OTS 2019
SODOBNE INFORMACIJSKE TEHNOLOGIJE IN STORITVE
M. Heričko in K. Kous (ur.)

Univerzitetna založba
Univerze v Mariboru

KORAK K SAMODEJNI INTEGRACIJI IN TESTIRANJU NAPOVEDNIH MODELOV STROJNEGA UČENJA

GREGA VRBANČIČ, VILI PODGORELEC

Povzetek: *Z razcvetom strojnega učenja in prodorom v vse veje gospodarstva, se nenehno povečuje tudi potreba po ustaljenih razvojnih procesih pri razvoju napovednih modelov strojnega učenja, ki bi olajšala njihovo vpeljavo in integracijo v obstoječe informacijske sisteme. Podobno kot je to že ustaljena praksa pri programskem inženirstvu, je potreba in želja gospodarstva, da se tudi pri razvoju napovednih modelov strojnega učenja razvijejo smernice in prakse, ki bi omogočale enostavno vpeljavo, hiter in prilagodljiv razvoj ter nadzor nad kvaliteto dostavljenih napovednih modelov. V prispevku bomo predstavili smernice, izzive in potencialne rešitve pri vpeljavi napovednih modelov v proces samodejne neprekinjene integracije ter prikazali praktičen primer vpeljave napovednega modela.*

Ključne besede: *samodejna neprekinjena integracija, napovedni modeli, strojno učenje, testiranje, razvoj programske opreme*

NASLOVA AVTORJEV: Grega Vrbančič, mladi raziskovalec, Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko, Maribor, Slovenija, e-pošta: grega.vrbancic@um.si.
dr. Vili Podgorelec, redni profesor, Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko, Maribor, Slovenija, e-pošta: vili.podgorelec@um.si.

https://doi.org/10.18690/978-961-286-282-4.3 ISBN 978-961-286-282-4
Dostopno na: http://press.um.si

1 UVOD

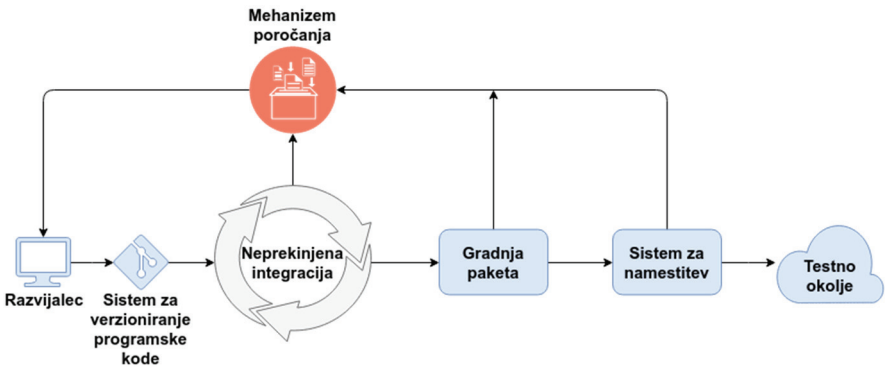
S prodorom umetne inteligence in tehnik strojnega učenja v praktično vse veje gospodarstva, še posebej pa na področje informacijskih tehnologij, se vedno pogosteje zastavlja tudi vprašanje o integraciji in zagotavljanju kakovosti takšnih rešitev. V nasprotju z na primer programskim inženirstvom, kjer so procesi integracije in testiranja programske kode ter programske metrike že dodobra opredeljene in uveljavljene, pa pri razvoju napovednih modelov trenutno še ni dobro definiranih okvirjev oz. strategij, kako jih pravilno in učinkovito integrirati, testirati in zagotavljati kakovost. Sprejeti in uveljavljeni procesi testiranja in zagotavljanja kakovosti napovednih modelov so nujni za njihovo množično integracijo v obstoječe ali celo kritične dele informacijskih sistemov. Da bi lahko dobro definirali procese testiranja in zagotavljanja kakovosti napovednih modelov, pa moramo najprej definirati, kaj na področju strojnega učenja sploh pomeni testiranje. Primarno se izraz testiranje, v povezavi z umetno inteligenco, navezuje na testiranje uspešnosti napovednih modelov. Na uspešnost napovednih modelov vpliva velik nabor odvisnih in neodvisnih spremenljivk, zato je na mestu vprašanje, kako sploh testirati napovedne modele? Kako zagotoviti, da bodo delovali v skladu z našimi zahtevami in pričakovanji?

V prispevku bomo predstavili problematiko oziroma izzive samodejne neprekinjene integracije in testiranja ter zagotavljanja kakovosti napovednih modelov ter poizkušali odgovoriti na zastavljena vprašanja. Predstavili bomo že uveljavljen koncept samodejne neprekinjene integracije ter ga razširili z vpeljavo korakov in komponent, specifičnih za razvoj napovednih modelov strojnega učenja. Prav tako bomo opisali obstoječe smernice in pristope zagotavljanja kakovosti napovednih modelov ter naslovili problematiko integracije napovednih modelov v obstoječe informacijske sisteme in na praktičnem primeru prikazali končen doprinos uvedbe testiranja na delovanje napovednega modela.

2 SAMODEJNA NEPREKINJENA INTEGRACIJA

Integracija programske opreme je pristop povezovanja podsistemov in komponent programske opreme z namenom izgradnje enotnega sistema. Integracija je del vsakega razvojnega življenjskega cikla programske opreme, saj ekipa inženirjev programsko opremo razvija skozi različne faze. [1] Leta je integracija programske opreme predstavljala tiičen in nepredvidljivosti poln korak v življenjskem ciklu razvoja programske opreme. Že več kot dve desetletji pa problematika tega koraka počasi pojenja, zahvaljujoč pojavitvi samodejne neprekinjene integracija (angl. Continuous Integration, CI). Začetki CI izhajajo iz Kent Blockovih dvanajstih praks razvoja programske opreme, poznanega kot eXtreme Programming (XP) [2]. Avtor je CI okarakteriziral kot proces, v katerem je nova programska koda integrirana v trenuten sistem v največ nekaj urah, pri čemer je sistem vedno zgrajen v celoti od začetka, spremembe programske kode pa se ohranijo v primeru, da so bili uspešno prestani vsi testi. [3].

Tipičen scenarij procesa neprekinjene integracije programske kode (Slika 1) se začne z razvijalcem, ki prispeva (angl. commit) programsko kodo v skupen repozitorij. Pri delu na tipičnem projektu lahko deležniki, pogosto v različnih vlogah, prispevajo spremembe, ki sprožijo cikel CI. Na primer, razvijalci sprožijo cikel s spremembami programske kode, administratorji podatkovnih baz s spremembami definicij entitet, ekipe, ki skrbijo za izgradnjo in namestitev programske opreme, s spremembami konfiguracijskih datotek, itd.



Slika 1: Konceptualni prikaz procesa neprekinjene integracije programske kode.

Koraki v omenjenem tipičnem scenariju procesa integracije programske kode si v splošnem sledijo [4]:

1. Najprej deležnik prispeva spremembo v skupen repozitorij. Medtem CI strežnik neprekinjeno na določen interval opreza za morebitnimi spremembami na repozitoriju ali posluša na izpostavljenem spletnem vmesniku (angl. webhook) za morebiten klic, ki sporoča o dogodku – spremembi na skupnem repozitoriju.
2. Kmalu po oddanem prispevku v repozitorij, CI strežnik zazna spremembo in pridobi najnovejšo različico programske kode ter prične z izvajanjem CI cikla.
3. Po koncu cikla CI strežnik generira poročilo, ki ga posreduje vsem deležnikom oz. poročilo ponudi na voljo v enostavno dostopni spletni obliki.
4. Opcijsko zapakira uspešno integrirano, prevedeno programsko kodo v izvršljiv paket in ga namesti na testno izvajalno okolje.

2.1 Komponente neprekinjene integracije

Kljub temu, da je samodejna neprekinjena integracija postala standard v programskem inženirstvu, v stroki še vedno ni konsenza, kako bi takšen sistem moral biti implementiran. Na voljo so različna orodja (Jenkins, Travis, GitLab CI...), ki omogočajo razvoj programske opreme z uporabo pristopa samodejne neprekinjene integracije, sicer pa pristop sam ne zahteva uporabe nobenega specifičnega orodja. Glede na omenjeno, obstaja množica napotkov in dobrih praks [4], [5], povezanih s CI in vpeljavo le-te, ki jih lahko obravnavamo kot nekakšne smernice. Martin Fowler v njegovem množično citiranem prispevku [5] predstavi deset ključnih praks, potrebnih za vzpostavitev učinkovitega CI sistema:

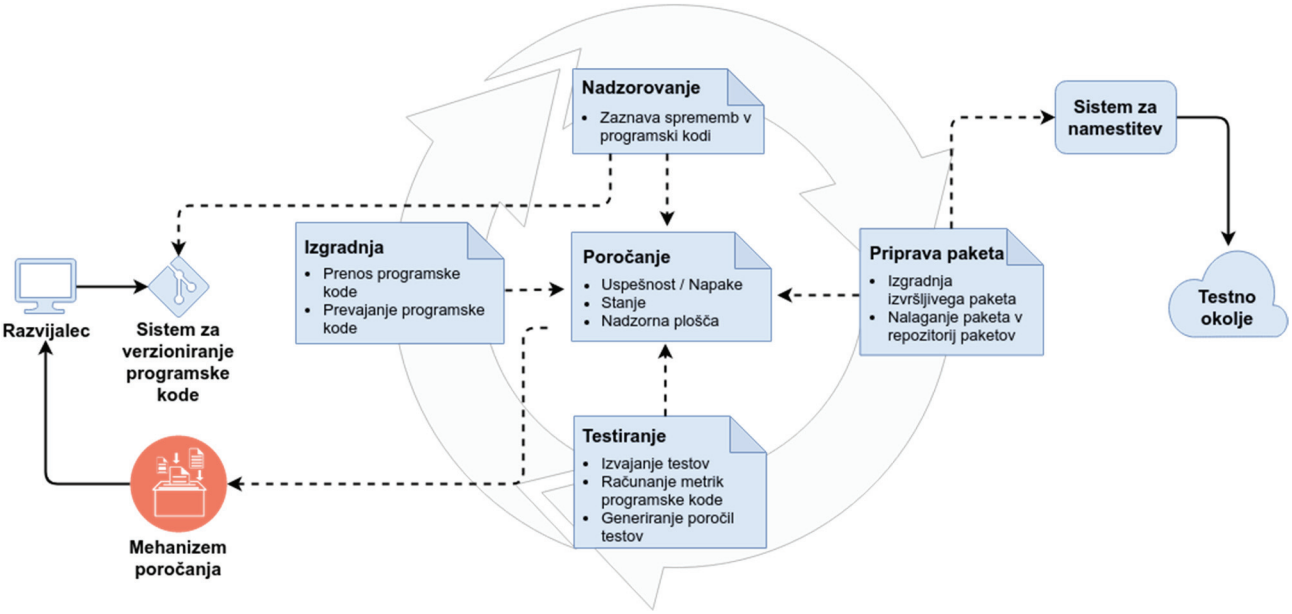
- vzdrževanje enotnega vira – repozitorija programske kode,
- avtomatizacija izgradnje paketa,
- izgradnja naj bo samo testirajoča (angl. self-testing),
- vsi razvijalci dnevno prispevajo kodo v repozitorij,
- vsak prispevek mora biti zgrajen na strežniku, ki poganja integracijskih sistem,
- nedelujoče izgradnje morajo biti takoj popravljene,
- izgradnja naj bo hitra,
- testiranje se naj izvaja na klonu produkcijskega okolja,
- dostopnost do zadnjega zgrajenega izvajalnega paketa naj bo enostavna in omogočena vsem deležnikom,
- vsak ima možnost spremljati, kaj se dogaja,
- avtomatizacija namestitve.

Podobno kot Fowler, tudi Duvall [4] predstavi sedem temeljev neprekinjene integracije programske opreme, ki v splošnem povzamejo Fowlerjeve ključne prakse, dodatno pa eksplicitno izpostavi, da naj ima popravilo neuspešnih integracij najvišjo prioriteto. Kot enega izmed ključnih temeljev izpostavi tudi pregled generiranih poročil, ki vsebujejo analizo kvalitete programske kode.

2.2 Integracijski cikel

Kot predstavljeno je proces samodejne neprekinjene integracije programske opreme sestavljen iz več gradnikov. Ključna komponenta integracijskega procesa, sestavljena iz več gradnikov, je integracijski cikel. Vsak integracijski cikel (Slika 2), se izvaja na CI strežniku, ki v splošnem izvede naslednje korake [6]:

1. Zazna spremembe na skupnem repozitoriju (npr.: Git, Subversion) programske kode.
2. Pridobi programsko kodo iz repozitorija ter jo prenese na CI strežnik.
3. Prevede programsko kodo, v kolikor je to potrebno.
4. Izvede preverjanje kakovosti – izvede sorodne naloge kot so testiranje enot (angl. unit testing) in verifikacija avtomatizirane arhitekture, analiza kvalitete programske kode, itd. Te naloge so v procesu samodejne neprekinjene integracije opcijske, vendar priporočljive, saj je čimprejšnje vpeljevanje nadzora kvalitete v proces integracije ena izmed ključnih lastnosti pristopa samodejne neprekinjene integracije.
5. Prevedeno programsko kodo vrne nazaj v repozitorij.
6. Poroča o stanju izgradnje vsem deležnikom.
7. Opcijsko zapakira uspešno integrirano prevedeno programsko kodo in jo namesti v testno okolje.



Slika 2: Podrobna predstavitev cikla neprekinjene integracije.

2.3 Testiranje

Faza testiranja, poznana tudi kot neprekinjeno testiranje (angl. Continuous testing), v ciklu CI uporablja avtomatizirane pristope ter tako imenovan pomik v levo (angl. shift-left) pristop z namenom pohitritve testiranja, ki posledično v proces CI in faze razvoja programske opreme vključuje tudi problematiko zagotavljanja kakovosti. Uporaba takšnega pristopa lahko vključuje nabor samodejnih testnih delovnih tokov, kateri lahko zajemajo tudi analitiko in metrike programskega inženirstva (LOC[1], CBO[2], CC[3]...), z namenom zagotavljanja čiste, pregledne, na dejstvih temelječe slike kvalitete dostavljene programske opreme. [6]

Z uporabo pristopa neprekinjenega testiranja deležniki dobijo povratne informacije o kakovosti programske opreme, ki jo gradijo. Prav tako jim omogoča, da testirajo prej ter z večjo pokritostjo z odstranitvijo ozkega grla kot je dostop do deljenega testnega okolja in čakanjem na razvoj oz. stabilizacijo uporabniškega vmesnika.

Torej, če želimo razviti programsko opremo, ki je zanesljiva, moramo zagotoviti zanesljivost na nivoju objektov, katero pa lahko dosežemo le s skozi uspešno testiranje enot. Seveda pa nam zgolj napisani testi enot nujno ne zagotavljajo zanesljivosti. Testi morajo efektivno preverjati uporabo objektov in morajo biti pogosto zagnani. Ker objekti komunicirajo med seboj, morajo biti testi pognani, kadarkoli v programski opremi pride do kakršnekoli spremembe. Za celovito in učinkovito testiranje programske opreme je nujno potrebno avtomatizirati skupek raznovrstnih testov [4]:

- **Testi enot**: preverjajo obnašanje manjših elementov programske opreme, ki so navadno predstavljeni kot samostojen razred. Nekateri testi enot zahtevajo minimalne zunanje odvisnosti, ki so implementirane samo kot drugi samostojni razredi. Takšni razredi so sami po sebi enostavni in nimajo globokega grafa objektov. Občasno testi enot uporabljajo t.i. prototipe (angl. mock), ki so prav tako enostavni razredi, katerih vloga je nadomestitev bolj kompleksnih enot za namene testiranja.
- **Testi komponent:** testi komponent ali testi podsistemov preverjajo specifične dele oz. komponente programske opreme in navadno zahtevajo kakšne zunanje odvisnosti kot so podatkovne baze, datotečni sistemi itd. Takšni testi preverjajo, ali komponente delujejo na način, da ustvarjajo pričakovano agregirano obnašanje. Ker tovrstni testi zajemajo večjo količino programske kode v vsakem testnem primeru, se temu primerno tudi dalj časa izvajajo kot na primer testi enot. Pogosto posamezni testi komponent testirajo obnašanje določenih komponent preko izpostavljenega aplikacijskega programskega vmesnika. Takšni testi so poznani tudi kot integracijski testi.

[1] Število vrstic kode (angl. Lines Of Code – LOC)
[2] Število razredov, ki so vezani na specifičen razred (angl. Coupling Between Objects – CBO)
[3] Ciklična kompleksnost (angl.Cyclomatic Complexity – CC)

- **Sistemski testi:** preverjajo delovanje celotne programske opreme in tako zahtevajo popolno nameščen sistem, kot na primer vsebnik servletov in povezano podatkovno bazo. Takšni testi preverjajo, ali zunanji vmesniki kot so spletne strani, spletne storitve ali grafični uporabniški vmesnik delujejo v povezavi s preostalim razvitim sistemom po pričakovanjih.
- **Funkcionalni testi:** kot nakazuje ime, testi funkcionalnosti preverjajo delovanje posameznih funkcionalnosti programske opreme s stališča uporabnika, kar pomeni da testi sami posnemajo interakcijo uporabnikov s programsko opremo.

Glavne koristi, ki jih deležniki z vpeljavo neprekinjenega testiranja pridobijo, so tako [6]:

- Pomik aktivnosti testiranja "v levo" v življenjskem ciklu razvoja programske opreme in integracijo le-teh v razvojne aktivnosti.
- Združevanje testnih, razvojnih in administrativnih ekip v vsakem koraku življenjskega cikla razvoja programske opreme.
- Avtomatizacija testiranja v največji meri z namenom neprekinjenega testiranja ključnih lastnosti oz. zmogljivosti dostavljene programske opreme.
- Omogoča, da poslovnim partnerjem dostavljamo zgodnjo in neprekinjeno povratno informacijo o lastnostih in zmogljivostih razvijajoče se programske opreme.
- Odstranjuje ozka grla dostopnosti do testnega okolja.
- Aktivno in neprekinjeno nadzoruje kvaliteto skozi celoten razvojni cikel programske opreme.

2.4 Izzivi vpeljave neprekinjene integracije

Skozi leta razvoja in vpeljav CI procesa v proces razvoja programske opreme je faza testiranja postala ena izmed najpomembnejših delov uspešnega izvajanja CI. Po drugi strani pa glede na poročanje številnih študij, faza testiranja predstavlja izvor največjih ovir in izzivov, povezanih z vpeljavo CI v proces razvoja programske opreme.

Med pogostejšimi izzivi, povezanimi s testiranjem programske kode, se tako pojavlja problem avtomatizacije različnih tipov testov kot so testiranje enot, integracijsko testiranje, testiranje uporabniških vmesnikov [7], [8]. Vzrok za težave pri avtomatizaciji testov lahko bodisi izvira iz pomanjkljive infrastrukture, odvisnosti od specifične strojne opreme, bodisi iz slabega izvajanja testno vodenega razvoja (angl. test driven development, TDD) [9]. Drug pogost izziv, povezan s testiranjem programske opreme, pa je nizka kvaliteta napisanih testov, kar vključuje nezanesljive teste, nizko pokritost s testi, dolgo izvajajoče se teste, ipd. [7], [10]

Poleg izzivov, povezanih s testiranjem programske opreme, so avtorji članka [9] kot pomemben izziv vpeljave CI identificirali tudi odpravljanje konfliktov v programski kodi. Med pogostejšimi razlogi za težave pri odpravljanju konfliktov sta izpostavljena odvisnost od modulov tretjih oseb ter močno sklopljena zasnova oz. arhitektura programske opreme. [7], [10]

2.5 Koristi neprekinjene integracije

Raziskave, opravljene na področju programskega inženirstva, kot pomembnejše koristi neprekinjene integracije izpostavljajo [5], [9]:

- Pridobivanje hitrejšega povratnega odziva o uspešnosti izgradnje programske opreme kot tudi odziva uporabnikov.
- Hitrejša identifikacija, lažje obvladovanje ter hitrejša odprava hroščev v programski opremi.
- Pogoste in zanesljive izdaje programske opreme, ki vodijo k zvišanju stopnje zadovoljstva končnih uporabnikov programske opreme.
- Skozi vpeljavo CI v proces razvoja programske opreme podjetja pogosto vpeljejo tudi proces neprekinjene dostave, kar ima za posledico izboljšanje povezovanja in komunikacije med razvijalci ter skrbniki infrastrukture ter posledično povečano stopnjo avtomatizacije procesov.

3 VKLJUČITEV NAPOVEDNIH MODELOV V PROCES SAMODEJNE NEPREKINJENE INTEGRACIJE IN TESTIRANJA

Razvoj napovednih modelov strojnega učenja v sami osnovi ni precej drugačen od razvoja tradicionalne programske opreme. Tako kot pri razvoju običajne programske opreme je tudi pri razvoju napovednih modelov to proces s celotnim življenjskim ciklom, ki vključuje načrtovanje, implementacijo, optimizacijo, testiranje in dostavo oziroma namestitev. S tem ko napovedni modeli postajajo iz leta v leto čedalje bolj vključeni v raznovrstne informacijske sisteme, se tudi njihova vloga čedalje bolj povečuje. Z vedno bolj pomembno vlogo napovednih modelov pa se zvišuje tudi potreba, da je življenjski cikel razvoja ter integracije napovednega modela upravljan na sistematičen in bolj rigiden način kot je to do sedaj že uveljavljena praksa pri razvoju konvencionalne programske opreme. [11] O tem priča tudi porast znanstvenih in strokovnih člankov [2], [11], [12], ki naslavljajo temo razvojnega življenjskega cikla napovednih modelov strojnega učenja in integracijo ter namestitev takšnih modelov v že obstoječe programske rešitve.

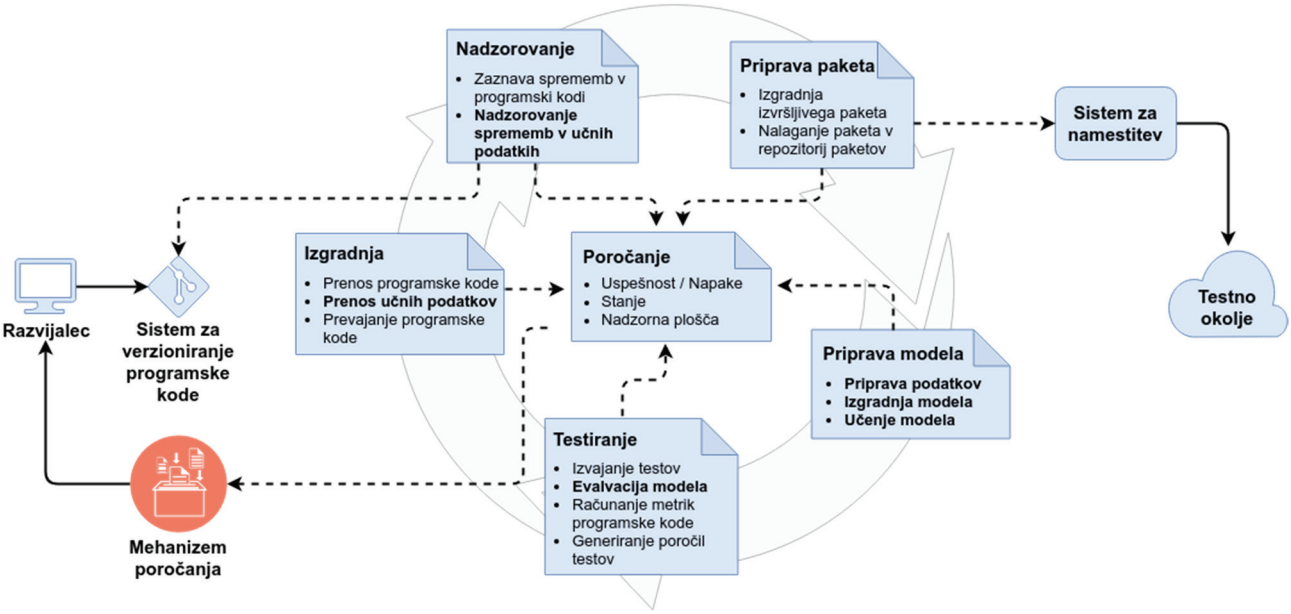
Kot rečeno, se sam pristop razvoja napovednega modela v samem bistvu kaj dosti ne razlikuje od razvoja tradicionalne programske opreme, ampak ga v splošnem samo razširja. Tipičen proces razvoja napovednega modela bi lahko opredelili s sledečimi koraki:

1. Identifikacija problema
2. Načrtovanje rešitve
3. Pridobivanje in pred-obdelava podatkov
4. Izgradnja napovednega modela
   4.1. Izbira algoritma
   4.2. Učenje modela
5. Evalvacija modela
6. Namestitev

3.1 Samodejna neprekinjena integracija napovednih modelov

Izmed prej naštetih korakov procesa razvoja napovednih modelov sta za vpeljavo napovednih modelov strojnega v proces CI ključna izgradnja napovednega modela in evalvacija modela. Izgradnja uspešnega napovednega modela v splošnem zahteva veliko učno množico (podatki), bolj ali manj kompleksen algoritem, s katerim bomo zgradili napovedni model ter strategijo učenja in ovrednotenja napovednega modela. Ob tem je pomembno dodati, da je izgradnja napovednega modela tipično dolgo trajajoč postopek, katerega rezultat ni deterministične. Ne-determinističnost napovednih modelov pa prinaša tudi številne izzive v področje nadzorovanja in zagotavljanja kakovosti delovanja napovednega modela, zato je ključnega pomena, da v obstoječ CI proces pri vpeljavi napovednih modelov strojnega učenja poleg mehanizmov priprave modela vključimo tudi potrebne mehanizme za evalvacijo le-teh.

Konceptualna zasnova (Slika 3) prikazuje razširjen proces CI z vključitvijo napovednega modela strojnega učenja. Kot je razvidno iz slike je poleg razširitev obstoječih komponent procesa dodana tudi nova komponenta – priprava modela. Priprava modela vsebuje korake priprave podatkov, izgradnjo samega modela ter učenje modela. V komponenti nadzorovanje je dodan korak nadzorovanja sprememb nad učnimi podatki, ti so navadno hranjeni bodisi v obstoječem sistemu za verzioniranje programske kode, bodisi v za to namenjenem sistemu za hrambo podatkov kot na primer v podatkovnih bazah, datotečnih sistemih ipd. Razširjena je tudi komponenta izgradnja, ki poleg prenosa ter opcijsko prevajanja programske kode vsebuje tudi korak, v katerem se opravi prenos učnih podatkov iz specifičnega vira v sam proces CI. Prav tako ključna je razširitev komponente testiranje, ki v razširjenem procesu CI vsebuje tudi korak evalvacije modela.



Slika 3: Konceptualna zasnova razširjenega procesa neprekinjene integracije z vključitvijo napovednih modelov strojnega učenja.

3.2 Izzivi

Z vključitvijo dodatne komponente ter razširitvijo omenjenih komponent procesa samodejne integracije in testiranja napovednih modelov, se moramo soočiti tudi z izzivi, ki jih le-te prinesejo. Kot ključne izzive so avtorji v prispevkih [11]–[13] izpostavili predvsem problematiko kompleksnosti testiranja in zagotavljanja kakovosti napovednih modelov ter izzive, povezane z učenjem napovednih modelov.

Med izzive, povezane z učenjem napovednih modelov, prav gotovo spadajo izzivi, ki se navezujejo na učne in testne podatke. Kvaliteta in kvantiteta podatkov, tako učnih kot tudi testnih, je kritična za razvoj uspešnega napovednega modela. V primeru razvoja napovednih modelov, temelječih na algoritmih globokega učenja, kot so na primer konvolucijske nevronske mreže ali nevronske mreže s povratno zanko, pa je predvsem količina podatkov še toliko bolj pomembna, saj takšni algoritmi za uspešno učenje potrebujejo večjo količino prečiščenih podatkov kot konvencionalni algoritmi. Na tej točki velja izpostaviti pomembnost čiščenja podatkov, ki je pri razvoju in integraciji napovednih modelov pogosto spregledana. V splošnem obstajata dve skupini tehnik oziroma pristopov, s katerimi naslavljamo negativni efekt neprečiščenih podatkov oziroma podatkov s šumom: odpornost na šum v podatkih ter odstranitev šuma. Pristopi, odporni na šum, vključujejo izgradnjo robustnih algoritmov, ki so tolerantni do podatkov, ki vključujejo šum in tako ne potrebujejo predprocesiranja in prečiščevanja podatkov. Tehnike za odstranjevanje šuma pa delujejo po principu odstranjevanja učnih primerkov, v kolikor le-ti vsebujejo šum. [2]

Prav tako pogosti izzivi, povezani z učenjem napovednih modelov, so hranjenje podatkov oziroma izzivi z uporabo sistema za verzioniranje. Velikost napovednih modelov lahko presega nekaj gigabajtov, povrhu vsega pa so ti navadno hranjeni v binarni obliki, kar zna predstavljati problem ob uporabi sistema za verzioniranje kot je Git, saj ta ni bil zasnovan za takšne namene. Sicer lahko tovrsten problem rešimo z uporabo razširitve Git LFS (Git Large File Storage), vendar pa ni nujno da je uporaba le-te podprta s strani izbranega sistema za neprekinjeno integracijo. Z vpeljavo napovednih modelov v proces CI pa je potrebno poleg napovednih modelov slediti in hraniti tudi spremembe v učnih in testnih podatkih. Sicer je mogoče to opravljati v sklopu repozitorija programske kode, lahko pa tak pristop hitro privede do težje sledljivosti in zmede pri nadzorovanju in upravljanju sprememb. Obetajoč kandidat, ki naslavlja omenjeno problematiko, je orodje Data Version Control (DVC), splavljeno leta 2018, ki je razširitev Git orodja s poudarkom na upravljanju z velikimi količinami podatkov in možnostjo reprodukcije napovednih modelov. [2]

Ena izmed pomembnejših lastnosti procesa CI je hitra in pogosta integracija, ki vzpodbuja, da razvijalci večkrat dnevno prispevajo programsko kodo v repozitorij. Kot poročajo avtorji, se že pri razvoju konvencionalne programske opreme pogosto pojavljajo izzivi, povezani z dolgo trajajočimi integracijskimi cikli. Ti pa se z

vpeljavo napovednih modelov v proces CI še dodatno drastično povečajo, saj je učenje napovednih modelov računsko zahteven in dalj časa trajajoč proces. Tipično se pri razvoju konvencionalne programske opreme težava dolgo trajajočega CI cikla rešuje z razbijanjem izgradnje programske opreme na več manjših komponent. Enakega pristopa pa žal ne moremo uporabiti pri razvoju napovednih modelov, saj je proces izgradnje in učenja močno sklopljen. Omenjene težave lahko naslovimo z delegiranjem učenja napovednih modelov na specifično strojno opremo, optimizirano za učenje napovednih modelov, ali pa z uporabo ogrodij in sistemov za porazdeljeno učenje. [2]

Kot poznamo že s področja programskega inženirstva je testiranje pomembna strategija za povečanje zanesljivosti, zmanjševanje tehničnega dolga in dolgoročno zniževanje stroškov vzdrževanja. Ključna razlika testiranja napovednih modelov strojnega učenja v primerjavi s testiranjem konvencionalne programske opreme je v kompleksnosti. Kompleksnost testiranja napovednih modelov izvira iz njihove narave, saj je obnašanje le-teh močno odvisno od podatkov in dejstva, da jih ni mogoče strogo specificirati. Na omenjen izziv lahko gledamo z vidika prispodobe učenja napovednega modela s prevajanjem programske kode, kjer je v primeru razvoja napovednega modela programska koda prispodoba tako za dejansko programsko kodo kot tudi učne podatke. Tako je potrebno, z vidika te prispodobe, testirati tako programsko kodo kot tudi podatke, naučen napovedni model pa moramo obravnavati na podoben način kot obravnavamo izvršljive pakete programske opreme. [13] Na tej točki si je torej smiselno zastaviti vprašanje: Kaj in koliko sploh testirati?

3.3 Testiranje in evalvacija napovednih modelov

Testiranje programske opreme je v programskem inženirstvu v splošnem dobro raziskano, podobno je tudi s področjem strojnega učenja. Težava pa se pojavlja na preseku omenjenih področij, torej testiranju napovednih modelov strojnega učenja. Še najboljši približek smernicam oziroma dobrim praksam, kot jih poznamo na področju programskega inženirstva, je predstavljen v prispevku [13], pripravljenem s strani Googlovih zaposlenih. V nadaljevanju bomo predstavili nekaj pomembnejših smernic za celovito testiranje napovednih modelov ter zmanjševanje potencialnega tehničnega dolga.

Ključna razlika med konvencionalno programsko opremo ter napovednimi modeli strojnega učenja je, da obnašanje napovednih modelov ni določeno neposredno s programsko kodo, ampak na njegovo obnašanje vplivajo podatki. Zato je posledično smiselno v procesu neprekinjene integracije z vpeljanimi napovednimi modeli testirati tudi učne podatke. Pri testiranju podatkov je smiselno preverjati ali so le-ti v pravilni oz. v pričakovani obliki ter ali so vrednosti posameznih značilk znotraj predvidenega definiranega območja. Dodatno je smiselno s testi enot preverjati tudi programsko kodo za izluščevanje in/ali izgradnjo značilk, čeprav se navadno zdi, da je precej enostavna in ni potrebe po testiranju. Napake v vrednosti značilk, ko so že v procesu izluščevanja in nadalje v procesu učenja napovednega modela, je namreč skorajda nemogoče zaznati. [13]

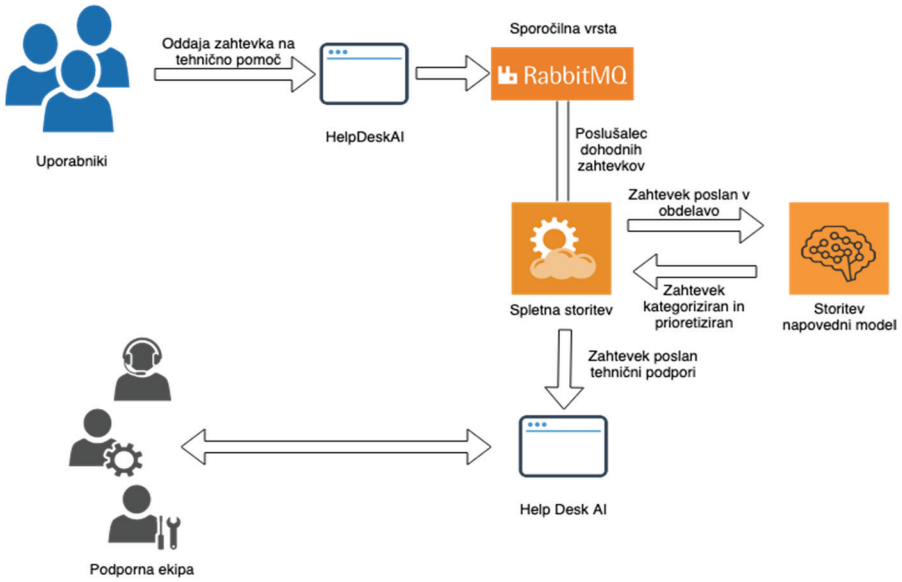
Za razliko od področja strojnega inženirstva, kjer se je že razvil širok nabor smernic in dobrih praks, so smernice in dobre prakse razvoja zanesljivih napovednih modelov strojnega učenja še v razvoju. Algoritmi strojnega učenja navadno dopuščajo oz. celo zahtevajo nastavitve velikega števila t.i. hiper-parametrov, s katerimi definiramo delovanje algoritmov, učnih strategij, ipd. Izbira primernih hiper-parametrov ima velik vpliv na sposobnost učenja modela kot tudi na končno uspešnost napovednih modelov. Hiper-parametre je potrebno skozi razvoj prav tako prilagajati glede na spremembe v zasnovi učnega algoritma kot tudi glede na spremembe v podatkih. V ta namen je priporočljiva uporaba bolj ali manj enostavnih strategij iskanja hiper-parametrov, ki lahko v splošnem poleg zvišanja kvalitete napovedi razkrije tudi težave, povezane z zanesljivostjo napovednih modelov. Za namene zagotavljanja kakovosti napovednih modelov je ključno vpeljati teste, kjer preverjamo ali je uspešnost napovednega modela na vnaprej določeni učni podmnožici v skladu z našimi pričakovanji. Na primer, »uspešnost oz. točnost napovednega modela za podmnožico x mora biti nad 95 %« je lahko eden izmed takšnih testov. Priporočljiva je tudi vpeljava več testov takšnega tipa, za različne podmnožice, saj je na tak način lažje zaznati večje padce v uspešnosti napovednega modela iz iteracije v iteracijo. Na tej točki je smiselno izpostaviti tudi, da je validacijo kakovosti napovednega modela nujno potrebno opraviti pred namestitvijo modela v izvajalno okolje. [13] Pri procesu zagotavljanja kakovosti s preverjanjem uspešnosti modela pa ima velik pomen tudi izbira metrike, s katero merimo uspešnost napovednega modela. Med pogosteje uporabljanimi metrikami je zagotovo metrika točnost (angl. accuracy), ki predstavlja delež pravilno napovedanih primerkov. Zaradi enostavnosti same metrike pa lahko le-ta prikrije

pomembne informacije o uspešnosti napovednega modela. Zato je priporočljiva uporaba kombinacij metrik oz. uporaba metrik, ki bolj celovito kot metrika točnost predstavljajo uspešnost napovednega modela (npr.: F1 metrika, AUC metrika). [2] Prav tako kot pri tipični programski opremi je tudi pri napovednih modelih nujno zagotoviti možnost enostavne povrnitve modela na prejšnjo verzijo v primeru, da je zaznana nepravilnost v delovanju oz. v primeru nepričakovanega padca uspešnosti napovednega modela strojnega učenja. [13]

4 PRAKTIČNI PRIMER

Za praktični prikaz primera vpeljave razvoja napovednih modelov z uporabo razširjenega procesa CI smo si zastavili razvoj spletne aplikacije – sistem za tehnično podporo s samodejno kategorizacijo in prioretizacijo uporabniških zahtevkov. Razvit sistem naslavlja scenarij (Slika 4), kjer uporabniki preko pročelne spletne aplikacije vnesejo zahtevek tehnični podpori, ta pa je preko sporočilne vrste dostavljen zalednemu delu spletne aplikacije. Zaledni del spletne aplikacije preko REST komunicira z napovednim modelov strojnega učenja, izpostavljenim v obliki spletne storitve, ki prejet zahtevek kategorizira in mu določi prioriteto. Obdelan uporabniški zahtevek je nato posredovan na pročelni del spletne aplikacije, namenjen tehnični podpori.

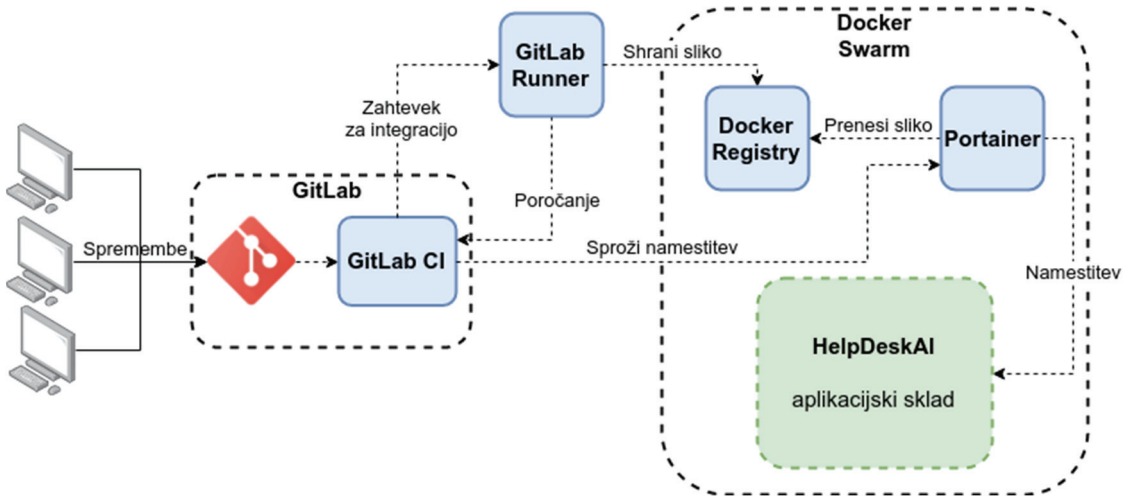
Tehnološki sklad, izbran za implementacijo praktičnega primera, sestoji iz sporočilne vrste RabbitMQ, zaledni del spletne storitve je implementiran z uporabo Java ogrodja Spring Boot, pročelna aplikacija temelji na knjižnici React, napovedna modela (en zadolžen za kategorizacijo, drugi pa za prioretizacijo) sta razvita s pomočjo Pythonove knjižnice sklearn ter servirana v obliki REST s pomočjo ogrodja Flask. Vsak izmed omenjenih delov spletne aplikacije je za namen namestitve na gruči Docker Swarm zapakiran v Docker sliko, za orkestracijo zabojnikov skrbi Portainer, za verzioniranje kode ter kot sistem za neprekinjeno integracijo in dostavo pa služi GitLab.



Slika 4: Scenarij praktičnega primera – sistem za tehnično podporo s samodejno kategorizacijo in prioretizacijo uporabniških zahtevkov.

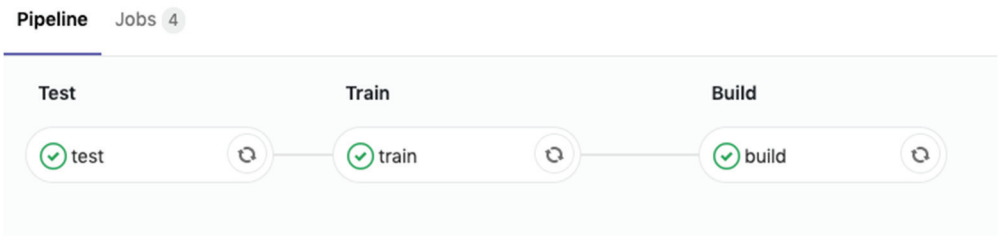
Pri tem praktičnem primeru se bomo osredotočili na razvoj in integracijo storitve, upoštevajoč smernice, ki smo jih nakazali oziroma predstavili v prejšnjem poglavju. Cilj je vpeljati razvoj napovednega modela strojnega učenja v predstavljen razširjen proces CI. Slika 5 prikazuje prilagojen koncept vpeljave razvoja napovednega modela strojnega učenja v proces samodejne neprekinjene integracije. V našem primeru je programska koda kot tudi učni podatki shranjena v sistemu za verzioniranje programske kode Git. Ob vsaki spremembi se torej sproži proces neprekinjene integracije, ki ga v našem primeru izvaja t.i. GitLab Runner. GitLab Runner je uraden GitLabov agent, namenjen za izvajanje tovrstnih nalog pri uporabi sistema GitLab, ki ima vgrajeno samodejno poročanje, kar nam omogoča sprotno spremljanje poteka procesa samodejne neprekinjene integracije. V zadnji fazi omenjen agent zgradi Docker sliko, ki napovedna modela zapakira v obliko, neodvisno od platforme in pripravljeno za takojšnje izvajanje. Sliko nato shrani v privaten register slik, integracijski sistem pa na orkestracijskem sistemu sproži zahtevek za namestitev. Kot omenjeno smo za

orkestracijski sistem izbrali Portainer, ta je zadolžen za izvajanje, orkestracijo ter posodabljanje storitev, definiranih znotraj našega aplikacijskega sklada (HelpDeskAI). V primeru, da kakšen izmed korakov integracije ni uspešen, se celoten proces zaustavi, s čimer ne pridemo do situacije, ko bi namestili nedelujočo storitev oz. storitev, ki ni v skladu z našimi zahtevami in pričakovanji.



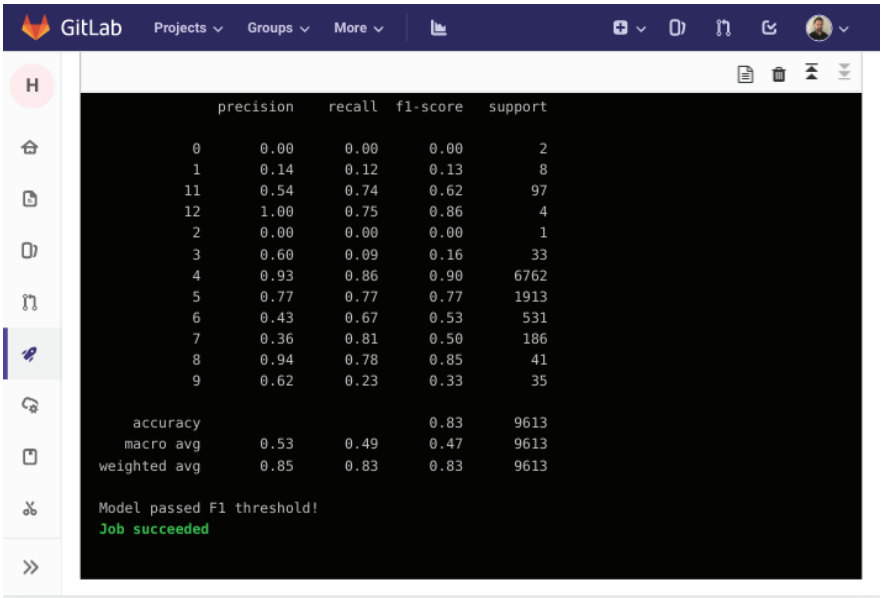
Slika 5: Proces samodejne neprekinjene integracije za razvoj napovednega modela strojnega učenja.

Sam proces integracije napovednega modela smo opravili v treh korakih (Slika 6): testiranje, učenje in gradnja. Ker so uporabljeni učni podatki bili že predhodno prečiščeni in preoblikovani v obliko, primerno za učenje napovednih modelov, smo korak pred-procesiranja izpustili, sicer bi v tipičnem primeru tak korak bil nujno potreben. V našem primeru se je tako integracija pričela s korakom testiranja, kjer smo poleg poznanih konceptov iz programskega inženirstva testirali tudi podatke. Preverjali smo predvsem nabor vrednosti posameznih značilk znotraj učne množice, mogoče pa bi bilo opraviti preverjanje distribucije testnih primerkov, preverjanje informacijskih vrednosti posameznih značilk, ipd.



Slika 6: Koraki integracije napovednega modela.

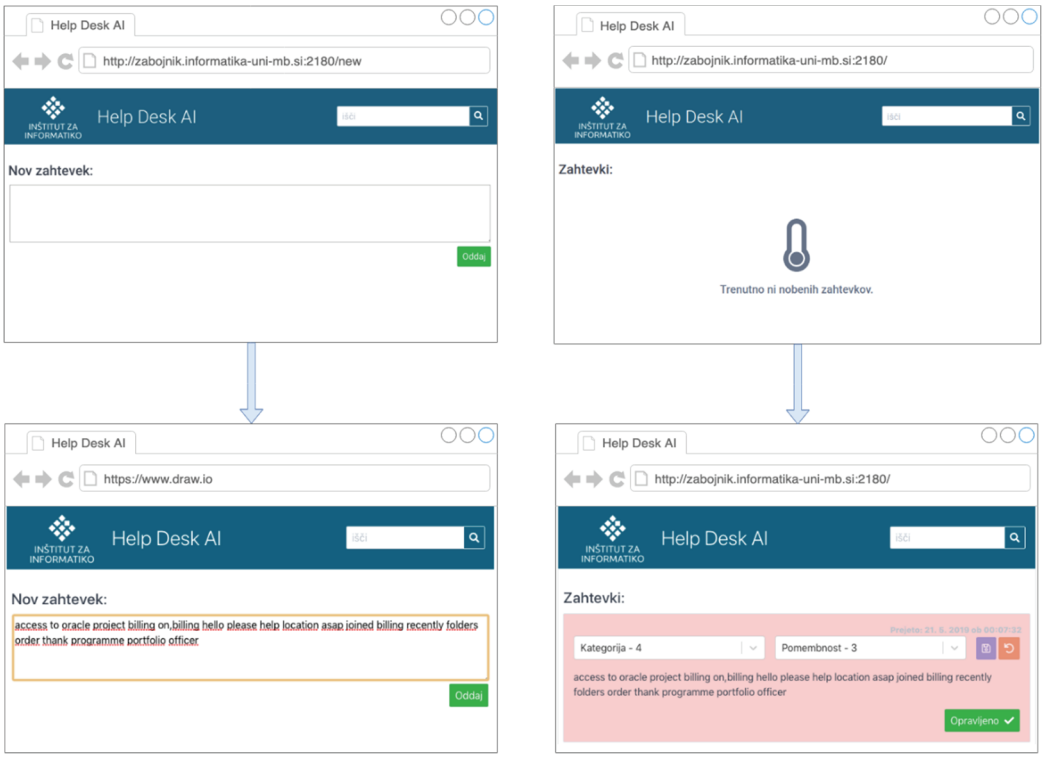
V drugem koraku smo izvedli učenje in evalvacijo obeh naučenih modelov. Kot rečeno smo učili dva specializirana modela. Naloga prvega je glede na podano vsebino zahtevka določiti kategorijo, v katero zahtevek spada, naloga drugega pa je določiti prioriteto posameznega zahtevka. Oba modela smo učili na enak način, z uporabo klasifikatorja naivni Bayes ter razdelitvijo učnih in testnih podatkov v razmerju 80 % proti 20 %. Uporabili pa smo tudi pristop grid search za iskanje optimalne nastavitve hiper-parametrov klasifikatorja. Po učenju posameznega modela smo izračunali ter izpisali tudi matriko zmede z osnovnimi vrednostmi klasifikacijskih metrik (Slika 7) kot so točnost, natančnost (angl. Precision), priklic (angl. Recall) ter F1 metriko. Uspešnost posameznega napovednega modela smo evalvirali na podlagi povprečne obtežene vrednosti F1 metrike ter določili mejo 80 %, nad katero je napoveden model dovolj uspešen za uporabo oz. namestitev. V primeru, da model ne doseže vnaprej določene meje, se faza zaključi kot neuspešna ter se tako prekine celoten proces samodejne neprekinjene integracije. Na tej točki bi veljalo izpostaviti, da bi za namene evalvacije napovednih modelov, podobno kot je za ocenjevanje kvalitete programske kode, prav prišla razna ogrodja, knjižnice in sistemi, ki bi omogočali lažjo integracijo in pregled nad rezultati evalvacije. V našem primeru je bilo namreč potrebno evalvacijo in mehanizme preverjanja in prestajanja uspešnosti implementirati ročno.



Slika 7: Matrika zmede in vrednosti izbranih metrik napovednega modela za klasifikacijo zahtevkov v kategorije.

Po uspešno prestani fazi učenja in evalvacije napovednih modelov se v fazi grajenja naučena modela s pomočjo ogrodja Flask izpostavita v obliki REST, celotna storitev pa se nato zapakira v Docker sliko, ki jo naložimo na privaten register Docker slik. Integracija je z uspešno izvedenim zadnjim korakom tako zaključena, integracijski sistem pa za potrebe namestitve izvrši zahtevek na orkestracijski sistem Portainer. Ta poskrbi za prenos najnovejše Docker slike iz registra slik ter zagon zabojnika na podlagi prenesene slike. V kolikor je zagon zabojnika uspešen, je predhodna različica storitve zabojnika zamenjana za novejšo.

Iz slike 8 so razvidni zaslonski posnetki pročelnega dela razvitega sistema za tehnično podporo. Zgornji dve sliki prikazujeta prazna pogleda za uporabnika (levo) in za člana tehnične podpore (desno). Spodaj levo je prikazan vnos zahtevka s strani uporabnika, na desni strani pa je prikazan ta isti zahtevek po opravljeni kategorizaciji in prioretizaciji s strani naših razvitih napovednih modelov.



Slika 8: Zaslonske slike spletne aplikacije za tehnično podporo.

Obstoječ sistem bi bilo smiselno razširiti s hrambo sprememb kategorij oz. stopnje pomembnosti s strani članov tehnične podpore. Hrambo teh sprememb bi lahko izrabili za pridobivanje novih oz. izboljšanih učnih podatkov, s čimer bi dodatno pripomogli k večji uspešnosti napovednega modela. V takšnem primeru bi dodatno veljalo uvesti tudi periodičen zagon cikla neprekinjene integracije napovednega modela, saj bi bilo ob pogostih spremembah učnih podatkov s strani članov tehnične podpore smiselno tudi pogosteje učiti in evalvirati napovedna modela.

5 ZAKLJUČEK

V prispevku smo na začetku predstavili osnoven koncept samodejne neprekinjene integracije pri razvoju konvencionalne programske opreme ter izpostavili ključne prednosti uporabe takšnega pristopa. V nadaljevanju smo obstoječ, uveljavljen koncept, razširili ter prilagodili za potrebe razvoja in integracije napovednih modelov strojnega učenja. Naslovili smo nekaj ključnih izzivov, ki jih prinaša vpeljava razvoja napovednih modelov, ter izpostavili morebitne smernice oziroma dobre prakse.

Predlagane smernice in dobre prakse smo predstavili tudi v obliki praktičnega primera, kjer smo z vpeljavo napovednih modelov strojnega učenja razvili sistem za tehnično podporo, ki samodejno, z uporabo napovednih modelov, kategorizira in prioretizira posamezne uporabniške zahtevke.

Izpostaviti velja, da je poleg manka uveljavljenih dobrih praks in smernic, kot je to pri programskem inženirstvu, mogoče zaznati tudi manko namenskih orodij ali knjižnic, še posebej za potrebe evalvacije napovednih modelov ter generiranja in pregleda poročil. Menimo, da bo v naslednjih letih področje integracije in evalvacije napovednih modelov ter zagotavljanje kakovosti le-teh doživelo silovit razcvet, saj potreba po rešitvah, ki naslavljajo omenjeno problematiko v gospodarstvu iz leta v leto narašča.

6 LITERATURA

[1] S. Hamdan in S. Alramouni, „A Quality Framework for Software Continuous Integration“, *Procedia Manuf.*, let. 3, str. 2019–2025, 2015.
[2] A. Herczeg, „Continuous Integration in Machine Learning Towards fully automated continuous learning“, 2018.
[3] K. Beck, „Embrance Change with Extreme Programming“, *IEEE Comput. Mag.*, št. c, str. 70–77, 1999.
[4] P. M. Duvall, S. Matyas, in A. Glover, *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.
[5] M. Fowler in M. Foemmel, „Continuous integration“, *Thought-Works) http//www. thoughtworks. com/Continuous Integr. pdf*, let. 122, str. 14, 2006.
[6] M. Erder in P. Pureur, „Continuous Architecture and Continuous Delivery“, v *Continuous Architecture*, Elsevier, 2016, str. 103–129.
[7] M. Leppänen *idr.*, „The highways and country roads to continuous deployment“, *IEEE Softw.*, let. 32, št. 2, str. 64–72, mar. 2015.
[8] A. Debbiche, M. Dienér, in R. Berntsson Svensson, „Challenges When Adopting Continuous Integration: A Case Study“, Springer, Cham, 2014, str. 17–32.
[9] M. Shahin, M. Ali Babar, in L. Zhu, „Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices“, *IEEE Access*, let. 5, št. Ci, str. 3909–3943, 2017.
[10] G. G. Claps, R. Berntsson Svensson, in A. Aurum, „On the journey to continuous deployment: Technical and social challenges along the way“, v *Information and Software Technology*, 2015, let. 57, št. 1, str. 21–31.
[11] C. Renggli *idr.*, „Continuous Integration of Machine Learning Models with ease.ml/ci: Towards a Rigorous Yet Practical Treatment“, 2019.
[12] B. Derakhshan, „Continuous Deployment of Machine Learning Pipelines“, str. 397–408, 2019.
[13] E. Breck, S. Cai, E. Nielsen, M. Salib, in D. Sculley, „The ML test score: A rubric for ML production readiness and technical debt reduction“, v *Proceedings - 2017 IEEE International Conference on Big Data, Big Data 2017*, 2018, let. 2018-Janua, str. 1123–1132.