvatnog skupa podataka, model može imati problema sa generalizacijom. Takođe, njihova složenost i niska interpretabilnost (tzv. "crna kutija") često predstavljaju prepreku za upotrebu u oblastima koje zahtijevaju objašnjive odluke, kao što su medicina, pravo ili bezbjednosni sistemi.

Još jedan izazov je potreba za stručnim znanjem iz oblasti dubokog učenja, što ograničava upotrebu u manjim timovima bez pristupa naprednim resursima i računarskoj infrastrukturi.

U praksi, Random Forest je pogodan za brze prototipe, inženjerske aplikacije i domene sa strukturiranim ulazima. SVM daje izuzetne rezultate kada su podaci ograničeni i jasno klasifikovani, dok su neuronske mreže nezamjenjive u radu sa slikama, sekvencama i kompleksnim senzorskim nizovima**.**

Efikasna detekcija defekata zahtijeva pažljiv balans između tehničke složenosti, resursa, kvaliteta podataka i konkretnih ciljeva projekta. Iako nema univerzalnog rješenja, razumijevanje osobina različitih algoritama omogućava informisanu odluku o tome koji pristup je najprikladniji.

U kontekstu proizvodnje, gdje su brzina i pouzdanost ključni, Random Forest predstavlja odličnu polaznu tačku. Kada je preciznost presudna, a obim podataka ograničen, SVM može biti najefikasnija opcija. U složenim sistemima, gdje dominiraju nestrukturirani podaci i visoka varijabilnost, neuronske mreže nude najviši potencijal za preciznu i adaptivnu detekciju defekata, pod uslovom da su dostupni dovoljni resursi i kvalitetni podaci.

U narednim fazama industrijskog razvoja, vjerovatno je da će se ovi algoritmi koristiti u hibridnim sistemima**,** gdje se kombinovanjem različitih pristupa postiže optimalna tačnost, objašnjivost i efikasnost u detekciji defekata. (PC Chip, 2025)

[14] RNN (Recurrent Neural Networks) - rekurentne neuronske mreže su posebna vrsta neuronskih mreža koja se koristi za obradu sekvencijalnih podataka, podataka koji imaju vremenski redoslijed.

4. INTEGRACIJA MAŠINSKOG UČENJA I PYTHON-a

U prethodnim poglavljima analizirani su koncepti testiranja softvera i primjena mašinskog učenja u otkrivanju defekata. Kako bi se ti koncepti mogli realizovati u praksi, neophodno je koristiti odgovarajuće programske jezike i alate. U tom kontekstu, Python se izdvojio kao dominantan jezik u oblasti mašinskog učenja, data science-a i automatizovanog testiranja. Ovo poglavlje analizira vezu između mašinskog učenja i Pythona, kao i razloge zbog kojih je upravo Python postao standard u ovoj oblasti.

4.1 Upotreba Python-a za mašinsko učenje

Python je jedan od najkorišćenijih programskih jezika u oblasti mašinskog učenja, kako u industriji, tako i u akademskim krugovima. Njegova popularnost ne proizilazi isključivo iz njegove jednostavne sintakse, već iz čitavog ekosistema alata, biblioteka, zajednice i podrške za različite zadatke vezane za obradu podataka, izgradnju modela i vizualizaciju rezultata. U ovom dijelu rada biće analizirani ključni faktori koji su doprinijeli sveprisutnosti Pythona u primjeni mašinskog učenja.

Prvi značajan faktor je jednostavnost jezika. Python ima veoma čitljivu sintaksu, što ga čini pogodnim kako za početnike, tako i za iskusne istraživače i inženjere. U kontekstu mašinskog učenja, gdje su eksperimenti i brze iteracije od ključne važnosti, mogućnost da se brzo napiše, pročita i izmijeni kod je neprocjenjiva. Ova čitljivost smanjuje broj grešaka, olakšava održavanje i timski rad, te omogućava bržu validaciju rezultata.

Drugi važan razlog je široka dostupnost biblioteka i alata. Python ekosistem uključuje specijalizovane biblioteke za sve faze rada u mašinskom učenju: od prikupljanja i obrade podataka (pandas, NumPy), preko građenja modela (scikit-learn, TensorFlow, PyTorch, Keras), do evaluacije i vizualizacije rezultata (matplotlib, seaborn). Ove biblioteke ne samo da nude robustan skup funkcionalnosti, već i omogućavaju korišćenje naprednih algoritama bez potrebe za implementacijom od nule.

Treći razlog odnosi se na kompatibilnost i integraciju sa drugim tehnologijama. Python se jednostavno integriše sa web servisima, bazama podataka, alatima za vizualizaciju i sistemima za upravljanje verzijama, što ga čini pogodnim za razvoj sveobuhvatnih rješenja. U kontekstu testiranja softvera, mogućnost da se Python skripte povežu sa testnim okruženjima i CI/CD platformama (kao što su Jenkins, GitHub Actions) pruža dodatnu fleksibilnost i automatizaciju.

Jedan od najvažnijih aspekata Pythona u kontekstu mašinskog učenja je njegova upotrebljivost u istraživanju i prototipiranju. Naučnici i istraživači često koriste Python u interaktivnim okruženjima kao što je Jupyter Notebook, gdje mogu kombinovati kod, tekst, formule i vizualizacije u jednom dokumentu. Ova mogućnost znatno olakšava dokumentovanje eksperimenta i dijeljenje rezultata sa kolegama.

Python takođe ima izuzetno aktivnu i raznovrsnu zajednicu korisnika. Ova zajednica doprinosi razvoju i unapređenju alata, pisanju dokumentacije, tutorijala i open-source kodova koji su dostupni svima. Veliki broj kurseva, udžbenika i online materijala omogućava lakše usvajanje znanja i bržu integraciju novajlija u kompleksne projekte. Pored toga, brojne

akademske studije i istraživanja u oblasti vještačke inteligencije se baziraju upravo na Python implementacijama, što dodatno učvršćuje njegovu poziciju.

Još jedan bitan razlog jeste njegova platformska nezavisnost. Python može da se koristi na različitim operativnim sistemima – Windows, Linux, macOS – bez većih modifikacija u kodu. Ovo ga čini pogodnim za timove koji koriste heterogene razvojne okoline. Takođe, Python omogućava jednostavno upravljanje zavisnostima i radnim okruženjima pomoću alata kao što su pip i virtualenv, što je posebno važno u reproduktivnim eksperimentima.

U savremenoj praksi, Python je često prvi izbor za razvoj modela mašinskog učenja u startup kompanijama, istraživačkim institutima i velikim korporacijama. Pored toga, brojni cloud servisi (kao što su Google Colab, AWS SageMaker, Microsoft Azure ML) nude direktnu podršku za Python, omogućavajući korisnicima da treniraju modele na daljinu bez potrebe za lokalnim resursima.

Python omogućava brzu tranziciju od prototipa do produkcijskog koda. Jednom kada se razvije i testira model, može se relativno jednostavno integrisati u aplikaciju ili sistem, bilo da je riječ o web aplikaciji, REST API servisu ili desktop softveru. Ovo ubrzava razvojni ciklus i smanjuje troškove implementacije.

Python je postao standardni jezik u oblasti mašinskog učenja zbog svoje jednostavne sintakse, bogatstva biblioteka, aktivne zajednice i visoke fleksibilnosti. Njegova uloga u razvoju sistema za otkrivanje defekata u softveru je ključna jer omogućava brzo prototipiranje, analizu i evaluaciju različitih algoritama, kao i integraciju u realne softverske sisteme. (W3Schools, 2025)

4.2 Najvažnije biblioteke

Jedan od glavnih razloga zbog kojih se Python koristi u mašinskom učenju jeste široka dostupnost moćnih biblioteka koje omogućavaju jednostavnu i efikasnu obradu podataka, izgradnju modela, evaluaciju rezultata i njihovu vizualizaciju. U ovom potpoglavlju biće predstavljene najvažnije biblioteke koje se koriste u ovoj oblasti, sa akcentom na njihove osnovne funkcionalnosti i doprinos procesu razvoja softverskih rješenja zasnovanih na mašinskom učenju.

Pandas je biblioteka za manipulaciju i analizu podataka koja omogućava efikasan rad sa tabelarnim strukturama, kao što su DataFrame objekti. Pandas omogućava čitanje i pisanje podataka iz različitih izvora (CSV, Excel, SQL baze, JSON), kao i izvođenje operacija kao što su filtriranje, agregacija, grupisanje i transformacija podataka. U kontekstu mašinskog učenja, pandas je neizostavan alat u fazi pripreme podataka. Kvalitet i tačnost ulaznih podataka direktno utiču na performanse modela, te se smatra da analitičar u ML projektima najveći dio vremena provodi upravo u obradi podataka koristeći pandas.

NumPy je osnovna biblioteka za numeričke operacije u Pythonu, koja obezbjeđuje podršku za višedimenzionalne nizove (array) i brojne matematičke funkcije. Iako korisnici često direktno ne primjećuju NumPy, gotovo sve ostale biblioteke (uključujući pandas, scikit-learn i TensorFlow) internu logiku zasnivaju na NumPy objektima i operacijama. NumPy omogućava brzo i efikasno izvođenje linearne algebre, operacija nad matricama i statističkih izračunavanja, što je osnova za sve ML algoritme

Scikit-learn je jedna od najpopularnijih biblioteka za učenje sa nadzorom (supervised learning) i bez nadzora (unsupervised learning). Pruža širok spektar algoritama, uključujući regresiju, klasifikaciju, klasterizaciju, redukciju dimenzionalnosti i validaciju modela. Posebna vrijednost scikit-learn biblioteke ogleda se u njenoj intuitivnoj strukturi i API-ju, koji omogućavaju korisnicima da brzo implementiraju modele i uspoređuju njihove performanse uz minimalan broj linija koda. Scikit-learn je naročito pogodan za prototipiranje i analizu efikasnosti različitih pristupa u otkrivanju defekata softvera.

Keras je visokonivojska biblioteka za izgradnju neuronskih mreža, koja radi kao interfejs za niženivojske biblioteke kao što je TensorFlow. TensorFlow, koji je razvijen od strane Google-a, predstavlja sveobuhvatni okvir za razvoj i treniranje dubokih neuronskih mreža. Keras omogućava jednostavnu definiciju neuronskih slojeva, funkcija greške, optimizatora i metrika, čime korisniku omogućava da se fokusira na arhitekturu modela, a ne na niskonivojsku implementaciju. Kombinacija Keras-TensorFlow omogućava treniranje modela na matičnoj ili grafičkoj ploči, što je ključno za skalabilnost i efikasnost.

Vizualizacija podataka je sastavni dio svakog projekta u mašinskom učenju, kako u fazi analize podataka tako i prilikom interpretacije rezultata modela. Matplotlib je osnovna biblioteka za pravljenje statičkih, animiranih i interaktivnih grafika u Pythonu. Seaborn je nadogradnja na matplotlib koja omogućava lakšu izradu sofisticiranih statističkih vizualizacija, uključujući heatmap-ove, boxplot-ove, pairplot-ove i druge oblike grafikona. Vizualizacije pomažu korisnicima da bolje razumiju strukturu podataka, otkriju obrasce i donesu zaključke o performansama modela.

Sinergija ovih biblioteka omogućava istraživačima i inženjerima da brzo i efikasno razvijaju modele mašinskog učenja i integrišu ih u realne sisteme za otkrivanje softverskih defekata. Zahvaljujući otvorenom kodu i aktivnoj zajednici, korisnici imaju pristup ne samo alatima, već i velikom broju resursa za učenje i razvoj. Stoga, poznavanje i pravilna upotreba ovih biblioteka predstavlja osnovu svakog uspješnog ML projekta u Python okruženju.

4.3 Primjeri jednostavnih ML zadataka u Pythonu

U nastavku su predstavljena tri osnovna zadatka iz oblasti mašinskog učenja implementirana u programskom jeziku Python. Svaki primjer ilustruje različitu kategoriju problema: klasifikaciju**,** regresiju i nenadgledano učenje (klasterovanje)**.** Primjeri koda koriste široko prihvaćene biblioteke kao što su scikit-learn, numpy, pandas i matplotlib, koje čine temelj modernog Python ekosistema za mašinsko učenje.

Primjer 1: Klasifikacija – Prepoznavanje vrste cvijeta pomoću Iris skupa podataka

Klasifikacija predstavlja nadgledanu metodu mašinskog učenja kojom se instanca svrstava u jednu od unaprijed definisanih kategorija. U ovom primjeru koristi se Iris dataset, jedan od najpoznatijih i najčešće korišćenih skupova podataka za testiranje klasifikacionih algoritama. Skup podataka sadrži 150 uzoraka cvijeta, sa četiri atributa: dužina i širina latice i čašice, te ciljnim atributom koji označava vrstu cvijeta (Setosa, Versicolor, Virginica).

Kod:

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score

# Učitavanje skupa podataka
iris = load_iris()
X = iris.data
y = iris.target

# Podjela na trening i test skup (70% : 30%)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Treniranje modela korišćenjem Random Forest klasifikatora
model = RandomForestClassifier()
model.fit(X_train, y_train)

# Izvođenje predikcija i evaluacija performansi
y_pred = model.predict(X_test)
print("Tačnost modela:", accuracy_score(y_test, y_pred))
```

Objašnjenje**:**
U ovom primjeru, koristi se Random Forest, ansambl algoritam koji kombinuje više stabala odlučivanja (decision trees) kako bi se povećala tačnost i otpornost modela na prekomjerno učenje (overfitting). Nakon podjele skupa podataka na trening i test skup, model se trenira, a zatim se na test podacima mjeri tačnost kao osnovna mjera uspješnosti klasifikacije. Visoka tačnost u ovom kontekstu ukazuje na sposobnost modela da efikasno klasifikuje nepoznate primjerke.

```
Tačnost modela: 1.0
```

Primjer 2: Regresija – Predviđanje cijene kuće na osnovu kvadrature

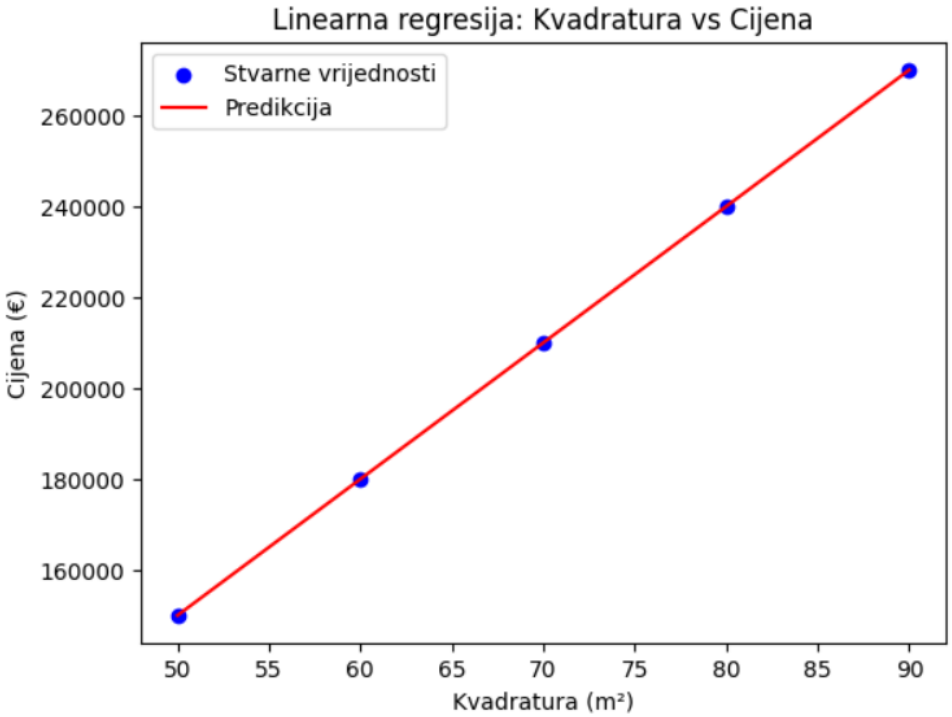
Regresija je statistička metoda predviđanja kontinuiranih vrijednosti, za razliku od klasifikacije koja predviđa diskretne klase. U ovom primjeru koristi se jednostavan linearni model kako bi se na osnovu kvadrature predvidjela cijena nekretnine. U cilju ilustracije koristi se manji broj vještački generisanih podataka, što omogućava vizualizaciju linearnog odnosa između varijabli.

Kod:

```
import numpy as np
from sklearn.linear_model import LinearRegression
import matplotlib.pyplot as plt
```

```
# Podaci: kvadratura kuća (m2) i njihove cijene (€)
X = np.array([[50], [60], [70], [80], [90]])
y = np.array([150000, 180000, 210000, 240000, 270000])

# Treniranje linearnog regresionog modela
model = LinearRegression()
model.fit(X, y)

# Predikcija cijena
predicted = model.predict(X)

# Vizualizacija rezultata
plt.scatter(X, y, color='blue', label='Stvarne vrijednosti')
plt.plot(X, predicted, color='red', label='Predikcija')
plt.title("Linearna regresija: Kvadratura vs Cijena")
plt.xlabel("Kvadratura (m²)")
plt.ylabel("Cijena (€)")
plt.legend()
plt.show()
```

Objašnjenje**:**
Linearna regresija modelira odnos između nezavisne varijable (kvadratura) i zavisne varijable (cijena). U konkretnom slučaju, model uči pravac (linearnu funkciju) koji najbolje aproksimira podatke. Dobijena regresiona linija omogućava da se cijena procijeni za bilo koju datu kvadraturu u okviru domena podataka. Vizualizacija pruža intuitivno razumijevanje preciznosti modela u odnosu na realne podatke.



Slika 7. Dijagram prikaza rezultata izvršavanja koda

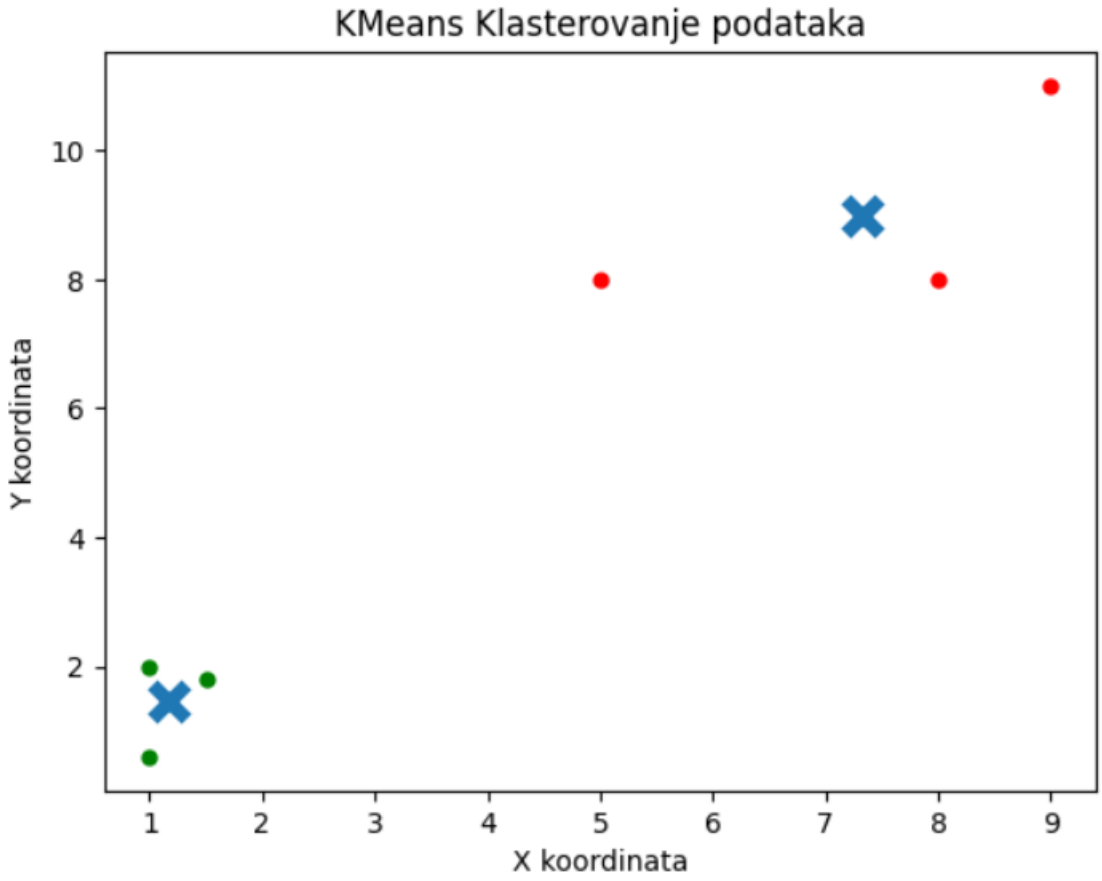
Primjer 3: Klasterovanje – Grupisanje podataka pomoću KMeans algoritma

Klasterovanje predstavlja tehniku nenadgledanog učenja kojom se skup podataka automatski dijeli u grupe (klastere) na osnovu unutrašnje sličnosti. Za razliku od klasifikacije, kod klasterovanja ne postoji unaprijed definisana oznaka klase. U ovom primjeru koristi se KMeans algoritam, koji dijeli podatke u određeni broj klastera minimiziranjem unutargrupne varijanse.

Kod:

```
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
import numpy as np

# Definisanje prostih 2D podataka
X = np.array([[1, 2], [1.5, 1.8], [5, 8],
              [8, 8], [1, 0.6], [9, 11]])

# Primjena KMeans algoritma sa 2 klastera
kmeans = KMeans(n_clusters=2)
kmeans.fit(X)

# Ekstrakcija centara klastera i oznaka
centroids = kmeans.cluster_centers_
labels = kmeans.labels_

# Vizualizacija rezultata
colors = ["g.", "r."]
for i in range(len(X)):
    plt.plot(X[i][0], X[i][1], colors[labels[i]], markersize=10)

plt.scatter(centroids[:, 0], centroids[:, 1], marker="x", s=150, linewidths=5)
plt.title("KMeans Klasterovanje podataka")
plt.xlabel("X koordinata")
plt.ylabel("Y koordinata")
plt.show()
```

Objašnjenje:
KMeans algoritam inicijalno nasumično pozicionira k centara, zatim iterativno dodjeljuje podatke najbližem centru i ažurira pozicije centara dok se ne postigne konvergencija. U ovom primjeru, podaci se automatski grupišu u dvije grupe, a rezultati se prikazuju grafički. Svaka boja označava pripadnost određenom klasteru, dok su centri klastera označeni simbolom „x“. Ova tehnika se često koristi u analizi tržišta, segmentaciji korisnika, kompresiji podataka i otkrivanju obrazaca bez prethodnog znanja o etiketama.



Slika 8. Dijagram prikaza rezultata izvršavanja koda

Prikazani primjeri demonstriraju tri temeljna pristupa u mašinskom učenju: klasifikaciju, regresiju i klasterovanje. Korišćenjem programskog jezika Python i biblioteka kao što su scikit-learn i matplotlib, omogućeno je intuitivno, brzo i efikasno modeliranje i vizualizacija različitih problema. Ovi primjeri čine osnovu za dalje istraživanje i razvoj složenijih modela u realnim aplikacijama, uključujući medicinsku dijagnostiku, finansijske predikcije i inteligentne sisteme preporuke.

4.4 Prednosti Pythona u razvoju i testiranju softvera

Python je jedan od najpopularnijih i najzastupljenijih programskih jezika današnjice. Njegova primjena obuhvata širok spektar oblasti — od nauke o podacima i vještačke inteligencije do razvoja web aplikacija, automatizacije procesa i softverskog testiranja. Razlozi sveprisutnosti Pythona u razvoju i testiranju softvera leže u njegovoj jednostavnosti, izražajnosti, bogatom ekosistemu biblioteka, i aktivnoj zajednici. U nastavku su detaljno analizirane ključne prednosti koje Python pruža prilikom razvoja i testiranja savremenih softverskih rješenja.

Jedna od najistaknutijih karakteristika Pythona je njegova jednostavna i intuitivna sintaksa, koja je bliska prirodnom jeziku. To omogućava programerima da brzo razvijaju prototipove,

ali i da lako održavaju i nadograđuju postojeći kod. Ova osobina značajno doprinosi smanjenju broja grešaka u kodu, kao i poboljšanju kolaboracije među timovima.

Na primjer, upoređujući implementaciju istih algoritama u Pythonu i jezicima kao što su Java ili C++, Python kod je obično značajno kraći i pregledniji. Ova prednost dolazi do izražaja posebno u fazi testiranja softvera, gdje je brzo pisanje i čitanje testova ključno za efikasan razvoj.

Python podržava različite paradigme programiranja — imperativno, objektno-orijentisano i funkcionalno — što omogućava prilagođavanje jezika različitim vrstama projekata. Zbog toga se Python koristi kako u razvoju malih skripti za automatizaciju, tako i u velikim, kompleksnim sistemima.

Kada se radi o testiranju softvera, ova fleksibilnost omogućava jednostavnu integraciju Pythona u bilo koji dio razvojnog ciklusa — bilo kao alat za pisanje unit testova, automatizaciju regresionih testova ili kao skriptni jezik za integraciju i testiranje API-ja. Njegova kompatibilnost sa alatima i platformama poput Jenkins-a, Docker-a, Selenium-a i CI/CD sistema dodatno olakšava njegovu primjenu u modernim DevOps okruženjima.

Jedna od najvažnijih prednosti Pythona je izuzetno bogata zbirka biblioteka koje omogućavaju testiranje softverskih komponenti na svim nivoima: od jedinica (unit testing), preko integracionih, do sistemskih i funkcionalnih testova.

- unittest - Ugrađeni Python modul koji omogućava pisanje unit testova po uzoru na JUnit iz Jave. Pruža funkcionalnosti za kreiranje test slučajeva, test suita i izvještavanja.
- pytest - Jedan od najmoćnijih i najpopularnijih frameworka za testiranje u Pythonu. Omogućava jednostavno pisanje testova uz minimalnu sintaksu, a podržava i napredne opcije kao što su fixture-i, parametrisani testovi i integracija s drugim alatima.
- Selenium - Alat za automatizaciju testiranja web aplikacija. Python biblioteka selenium omogućava upravljanje browserima i simulaciju korisničke interakcije sa web interfejsima.
- tox omogućava automatizaciju testiranja u više okruženja i verzija Pythona.
- coverage mjeri pokrivenost koda testovima.
- hypothesis koristi tehnike generativnog testiranja (property-based testing), automatski otkrivajući rubne slučajeve.

Zahvaljujući svom interpretiranom karakteru, Python omogućava brzu izradu i izvršavanje koda bez potrebe za komplikovanim procesom kompilacije. Ovo je od izuzetne važnosti prilikom istraživanja novih rješenja, razvoja prototipa, testiranja ideja i pisanja privremenih skripti za provjeru funkcionalnosti.

U realnim uslovima razvoja softvera, često je potrebno brzo provjeriti određenu logiku, validaciju podataka, performanse API poziva ili logiku komunikacije između komponenti. Python, u kombinaciji s alatima poput requests za HTTP testiranje ili sqlite3 za rad sa bazama podataka, omogućava lako i efikasno izvođenje ovih zadataka.

Python ima jednu od najaktivnijih i najbrojnijih programerskih zajednica na svijetu. To znači da je podrška u vidu dokumentacije, foruma, tutorijala i besplatnih resursa izuzetno razvijena. Otvorenost ekosistema i dostupnost izvornog koda velikog broja biblioteka omogućavaju dubinsko razumijevanje rada softverskih komponenti, što je od ključnog značaja u naprednom testiranju, posebno kada se testira stabilnost, sigurnost i pouzdanost softverskog sistema.

U savremenim softverskim sistemima, automatizacija procesa testiranja i isporuke (CI/CD) predstavlja ključnu komponentu brze i sigurne isporuke proizvoda. Python se savršeno uklapa u ovu paradigmu zahvaljujući alatima kao što su:

- fabric**,** invoke**,** ansible – alati za automatizaciju deploy procesa i konfigurisanje servera.
- pytest + GitHub Actions/Jenkins/GitLab CI – omogućavaju automatsko pokretanje testova nakon svakog commita.
- Docker integracija – jednostavna upotreba Pythona u kontejnerizovanim aplikacijama.

Zbog toga je Python postao nezamjenjiv alat u automatizaciji i upravljanju softverskim životnim ciklusom.

Python radi na svim glavnim operativnim sistemima (Windows, Linux, macOS), što ga čini idealnim za razvoj softverskih rješenja koja treba da budu prenosiva. Takođe, alati napisani u Pythonu mogu se lako prilagoditi za različita razvojna i testna okruženja, što pojednostavljuje upravljanje verzijama i okruženjima u složenim softverskim sistemima.

Python se pokazao kao jedan od najefikasnijih jezika za razvoj i testiranje savremenih softverskih rješenja. Njegova jednostavnost, fleksibilnost, dostupnost alata i biblioteka za sve vrste testiranja, kao i mogućnosti automatizacije i integracije u CI/CD okruženja, čine ga idealnim izborom kako za početnike, tako i za iskusne inženjere. U savremenoj praksi, gdje se od razvojnih timova zahtijeva brza isporuka visokokvalitetnog koda, Python ne samo da olakšava razvoj funkcionalnosti, već značajno doprinosi kvalitetu softvera kroz moćne mehanizme testiranja i automatizacije.

5. PRAKTIČNA IMPLEMENTACIJA MODELA U PYTHON-u

U ovom poglavlju prikazana je praktična primjena tehnika mašinskog učenja za otkrivanje defekata u softveru, korištenjem programskog jezika Python. Cilj ove implementacije je da se kroz konkretan eksperiment, baziran na stvarnim skupovima podataka, demonstrira kako različiti algoritmi mašinskog učenja mogu poslužiti u detekciji softverskih defekata prije nego što se softver pusti u produkciju. Ovakav pristup je naročito značajan u kontekstu sve veće potrebe za automatizovanom kontrolom kvaliteta u softverskom inženjerstvu.

Cjelokupan proces modeliranja obuhvata niz međusobno povezanih koraka. Prvi korak odnosi se na odabir i pripremu odgovarajućeg skupa podataka, koji mora biti reprezentativan za problematiku detekcije defekata i sadržavati dovoljan broj primjera defektnih i nedefektnih softverskih modula. Nakon toga slijedi faza predobrade podataka, koja uključuje obradu nedostajućih vrijednosti, transformaciju karakteristika, te balansiranje skupa podataka ukoliko je on klasno neuravnotežen – što je česta pojava u ovoj oblasti (Khoshgoftaar et al., 2007). Pravilna obrada podataka u ovom koraku ima direktan uticaj na performanse modela i pouzdanost rezultata.

Nakon pripreme podataka, slijedi ključna faza – treniranje modela. U ovom koraku, koriste se različiti algoritmi nadgledanog učenja, kao što su logistička regresija, stablo odlučivanja (Decision Tree), nasumične šume (Random Forest), podržavajuće mašine (Support Vector Machines – SVM), kao i neuronske mreže. Osim toga, hiperparametri svakog modela optimizovani su korištenjem metoda poput unakrsne validacije (cross-validation) i pretrage mreže (grid search), kako bi se postigle najbolje moguće performanse.

Evaluacija modela vrši se pomoću standardnih metrika klasifikacije, uključujući tačnost (accuracy), preciznost (precision), odziv (recall), F1-mjeru, kao i područje ispod ROC krive (AUC-ROC). Poseban fokus stavljen je na metrike koje bolje oslikavaju ponašanje modela u uslovima klasne neravnoteže, jer jednostavna tačnost može biti zavaravajuća u slučaju kada postoji velika dominacija nedefektnih modula. Na primjer, u skupu gdje je samo 10% modula defektno, model koji sve module klasifikuje kao nedefektne imat će tačnost od 90%, ali njegov odziv za defektne module bit će nula – što je neprihvatljivo u realnoj primjeni. Stoga je važno posmatrati kombinovane metrike koje daju uravnotežen uvid u performanse modela na svim klasama.

Python je odabran kao razvojno okruženje iz više razloga. Prvo, Python nudi izuzetno bogat i dobro dokumentovan skup biblioteka za obradu podataka i implementaciju algoritama mašinskog učenja. Biblioteke poput pandas i numpy omogućavaju efikasnu manipulaciju podacima i izvođenje numeričkih operacija, dok scikit-learn pruža širok spektar gotovih algoritama, funkcija za validaciju modela i evaluaciju rezultata. Vizuelizacija podataka i rezultata implementirana je pomoću matplotlib i seaborn biblioteka, koje omogućavaju jasan i intuitivan prikaz složenih odnosa između varijabli i performansi modela.

Dodatno, Python kao programski jezik karakteriše visoka čitljivost i jednostavnost pisanja koda, što omogućava brzu iteraciju i eksperimentisanje sa različitim pristupima. Njegova fleksibilnost čini ga idealnim za istraživački rad i prototipizaciju modela, kao i za skaliranje rješenja na industrijskom nivou. Sve veća popularnost Pythona u zajednici istraživača i inženjera doprinosi bogatstvu dostupne dokumentacije, otvorenog koda i naučnih radova koji omogućavaju kontinuirano unapređenje znanja i praksi u oblasti mašinskog učenja.

U okviru ovog poglavlja prikazana je praktična implementacija algoritma mašinskog učenja za otkrivanje defekata softvera korišćenjem programskog jezika Python. Cilj je demonstrirati kako se stvarni skup podataka o softverskim modulima može analizirati korišćenjem tehnika nadgledanog učenja kako bi se predvidjela prisutnost defekta (greške) u modulu. Ovaj proces uključuje pripremu podataka, obuku modela, evaluaciju rezultata i diskusiju o dobijenim mjerenjima tačnosti i učinkovitosti modela. Korišćen je realan dataset poznat kao NASA CM1, a model je realizovan korišćenjem biblioteka pandas, scikit-learn i matplotlib. Ovaj primjer predstavlja jednostavan ali dovoljno reprezentativan eksperiment primjene mašinskog učenja u kontekstu softverskog inženjeringa.

Za ovaj eksperiment odabrana je Naivni Bajes (Naive Bayes) klasifikacija kao model mašinskog učenja zbog svoje jednostavnosti i efikasnosti kod binarne klasifikacije. Kao tehnika evaluacije performansi modela koristi se podjela podataka na trening i test skup pomoću funkcije train_test_split, gdje 70% podataka pripada trening skupu, a 30% test skupu.

Cilj eksperimenta je da se model obuči da prepozna karakteristike softverskih modula koji sadrže greške i da potom, na osnovu novih podataka, klasifikuje da li su oni defektni ili ne. Ovakav pristup je primjenjiv u realnim softverskim projektima gdje se unaprijed želi znati koji dijelovi sistema su potencijalno problematični. (International Journal of Computer Applications, 2025)

5.1 Opis i priprema skupa podataka

U savremenim istraživanjima iz oblasti softverskog inženjerstva, a naročito u domenu otkrivanja defektnih softverskih komponenti, značajnu ulogu imaju istorijski skupovi podataka koji sadrže informacije o karakteristikama softverskih modula i njihovoj sklonosti ka greškama. U okviru ovog istraživanja, korišćen je NASA-in javno dostupan skup podataka CM1, koji je dio serije dataset-ova namjenjenih za istraživanja o predikciji defekata u softveru.

5.1.1 Opis skupa podataka CM1

Skup podataka CM1 potiče iz NASA-inog projekta ugrađenih sistema, konkretno iz domene svemirskih i satelitskih softverskih komponenti. Svaki zapis u ovom dataset-u predstavlja jedan softverski modul – osnovnu jedinicu analize u kontekstu ovog rada. Modul se ovde može posmatrati kao zasebna funkcionalna cjelina koja ima svoje metrike kompleksnosti i performansi.

Podaci sadrže brojne metričke varijable koje kvantitativno opisuju softverske module. Među najvažnijim metrikama nalaze se:

- LOC (Lines of Code) – broj linija koda;
- V(G) – ciklomatska kompleksnost (mjera broja nezavisnih puteva kroz program);
- Halsteadove metrike – uključujući broj operatora i operanada, kao i izvedene mjere kao što su "difficulty", "effort" i "volume";
- Broj funkcija – ukupna količina funkcija definisanih unutar modula;
- Broj komentara – uključuje informaciju o nivou dokumentovanosti koda.

Pored ovih kvantitativnih karakteristika, dataset uključuje i binarnu klasifikacionu varijablu nazvanu "defective". Ova kolona ima vrijednost 1 ukoliko je softverski modul identifikovan kao defektan (sadrži jednu ili više grešaka), odnosno vrijednost 0 ako nije detektovan nijedan defekt. Ova ciljana varijabla predstavlja osnovu za sve klasifikacione modele koji će se koristiti u daljoj analizi.

5.1.2 Priprema podataka za analizu

Prije nego što se pristupi samom modeliranju, neophodno je obaviti odgovarajuću pripremu podataka. Ovaj proces uključuje nekoliko ključnih faza:

1. Učitavanje podataka

Za učitavanje skupa podataka koristi se biblioteka pandas, koja je standardni alat u Python ekosistemu za manipulaciju tabelarnim podacima. Dataset se nalazi u formatu CSV, pa se učitava pomoću funkcije read_csv().

```
import pandas as pd
data = pd.read_csv("cm1.csv")
```

U ovom koraku se učitavaju svi redovi i kolone iz fajla cm1.csv, a rezultat je DataFrame objekat koji omogućava jednostavan pristup kolonama i redovima.

2. Izdvajanje ulaznih i ciljne varijable

Za potrebe klasifikacije, neophodno je odvojiti ulazne varijable (features) od ciljne varijable (target). U ovom slučaju, ciljna varijabla je defective, dok su sve ostale kolone ulazne karakteristike.

```
X = data.drop("defective", axis=1)
y = data["defective"]
```

Ovim pristupom se formira matrica X dimenzija (n × m), gdje je n broj modula, a m broj karakteristika. Vektor y je jednodimenzionalni niz koji sadrži binarne vrijednosti (0 ili 1) za svaki modul.

3. Normalizacija podataka

S obzirom na to da se različite metrike izražavaju u različitim numeričkim opsezima (broj linija koda može imati znatno veće vrijednosti u odnosu na ciklomatsku kompleksnost), preporučljivo je izvršiti normalizaciju podataka. Ovo je posebno važno kada se koriste algoritmi osjetljivi na skalu varijabli, kao što su logistička regresija, SVM, KNN i neuronske mreže.

Primjer normalizacije pomoću MinMaxScaler:

```
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()
X_scaled = scaler.fit_transform(X)
```

Rezultat normalizacije je da sve vrijednosti budu u opsegu [0,1], čime se postiže uniformnost i ubrzava konvergencija algoritama.

4. Obrada nedostajućih vrijednosti

U stvarnom svijetu, skupovi podataka često sadrže nedostajuće vrijednosti, koje mogu nastati zbog grešaka u mjerenju, prikupljanju ili obradi podataka. Neobrađene nedostajuće vrijednosti mogu značajno narušiti performanse modela. Moguće strategije uključuju:

- Uklanjanje redova ili kolona sa previše NaN vrijednosti;
- Popunjavanje sa srednjom vrijednošću (mean), medijanom ili najčešćom vrijednošću;
- Korišćenje naprednijih metoda imputacije.

U ovom istraživanju, pretpostavlja se da dataset ne sadrži nedostajuće vrijednosti, ili su one prethodno uklonjene/imputirane.

5. Podjela skupa na trening i test dio

Za evaluaciju modela, podaci se dijle na trening i test podskup. Trening skup se koristi za obučavanje modela, dok se test skup koristi za provjeru performansi na do tada neviđenim podacima. Standardna podjela uključuje 70% podataka za trening i 30% za test.

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    X_scaled, y, test_size=0.3, random_state=42
)
```

Parametar random_state služi za reproduktivnost eksperimenta, obezbjeđuje da se prilikom svakog pokretanja podjela izvrši na isti način.

Prilikom analize ovakvih dataset-ova, često se primjećuje značajna neuravnoteženost klasa, pri čemu defektnih modula ima znatno manje u odnosu na one koji nisu defektni. Ova pojava može negativno uticati na performanse modela ukoliko se ne adresira pravilno. Tipične tehnike za rješavanje ovog problema uključuju:

- Oversampling manjinske klase (SMOTE algoritam),
- Undersampling većinske klase,
- Korišćenje algoritama otpornih na neuravnoteženost (Random Forest, XGBoost),
- Uvođenje pondera (weights) za klase.

U nastavku, biće opisani različiti klasifikacioni algoritmi koji su trenirani i evaluirani na osnovu prethodno pripremljenog CM1 skupa podataka, s ciljem izgradnje pouzdanog prediktivnog modela za detekciju defektnih softverskih modula.

5.2 Treniranje i evaluacija modela

Efikasna predikcija defektnih softverskih modula oslanja se na primjenu mašinskog učenja, gdje je izbor modela, način treniranja i evaluacija od suštinske važnosti za validnost rezultata. U okviru ovog istraživanja, za klasifikaciju modula kao defektnih ili ne, primjenjen je Naivni Bajesov klasifikator, konkretno njegova varijanta Gaussian Naive Bayes (GaussianNB), koja je pogodna za podatke koji prate normalnu raspodjelu. Ovaj metod je odabran zbog jednostavnosti, brzine treniranja i dobre performanse na zadacima binarne klasifikacije, čak i kada je količina podataka ograničena.

5.2.2 Implementacija i treniranje modela

Za implementaciju Naivnog Bajesovog klasifikatora korišćena je biblioteka scikit-learn, koja pruža jednostavan API za treniranje i evaluaciju modela.

```
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import classification_report

# Inicijalizacija i treniranje modela
model = GaussianNB()
model.fit(X_train, y_train)
```

U ovoj fazi, model "uči" iz podataka – koristi ulazne karakteristike (X_train) i pripadajuće klase (y_train) kako bi izračunao srednje vrijednosti i varijanse za svaku karakteristiku, u okviru svake klase. Na osnovu toga, kasnije se može procijeniti kojoj klasi najvjerovatnije pripada nova instanca.

Važno je napomenuti da je GaussianNB vrlo efikasan u smislu vremena treniranja, jer koristi jednostavne analitičke formule bez iterativnog optimizovanja.

5.2.3 Testiranje modela

Nakon treniranja modela, isti se primjenjuje na test skup, koji sadrži podatke koje model ranije nije vidio. Ovim se simulira situacija iz realnog svijeta, primjena modela na novim softverskim modulima.

```
# Predikcija klasa na test skupu
y_pred = model.predict(X_test)
```

Metoda predict() koristi naučene parametre da izračuna vjerovatnoće i donese odluku o klasi za svaki primjer iz test skupa. Rezultat je niz predikcija koji se dalje upoređuje sa stvarnim vrijednostima (y_test) u cilju evaluacije.

5.2.4 Evaluacija performansi

Za evaluaciju modela korišćene su sljedeće standardne metrike:

1. Preciznost (Precision)

Preciznost predstavlja procenat tačnih pozitivnih predikcija u odnosu na sve pozitivne predikcije.

2. Odziv (Recall)

Odziv mjeri koliko model uspješno pronalazi sve stvarno defektne modul. Lažno negativni - defektni moduli klasifikovani kao ispravni.

3. F1-mjera

F1-mjera predstavlja harmonijsku sredinu između preciznosti i odziva. Ova metrika je posebno korisna kod neuravnoteženih podataka, gdje je balans između preciznosti i odziva važan.

4. Tačnost (Accuracy)

Ukupna tačnost mjeri procenat tačnih predikcija (i pozitivnih i negativnih) u odnosu na ukupan broj primjera:

```
# Evaluacija
print(classification_report(y_test, y_pred))
```

Metoda classification_report prikazuje sve gore navedene metrike za obe klase (defektne i nedefektne), kao i makro i mikro prosjeke, što omogućava uvid u uravnoteženost klasifikacije.

5.2.5 Interpretacija rezultata

Na osnovu dobijenih metrika moguće je ocjeniti kvalitet klasifikatora. Na primjer:

- Ako model ima visoku tačnost ali nizak odziv za klasu "1", to znači da uspješno identifikuje većinu nedefektnih modula, ali propušta defektne – što može biti kritično u stvarnim sistemima**.**
- Sa druge strane, visok odziv uz nižu preciznost ukazuje da model detektuje većinu defektnih modula, ali uz veći broj lažnih uzbuna – što može biti prihvatljivo u bezbjednosno osjetljivim sistemima, gdje je bolje spriječiti nego liječiti.

U softverskom inženjerstvu, često se teži maksimizaciji F1-mjere za klasu "defective", jer ona uzima u obzir i preciznost i odziv, čime se dobija uravnotežen indikator efikasnosti.

5.2.6 Prednosti i ograničenja pristupa

Prednosti GaussianNB:

- Brzina treniranja, idealno za velike dataset-ove.
- Dobar inicijalni model za referencu (baseline).
- Pogodan za visoko-dimenzionalne podatke.
- Otporan na pretreniranje kod jednostavnijih problema.

Ograničenja:

- Pretpostavka normalne raspodjele može biti nerealna.
- Uslovna nezavisnost karakteristika rijetko se ispunjava u stvarnosti.
- Osjetljivost na varijabilnost podataka i outliere.

Zbog navedenih ograničenja, GaussianNB se često koristi kao početna tačka u analizi, nakon čega se testiraju kompleksniji modeli (Random Forest, SVM, XGBoost, i drugi).

Primjena Naivnog Bajesovog klasifikatora u kontekstu predikcije softverskih defekata pokazuje se kao efikasna metoda u situacijama kada su podaci relativno čisti, a klasifikacija binarna. Uprkos svojoj jednostavnosti, model daje konkurentne rezultate u pogledu F1-mjere i odziva, posebno kada je cilj brza i robusna procjena defektnih modula.

Međutim, za postizanje maksimalnih performansi preporučuje se dodatna analiza: optimizacija metrika, evaluacija na više dataset-ova i upoređivanje sa naprednijim algoritmima.

5.3 Prikaz rezultata

Nakon treniranja modela i testiranja na izdvojenom skupu podataka, rezultati su analizirani kroz standardne metrike: preciznost, odziv, F1-mjera i tačnost. Posebna pažnja posvećena je interpretaciji metrika u kontekstu problema predikcije softverskih defekata, kao i uočenim izazovima u vezi sa strukturom i uravnoteženošću podataka.

U dataset-u CM1, kao i u mnogim dataset-ovima iz oblasti softverskog inženjeringa, prisutna je neuravnoteženost klasa: broj nedefektnih modula značajno nadmašuje broj defektnih. Ova disproporcija otežava zadatak klasifikatora, jer model može da postigne visoku ukupnu tačnost jednostavno favorizujući većinsku klasu.

Na primjer, model koji bi uvijek predviđao da su svi moduli nedefektni imao bi tačnost preko 80%, ali bi bio potpuno beskoristan za praktičnu upotrebu jer ne bi identifikovao nijedan defektni modul.

Ova neuravnoteženost se direktno odražava na niži odziv za klasu 1. GaussianNB, iako robustan, ne koristi mehanizme za kompenzaciju neuravnoteženih podataka, što objašnjava njegovu slabiju performansu u detekciji defektnih modula. U realnom svijetu, takva situacija može dovesti do ozbiljnih posljedica, jer neotkriveni defektni moduli predstavljaju rizik za stabilnost i bezbjednost softverskog sistema.

S obzirom na pomenute izazove, performanse modela mogu se interpretirati kroz sljedeće tačke:

1. Visoka tačnost ali ograničena korisnost

Ukupna tačnost od 84% može izgledati zadovoljavajuće, ali s obzirom na nisku F1-mjeru za defektne module, jasno je da model nije optimalno pouzdan za detekciju rizika. U kontekstu predikcije defekata, gdje je cilj upravo otkrivanje potencijalno problematičnih modula, važnije je maksimizirati odziv i F1-mjeru za klasu 1.

2. Podložnost pretpostavkama raspodjele

GaussianNB pretpostavlja da podaci slijede normalnu raspodjelu. Međutim, u softverskoj metriki često se susreće sa asimetričnim raspodjelama, outlier-ima i nelinearnim zavisnostima, što može negativno uticati na procjenu vjerovatnoća unutar modela.

3. Jednostavnost modela kao prednost i ograničenje

Jedna od prednosti GaussianNB je njegova jednostavnost i transparentnost. S obzirom da model koristi zatvorene formule za izračunavanje vjerovatnoće, lako je analizirati uzrok grešaka. Međutim, u kompleksnijim problemima, gdje odnosi između metrika nisu linearni niti nezavisni, jednostavnost može postati ograničavajući faktor u tačnosti klasifikacije.

Iako rezultati pokazuju da GaussianNB uspješno klasifikuje većinu nedefektnih modula, njegova niska sposobnost da detektuje defektne module smanjuje njegovu upotrebnu vrijednost u fazi verifikacije i validacije softvera. Praktična implikacija ovog nalaza je da bi se GaussianNB mogao koristiti kao početni filter za eliminaciju očigledno ispravnih modula, dok bi se za detaljniju analizu koristili složeniji modeli.

Takođe, rezultati sugerišu potrebu za dodatnom obradom podataka prije treniranja modela, kao što su:

- Upotreba tehnika balansiranja
- Uvođenje ponderisane klasifikacije, gdje se većoj klasi dodjeljuje manja težina,
- Korišćenje ensemble metoda (Random Forest ili XGBoost), koji bolje prepoznaju složenije obrasce.

Na osnovu izvedene evaluacije može se utvrditi:

1. Primjena alternativnih algoritama: Testirati naprednije klasifikatore poput Support Vector Machines (SVM), Random Forest, ili neuronskih mreža, koji bolje podnose nelinearnosti i neuravnoteženost.
2. Eksperimentisanje sa inženjeringom karakteristika: Uvođenje dodatnih softverskih metrika (broj komentara, zasićenost grananja, coupling/ cohesion indeksi) može poboljšati model.
3. Kros-validacija: Umjesto jedne podjele na trening i test skup, koristiti višestruku kros-validaciju radi pouzdanije procjene performansi.
4. Analiza grešaka: Detaljna analiza primjera koje je model pogrešno klasifikovao može otkriti specifične obrasce i pomoći u unapređenju budućih modela.

Rezultati pokazani u ovom poglavlju ukazuju da GaussianNB, iako efikasan i brz, ima određena ograničenja u domenu detekcije softverskih defekata, posebno kada su podaci neuravnoteženi. Dok tačnost može biti visoka, performanse u detekciji defektnih modula zahtijevaju dodatnu pažnju.

Uprkos tome, vrijednost ovakvog modela ne treba potcijeniti – kao početni korak u detekciji defekata, on može igrati važnu ulogu u širem sistemu automatizovane evaluacije kvaliteta softvera. Dodatne analize i upotreba naprednijih metoda su preporučeni sljedeći koraci u cilju povećanja tačnosti i upotrebljivosti sistema u realnim softverskim projektima.

6. ZAKLJUČAK

U ovom radu detaljno je istražena mogućnost primjene mašinskog učenja u oblasti detekcije softverskih defekata, sa posebnim fokusom na jedan od najjednostavnijih, ali i veoma efikasnih algoritama – Naivni Bajesov klasifikator (GaussianNB). Implementacija modela je realizovana u programskom jeziku Python, koristeći popularne biblioteke za obradu podataka i mašinsko učenje, što omogućava lakoću reprodukcije i primjene u praktičnim okruženjima.

Analiza je sprovedena na osnovu javno dostupnog dataset-a CM1, koji sadrži skup metrika o softverskim modulima iz projekta ugrađenih sistema. Ovaj skup podataka karakteriše prisustvo raznovrsnih metrika koje opisuju složenost, veličinu i druge karakteristike modula, kao i binarni indikator da li je modul defektan ili ne. Takav pristup omogućava izgradnju modela koji može predvidjeti prisustvo defekata na osnovu relevantnih softverskih parametara.

Rezultati dobijeni treniranjem Naivnog Bajesovog klasifikatora na ovom dataset-u ukazuju da model postiže ukupnu tačnost od približno 84%. Ova vrijednost predstavlja mjeru koliko je model generalno uspješan u pravilnoj klasifikaciji modula kao defektnih ili nedefektnih. Ipak, detaljnija analiza performansi kroz druge metrike kao što su preciznost, odziv i F1-mjera pokazuje značajne razlike u sposobnosti modela da prepozna defektne module u odnosu na nedefektne.

Naime, model pokazuje znatno bolje rezultate u identifikaciji nedefektnih modula, dok je njegova sposobnost detekcije defektnih modula ograničena, što se odražava kroz znatno niži odziv i F1-mjeru za klasu defektnih modula. Ovakvo ponašanje modela može se objasniti prirodom podataka, koji su neuravnoteženi, odnosno broj modula koji nisu defektni značajno prevazilazi broj onih koji su defektni. Neuravnoteženost podataka predstavlja jedan od ključnih izazova u oblasti primjene mašinskog učenja na problemima klasifikacije i zahtijeva dodatne tehnike za balansiranje ili specifične pristupe u samom procesu treniranja modela.

Pored izazova vezanih za neuravnoteženost, vrijedno je istaći i inherentna ograničenja Naivnog Bajesovog klasifikatora. Ovaj algoritam polazi od pretpostavke da su ulazne promjenljive nezavisne i da slijede normalnu raspodjelu unutar svake klase, što u realnosti softverskih metrika često nije slučaj. Softverski moduli i njihove metrike mogu imati složene međuzavisnosti i distribucije koje odstupaju od Gaussove krive, što može uticati na smanjenje preciznosti modela.

U radu je takođe dat poseban osvrt na pripremu podataka, koja je ključni korak u svakom procesu primjene mašinskog učenja. Priprema dataset-a obuhvatila je izdvajanje relevantnih karakteristika (features) i ciljne promjenljive (target), obradu eventualnih nedostataka i nedostajućih vrijednosti, kao i normalizaciju podataka. Posebno je naglašena važnost podjele podataka na trening i test skupove radi validacije modela i izbjegavanja prenaučenosti. Ovaj pristup omogućava objektivnu procjenu performansi modela na podacima koje model ranije nije video.

Kroz praktični dio rada, implementacija modela u Python-u pokazala je da je moguće uz relativno jednostavne korake i pristupe postići solidne rezultate u zadatku predikcije defekata. Ujedno, ukazano je na to da za kompleksnije i zahtjevnije aplikacije, kao što su softverski

projekti sa velikim brojem modula i složenim arhitekturama, postoji potreba za korištenjem sofisticiranijih metoda i tehnika.

Rad potvrđuje da mašinsko učenje, čak i u svojoj osnovnoj formi, predstavlja značajan alat u oblasti testiranja softvera, sa potencijalom da unaprijedi postojeće metode otkrivanja defekata, poveća efikasnost procesa kontrole kvaliteta i smanji vrijeme i troškove razvoja softvera.

Rezultati istraživanja imaju praktičnu vrijednost za razvojne timove i QA inženjere, jer ukazuju na potencijal korišćenja mašinskog učenja kao dopune tradicionalnim metodama testiranja. Naivni Bajesov klasifikator može poslužiti kao brz i jednostavan filter za preliminarnu klasifikaciju modula, dok bi složeniji algoritmi mogli biti korišćeni za detaljniju analizu.

Poseban značaj leži u mogućnosti integracije ovakvih modela u CI/CD alate, čime bi se omogućila automatska provjera kvaliteta modula već u fazi razvoja, prije nego što se softver plasira u proizvodno okruženje.

Rad potvrđuje da mašinsko učenje ima značajan potencijal u domenu detekcije softverskih defekata, ali i da uspješnost njegove primjene zavisi od kvalitetne pripreme podataka, izbora odgovarajućeg algoritma i dublje domenske ekspertize u oblasti softverskog inženjeringa.

LITERATURA I TABELA SLIKA

(2025, Jul 3). Preuzeto sa International Software Testing Qualifications Board: https://www.istqb.org/

(2025, Jul 3). Preuzeto sa Katalon: https://katalon.com/resources-center/blog/bug-defect-life-cycle

(2025, jul 2). Preuzeto sa Geeks for Geeks: https://www.geeksforgeeks.org/software-engineering/software-engineering/

(2025, jul 2). Preuzeto sa Geeks for Geeks: https://www.geeksforgeeks.org/software-testing/software-engineering-differences-between-manual-and-automation-testing/

(2025, jul 2). Preuzeto sa IBM: https://www.ibm.com/think/topics/machine-learning

(2025, jul 3). Preuzeto sa PC Chip: https://pcchip.hr/helpdesk/sto-je-to-machine-learning-ili-strojno-ucenje/

(2025, jul 3). Preuzeto sa W3Schools: https://www.w3schools.com/python/python_ml_getting_started.asp

(2025, jul 3). Preuzeto sa International Journal of Computer Applications: https://www.ijcaonline.org/

Pressman, Roger S.; Maxim, Bruce R. (2014). *Software Engineering: A Practitioner's Approach* (8th izd.). New York,: McGraw-Hill Education.

Slika 1. Dijagram procesa testiranja softvera (https://www.helloworld.rs/blog/Kako-izgleda-proces-testiranja-softvera/12291) 8

Slika 2. Dijagram životnog ciklusa defekta (https://katalon.com/resources-center/blog/bug-defect-life-cycle) 12

Slika 3. Web platforma "My Knee Rehabilitation"..... 13

Slika 4. Random Forest algoritam (https://fr.pinterest.com/pin/848787861035795075/visual-search/?x=16&y=16&w=532&h=532&surfaceType=flashlight) 28

Slika 5. SVM algoritam (https://artpictures.club/autumn-2023.html) 28

Slika 6. Dijagram neuronske mreže (https://hr.education-wiki.com/4761295-classification-of-neural-network) 29

Slika 7. Digagram prikaza rezultata izvršavanja koda 35

Slika 8. Digagram prikaza rezultata izvršavanja koda 37