Zahvaljujem svome mentoru izv. prof. dr. sc. Igoru Mekterovi\'cu na pomoći pri izradi završnog rada, svom stečenom znanju u toku suradnje te izgrađenom odnosu.

Zahvaljujem svojim roditeljima te sestri na pruženoj podršci prilikom studiranja na preddiplomskom studiju.

Zahvaljujem svojem fakultetskom okruženju na atmosferi koja motivira na rad.

Zahvaljujem sebi na svom uloženom trudu i vremenu te na tome što uvijek radim, dajem, stvaram te savladavam više nego što mislim da mogu.

SADRŽAJ

POPIS SLIKA

Popis oznaka i kratica

API	Application Programming Interface
APAS	Automated Programming Assessment System
CEE	Code Execution Engine
CPU	Central Processing Unit
CSV	Comma Separated Values
OCES	Online Code Execution System
FER	Fakultet Elektrotehnike i Računarstva
ID	Identification
PDF	Portable Document Format
REST API	Representational State Transfer API
RPC	Remote Procedure Call
URL	Unique Resource Locator
SQL	Structured Query Language

1. Uvod

Sustav *Edgar* je informacijski sustav koji se koristi na FER-u za potrebe automatizirane provjere znanja. On omogućuje testiranje studenata na nekoliko tipova pitanja od kojih su programska pitanja, a posebice SQL pitanja, od iznimne važnosti te su potaknule razvoj samog sustava. Sustav je u upotrebi nekoliko godina te se pokazao kao prikladno rješenje. Sustav je kroz godine prošao kroz mnoge nadogradnje što je omogućeno izvedbom prikladne modularne arhitekture te je upravo zbog svoje primjene izgrađen na način koji omogućuje jednostavno dodavanje novih funkcionalnosti. Korištenje takvog sustava omogućuje prikladan način testiranja studenata u programskog znanju, a ujedno smanjuje teret s ionako malog broja nastavnog osoblja. Sustav *Edgar* je tijekom svoje uporabe omogućio pristup mnogim podacima koji opisuju performanse studenata na određenim kolegijima. Ti se podaci sastoje od performansi studenata na pojedinim nastavnim aktivnostima te informacije jesu li položili određeni kolegij. Sadržaj je takvih podataka potaknuo ideju o izradi prediktivnog modela koji predviđa performanse studenata, a posebice njihovu prolaznost. Rezultati dobiveni izradom takvih modela su obećavajući te s dovoljno malom greškom mogu predvidjeti prolaznost pojedinog studenta već nakon polovice semestra. Prikupljanje podataka za takve modele te provedba predikcije modela zahtjeva određeno vrijeme i trud nastavnog osoblja. S obzirom na to da sam sustav zbog svoje namjene ima sposobnost izvoditi proizvoljan programski kod, pojavljuje se ideja o integraciji modela strojnog učenja u sam sustav kako bi se predikcija modela te dostava rezultata modela studentima mogli izvoditi automatizirano u svrhu pravovremene povratne informacije studentima o njihovim performansama te kako bi ih se potaknulo na rad u svrhu povećanja prolaznosti studenata. Prolaznost studenata je od velike važnosti FER-u, a i ostalim obrazovnim ustanovama svijeta. Takva bi nadogradnja omogućila razvoj proizvoljnog broja prediktivnih modela te njihovu integraciju u sustav *Edgar* s ciljem pravovremenog informiranja o potencijalnim problemima odnosno povećanjem prolaznosti studenata. U sklopu ovog rada se proširuje postojeći sustav *Edgar* u skladu sa servisno orijentiranog arhitekturom, a proširenje se ostvaruje nadogradnjom postoje-

ćih kontejneriziranih aplikacija koje samom sustavu omogućuju pokretanje proizvoljnog koda. Također se pokazuju određene prepreke prilikom nadogradnje sustava te se daju smjernice za moguću nadogradnju rješenja, uspostavu automatiziranog procesa dubinske analize podataka te strukturiranog prikaza rezultata studentima.

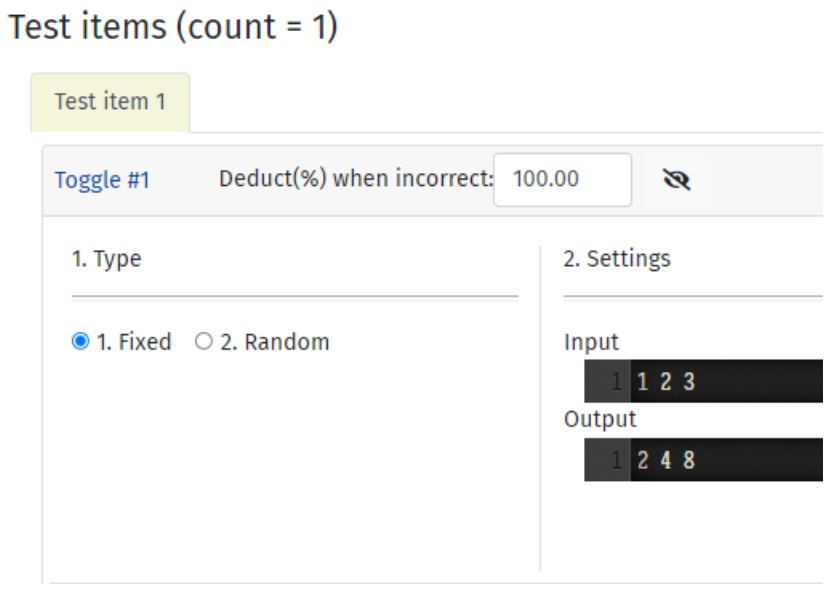
2. Sustav *Edgar* i postojeće funkcionalnosti

Broj studenata na području računarskih znanosti se iz godine u godinu povećava, kako na FER-u tako i na ostalim sveučilištima. Studenti u ovom području znanosti često moraju pokazati znanje i vještine iz rješavanja programskih zadataka. Činjenica da je pojedini programski zadatak moguće riješiti koristeći mnogobrojne algoritme te je svaki od algoritama moguće implementirati na mnogo načina uvelike otežava ručnu provjeru programskih zadataka. Navedena činjenica, osim što ima utjecaj na složenost ispravljanja programskih zadataka, opterećuje nastavno osoblje, upravo zbog zahtjevnosti te vremenske složenosti ocjenjivanja.

Sustav *Edgar*[6], kao automatizirani sustav za programsku provjeru znanja (APAS), nastoji ukloniti potrebu za ručnim ispravljanjem programskih zadataka te u nezanemarivoj količini ukloniti teret s nastavnog osoblja, a ujedno omogućiti jednostavan, kvalitetan te ispravan proces ocjenjivanja računalnog znanja. Sustav također nastoji pružiti pravovremenu povratnu informaciju studentima, što je otežano ručnim ispravljanjem, kako bi se poboljšao sveukupni prijenos znanja na studente. Za takvu se potrebu sustav *Edgar* pokazao kao prikladno rješenje te se danas koristi na više od deset kolegija na FER-u, ali je stvoren i oblikovan na način da ga mogu koristiti i ostale obrazovne ustanove svijeta. Početno je razvijen za potrebe kratkih provjera znanja, posebice na kolegiju Baze podataka gdje se od ispitanika traži unos SQL upita, a njegova se točnost ocjenjuje usporedbom rezultirajućih redaka studentskog upita na bazu podataka s rezultirajućim recima točnog upita. Takav oblik provjere znanja prije sustava *Edgar* nije bio podržan. Danas je sustav u stanju vršiti provjere nad nekoliko programskih jezika te na većem broju tipova pitanja.

Za potrebe izrade programskih zadataka, nastavno osoblje ima mogućnost definirati testne primjere odnosno primjere rješenja koje bi određeni algoritam trebao imati kada mu je dodijeljen određeni ulazni skup (Slika 2.1). Na temelju danih testnih primjera, sustav *Edgar* provjerava točnost studentskih algoritama u stvarnom vremenu

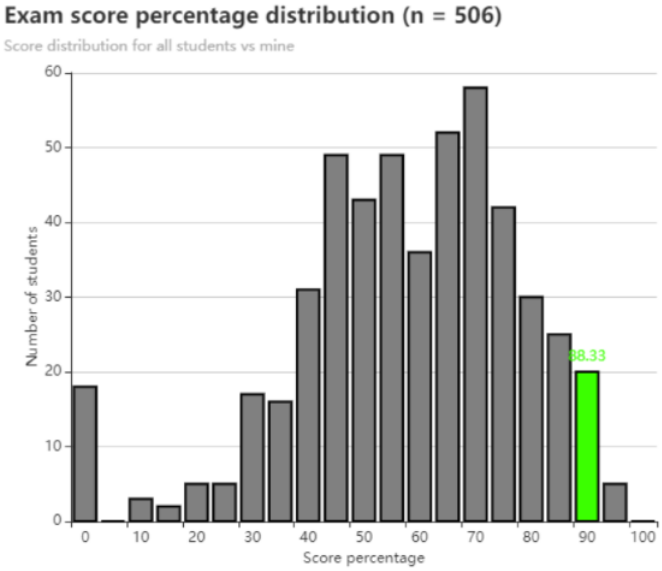
čime se studentima daje uvid u točnost njihovih rješenja tijekom rješavanja programskog zadatka te im se, u količini u kojoj određena vrsta zadataka to omogućuje, prikazuje učinjena pogreška. Ovakvim se pristupom studentima omogućuje ispravak koda na način koji nalikuje stvarnom načinu otklanjanja pogrešaka (engl. *debug*). Kako bi riješavanje programskih zadataka nalikovalo papirnatom obliku riješavanja zadataka, student ima pravo na određeni broj pokretanja vlastitog koda čime se potiče efikasnost pisanja algoritama.



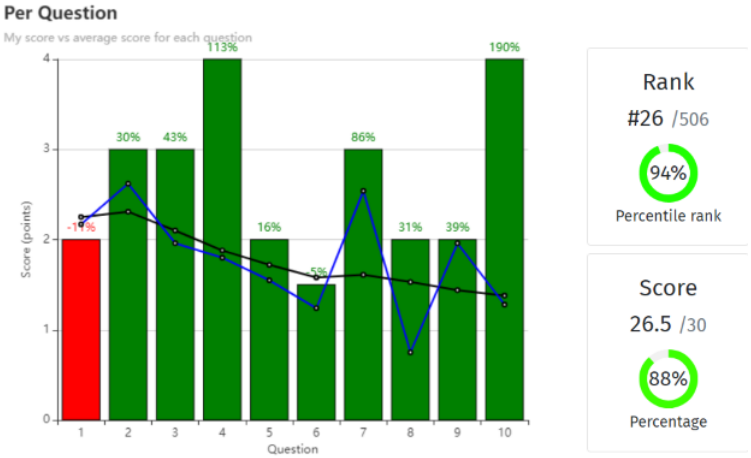
Slika 2.1: Primjer definicije testnog slučaja

Nakon rješavanja pojedinih nastavnih aktivnosti nekog od kolegija, studenti imaju uvid u statistiku rješavanja zadataka svih studenata koji u određenoj nastavnoj godini sudjeluju u dotičnom kolegiju. Slika 2.2 prikazuje distribuciju bodova prvog kolokvija iz kolegija Baze podataka, a Slika 2.3 prikazuje statistiku bodova po zadacima prvog kolokvija iz kolegija Baze podataka. Student ima uvid u sveukupnu poziciju na kojoj se nalazi unutar distribucije bodova svih studenata koji su sudjelovali u istoj nastavnoj aktivnosti te za svaki pojedini zadatak neke aktivnosti može vidjeti vlastiti te prosječan broj bodova svih studenata koji su rješavali isti zadatak. Ovim se pristupom potiče povećanje truda i vremena koje student ulaže u određeni kolegij s pretpostavkom natjecateljskog duha studenat

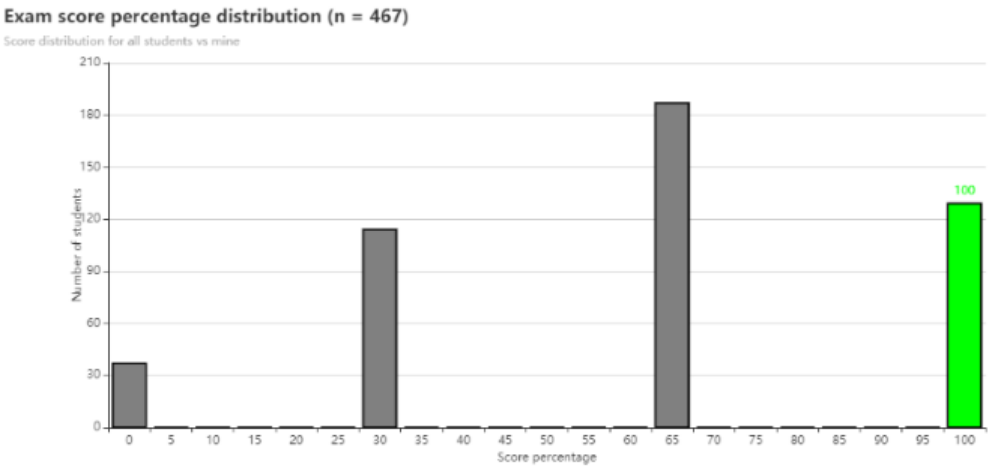
Slika 2.4 prikazuje studentovu poziciju u ukupnoj distribuciji bodova četvrte laboratorijske vježbe kolegija Baze podataka. Student također za pojedini zadatak ima uvid u vlastite bodove kao i prosjek bodova svih studenata, što je prikazano na Slici 2.5.



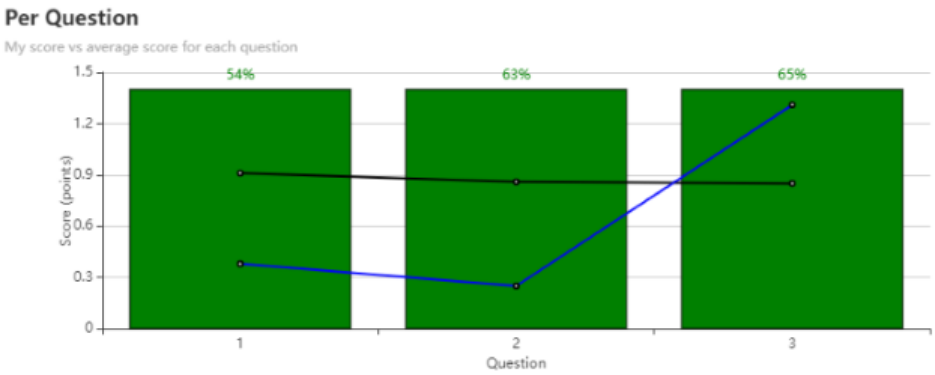
Slika 2.2: Prikaz distribucije bodova prvog kolokvija iz kolegija Baze podataka



Slika 2.3: Prikaz statistike bodova po zadacima prvog kolokvija iz kolegija Baze podataka



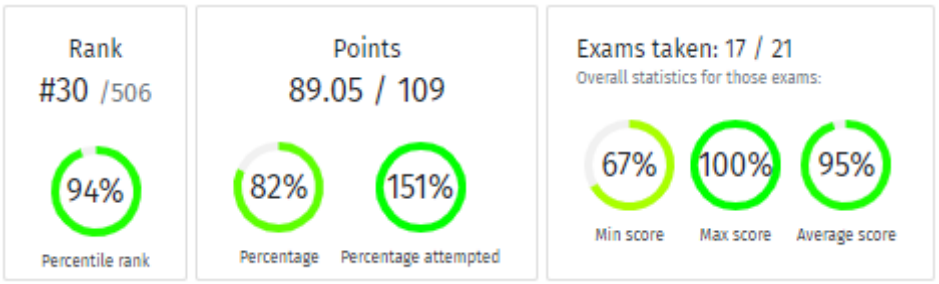
Slika 2.4: Prikaz sveukupne statistike bodova iz četvrte laboratorijske vježbe kolegija Baze podataka



Slika 2.5: Prikaz statistike bodova pojedinih zadataka iz četvrte laboratorijske vježbe kolegija Baze podataka

Student osim pregleda podataka o pojedinoj nastavnoj aktivnosti kolegija ima mogućnosti i u uvid sveukupne statistike bodova iz svih aktivnosti cijelog kolegija kao što je prikazano na Slici 2.6.

Osim programskih zadataka, *Edgar* omogućuje sastavljane te ispravljanje ABC pitalica, pitanja koja zahtijevaju tekstualni odgovor, skaliranih pitanja koja zahtijevaju odgovor u intervalu od 1 - 5 (engl. *scale*) te pitanja koja će studenti međusobno ispravljati te bodovati (engl. *peer assessment*). Ovakvim se opsegom mogućih tipova pitanja neke nastavne aktivnosti pojedinih kolegija mogu u velikoj većini, ako ne i u potpunosti, održavati preko sustava *Edgar*.



Slika 2.6: Prikaz sveukupne statistike bodova iz kolegija Baze podataka

Navedene mogućnosti daju uvid u opsežnost informacija s kojima je sustav *Edgar* u stanju djelovati. Podaci o svim prethodno pisanim provjerama svih studenata preko sustava *Edgar* tijekom svih nastavnih godina te informacija jesu li položili određeni kolegij, su spremljeni u bazi podataka. Ovako opsežan izvor podataka potiče ideju o razvoju određenih sustava strojnog učenja.

3. Motivacija za izradu prototipa integracije sustava za strojno učenje

Visoka učilišta kojima je stalo do kvalitete i poboljšanja nastavnog procesa prikupljaju i analiziraju podatke o učinku studenata na kolegijima. Jedan od mogućih načina procjene studentskih performansi je dubinska analiza podataka (engl. *data mining*). Cilj ovakvog pristupa je pronaći prosječan ishod studenata na kolegijima i na pojednim nastavnim aktivnostima, pronaći povezanost između performansi studenata tokom trajanja semestra i postotka prolaznosti te iskoristiti dobivene podatke u svrhu poboljšanog prijenosa znanja s nastavnog osoblja na studente. Pristup svim podatcima koje sustav Edgar pruža omogućuje izradu prediktivne analize studentskih performansi te u skladu s time unaprijeđenje pristupa učenju u svrhu poboljšanja studentskih rezultata. Na ovakav se način može učiniti novi korak u području sustava za automatiziranu provjeru znanja.

U nastavku je opisan proces dubinske analize podataka u svrhu predviđanja prolaznosti studenata sa svim korištenim podacima i dobivenim rezultatima preuzet iz članka [5].

3.1. Struktura kolegija Baze podataka

Kolegij Baze podataka u sklopu FER-a se sastoji od 100 mogućih bodova, a struktura nastavnih aktivnosti je prikazana u Tablici 3.1. Studenti s više od 50 bodova ostvaruju prolaz na dotičnom kolegiju. *Online* ispitima je moguće ostvariti 25% bodova, oni mogu biti nadzirani (u trajanju od primjerice 2 sata u prostorijama fakulteta uz prisutnost nastavnog osoblja) te nenadzirani (tipično u trajanju od tjedan dana, pišu se kod kuće). Prikazana struktura sa 8 *online* ispita te međuispitom koji se održava u sredini semestra olakšava praćenje napretka pojedinih studenata te pravovremenu intervenciju u skladu s njihovim rezultatima. Takva struktura, sa semestrom podijeljenim na dvije cjeline jednakog trajanja te s ispitima, koji donose najveći udio bodova, raspodijelje-

Tablica 3.1: Struktura nastavnih aktivnosti kolegija Baze podataka

Naziv	Tip	Broj bodova
1. domaća zadaća (SQL)	Online/nenadziran	1
2. domaća zadaća (SQL)	Online/nenadziran	1
1. test s višestrukim odgovorima	Online/nadziran	5.8
1. SQL programski test	Online/nadziran	4.2
Međuispit	Papirnati oblik	30
3. domaća zadaća (SQL)	Online/nenadziran	1
4. domaća zadaća (SQL)	Online/nenadziran	1
2. test s višestrukim odgovorima	Online/nadziran	8
2. SQL programski test	Online/nadziran	3
Nastavnički bodovi	Proizvoljno	5
Završni ispit	Papirnati oblik	40

nim na krajevima obiju cjelina, pruža kvalitetne referentne točke za izradu prediktivnih modela.

3.2. Predikcija prolaznosti studenata iz kolegija Baze podataka

Za predikciju prolaznosti studenata, korišteni su podaci dobiveni tijekom trajanja kolegija putem sustava Edgar te podaci dobiveni prije nego što su studenti upisali kolegij, a koji uključuju spol, prosjek ocjena srednjoškolskog obrazovanja, bodove dobivene polaganjem ispita državne mature, ukupan broj prijemnih bodova prilikom upisa na fakultet, poredak prilikom upisa na fakultet, podatak sudjeluje li student na kolegiju Baze podataka po prvi puta te prosjek ocjena studenta u fazi studiranja prethodno od upisa kolegija Baze podataka.

Skup podataka od 361 promatranih pojedinaca pojedijeljen je u 75%-tni skup za treniranje te 25%-tni skup za testiranje. Skup za treniranje je primijenjen za evaluaciju nekoliko modela dubinske analize podataka koristeći unakrsnu provjeru (engl. *cross validation*). Tijekom analize je korišten skriptni jezik *R* te alat *RStudio*. Modeli su procijenjeni nad tri različita skupa nepoznanica u odnosu na vrijeme u kojem su podaci prikupljeni. Prvi skup uključuje podatke dobivene prije nego što su studenti upisali kolegij, drugi skup uključuje podatke dobivene tijekom provedbe kolegija, a treći skup

uključuje uniju prethodno navedena dva skupa. Modeli su procijenjeni u odnosu na činjenicu je li student položio kolegij jer je to najbitniji referentna točka u području visokih obrazovnih ustanova. Najbolji je model u mogućnosti s približno 80% predvidjeti hoće li student položiti kolegij već nakon prve polovice semestra, odnosno nakon održavanja međuispita.

Ovakvi rezultati uvelike motiviraju izradu prototipa modela strojnog učenja te njegovo integriranje u sustav Edgar zbog ranog pružanja povratne informacije nastavnom osoblju te poduzimanja sukladnih akcija kako bi se povećala prolaznost studenata, smanjio opseg tereta nastavnog osoblja, a ujedno zadržala razina i kvaliteta znanja koja se prenosi na studente tijekom provedbe kolegija. Kako bi se dubinska analiza podataka i izvršavanje algoritama strojnog učenja mogli izvesti preko sustava *Edgar* potrebno je ustrojiti sustav na način koji podržava izvođenje proizvoljnog *R* koda te dohvat rezultata pokrenutog koda, koji će osim ispisa na standardni izlaz sadržavati i popratni generirani sadržaj poput slika (npr. histogram ili stablo odluke).

4. Korištene tehnologije

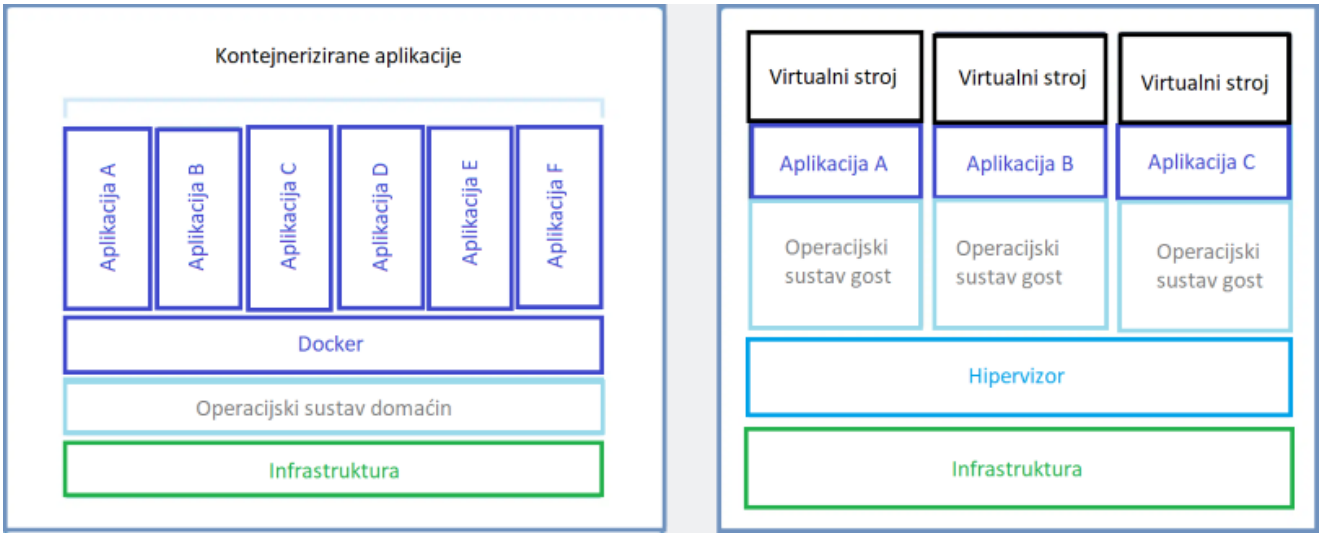
4.1. Docker

Docker[2] je otvorena platforma za razvoj, isporuku te pokretanje aplikacija. *Docker* je virtualizacijska te kontejnerizacijska (engl. *containerization*) tehnologija koja omogućuje stvaranje *Linux* kontejnera (engl. *container*). On uvelike smanjuje vrijeme između pisanja koda i pokretanja aplikacije u produkciji te pruža skalabilnost i portabilnost. *Docker* omogućuje pakiranje (engl. *package*) i pokretanje aplikacije u labavo izoliranom okruženju zvanom kontejner.

Kontejner[1] je, za razliku od virtualnog stroja (engl. *virtual machine*) izolirani proces koji se izvodi na operacijskom sustavu, bez potrebe za dodatnim procesima i hipervizorom (Slika 4.1[1]). Virtualni strojevi za svoje pokretanje zahtijevaju dodatan operacijski sustav zvan gost. Zbog toga bi za pokretanje većeg broja aplikacija bilo potrebno imati veći broj gosta, a samim time se javlja i potreba za hipervizorom. Kontejneri ovdje imaju prednost jer je više kontejnera moguće istodobno izvoditi na jednom operacijskom sustavu. Kontejner omogućuje izvođenje aplikacije u drugačijim okruženjima, a može se koristit i na *Windows* i *Linux* operacijskom sustavu domaćinu. *Docker* koristi upravo ovu činjenicu kako bi programerima olakšao stvaranje aplikaciju, njihovu isporuku te upravljanje samim procesom izgradnje aplikacije unutar tima. On svaku aplikaciju stavlja u poseban kontejner čime je omogućeno pokretanje većeg broja izoliranih aplikacija na jednom operacijskom sustavu. *Docker* također korisniku uvelike olakšava upravljanje kontejnerima.

4.1.1. Arhitektura

Docker koristi klijent-server (engl. *client-server*) arhitekturu. Klijent komunicira s *Docker* pozadinskim procesom (engl. *daemon*) koji izgrađuje, pokreće te distribuira *Docker* kontejnere. Pozadinski proces je proces malog prioriteta koji obavlja zadatke u pozadini poput sakupljanja smeća (engl. *garbage collection*). *Docker* klijent i pozadin-



Slika 4.1: Razlika kontejnera i virtualnog stroja

ski proces mogu biti pokrenuti na istom sustavu, a klijent također može komunicirati s udaljenim (engl. *remote*) pozadinskim procesom. *Docker* klijent i pozadinski proces komuniciraju pomoću REST API-a. Također postoji klijent *Docker Compose* koji se koristi za upravljanje aplikacijama koje se sastoje od skupa kontejnera, kao što je to kod Judge0a, sustava za izvršavanje koda.

Docker pozadinski proces sluša API zahtjeve te upravlja *Docker* objektima poput slika, kontejnera, mreža te volumena (engl. *volume*).

Docker klijent je primaran način na koji korisnik komunicira s *Dockerom*. Prilikom izvršavanja *Docker* naredaba , klijent šalje naredbe *Docker* pozadinskom procesu koji ih izvršava.

Docker registar (engl. *registry*) sprema *Docker* slike.

Docker Hub je javan registar s kojeg *Docker* primarno traži slike.

Docker objekti

Slika (engl. *image*): predložak (engl. *template*) s instrukcijama za izgradnju *Docker* kontejnera. Slike mogu biti izgrađene na temelju drugih slika s dodatnim prilagodbama. Slike se grade pomoću *Dockerfilea* (Slika 4.2) koji se sastoji od poprilično jednostavne sintakse za definiciju koraka potrebnih za izgradnju i pokretanje slike. Svaka instrukcija *Dockerfilea* stvara jedan sloj slike što sliku čini laganom (engl. *lightweight*), malom i brzom u usporedbi s ostalim virtualizacijskim tehnologijama.

Kontejner (engl. *container*): pokretljiva instanca unutar slike. On se može povezati s jednom ili više mreža, može mu se pridijeliti memorija te se može kreirati slika na temelju njegovog trenutnog stanja. U zadanom (engl. *default*) stanju on je relativno labavo izoliran od ostalih kontejnera te operacijskog sustava domaćina.

Svezak (engl. *volume*): datotečni sustav povezan s *Docker* kontejnerom koji čuva podatke generirane od strane kontejnera. Podaci su spremljeni na domaćinu te su neovisni o životnom vijeku kontejneru. Ova činjenica pruža korisnicima mogućnost dijeljenja datotečnih sustava između kontejnera.

Docker je napisan u programskom jeziku *Go* te koristi nekoliko svojstava *Linux kernela* kako bi pružao opisane funkcionalnosti. Koristi tehnologiju zvanu imenski prostor (engl. *namespace*) koja pruža izolirani radni prostor zvan kontejner. Imenski prostori pružaju sloj izolacije na način da je svaki kontejner pokrenut unutar odvojenog imenskog prostora te ima pristup samo svojem imenskom prostoru.

4.1.2. Upute za korištenje

Primarna datoteka za izgradnju slika je *Dockerfile*. *Dockerfile* je tekstualna dototeka koja sadrži upute za izgradnju slika u kojem su naredbe jednake onima koje bi korisnik unio u komandnu liniju kako bi izgradio istovjetnu sliku. Prilikom izvođenja naredaba, u pozadini se slika stvara ″sloj po sloj″ što omogućuje lakše verzioniranje i upravljanje izgradnje same aplikacije. Slika 4.2 prikazuje osnovan primjer *Dockerfilea* koji gradi sliku ovisnu o jeziku *Go*, ova će slika biti dostupna na vratima (engl. *port*) 8080 unutar *Docker* kontejnera.

```
FROM golang:1.16-alpine

WORKDIR /app

COPY go.mod ./
COPY go.sum ./
RUN go mod download

COPY *.go ./

RUN go build -o /docker-gs-ping

EXPOSE 8080

CMD [ "/docker-gs-ping" ]
```

Slika 4.2: *Dockerfile*

Alpine je lagana (engl. *lightweight*) distribucija *Linuxa* koja se koristi u sigurnosne svrhe, a također omogućuje kratko vrijeme pokretanje (engl. *boot-up time*) kontejnera.

Većina se upravljanja, pokretanja te zaustavljanja aplikacije izgrađene *Dockerom* izvodi pomoću komandne linije. Također je razvijena aplikacija radne površine (engl. *desktop*) *Docker Desktop* koja omogućuje jednaku funkcionalnost uz bolje korisničko iskustvo.

4.2. Servisno orijentirana arhitektura

Servisno orijentirana arhitektura (engl. *Service oriented architecture*), SOA , je arhitekturni stil izgradnje aplikacije koji se sastoji od labavo povezanih dijelova, servisa (engl. *service*), koji međusobno komuniciraju te izmjenjuju poruku ili usklađuju obavljanje određene aktivnosti. Glavni koncepti servisno orijentirane arhitekture uključuju:

- strateški ciljevi su bitniji od specifičnih projekata
- interoperabilnost je važnija od prilagođene integracije
- dijeljeni servisi su važniji od implementacije specifične svrhe
- kontinuirano napredovanje je važnije od trenutne savršenosti

Servisno orijentirana arhitektura je implementacija servisnog modela programiranja. U ovom je arhitekturnom stilu poslovni proces implementiran kao servis koji pruža pristup preko striktno definiranih programskih sučelja, API-ja. Pojava RPC-a je potaknula razvoj servisno orijentirane arhitekture jer je RPC omogućio poziv procesa strane aplikacije kao da je pozvan unutar izvorišne aplikacije. Servisno orijentirana arhitektura omogućuje jednostavnu povezanost s već razvijenim aplikacijama. Ovaj arhitekturni stil ima brojne prednosti od kojih su neke:

Ponovna iskoristivost: servisi se mogo ponovno iskoristiti kroz mnogo različitih aplikacija

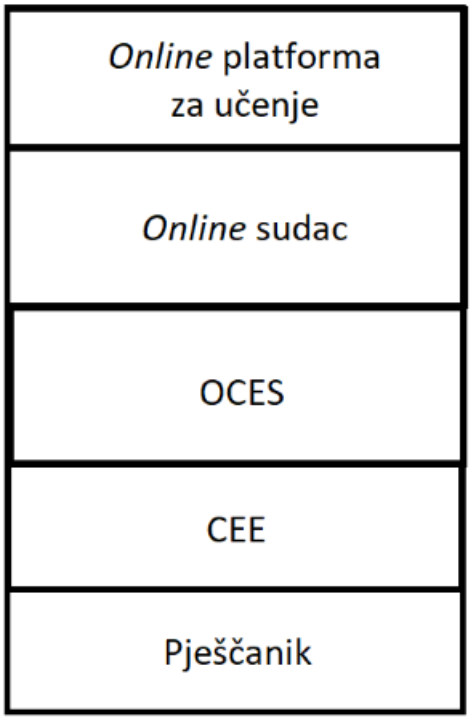
Jednostavna održivost: jer su svi servisi međusobno neovisni, njihovo je unaprjeđenje jednostavno te nema utjecaj na ostale servise koji s njim komuniciraju

Interoperabilnost: korištenje standardiziranog protokola (poput REST API-a) dozvoljava komunikaciju servisa te prijenos podataka neovisno o jeziku u kojem su pisani

Skalabilnost: servisi mogu biti pokrenuti na različitim serverima, dodatno, korištenje standardiziranog protokola dozvoljava smanjenje interakcije između klijenata i servisa

4.3. Judge0

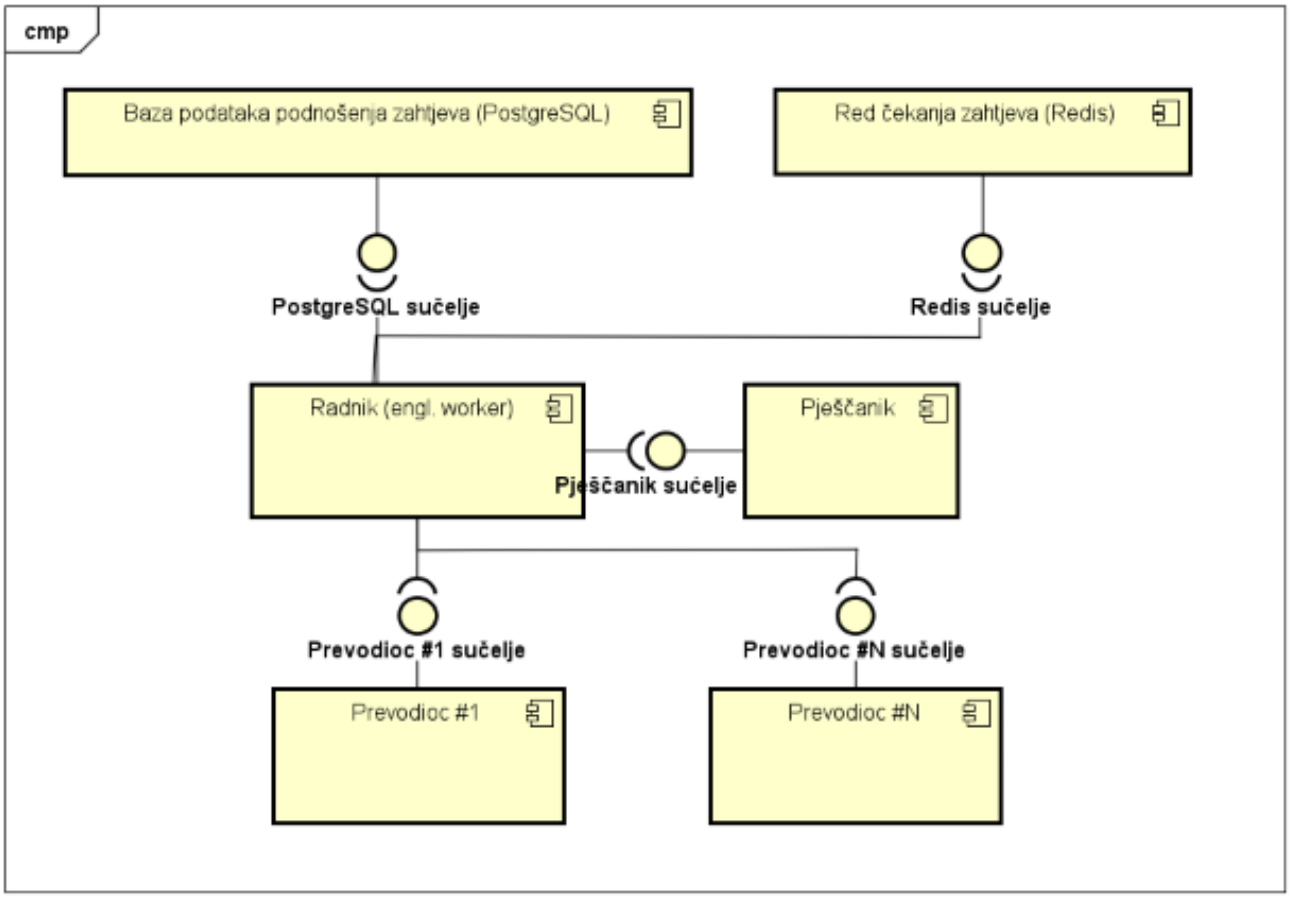
Judge0[4] je općenit, robustan (engl. *robust*), skalabilan (engl. *scalable*), podesiv sustav otvorenog pristupa (engl. *open-source*) za izvršavanje koda (OCES). Sastoji se od moderne modularne arhitekture koju je moguće upogoniti preko proizvoljnog broja računala i operacijskih sustava. Napravljen je kako bi pružao više funkcionalnosti te jednostavniju uporabu nego što to pružaju dosadašnja rješenja u području *online* sudaca te *online* sustava za izvršavanje koda. *Online* suci su sustavi dizajnirani za ocjenjivanje te evaluaciju korisnikovih kodova nad predefiniranom skupu testnih slučaja. *Judge0* je uvelike korišten na FER-u unutar sustava *Edgar*, a u sklopu automatiziranih provjera na desetak kolegija. Slika 4.3 sadrži prikaz arhitekture ekosustava *online* sudaca (engl. *online judge*) koja se sastoji od različitih slojeva sa različitim zadaćama.



Slika 4.3: Arhitektura ekosustava *online* sudaca

Arhitektura se sastoji od pješčanika (engl. *sandbox*) koji osigurava sigurnost te se učestalo koristi i za potrebe testiranja malicioznog koda, stroja za pokretanje koda

(CEE), prikazan na Slici 4.4[4], koji koristi pješčanik kako bi preveo (engl. *compile*) te pokrenuo (engl. *run*) kod predan na izvođenje te *online* sustava za izvršavanje koda koji pruža API za prevođenje i pokretanje proizvoljnog izvršnog koda. U sklopu *Judge0a*, pješčanik se osim u sigurnosne svrhe koristi i za osiguravanje granica zauzeća memorije te korištenja centralne procesorske jedinice (CPU). Ova je mogućnost posebno korisna u sklopu programskih natjecanja, a i kod *online* sustava za učenje jer se mogu odrediti granice vremena izvođenja programa te maksimalne količine memorije koju programi koji se testiraju mogu zauzeti.



Slika 4.4: CEE arhitektura

Online sudac je online servis (engl. *service*) koji provodi podnošenje (engl. *submission*), procjenu te bodovanje. U fazi se podnošenja korisnikov kod prevodi, u fazi procjene se rezultati korisnikovog koda provjeravaju u odnosu na predefinirani skup testnih slučajeva te se u fazi bodovanja rezultati koda ocjenjuju u odnosu na rezultate faze procjene. *Judge0* uvodi ideju odvajanja *online* sudaca od OCES-a kako bi njegovo područje primjene bilo raširenije. *Judge0* omogućuje skalabilnost na više načina. Prvi je način vertikalno skaliranje, što podrazumijeva korištenje određenog broja

Tablica 4.1: Atributi tablice *languages*

Naziv	Tip	Opis
id	cijeli broj	Jedinstvena oznaka programskog jezika.
name	znakovni niz	Naziv programskog jezika.
compile_cmd	tekst	Naredba za prevođenje koda.
run_cmd	tekst	Naredba za pokretanje koda.
source_file	znakovni niz	Naziv dokumenta u kojem će kod biti spremljen (npr. Glavni.java).
is_archived	znakovni niz	? (može biti 'f' ili 't')

CEE-eva koji su pokrenuti na istom računalu čime se omogućuje paralelno pokretanje više podnesenih zahtjeva. Drugi je način horizontalno skaliranje koje podrazumijeva pokretanje više OCES-ova na više računala te njihovo upravljanje pomoću sustava za raspodjelu opterećenja (engl. *load-balancer*). Horizontalno je skaliranje moguće ostvariti zajedno sa vertikalnim skaliranjem, pri čemu je vertikalno skaliranje ostvareno na pojedinim računalima. Također je moguće koristi *Docker* tehnologiju gdje je svaki modul moguće isporučiti u obliku *Docker* kontejnera čime se olakšava instanciranje i instalacija.

4.3.1. *Judge0* API

Judge0 koristi jednostavan JSON API[3] koji se sastoji od dvije krajnje točke (engl. *endpoint*). Prva krajnja točka omogućuje stvaranje novog zahtjeva za jedan od dostupnih programskih jezika. Druga krajnja točka omogućuje provjeru statusa prethodno stvorenih zahtjeva te pregled rezultata izvođenja koda predanog u podnesenom zahtjevu. Zahtjev predan na prvu krajnju točku mora sadržavati barem izvršni kod te identifikator programskog jezika. Ostali su ulazni parametri korišteni u sklopu ovog rada s njihovim opisom prikazani u Tablici 4.2. Podržavanje izvršavanja izvornog koda pisanog u proizvoljnog jeziku se jednostavno omogućuje dodavanjem novog retka u tablicu *languages* (Tablica 4.1) koja se nalazi u bazi podataka podnošenja zahtjeva.

Druga krajnja točka pruža informacije o rezultatima izvršavanja poput standardnog izlaza, količine korištenje memorije te vremena izvođenja. Izlazni parametri dobiveni kao rezultat izvođenja izvršnog koda s druge krajnje točke, a koji su korišteni u sklopu ovog rada, su detaljnije opisani u Tablici 4.3.

Tablica 4.2: Ulazni parametri prilikom predaje zahtjeva na *Judge0* sustav

Ime	Tip	Opis
source_code	znakovni niz	Izvršni kod programa.
language_id	cijeli broj	ID jezika programa.
compiler_options	znakovni niz	Opcije za prevodioca.
callback_url	znakovni niz	URL na koji *Judge0* izvršava PUT zahtjev s informacijama o podnesenom zahtjevu nakon što je obrađen.
additional_files	Base64 enkodirani niz znakova	Dodatne datoteke koje su potrebne za izvršavanje koda. Vrijednost ovog parametra predstavlja sadržaj komprimirane (.zip) datoteke koja sadrži dodatne datoteke.

4.3.2. Komunikacija korisnika i sustava

Komunikacija između korisnika i *Judge0* sustava se može odvijati na više načina. Prvi način započinje korisnikovim podnošenjem zahtjeva na prvoj krajnjoj točci. U sljedećem se koraku unutar sustava kreira zapis o podnesenom zahtjevu u bazi podataka prikazanoj na Slici 4.3 te se zahtjev dodaje u red čekanja zahtjeva. Dodavanjem u red čekanja, vraća se podatak o jedinstvenoj oznaci (token) predanog zahtjeva ko-

Tablica 4.3: Izlazni parametri nakon izvršavanja zahtjeva na *Judge0* sustavu

Ime	Tip	Opis
callback_url	znakovni niz	URL na kojem *Judge0* izvršava PUT zahtjev nakon što je podnesen zahtjev za izvršavanjem koda obavljen.
stdout	tekst	Sadržaj standardnog izlaza (engl. *standard output*) programa nakon izvršavanja.
stderr	tekst	Sadržaj standardnog izlaza za pogreške (engl. *standard error*) programa nakon izvršavanja.
token	znakovni niz	Jedinstven znakovni niz pridijeljen pojedinom zahtjevu.

jim korisnik može pristupiti rezultatima zahtjeva na drugoj krajnjoj točci. CEE kontinuirano dohvaća zahtjeve iz reda čekanja te ih inicijalizita u pješčaniku, priprema datoteke u pješčaniku, prevodi izvršni kod, izvršava prevedeni kod, određuje status zahtjeva, čisti ekosustav pješčanika te ažurira stanje zahtjeva u bazi podataka. Korisnik može kontinuirano provjeravati status zahtjeva sve dok njegova obrada nije završena. Ovakav je način komunikacije između korisnika i *Judge0* sustava asinkron (engl. *async*) jer korisnik nema informaciju o tome kada će zahtjev biti obrađen. U drugom načinu asinkrone komunikacije korisnik prilikom podnošenja zahtjeva predaje *callback_url* parametar koji označuje URL na kojem će dobiti informacije o izvršenom zahtjevu nakon njegove obrade. *Judge0* također pruža mogućnosti sinkrone (engl. *sync*) komunikacije, ali taj način nije preporučen od strane autora.

4.4. Jezici podržani za predviđanje rezultata

4.4.1. *R*

Skriptni jezik *R* je uvelike korišten u području statistike i strojnog učenja. Jedan od glavnih razloga raširenog korištenja jezika *R* je taj što je jezik u suštini izrađen od strane statističara te upravo za primjenu u statistici. Ovaj jezik ima brojne prednosti u području statistike i strojnog učenja. Neke od njih su prikladnost jezika za provođenje podatkovne analize, brojne biblioteke koje jezik uključuje, a koje su razvijene za potrebe statističke analize te prikladnost za istraživačke zadatke (engl. *exploratory work*). *R* je prikladan za provedbu analize jer su podatkovna analiza te vizualizacija uvelike podržani bibliotekama napisanim za jezik *R*. *R* također pruža podršku raznih struktura podataka koje su pogodne upravo za navedena područja. Jezik *R* omogućuje brzu izradu prototipa modela strojnog učenja te uporabu podatkovnih skupova u izradi modela strojnog učenja. U usporedbi sa skriptnim jezikom Python, *R* je pogodniji za ad-hoc analizu (analizu već postojećih podataka) te je upravo zbog navedenih funkcionalnosti izabran za potrebe ovog rada. Za potrebe podatkovne analize te izrade modela strojnog učenja, najraširenije su biblioteke poput:

Caret: Korišten za klasifikaciju te regresiju. Sadrži alate za razdvajanje podataka (engl. *data splitting*), pre-procesiranje (engl. *pre-processing*), odabir značajki (engl. *feature selection*) te mnoge druge funkcionalnosti.

DataExplorer: Popularna biblioteka s primjenom u strojnom učenju. Područja fokusa ove biblioteke su istraživačka podatkovna analiza (engl. *exploratory data*

analysis) te podatkovno izvještavanje (engl. *data reporting*). Ova biblioteka automatizira proces podatkovnog istraživanja (engl. *data exploration*) za analitičke zadatke i izradu prediktivnih modela.

Dplyr: Brz i konzistentan alat za rad s podatkovnim okvirima (engl. *data frame*), olakšava najčešće manipulacijske izazove.

Ggplot2: Jedna od najraširenijih biblioteka za vizualizaciju podataka.

kernLab: Biblioteka za klasifikaciju, regresiju, grupiranje (engl. *clustering*), detekciju anomalije (engl. *novelty detection*) te smanjenje dimenzionalnosti (engl. *dimensionality reduction*).

4.4.2. *R Markdown*

Za potrebe ovog rada, također je potrebno predvidjeti upotrebu *R Markdowna (RMD)*. *R Markdown* je tekstualna datoteka koja uz sam *R* kod omogućuje i korištenje deskriptivnog teksta te prikaz izlaza pojedinih odsječaka koda. Izvršavanje *R Markdown* datoteke je omogućeno u *R Studiju*. *R Studio* je integrirano razvojno okruženje namijenjeno za jezik *R*. Osim korištenjem *R Studia*, *R Markdown* datoteke je moguće pokrenuti izravno iz *R* skripte uporabom *rmarkdown* biblioteke kao što je prikazano u nastavku.

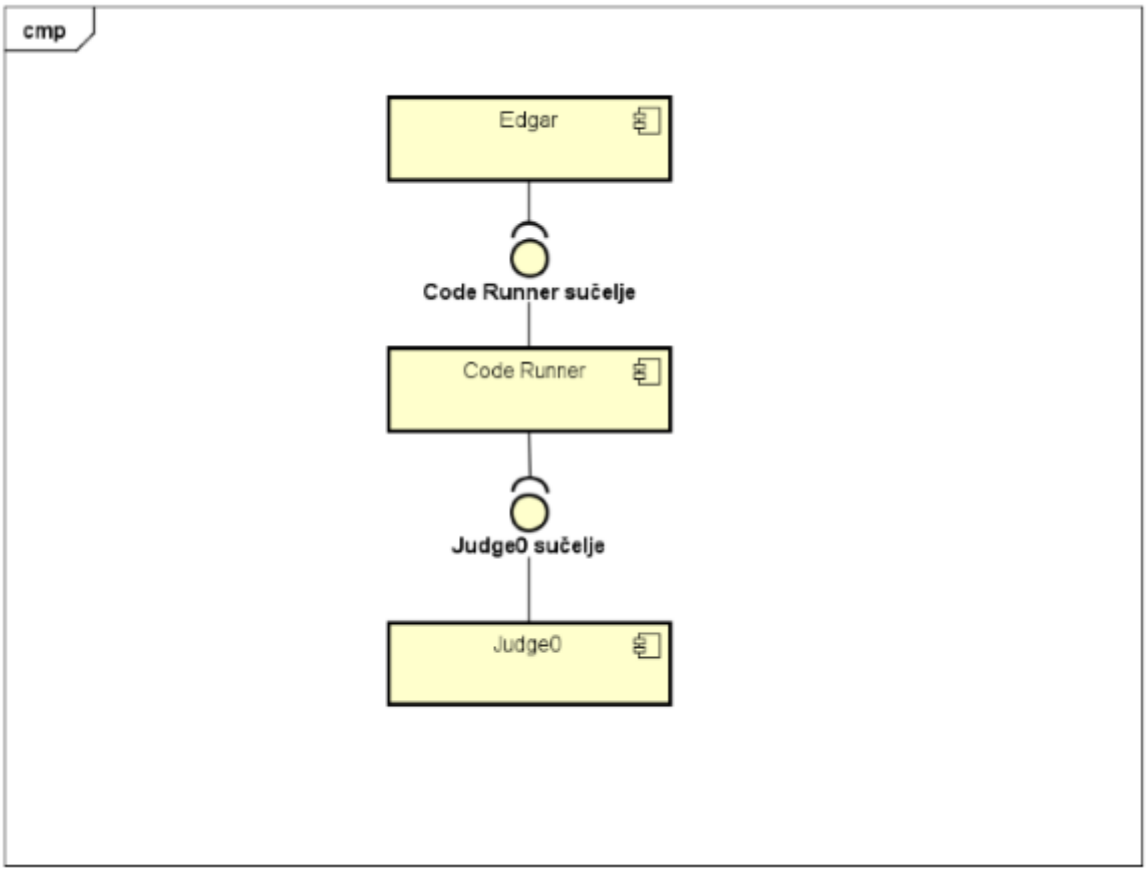
```
rmarkdown::render('script.Rmd')
```

Izvršavanjem *R Markdowna* se mogu kreirati primjerice *PDF* i *HTML* datoteke, čiji je format mnogo prikladniji za izvještavanje statističkih rezultata te uključuje i mogućnost strukturiranog prikaza matematičkih formula.

4.5. *Edgar*

Sustav *Edgar* se temelji na servisno orijentiranoj arhitekturi. Odabir SOA arhitekture je prigodan izbor za izgradnju ovakvog sustava jer su za njegovu veličinu, složenost te opseg funkcionalnosti koje pruža vrlo bitni skalabilnost, interoperabilnost, ponovna iskoristivost te jednostavna održivost. Ovakav je sustav konstantno otvoren za dodavanje novih funkcionalnosti te je stoga mogućnost ponovne uporabivosti određenih svojstava aplikacije bitna za jednostavnu nadogradnju sustava.

Sam sustav sadrži niz opcionalnih funkcionalnosti poput izvršavanja koda proizvoljnog programskog jezika koristeći *Judge0* sustav. Sustav *Edgar* te *Judge0* sustav komuniciraju preko programa zvanog *Code Runner* koji služi kao premosnik između dvaju sustava osiguravajući slanje i primanje pogodno strukturiranih podataka. Slika 4.5 prikazuje dio arhitekture *Edgar* sustava koje se odnosi na pokretanje nepovjerljivog koda. Slika 4.6 prikazuje strukturu podataka dobivenih slanjem zahtjeva za izvršavanjem koda na *Code Runner* program. Podaci uključuju status predanog zahjteva, ispis sa standardnog izlaza nakon izvršavanja koda, podatke o programskom jeziku predanog na izvršavanje te trenutke podnošenja i kraja izvršavanja zahtjeva.



Slika 4.5: Struktura izlaznih podataka *Code Runner* API-a

Slika 4.6: Struktura izlaznih podataka *Code Runner* API-a

Code Runner interno koristi prioritetni red zahtjeva za izvršavanjem te komunicira sa *Judge0* sustavom koristeći asinkoron način komunikacije koji uključuje predaju *callback_url* parametra. *Code Runner* također omogućuje skladištenje raznih metrika o pristiglim zahtjevima. On je ostvaren kao kontejnerizirana aplikacija koristeću *Docker* sustav, a osim s njim, sustav *Edgar* također ima mogućnost komunikacije s dodatnim kontejneriziranim aplikacijama poput *pg runner* i *mongo runnera* koji omogućuju evaluaciju SQL te NoSQL upita no njihova je realizacija izvan dosega ovog rada.

5. Opis rješenja

Kako bi se mogao vršiti algoritam strojnog učenja, potrebno je proširiti sustav *Edgar* na način koji omogućuje pokretanje proizvoljnog *R* ili *R Markdown* koda te dohvat rezultata (ispisa na standardni izlaz) pokrenutog koda kao i svih popratnih rezultata koji uključuju slike, dijagrame, stabla odluke ili pak cjelokupnu stvorenu *PDF* ili *HTML* datoteku kao što je to slučaj prilikom izvršavanja *R Markdown* datoteke. Za ovakvu je potrebu prigodno koristit već upogonjen *Code Runner* program koji komunicira s *Judge0* sustavom te proširiti njegovu funkcionalnost prema potrebama integracije modela strojnog učenja jer cjelokupan sustav *Edgar* već podržava izvršavanje proizvoljnog koda. Ovakvo rješenje ima prednost nad ostalim zato što *Code Runner* program ima dugogodišnju primjenu u sklopu *Edgar* sustava te se pokazao kao ispravno i nadogradivo rješenje za potrebe izvršavanja proizvoljnog koda.

5.1. Proširenje *Judge0* sustava

Kako bi se omogućila integracija sustava za strojno učenje, *Code Runner* sustav je proširen manipulacijom 'run_cmd' atributa tablice *languages* povezanog s *R* jezikom sustava *Judge0* na način prikazan u nastavku. Istovjetan je kod, ali sa cjelokupnim izgledom retka u tablici *languages* baze podataka sustava *Judge0* prikazan na Slici 5.1.

```
/usr/local/r-4.0.0/bin/R -e "
    dir.create('logs');
    dir.create('files');
    files_before = list.files('.');
    sink('logs/log.txt');
    library('base64enc');
    source('script.R');
    files_after = list.files('.');
    files_new = setdiff(files_after, files_before);
    files_new_path = paste('./files', files_new, sep='/');
    file.copy(files_new, files_new_path);
    sink();
```

```
        tar('source.tar.gz',
            c('logs', 'files'),
            compression='gzip');
        print(base64encode('source.tar.gz'));
    "
```

```
 id |  name   | compile_cmd |
          run_cmd
   | source_file | is_archived
----+---------+-------------+-----------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------
--------------------+-------------+-------------
 91 | R-4.0.0 |            | /usr/local/r-4.0.0/bin/R -e "dir.create('logs'); dir.create('files'); files_before =
 list.files('.'); sink('logs/log.txt'); library('base64enc'); source('script.R'); files_after = list.files('.'); f
iles_new
= setdiff(files_after, files_before); files_new_path = paste('./files', files_new, sep='/'); file.copy(files_new,
files_new_path); sink(); tar('source.tar.gz', c('logs', 'files'), compression='gzip'); print(base64encode('source.
tar.gz'));" | script.R    | f
(1 row)

judge0=#
```

Slika 5.1: Sadržaj tablice *languages* vezan uz jezik *R Judge0* sustava

Ovakvom se manipulacijom *R* koda predanog na izvršavanje u prvom koraku stvaraju direktoriji "logs" i "files" u koje će se redom spremati ispis na standardni izlaz predanog koda te datoteke stvorene izvršavanjem koda. U sljedećem se koraku sprema informacija o trenutnim datotekama i direktorijima koji se nalaze u radnom direktoriju. Tada se pokreće predana *R* skripta, spremljena kao "script.R" kako je definirano u tablici *languages*. Nakon izvršavanja predanog koda, provjerava se koje se datoteke i direktoriji nalaze u tekućem direktoriju te se usporedbom sa skupom datoteka i direktorija prije izvršavanja koda dolazi do skupa datoteka i direktorija koji su nastali tokom izvođenja predanog koda. Naredbom "sink()" se sav ispis na standardni izlaz zapisuje u datoteku "files/files.txt" koja se također nalazi u skupu datoteka nastalih tokom izvođenja predanog koda. Uporaba naredbe "sink()" je ključna kako taj sadržaj ne bi bio prepoznat od strane *Judge0* sustava kao ispis na standardan izlaz. Ovakvim se postupkom izdvajaju sve datoteke i sav ispis nastao tokom izvođenja predanog koda. Kreirane se datoteke poput histograma i stabla odluke dodatno premještaju u direktorij "files" jer *Judge0* sustav ne dozvoljava komprimiranje datoteka trenutnog direktorija, dok dozvoljava komprimiranje direktorija. Navedene se datoteke tada komprimiraju, a sam sadržaj tako nastale komprimirane datoteke se ispisuje na standardni izlaz u *Base64* formatu kojeg sustav *Judge0* prepoznaje te ga uklapa u odgovor na predani zahtjev u polju "*stdout*".

Istovjetna se manipulacija može izvesti za *R Markdown* zamjenom naredbe

```
source('script.R');
```

naredbom

```
rmarkdown::render('script.Rmd')
```

ukoliko je u tablici *languages* postavljanjem atributa "source_file" definirano spremanje izvršnog koda u datoteku "script.Rmd". Kako bi se omogućilo korištenje mogućnosti *R Markdowna*, potrebno je preuzeti *pandoc* tehnologiju. Preuzimanje je moguće učinit pokretanjem naredaba

```
$ docker exec -it judge0-v1130_workers_1 bash
$ sudo apt-get update -y
$ sudo apt-get install -y pandoc
```

gdje se prvom naredbom dobiva pristup komandnoj liniji komponente *worker Judge0* sustava.

Kako bi se omogućilo ispravno korištenje *R* ili *R Markdown* koda, potrebno je ručno preuzeti potrebne biblioteke jer sustav *Judge0* ne dozvoljava pokretanje naredbe

```
install.packages('biblioteka')
```

tijekom izvršavanja predanog koda. Preuzimanje potrebnih biblioteka se može učiniti tako što se pokretanjem naredbe

```
$ docker exec -it judge0-v1130_workers_1 bash
```

pristupi komandnoj liniji *worker* kontejnera. Tada se pokretanjem naredbe

```
$ sudo /usr/local/r-4.0.0/bin/R
```

pokreće *R* jezik za uporabu s komandne linije. Odavde se mogu preuzeti potrebne biblioteke.

5.2. Proširenje *Code Runner* sustava

Code Runner program prima poruku od strane *Judge0* sustava u kojoj se na mjestu polja *stdout* nalazi sadržaj komprimirane datoteke u *Base64* formatu. Potom se dekodirana datoteka raspakira te se njen sadržaj stavlja u JSON strukturu kakvu prikazuje Slika 5.2.

Prvi par JSON strukture sadrži "output" kao ključ te ispis na standardni izlaz koji je bio sadržan u datoteci "files/file.txt" kao vrijednost. Drugi par JSON strukture sadrži "files" kao ključ te dodatnu JSON strukturu kao vrijednost. Dodatna se JSON struktura sastoji od ključeva koje predstavljaju nazivi datoteka koje su stvorene izvršavanjem koda te vrijednosti koje se sastoje od JSON strukture koja sadrži *Base64* enkodiran sadržaj datoteke te nekoliko popratnih podataka o njoj. Nakon što su svi

```
{
    "output" : "<ispit na standarni izlaz>",
    "files":
    {
        "slika" :
        {
            "content" : "<Base64 enkodiran sadržaj datoteke>",
            "type" : "<tip datoteke (ekstenzija)>",
            "size" : "<veličina datoteke>",
            "height" : "<broj piksela visine slike>",
            "width" : "<broj piksela širine slike>"
        },
        "datoteka" :
        {
            "content" : "<Base64 enkodiran sadržaj datoteke>",
            "type" : "<tip datoteke (ekstenzija)>",
            "size" : "<veličina datoteke>"
        }
    }
}
```

Slika 5.2: Sadržaj JSON strukture polja *stdout* u odgovoru *Judge0* sustava

podaci o datotekama te njihov sadržaj spremljeni u strukturi, radni se direktorij čisti brisanjem raspakiranih datoteka. Opisana se struktura uz nekoliko dodatnih podataka šalje prema sustavu *Edgar*. Slika 5.3 prikazuje sadržaj JSON strukture s jednom stvorenom datotekom.

Slika 5.3: Sadržaj JSON-a s ispisom na standardni izlaz i stvorenim datotekama

Sustav *Edgar* prati servisno orijentiranu arhitekturu te je povezan s *Code Runner* programom putem servisa *CodeRunnerService* koji služi za krajnju komunikaciju sa

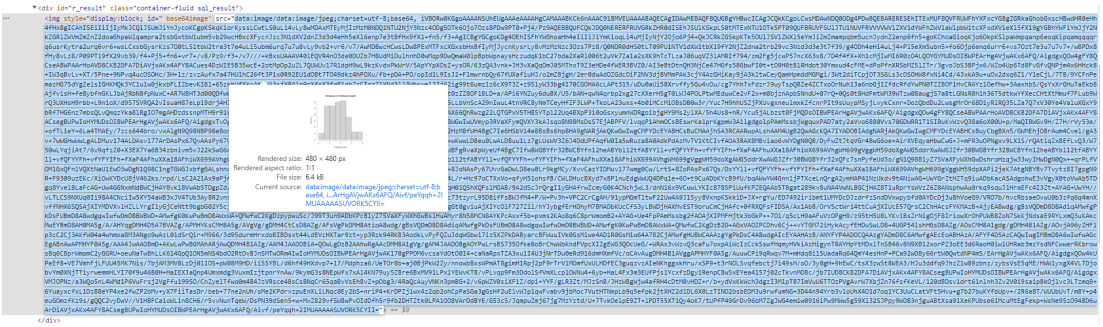
Code Runner API-em. Servis prati kvalitetnu arhitekturu te omogućuje jednostavnu nadogradnju u svrhu izvršavanja proizvoljnog koda novog jezika te manipulaciju rezultata izvršavanja koda. *Edgar* prethodno opisanu JSON strukturu dobivenu od *Code Runnera* koristi kako bi rezultate obavljanja proizvoljnog programskog koda predočio korisniku. Ovisno o programskom kodu predanom na izvršavanje, rezultati mogu uključivati retke tablice predanog upita i ispis na standardni izlaz. U predstavljenoj je nadogradnji naglasak na prikaz rezultata predviđanja koji uključuju slike poput histograma i stabla odluke. *Base64* enkodiran sadržaj datoteke, kakav se prima u poruci od *Code Runnera*, je zadržan zato jer se jednostavno može ugraditi u HTML bez potrebe da se, često privremene, datoteke trajno pohranjuju i poslužuju putem sustava *Edgar*.

```
<img style='display:block; id='base64image'
src='
LzlqLzRBQ... <!-- Base64 format datoteke -->' />
```

Rezultat umetanja Base64 enkodirane slike u oznaku *img* prikazuje Slika 5.4, dok Slika 5.5 prikazuje stvaran izgled *img* oznake koje je prethodno opisana.

Slika 5.4: Slika stvorena izvršavanjem *R* skripte korištenjem sustav *Edgar*

Ovakav se sustav može koristiti i za prikaz PDF ili HTML datoteke generiranih izvršavanjem *R Markdown* datoteke. Time se studentima mogu predočiti svi rezultati predviđanja uz kvalitetan i strukturiran opis same izrade prediktivnih modela kakav omogućuje sam *R Markdown*.



Slika 5.5: *Img* oznaka slike stvorena izvršavanjem *R* skripte korištenjem sustav *Edgar*

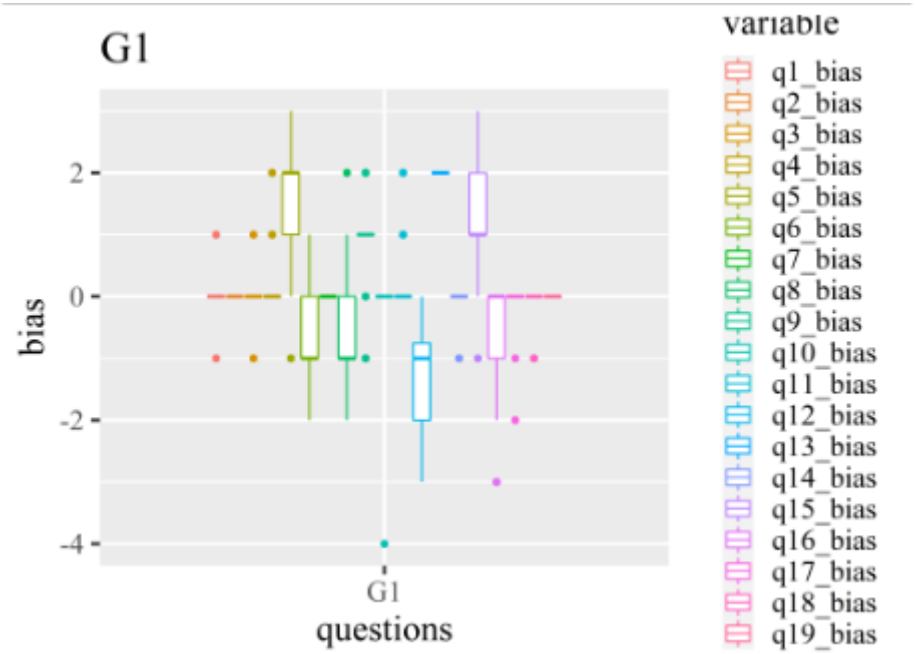
6. Smjernice za budući rad

6.1. Automatizacija dubinske analize podataka i izvršavanja algoritama strojnog učenja

Opisano proširenje cjelokupnog sustava *Edgar* koje podržava izvršavanje proizvoljnog *R* ili *R Markdown* koda može olakšati i ubrzati mnoge procese koji se u dosadašnjem obliku sustava izvršavaju ručno te zahtijevaju uložen trud i vrijeme. Izvršavanje samog algoritma strojnog učenja kome je svrha predvidjeti rezultate prolaznosti studenata ima mogućnost automatizacije. Baza podataka koja je povezana sa sustavom *Edgar* se može proširiti na način da se u njoj spremaju cjelokupne *R* skripte. Pojedina se skripta može izabrati na novoizrađenom korisničkom sučelju. Nakon odabira skripte, može postojati mogućnost odabira pohranjenih SQL upita koji bi izvukli potrebne podatke iz baze podataka i nad njima izvršili algoritme strojnog učenja. Podaci bi se mogli spremiti u CSV datoteku i mogu biti poslani zajedno sa *R* skriptom putem atributa "*additional_files*" prema sustavu *Judge0*. Ovakvim bi se procesom omogućila automatizirana dubinska analiza podataka i izvršavanje algoritama strojnog učenja uz mogućnost odabira određenog algoritma te odabira podataka nad kojima se želi izvršiti algoritam kao i odabira vrijednosti pojedinih varijabli koje se mogu nalaziti u pohranjenim SQL upitima. Rezultati takvih procesa mogu jednostavno biti prikazani studentima ili nastavnom osoblju, ovisno o njihovoj svrsi. Nadogradnja sustava koju ovaj rad obuhvaća je odrađena na način da omogućuje jednostavno dodatno proširenje. Sustav bi se mogao nadograditi složenijim i strukturiranijim procesom dohvaćanja i prijenosa rezultata algoritama ukoliko će buduće proširenje sustava to zahtijevati.

6.2. Automatizacija postojećih podatkovnih analiza sustava *Edgar*

U sustavu *Edgar* postoji mnogo slučajeva u kojima nastavno osoblje ručno provodi proces analize podataka te pokreće vlasite *R* skripte u svrhu različitih izračuna. Jedan je od takvih slučaja provjera pristranosti studenata prilikom rješavanja zadataka u kojima studenti međusobno boduju predana rješenja. Provjera pristranosti uključuje proces dubinske analize podataka te izrade podatkovne analize. Analiza pristranosti osim numeričkih pokozatelja pruža i niz grafičkih prikaza dobivenih *ggplot2* bibliotekom, kao što je *box and whisker* prikazan na slici 6.1.



Slika 6.1: *Box and whisker* dijagram pristranosti studenata

Izrada ovakve analize i dostava dobivenih rezultata se može automatizirati korištenjem proširenja *Edgar* sustava prikazanog u ovom radu.

6.3. Python

Sustav se također može proširit tako da podržava i obavljanje algoritama strojnog učenja u skriptnom jeziku *Python* jer je i on uvelike raširen u području strojnog učenja te također sadrži mnoštvo popratnih biblioteka koje olakšavaju sam proces izrade prediktivnih modela. Ukoliko se sustav proširi na ovaj način, njegova će integracija u

sustav nalikovati integraciji jezika *R* predstavljenoj u ovom radu. Izvršavanje *Python* skripte se može povezati sa proširenjem predloženim u poglavlju 6.1 jer i ovaj jezik podržava jednostavno čitanje podataka iz CSV datoteke uporabom biblioteke *pandas*.

7. Zaključak

Sustav *Edgar* je kroz godine uporabe na FER-u u nekoliko desetaka kolegija prošao kroz mnoge faze nadogradnje te je obogoćen mnogim funkcionalnostima poput podrške različitih tipova pitanja te prikaza bodovnih rezultata studenata u odnosu na ostale studente njegove generacije. Sustav je razvijen do stanja u kojem je moguće odraditi većinu, ako ne i sve nastavne aktivnosti određenih kolegija koji uključuju provjeru programskog znanja. Sustav *Edgar* je korišten od strane nekoliko tisuća studenata te je povezan s bazom podataka u kojoj su zabilježeni podaci o njihovim performansama, odnosno dobivenim bodovima na nastavnim aktivnostima iz nekoliko kolegija. Takvi su podaci, uz dodatan skup podataka koji je stvoren prije nego što su studenti ostvarili upis na FER, iskorišteni u izradi modela strojnog učenja na kolegiju Baze podataka koji, s obzirom na strukturu samog kolegija, omogućuje predikciju studentove prolaznosti s 80% uspješnosti već na polovici semestra, odnosno nakon odrađenog međuispita.

Takav je rezultat potaknuo ideju o izradi prototipa integracije sustava za strojno učenje u sustav *Edgar*. U ovom je radu opisan cjelokupan proces izvođenja programskog koda koji je već postojao u sustavu sa svim njegovim komponentama. Dan je i pregled komikacije između komponenata sa strukturom podataka koji se prenose u porukama. Opisana je nadogradnja sustava s naglaskom na obavljanje algoritama strojnog učenja u programskog jeziku *R* te dostavu dobivenih rezultata predviđanja studentima kako u tekstualnom, tako i grafičkom obliku. Raširena uporaba *R Markdown* datoteke zbog funkcionalnosti koje takav format datoteke pruža potiće nadogradnju procesa koja uključuje i njegovo korištenje zbog uske povezanosti sa skriptnim jezikom *R*. Nadogradnja je sustava u svrhu podrške izvršavanja *R Markdown* datoteke analogna onoj za jezik *R* uz male preinake. Predstavljena nadogradnja je otvorena za promjene koje su jednostavno ostvarive. Također je dan opis korištenih tehnologija te uputa za njihovo korištenje. Konačno su dane smjernice za daljnju nadogradnju zbog šireg skupa uporabe koji uključuju dostavu tekstualnog i grafičkog oblika predviđanja te različite uporabe povezane s izvršavanjem samog *R* koda.

LITERATURA

[1] Docker. *Comparing Containers and Virtual Machines*. URL https://www.docker.com/resources/what-container/.

[2] Docker. Docker overview, 2022. URL https://docs.docker.com/get-started/overview/.

[3] Herman Zvonimir Došilović. *Robust and Scalable Online Code Execution System*, 2020. URL https://ce.judge0.com/#submissions.

[4] Herman Zvonimir Došilović i Igor Mekterović. Robust and scalable online code execution system. U *2020 43rd International Convention on Information, Communication and Electronic Technology (MIPRO)*, stranice 1627–1632, 2020. doi: 10.23919/MIPRO48935.2020.9245310.

[5] Igor Mekterović i Ljiljana Brkić. Setting up automated programming assessment system for higher education database course. *International Journal of Education*, 02:287–294, 2017.

[6] Igor Mekterović. *Edgar*, 2017-present. URL https://gitlab.com/edgar-group/edgar.

Prototip integracije sustava za strojno učenje u sustav Edgar

Sažetak

Edgar je informacijski sustav razvijen na Fakultetu Elektortehnike i Računarstva, čiji je motiv za izradnju podržavanje automatizirane provjere programskih zadataka. U ovom je radu proces izvođenja programskog koda nadograđen za potrebe izvođenja algoritama strojnog učenja nad podacima koji se putem sustava Edgar spremaju u bazu podataka, a u svrhu dostave rezultata predviđanja studentima. Ovakvom se funkcionalnošću želi utjecati na sveukupnu prolaznost studenata jer je ta metrika od iznimne važnosti visokim obrazovnim ustanovama. Napravljena se nadogradnja može iskoristiti i u drugim područjima sustava, a postoji i opcija za automatiziranim procesom izvršavanja algoritama strojnog učenja i prezentacije rezultata.

Ključne riječi: automatizacija,baza podataka,online sudac,Judge0,dubinska analiza podataka, kontejnerizacija, previđanje

Prototype of machine learning system integration in Edgar

Abstract

Edgar is an information system developed at the Faculty of Electrical Engineering and Computing with the purpose of automated assessment of programing tasks. Within this bachelor's thesis, the execution process of programming code has been upgraded in order to perform machine learning algorithms with the data collected by Edgar, with the purpose of presenting such predictions to students. This kind of functionality can impact student's overall pass percentage, which is one the most valuable metrics to higher educational facilities. The upgrade presented in this thesis can be used in different areas of Edgar. There is also an option of developing an automated proces for executing machine learning algorithms and presenting such results.

Keywords: automatization,database,online judge,Judge0,data mining,containerization, prediction