Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Tomi Šebjanič

Implementacija in uporaba pametnega asistenta v izobraževanju

diplomsko delo

univerzitetni študijski program
prve stopnje
računalništvo in informatika

Mentor: doc. dr. Dejan Lavbič

Ljubljana, 2017

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Pametni asistenti so že nekaj časa prisotni na različnih ravneh informacijske podpore poslovanju, številna analitična podjetja pa jim napovedujejo zelo svetlo prihodnost. Eno izmed zanimivih področij za uporabo pametnih asistentov je tudi področje izobraževanja. V okviru diplomskega dela se osredotočite na razvoj prototipa pametnega asistenta, ki obstoječim komunikacijskim kanalom (elektronska pošta, govorilne ure ipd.) doda novega. Informacijska rešitev naj omogoča komunikacijo študentov z inteligentnim pomočnikom, ki bo sposoben odgovarjati na vprašanja, povezana z vsebino predmeta, in pridobitev osebnih podatkov o študentovem napredku pri predmetu (rezultati vmesnih preverjanj znanj, domačih nalog, izpitov ipd.).

Zahvaljujem se svojim bližnjim za vso podporo tekom študija in mentorju, doc. dr. Dejanu Lavbiču, za usmerjanje in nasvete med izdelavo diplomske naloge.

Družini.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
SQL	Structured Query Language	strukturirani poizvedovalni jezik
NoSQL	non SQL	ne SQL
RDBMS	Relational Database Management System	sistem za upravljanje relacijskih podatkovnih baz
IRC	Internet Relay Chat	internetni klepet
MVC	Model-View-Controller	model-pogled-nadzornik
A.L.I.C.E.	Artificial Linguistic Internet Computer Entity	umetna lingvistična internetna računalniška entiteta
AIML	Artificial Intelligence Mark up Language	označevalni jezik umetne inteligence
XML	Extensible Markup Language	razširljiv označevalni jezik
CSV	Comma-separated values	z vejico ločene vrednosti
API	Application Programming Interface	programski vmesnik
OIS	Introduction to Information Systems	Osnove informacijskih sistemov
VPS	Virtual Private Server	navidezni zasebni strežnik
OCR	Optical Character Recognition	optično prepoznavanje znakov

Povzetek

Naslov: Implementacija in uporaba pametnega asistenta v izobraževanju

Avtor: Tomi Šebjanič

Uporaba umetne inteligence v zadnjem času strmo narašča. Pametni asistenti (angl. *chatbots*) so prisotni že nekaj časa, vendar pa je v zadnjem času Facebook povzročil val uporabe in implementacije le-teh. Tako se vse več podjetij odloča za razvoj lastnega pametnega asistenta, ki uporabnikom pomaga pri enostavnih vprašanjih, ki bi jih lahko sicer našli tudi sami.

V diplomskem delu smo zasnovali in implementirali enostavnega asistenta, ki odgovarja na vprašanja, povezana s snovjo pri predmetu (v dotičnem primeru je to predmet Osnove informacijskih sistemov). V prvem delu smo analizirali obstoječe rešitve in uporabo pametnih asistentov v sklopu poučevanja ter predstavili uporabljene tehnologije in orodja. V osrednjem delu smo predstavili domeno, natančneje vrsto in strukturo podatkov, ter opisali in definirali uporabljene algoritme. Za tem smo opisali postopek razvoja in na koncu predstavili statistično analizo zajetih podatkov ter podali oceno rešitve in predloge za izboljšave.

Rezultati testiranja so pokazali, da je bila uporaba pametnega asistenta nizka, kar bi lahko bila posledica tega, da se je testiranje začelo en dan pred koncem obdobja pedagoškega dela. Večina uporabnikov je pametnega asistenta uporabljala le tisti dan, ko so se pridružili ekipi. Največkrat so ga uporabljali za iskanje poglavja, pridobivanje podatkov o uporabniku in razlago kratic.

Ključne besede: pametni asistent, Node.js, Docker, Neo4j, obdelava besedil

Abstract

Title: Development and use of a smart assistant in education

Author: Tomi Šebjanič

There has been a rapid growth in the use of artificial intelligence lately. Although smart assistants, also known as chatbots, have been present for quite some time, a massive implementation of chatbots has been brought by Facebook actually. As a result, more and more companies are developing their own smart assistant that would help users solve simple questions that could also be easily solved by the users themselves.

In the thesis we describe a design and implementation of a simple assistant that provides answers to questions related to the subject content (in this case the subject is the Introduction to Information Systems). In the first part we cross-check the solutions available at the time and analyze the employment of smart assistants in the context of teaching. We present the technologies used as well as the tools. The central part of the thesis presents the domain, more specifically its type and the data structure, and describes as well as defines the algorithms used. Further on we describe the development process and present a statistical analysis of the gathered data. We evaluate the solution and provide suggestions for its improvement.

The test results show that the use of a smart assistant is low, which could be a consequence of the testing being carried out just one day before the end of pedagogical work. Most users were only using it only the very day they joined the team. Most often it was used for chapters search, acquiring information about the user and the explanation of abbreviations.

Keywords: smart assistant, chatbot, Node.js, Docker, Neo4j, text processing

Poglavje 1

Uvod

Pametni asistenti so na trgu prisotni že nekaj let. Eden izmed prvih slovenskih pametnih asistentov (oz. asistentk) je bila "Vida"(virtualna davčna asistentka), ki je bila razvita za potrebe tedanje Davčne uprave Republike Slovenije. Vida je odgovarjala na splošna in najpogostejša vprašanja s področja dohodnine in pomagala pri iskanju pojasnil ter obrazcev. [21]

V današnjem času je trend razvoja pametnih asistentov naglo narasel. Tako je januarja 2017[1] na trg prišel pametni asistent *Troljo*, ki ga je razvilo podjetje Wawe group.[2] Ta asistent nam pove prihode avtobusov Ljubljanskega potniškega prometa na določeno postajo in razpoložljivost javnih koles na postajah. [17]

Leta 2016 je britansko podjetje Ubisend[3] izvedlo anketo o mobilni komunikaciji s podjetji. Anketiranci so bili potrošniki in lastniki podjetij, starejši od 15 let, obeh spolov. Želeli so izvedeti preference in način komunikacije s podjetji. Večina anketirancev je bila stara med 25 in 50 let, torej tisti z največjo kupno močjo. Več kot 50 % anketirancev se je strinjalo s trditvijo, da bi morala podjetja 24 ur na dan, 7 dni v tednu biti na razpolago za kakršnakoli vprašanja, odgovori pa bi morali biti takojšnji. Na vprašanje

[1] `https://www.facebook.com/troljo/posts/1768684540119377`; dostopano: 22. 5. 2017

[2] `https://wave.si`

[3] `https://www.ubisend.com`

„Ob predpostavki, da sta obe opciji (telefonski klic in komunikacijske aplikacije) zastonj, kako bi raje komunicirali s podjetjem“ je 49,4 % anketirancev odgovorilo, da bi raje uporabili aplikacijo. To nakazuje, da bi tudi uporabniki raje komunicirali s pomočjo aplikacije, kar pomeni, da je razvoj in uporaba pametnih asistentov dobra naložba v prid večje prodaje. [20]

Iz rezultatov ankete lahko razberemo, da bodo moč in uporabnost pametnih asistentov začela izkoriščati mnoga podjetja. Uporabnost pametnih asistentov je že začelo izkoriščati nizozemsko letalsko podjetje KLM.[4] Podjetje KLM je marca 2016 izdalo pametnega asistenta za platformo Facebook Messenger. Pametni asistent omogoča potnikom, ki so rezervirali let in izbrali pametnega asistenta za način obveščanja, prejemanje informacij o rezervaciji, informacije o letu, informacije o začetku *check-in* procesa, prejemanje e-vozovnice in ostale informacije. [8] S tem so deloma razbremenili zaposlene v klicnem centru v dotičnem podjetju.

V procesu izobraževanja pogosto naletimo na problem velikega števila študentov in omejenega števila izvajalcev predmeta. Izvajalci so pogosto preobremenjeni in ne morejo zagotoviti zadostne pozornosti vsem študentom. Na tem področju lahko najdemo nekaj uspešnih poskusov implementacije in uporabe pametnih asistentov. Pojavili so se primeri asistentov pri poučevanju na daljavo [6], pri pomoči pri učenju tujih jezikov [5] in pri splošni pomoči ob izobraževanju [7].

Rezultat diplomske naloge bo deloma razbremenil izvajalce in jim omogočal, da bodo ustvarili klepetalnico, v kateri bodo študentje najprej povprašali pametnega asistenta in jih šele ob neuspešni komunikaciji s pametnim asistentom kontaktirali.

V prvem delu bomo analizirali obstoječe rešitve uporabe pametnih asistentov v sklopu poučevanja ter predstavili uporabljene tehnologije in orodja. V osrednjem delu bomo predstavili domeno, natančneje vrsto in strukturo podatkov, ter opisali in definirali uporabljene algoritme. Za tem bomo opisali postopek razvoja in na koncu predstavili statistično analizo zajetih po-

[4] `https://www.klm.com/home/nl/en`

datkov ter podali oceno rešitve in predloge za izboljšave. Celoten projekt je odprtokoden in dostopen na repozitoriju Bitbucket na spletnem naslovu `https://bitbucket.org/tomisebjanic/ois-chatbot`.

Poglavje 2

Pametni asistenti

2.1 Definicija

Na spletu in v strokovni literaturi najdemo kar nekaj definicij, kaj naj bi bil pametni asistent (angl. *chatbot*) in kakšne naj bi bile njegove zmožnosti. Shawar in Atwell v članku [15] podata definicijo, da so pametni asistenti računalniški programi, ki komunicirajo z uporabniki s pomočjo naravnega jezika. Avtorja sta v članku raziskovala področja, na katerih bi lahko bili pametni asistenti uporabni. Tudi Heller in drugi so v članku [6] definirali pametnega asistenta kot agenta, ki je sprogramiran tako, da posnema človeški pogovor, tj. pogovor v naravnem jeziku. Prosta enciklopedija Wikipedija [24] nam ponuja podobno, vendar bolj razširjeno definicijo pametnega asistenta: računalniški program za pogovor s pomočjo zvoka ali besedil. Taki programi so načrtovani tako, da se obnašajo kot človeški sogovornik in zaradi tega uspešno prestanejo Turingov test.

2.2 Zgodovina

Začetek razvoja pametnih asistentov sega v 60. leta 20. stoletja. Takratni namen pametnih asistentov je bil preveriti, ali lahko takšni sistemi pretentajo uporabnike do te mere, da mislijo, da so pravi ljudje. Vendar pa se

je namembnost takšnih sistemov skozi zgodovino spremenila od posnemanja človeških pogovorov do uporabnosti v izobraževanju, iskanju informacij in podpori poslovanju v podjetjih. [15]

2.2.1 Turingov test

Alan Turing se je že leta 1950 v svojem delu [18] spraševal, ali so stroji zmožni razmišljanja. Ta problem je opisal kot igro imitacije (angl. *the imitation game*). Pravila igre so sledeča: igro igrajo trije ljudje, moški (A), ženska (B) in izpraševalec (C), ki je lahko kateregakoli spola. Izpraševalec je v sobi, ločen od moškega in ženske. Cilj igre za izpraševalca je, da skuša ugotoviti spol ostalih dveh udeležencev igre. Izpraševalec pozna ostala udeleženca igre po oznakah X in Y in na koncu igre pove: X je A in Y je B oz. X je B in Y je A. Izpraševalec lahko ostalima udeležencema postavlja vprašanja, kot je npr.: „Ali mi lahko X pove dolžino svojih las?“ Na podlagi te igre lahko zastavimo vprašanje: „Kaj bi se zgodilo, če bi vlogo osebe A prevzel stroj? Ali se bo izpraševalec zmotil v enakem številu primerov, kot če bi igro igrala moški in ženska?“

Turingov test bi lahko uporabljali kot eno izmed metrik, kako uspešen je pametni asistent v komunikaciji z uporabniki (ali so odgovori inovativni in pravilni, ali se zna pametni asistent popravljati, kako uspešen je v procesu prepričevanja in pogajanja z uporabnikom itd.).

2.2.2 ELIZA

Raziskovalci iz laboratorija za umetno inteligenco univerze MIT so leta 1966 izdali pametnega asistenta z imenom ELIZA [23], ki se smatra kot eden izmed prvih na tem področju. ELIZA je bila ustvarjena z namenom, pokazati, da je komunikacija med človekom in računalnikom mogoča tudi v naravnem jeziku.

Ta pametni asistent je deloval tako, da je analiziral vhodne povedi in kreiral odgovor na podlagi pravil, povezanih z dekompozicijo vhoda. Pomanjkljivost pametnega asistenta je bila ta, da ni hranil pogovora v spominu, tako

da uporabnik ni mogel stopiti v sodelovanje in pogajanje z asistentom.

2.2.3 A.L.I.C.E.

A.L.I.C.E. je pametni asistent, njegova kratica pa pomeni *Artificial Linguistic Internet Computer Entity*. Razvil ga je dr. Richard S. Wallace iz podjetja A.L.I.C.E. Artificial Intelligence Foundation, Inc. leta 1995. [22] Avtor je dobil navdih za njegovo izdelavo od Alana Turinga in njegovega članka z naslovom *Computing Machinery and Intelligence*. Nekateri znanstveniki gledajo na A.L.I.C.E. kot nadgradnjo pametnega asistenta ELIZA. Avtor pravi, da ta trditev ni tako zelo napačna, vendar navaja, da ima A.L.I.C.E. več kot 40.000 kategorij znanja, medtem ko jih ima ELIZA samo okrog 200. Baza znanja je shranjena v AIML datotekah, ki je izpeljanka XML.

2.3 Uporaba v praksi

Pametni asistenti so lahko v praksi zelo uporabni na velikem številu področij: strankam podjetij lahko pametni asistenti izboljšajo uporabniško izkušnjo na ta način, da lahko v kateremkoli trenutku dobijo (večino) želenih informacij, kot so npr.:

- delovni čas;
- čas dobave artiklov;
- splošne informacije o podjetju itd.

Zelo uporabni so lahko tudi v procesu izobraževanja. Njihova uporabnost se lahko pokaže v naslednjih primerih:

- iskanje splošnih informacij o fakulteti;
- iskanje splošnih informacij o posameznem predmetu;
- iskanje podatkov v bazah znanja o določeni temi;

- prikazovanje podatkov o ocenah posameznega študenta;
- interaktivno učenje itd.

Kot lahko vidimo, je uporabnost pametnih asistentov skoraj neskončna. Z uporabo le-teh bi lahko učence bolj motivirali za delo, saj bi lahko stalno imeli nekakšen stik z vsebino predmeta in pomoč pri učenju. Z uporabo pametnih asistentov v procesu izobraževanja se je ukvarjalo že nekaj znanstvenikov.

V članku [7] so se ukvarjali z obravnavo razvoja in zmožnosti pametnih asistentov in inteligentnih tutorjev. Ugotovili so, da se lahko, ob pravilni integraciji in implementaciji, razširi v šole, na univerze in v ostale izobraževalne ustanove.

Fryer in Carpenter [5] sta se ukvarjala z uporabo pametnih asistentov pri učenju tujih jezikov. Avtorja sta ugotovila, da imajo učenci tujih jezikov zelo malo možnosti za govor v jeziku, ki se ga učijo, in prav tu bi prišli v pomoč pametni asistenti. V članku [15] so prišli do zaključka, da pametni asistenti ne bodo v celoti nadomestili učiteljev, so jim pa vsekakor v pomoč.

Argentinski znanstveniki so predlagali uporabo pametnih asistentov na nekoliko drugačen način, in sicer so hoteli z uporabo pametnih asistentov v šolstvu (srednjih šolah) dvigniti raven zanimanja za računalništvo, natančneje zanimanje za programiranje. Država se sooča z nizkim številom diplomiranih inženirjev računalništva (3500 na leto) v primerjavi z diplomiranimi pravniki (10.000 na leto) in diplomiranimi ekonomisti (15.000 na leto). Tako so Benotti, Martínez in Schapachnik [1] razvili pametnega asistenta, ki so ga poimenovali *Chatbot*, s katerim poskušajo dvigniti raven zanimanja učencev za računalništvo. Naloga *Chatbot*-a je interaktivno učenje programiranja s pomočjo enostavnih programerskih nalog s področja spremenljivk, pogojnih stavkov, končnih avtomatov itd. Rezultati testiranj so pokazali pozitivne rezultate za nadaljnje raziskovanje, kako pritegniti večje število učencev za študij računalništva.

Poglavje 3

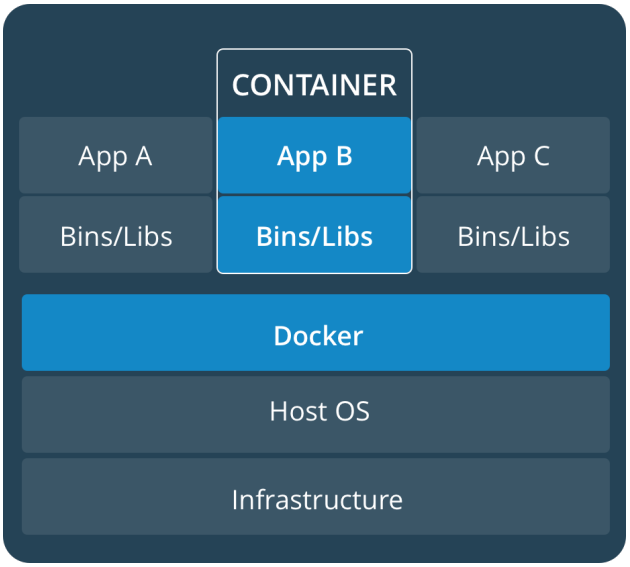
Pregled uporabljenih orodij, tehnologij in knjižnic

V tem poglavju bomo navedli in na kratko opisali vsa orodja, tehnologije in knjižnice, ki smo jih uporabili pri implementaciji pametnega asistenta. Pri implementaciji smo poleg navedenih knjižnic uporabljali tudi druge, ki jih bomo opisali v poglavju 5.

3.1 Docker

Docker je odprtokodni stroj, ki avtomatizira postavljanje (angl. *deployment*) aplikacij znotraj vsebnikov (angl. *containers*), ki so med seboj izolirani. Skica koncepta je prikazana na sliki 3.1. Omogoča lahko in hitro razvojno okolje z namenom krajšanja ciklov med pisanjem kode, testiranjem in prenašanjem aplikacije v produkcijo. Glavni princip Dockerja je, da naj bi vsak vsebnik izvajal samo eno aplikacijo oz. proces, kar pomeni, da spodbuja storitveno usmerjeno arhitekturo aplikacije oz. princip mikrostoritev. [19] Docker prav tako rešuje pogost programerski problem „delalo je na mojem računalniku“ ob sodelovanju več programerjev pri razvoju neke aplikacije. [3]

[1] `https://www.docker.com/what-container`; dostopano: 13. 7. 2017



Slika 3.1: Struktura vsebnikov v Dockerju.[1]

3.2 Node.js

Node.js je strežniško izvajalno okolje, napisano v programskem jeziku JavaScript, ki je dogodkovno usmerjen (angl. *event driven*), kar pomeni, da se aplikacija izvaja samo, kadar prejme zahtevke za določene dogodke, ki jih mora sprocesirati. Njegova arhitektura mu omogoča gradnjo skalabilnih spletnih aplikacij, ki lahko obvladajo več sočasno aktivnih povezav. [12]

3.3 Neo4j

Neo4j je podatkovna baza novejših generacij, bolj poznana pod kratico NoSQL. Je podatkovna baza v obliki grafa, načrtovana prav za ta namen (shranjevanje in procesiranje grafov). Vsak podatek v Neo4j podatkovni bazi ima eksplicitno določene povezave z vsemi povezanimi entitetami, kar omogoča hitrejše in učinkovitejše iskanje po takšni podatkovni strukturi, saj ob iskanju ne pregledujemo celotnega grafa, ampak samo del. [11]

3.4 MySQL

MySQL je odprtokodni RDBMS podjetja Oracle Corporation. Namenjen je shranjevanju podatkov v obliki vrstic v tabelah in uporablja SQL poizvedovalni jezik. [10]

3.5 Slack

Slack je oblačna spletna storitev, ki je sestavljena iz orodij za ekipno sodelovanje in storitev. Ponuja veliko funkcionalnosti, ki jih je imel včasih IRC, kot so npr.:

- pogovorne sobe (angl. *channels*), organizirane po tematikah;
- zasebne pogovorne sobe;
- zasebna sporočila med uporabniki.

Dodali so tudi integracije mnogih spletnih storitev in vtičnikov ter pametnih asistentov, ki povečajo storilnost skupine ljudi, npr. obveščanje o novih hroščih v aplikacijah, dodajanje novih datotek v spletno oblačno shrambo Google Drive, obveščanje o spremembah na Trello deskah itd. [16]

3.6 Botkit

Botkit je programsko ogrodje za hitro, enostavno razvijanje pametnih asistentov. Ogrodje podpira najpopularnejše platforme za komuniciranje, kot so npr. Slack,[2] Cisco Spark,[3] Facebook Messenger,[4] Twilio[5] in Microsoft Bot Framework.[6] Osnova ogrodja je Express.js,[7] ki je MVC ogrodje za Node.js. [2]

[2] `https://slack.com`
[3] `https://www.ciscospark.com`
[4] `https://www.messenger.com`
[5] `https://www.twilio.com`
[6] `https://dev.botframework.com`
[7] `https://expressjs.com`

Poglavje 4

Konceptualna zasnova pametnega asistenta

4.1 Vhodni podatki

Pametni asistent bo podajal odgovore z dveh področij:

- odgovarjal bo na vprašanja, povezana s študentovimi podatki (osnovni podatki, dosežene točke pri posameznih nalogah in na izpitu ter razne informacije glede uspešnosti pri predmetu);
- odgovarjal bo na vprašanja, povezana s snovjo pri predmetu.

Za to bomo imeli na voljo dva vira podatkov.

4.1.1 Podatki o študentih

Podatki o študentih bodo izvoženi v formatu CSV iz predmeta Osnove informacijskih sistemov, ki so shranjeni v spletni učilnici Fakultete za računalništvo in informatiko[1] (platforma Moodle[2]).

Na voljo imamo osnovne podatke o študentih (vpisna številka, e-mail naslov, dodeljeno skupino vaj itd.), podatke o posameznem preverjanju znanja

[1] `https://ucilnica.fri.uni-lj.si`
[2] `https://moodle.org`

(dosežene točke in odbitek), podatke o posamezni domači nalogi (dosežene točke, odbitek, točke medsebojnega ocenjevanja), podatke o doseženih točkah na predavanjih, število predlaganih popravkov in nasvetov na forumu, dodatne točke sodelovanja.

Pametni asistent bo do teh podatkov dostopal preko Googlovih preglednic[3] s pomočjo API dostopa, kar nam omogoča varovanje osebnih podatkov, saj potrebujemo za dostop račun s povabilom ter OAuth 2 žeton, ki se mora zgenerirati ob vsakem novem dostopu.

4.1.2 Gradivo za predmet Osnove informacijskih sistemov

Ključni del podatkov, ki jih bo uporabljal pametni asistent, bo izhajal iz gradiva pri predmetu Osnove informacijskih sistemov. Ti podatki bodo v HTML datotekah, ki so v obliki e-knjige v formatu GitBook.[4] Vsako poglavje je samostojna HTML datoteka, ki je po osnovni strukturi enaka vsem ostalim.

Vsako poglavje lahko sestavljajo:

- tekst;
- povzetek;
- podpoglavje;
- tabela;
- slika;
- ID poglavja;
- številka poglavja;
- logična povezava s predavanji in vajami;

[3] `https://www.google.com/sheets/about`

[4] `https://www.gitbook.com`

- dodatno branje in literatura.

Takšna oblika nam bo pri implementaciji omogočala izdelavo univerzalnega razčlenjevalnika za HTML datoteke, namesto da bi morali izdelati razčlenjevalnike za vsako datoteko (poglavje) posebej.

4.2 Uporabljeni algoritmi in postopki obdelave podatkov

4.2.1 Term frequency–inverse document frequency

Term frequency–inverse document frequency (*tf-idf*) je v procesu iskanja informacij v danem korpusu (*information retrieval*) numerična statistika, ki nam pove, kako pomembna je beseda v izbranem dokumentu znotraj korpusa. [13]

Ta statistika se pogosto uporablja kot utež v procesu iskanja informacij, rudarjenju po besedilih (*text mining*) in modeliranju uporabnikov (*user modeling*). Vrednost (*tf-idf*) narašča proporcionalno s številom pojavitev določene besede v dokumentu, ampak jo po navadi zniža frekvenca te besede znotraj korpusa, kar pomaga pri bolj realni uravnavi vrednosti. Tako lahko s to statistiko uspešno vrednotimo posamezne besede, saj imajo besede z višjo frekvenco pojavitev znotraj korpusa manjšo vrednost (*tf-idf*) in obratno.

Tf-idf je produkt dveh statistik, in sicer *term frequency* (*tf*) ter *inverse document frequency* (*idf*). *Tf* bazira na tem, da konstrukti besedila (besede, besedne zveze), ki se v besedilu pojavljajo pogosteje, imajo neko zvezo z vsebino besedila. *Tf* lahko zapišemo kot število pojavitev konstrukta t v dokumentu d:

$$tf_{t,d}$$

Poleg te enostavne enačbe obstajajo tudi druge možnosti za izračun *tf*:

- Booleanova frekvenca;
- frekvenca, normirana z dolžino dokumenta;

- logaritmična vrednost frekvence;
- povišana (*augmented*) frekvenca.

Pri implementaciji bomo uporabili frekvenco, normirano z dolžino dokumenta:

$$tf_{t,d} = \frac{tf_{t,d}}{|d|}$$

Idf meri, koliko informacije nam da beseda, tj. ali je beseda pogosta ali redka v celotnem korpusu. Je logaritmično normirana statistika z razmerjem med številom vseh dokumentov N in številom dokumentov, ki vsebujejo konstrukt t:

$$idf_t = \log \frac{N}{df_t}$$

To pomeni, da je *idf* vrednost konstrukta, ki je manj pogost, višji kot *idf* konstrukta, ki je bolj pogost v korpusu.

Na koncu združimo obe vrednosti, *tf* in *idf*, da dobimo pravilno in relevantno utež za vrednotenje teže posamezne besede v dokumentu:

$$tf\text{-}idf_{t,d} = tf_{t,d} \cdot idf_t$$

[9]

Opisano statistiko bomo v pametnem asistentu uporabljali za iskanje najbolj relevantnega poglavja, ki ga bodo uporabniki iskali. Določanje najbolj relevantnega poglavja bo potekalo s pomočjo iskanja po ključnih besedah.

4.2.2 Razdalja Jaro-Winkler

Avtor algoritma William E. Winkler je razvil različico Jaro razdalje, ki jo je leta 1989 razvil Matthew A. Jaro. Jaro razdalja meri podobnost med dvema nizoma znakov na intervalu $[0, 1]$: višja je vrednost, bolj sta si niza podobna. Podobnost med nizoma se določi na podlagi števila vstavljanj, izbrisov in transpozicij znakov v nizu, da postaneta niza enaka. Osnovni Jaro algoritem je sestavljen iz treh korakov:

1. izračun dolžine nizov;

2. iskanje skupnih znakov iz obeh nizov;

3. določitev števila transpozicij.

Jaro razdalja d_j dveh nizov s_1 in s_2 je:

$$d_j = \begin{cases} 0 & \text{če } m = 0 \\ \frac{1}{3}\left(\frac{m}{|s_1|} + \frac{m}{|s_2|} + \frac{m-t}{m}\right) & \text{sicer} \end{cases}$$

pri čemer:

- $|s_i|$ je dolžina niza;
- m je število enakih znakov;
- t je polovica števila vseh transpozicij.

Znaka iz posameznega niza sta skupna, če sta enaka in njuna razdalja ni večja od:

$$\left\lfloor \frac{\max(|s_1|, |s_2|)}{2} \right\rfloor - 1$$

Definicija transpozicije znaka je, da se znaka ujemata, vendar nista na isti poziciji. Polovica števila takšnih primerov se šteje kot število transpozicij.

Dopolnitev algoritma upošteva predpone besed, kar nam da boljše rezultate v prid nizom z istim začetkom. Če imamo podana dva niza s_1 in s_2, je njuna Jaro-Winkler razdalja:

$$d_w = d_j + (\ell p(1 - d_j)),$$

pri čemer:

- d_j je Jaro razdalja med nizoma;
- l je dolžina skupnega ujemanja od začetka do največ 4 znakov;
- p je faktor, za koliko je končna vrednost bolj nagnjena v prid nizom z enakim začetkom, ki ne sme presegati $0,25$. Avtor dopolnitve v svojem delu predlaga vrednost $0,1$.

[27, 25]

Algoritem Jaro-Winkler nam bo pomagal pri boljšem razumevanju vprašanj, na katera bo moral pametni asistent odgovoriti. Podane bomo imeli vnaprej definirane ukaze, ki jih bo pametni asistent „razumel“ in na njih znal nedvoumno odgovoriti. Ti ukazi bodo predstavljali vhod v algoritem skupaj z uporabniškim vnosom in tako bomo s pomočjo algoritma skušali najti ujemanje z vnaprej definiranimi ukazi. To bo prišlo še posebej do izraza, v kolikor se bo uporabnik zmotil pri pisanju vprašanja. Primer prepoznanega, napačno vtipkanega vnosa bi lahko bil „Kaj pomeni **kratca** crp?“, kar bomo s pomočjo algoritma prepoznali kot „Kaj pomeni **kratica** crp?“

Da bomo nekatere vnose še vedno prepoznali kot veljavne, bomo morali hevristično določiti najnižjo razdaljo oz. podobnost med definiranim ukazom ter vnosom. Prenizko postavljena meja bi pomenila ujemanje z nerelevantnim ukazom, previsoka meja pa bi pomenila, da nam algoritem ne bi nič koristil.

4.2.3 Lematizacija

Avtorja članka sta v [4] lematizacijo definirala kot „postopek, pri katerem neki besedni obliki v besedilu tvorimo lemo (geslo, iztočnico).“ Za slovenski jezik sta ugotovila, da lahko besede lematiziramo razmeroma natančno z uporabo oblikoslovnega leksikona, za besede, ki niso zajete v leksikonu, pa je lahko lematizacija le hevristična.

Lematizacija predstavlja pomemben korak v predprocesiranju v mnogih primerih procesiranja naravnega jezika in ostalih področjih lingvistike. Podoben postopek predprocesiranja je tudi krnjenje (*stemming*), ki je lahko tudi učinkovit postopek v drugih jezikih (npr. angleščini), v slovenščini ni priporočen, saj pogosto dobimo prekratke krne, ki se lahko zlijejo z besedami drugih pomenov in tako otežujejo nalogo postopkom, ki mu sledijo. [26]

Postopek lematizacije si lahko pogledamo na primeru besede „voziti“. Besede *vozimo*, *voziš*, *vozita*, *vozi* imajo skupno lemo *voziti*. V primeru analize besedila (če ne bi uporabljali postopka lematizacije) bi algoritem obravnaval vsako izmed besed posebej, s postopkom lematizacije pa bi obravnavali vse

besede kot eno. Uporaba lematizacije besed bo prišla do izraza predvsem pri določanju ključnih besed v dokumentih in njihovo vrednotenje s pomočjo *tf-idf* statistike.

4.2.4 Odstranjevanje *stop* besed

Tako kot lematizacija je tudi odstranjevanje stop besed (angl. *stop words*) pomemben korak v predprocesiranju besedil za uporabo v procesiranju naravnega jezika. Stop besede označujejo besede, ki same po sebi nimajo nekega specifičnega pomena, če se pojavijo same zase. Primeri takšnih besed v slovenskem jeziku so: *a, biti, dobro, in, ki, kako, na, nekdo* itd. Po navadi so v besedilu zelo pogoste in nam ne dajo dodane vrednosti na samo razumevanje, odstranjevanje le-teh pa pripomore k boljši učinkovitosti procesiranja ter manjšemu številu operacij s podatkovno bazo. Naravni jezik nima nekega pravila, ki bi nam povedalo, katera beseda je stop beseda in katera ni, zato si pri procesiranju naravnega jezika ustvarimo svojo zbirko stop besed.

4.2.5 Poizvedovanje po *graph-based* podatkih

Za hranjenje gradiva in podatkov o študentih bomo uporabljali Neo4j podatkovno bazo, ki je podatkovna baza v obliki grafa. Podatkovni model bo sestavljen iz vozlišč in oznak ter pripadajočih lastnosti in povezav med vozlišči.

Za poizvedovanje po nerelacijskih podatkovnih bazah, shranjenih v obliki grafa, uporabljamo posebno različico poizvedovalnega jezika, imenovanega *Cypher*. [14] S Cypher poizvedbami grafično opišemo strukturo vozlišč, povezav in relacij med njimi ter operacijo nad izbranimi strukturami, kar nam omogoča, da podatkovni bazi povemo, kaj želimo narediti, in ne na kakšen način (postopek) naj bo narejeno. Poizvedbe izgledajo podobno, kot če bi ročno narisali del grafa, ki nas zanima, ta ročno narisan graf pa bi potem prenesli v ASCII obliko.

Najenostavnejši primer Cypher poizvedbe lahko pokažemo na primeru

socialnih omrežij. Predpostavimo, da imamo dve osebi a in b ter množico skupnih prijateljev x. Želimo izvedeti, katere osebe so skupni prijatelji osebe a in osebe b. S pomočjo Cypher poizvedovalnega jezika lahko zapišemo enostavno poizvedbo:

```
MATCH (a:Person)-[:KNOWS]->(x:Person)<-[:KNOWS]-(b:Person)
RETURN x
```

Poglavje 5

Podrobnosti implementacije

V tem poglavju bomo opisali ključne dele pametnega asistenta oz. njihovo implementacijo. Pri razvoju smo se najbolj opirali na knjižnico Lodash[1] (v kodi se pojavlja kot _), ker nam omogoča lažje delo s podatkovnimi strukturami, kot jih ponuja Javascript. Lodash smo uporabili še iz dveh drugih razlogov. Prvi izmed njiju je ta, da knjižnica ponuja veliko število enostavnih (in tudi bolj kompleksnih) funkcij, kot so npr. testiranje za določen tip (`_.isNumber(value)`), iteriranje po zbirkah (`_.forEach(collection, [iteratee=_.identity])`) in mnoge druge. Drugi razlog za uporabo te knjižnice izhaja iz prejšnjega, saj postane z uporabo Lodash koda zelo pregledna in enostavno berljiva.

5.1 Razvojno in produkcijsko okolje

Razvojno in produkcijsko okolje si bomo postavili s pomočjo Dockerja. Docker nam omogoča postavitev in konfiguracijo okolja s prednostjo, da bo kjerkoli vse delalo enako kot pri razvijalcu, ki okolje pripravlja, kar pomeni, da ni potrebe po tem, da bi morali enak postopek priprave ponavljati, ko želimo prenesti aplikacijo v produkcijsko okolje.

Aplikacija se bo izvajala znotraj štirih ločenih Docker vsebnikov (angl.

[1] `https://lodash.com`

Docker container), pri čemer bo vsak vsebnik zadolžen za eno nalogo, in sicer: Node.js vsebnik, v katerem se bo izvajal pametni asistent; Neo4j vsebnik, v katerem se bo izvajala nerelacijska podatkovna baza; MySQL vsebnik, v katerem se bo izvajala relacijska podatkovna baza; in phpMyAdmin vsebnik, v katerem se bo izvajala aplikacija, s pomočjo katere bomo dostopali do MySQL podatkovne baze. Diagram vsebnikov aplikacije je prikazan na sliki 5.1. Za komunikacijo med posameznimi vsebniki bomo pripravili `docker-compose.yml` datoteko, v kateri bomo povedali, kateri vsebniki in preko katerih vrat (angl. *port*) naj med seboj komunicirajo. Za vse vsebnike bomo uporabljali standardne različice slik iz centralnega Docker repozitorija slik, razen za vsebnik, ki bo vseboval ključno logiko pametnega asistenta, tj. Node.js vsebnik. Za Node.js vsebnik bomo pripravili samostojno Dockerfile datoteko, v kateri bomo:

- naložili orodja za gradnjo in prevajanje aplikacij (angl. *build tools*);
- pognali ukaz `npm install`;
- zgradili aplikacijo za lematiziranje besed.

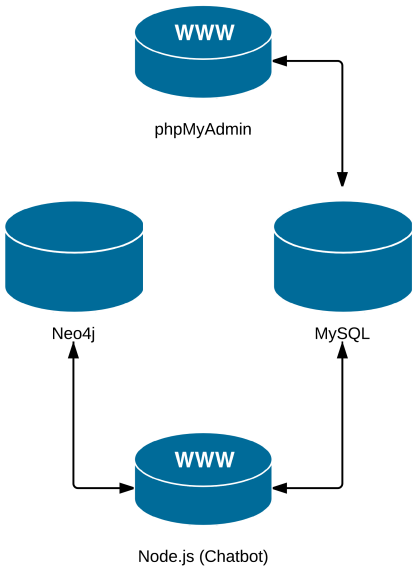
Node.js vsebnik bo za vstopno točko uporabljal skripto `wait-for-dbs-.sh` in ukaz `node slackbot.js`. Namen omenjene skripte je, da se ukaz `node slackbot.js` začne izvajati šele takrat, ko se oba vsebnika, Neo4j in MySQL, uspešno inicializirata in zaženeta.

Produkcijsko okolje bo gostil navidezni zasebni strežnik (*VPS*), ki bo ustvarjen pri ponudniku storitev Digital Ocean.[2] Za tega ponudnika smo se odločili, ker omogoča enostavno kreiranje instance, ki ima že prednaloženo vso programsko opremo, da lahko takoj začnemo gostiti Docker aplikacije.

5.2 Procesiranje teksta

Zaradi potreb po procesiranju razčlenjenega besedila bomo razvili lasten paket funkcij, ki nam bodo olajšale delo. Paket se bo imenoval `textProcessor`

[2] `https://www.digitalocean.com`



Slika 5.1: Docker vsebniki in komunikacija med njimi.

in bo vseboval funkcije, ki bodo:

- odstranjevale odvečne presledke, ločila in simbole ter številke;
- odstranjevale stop besede;
- iskale besede;
- lematizirale besede;
- določale ključne besede;
- iskale razliko med posameznimi povedmi.

Vsaka izmed funkcij prejme kot argument besedilo in vrne spremenjeno oz. sprocesirano besedilo.

Funkcije za odstranjevanje različnih delov besedil niso nič drugega kot regularni izrazi. Regularni izraz `/[\n\s]+/g` išče enega ali več znakov za novo vrstico in/ali presledek, medtem ko regularna izraza `/[→*.,?:;\-\/|]/g` in `/[0123456789]/g` iščeta simbole in številke ter jih zamenjata s presledkom (’

'). Odstranjevanje stop besed bo potekalo na podoben način, pri čemer se bomo sprehodili po seznamu stop besed in v zanki gradili regularne izraze in zamenjali posamezno stop besedo s presledkom.

```
var removeStopWords = function (text) {
    _.forEach(sloveneStopWords, function (stopWord) {
        var regex = new RegExp(
            '(\\s|^)(?:' + stopWord + ')(?=\\s|$)', 'g'
        );
        text = _.replace(text, regex, ' ');
    });

    _.forEach(englishStopWords, function (stopWord) {
        var regex = new RegExp(
            '(\\s|^)(?:' + stopWord + ')(?=\\s|$)', 'g'
        );
        text = _.replace(text, regex, ' ');
    });

    return text;
};
```

Izvorna koda 5.1: Funkcija za odstanjevanje stop besed.

Iskanja oz. ekstrakcije besed smo se lotili s pomočjo knjižnice Lodash, saj nam njene vgrajene funkcije omogočajo zelo elegantno in enostavno rešitev dotičnega problema. Rešitev ni nič drugega kot izločanje besed iz razbitega besedila, ki so bodisi celo število bodisi število s plavajočo vejico:

```
var extractWords = function (text) {
    return _.reject(_.words(text), function (w) {
        return
            !_.isNaN(parseInt(w)) ||
            !_.isNaN(parseFloat(w))
    });
};
```

Izvorna koda 5.2: Funkcija za iskanje besed.

Lematizacije besed se bomo lotili s pomočjo aplikacije, napisane v C++ programskem jeziku, saj (še) ne obstaja Javascript verzija lematizatorja za slovenski jezik. Avtorji aplikacije LemmaGen Slovene Lemmatizer module

so Domen Grabec, Jernej Virag in Gašper Žejn, izvorna verzija pa je nastala pod okriljem Inštituta Jožef Stefan. Projekt se nahaja na spletnem naslovu `https://bitbucket.org/mavrik/slovene_lemmatizer` in je prosto dostopen. Lematizator bomo ob vsakem zagonu Docker vsebnika prevedli (angl. *compile*), da bomo lahko potem znotraj pametnega asistenta izvajali ukaz `child_process.execFileSync(...)` z argumenti:

- `lemmatizerPath`: direktorijska pot do izvršljive datoteke lematizatorja;
- `languageFilePath`: direktorijska pot do jezikovne datoteke;
- `word`: beseda, ki jo želimo lematizirati;
- {`encoding: 'utf-8'`}: kodiranje.

Rezultat klica je standardni izhod aplikacije, tj. lematizirana beseda.

Postopka določanja ključnih besed se bomo lotili tako, da bomo najprej poiskali unikatne besede in za tem znotraj zanke iterirali po seznamu teh besed ter jih lematizirali, v kolikor niso v seznamu rezerviranih besed. Ker seznam unikatnih besed v začetku ni povsem unikaten, saj je možno, da vsebuje isto besedo, vendar v različni skladenjski obliki, bomo lematizirali vse omenjene unikatne besede in v nabor ključnih besed dodajali samo tiste besede, ki so zares unikatne. To funkcijo bomo uporabljali tudi za določanje ključnih besed iz uporabnikovega vnosa, zato bomo imeli tudi preverjanje in v kolikor bo beseda glagol, jo bomo zavrgli, v nasprotnem primeru pa bo dodana v nabor ključnih besed.

```
1  var extractKeywords = function (text, processText, isUserInput) {
2      if (processText) text = processContent(text);
3
4      var keywords = [];
5      var words = extractWords(text);
6      var uniqueWords = _.uniqBy(words, _.lowerCase);
7
8      _.forEach(uniqueWords, function (uniqueWord) {
9          var count = _.remove(words, function (word) {
10             return word === uniqueWord
```

```
        }).length;

        var word = uniqueWord.toLowerCase();
        if (reservedWords.indexOf(word) === -1) {
            word = lemmatizeSync(word);
        }

        if (isUserInput && sloveneVerbs.indexOf(word) !== -1) return;

        var existingWordIndex = _.findIndex(keywords, ['word', word]);
        if (existingWordIndex !== -1) {
            keywords[existingWordIndex].count += count;
        } else {
            if (isUserInput || count >= 1 && word.length > 0) {
                keywords.push({
                    word: word,
                    count: count
                });
            }
        }
    });

    return keywords;
};
```

Izvorna koda 5.3: Funkcija za iskanje ključnih besed.

Kot zadnja izmed nabora funkcij za procesiranje teksta je funkcija, ki poišče razliko med dvema nizoma znakov. S pojmom razlika povedi mislimo na besede daljše povedi, ki ne vsebujejo prvih n skupnih besed.

```
var stringDifference = function (firstString, secondString) {
    var a = firstString.replace('(.*)', '').trim().split(' ');
    var b = secondString.replace('(.*)', '').trim().split(' ');

    return a.length < b.length ?
        (b.splice(a.length, b.length - a.length) :
        a.splice(b.length, a.length - b.length))
        .join(' ');
};
```

Izvorna koda 5.4: Funkcija za iskanje razlike med nizoma.

Funkcijo 5.4 bomo uporabljali za pridobivanje argumentov ukaza, v kolikor bo najdeno popolno ujemanje z vnaprej definiranimi ukazi. Kot primer lahko podamo naslednji povedi: *„Kaj pomeni kratica (.*)“* in *„Kaj pomeni kratca preverjanje znanja“*. Funkcija `stringDifference` nam bo kot rezultat vrnila seznam z dvema elementoma `['preverjanje', 'znanja']`.

5.3 Razčlenjevanje gradiva

Za razčlenjevanje gradiva bomo implementirali funkcijo, ki bo kot argument prejela seznam imen datotek za uvoz v podatkovno bazo. Ta argument bo poimenovan `documents` in bo v obliki Javascript tabele:

```
1  var documents = [
2      'Arhitektura-IS-in-e-trgovanje.html',
3      'Dopolnitve-Node-js-spletne-klepetalnice.html',
4      'index.html',
5      'informatika-v-zdravstvu.html',
6      'literatura.html',
7      'Marshmallow-challenge.html',
8      'Nacrt-IS-in-aplikacija-e-Zdravje.html',
9      'Nacrtovanje-IS.html',
10     ...
11 ];
```

Izvorna koda 5.5: Seznam dokumentov za razčlenjevanje.

S pomočjo argumenta `documents` bomo v zanki odprli posamezno datoteko, jo prebrali in razčlenili. Datoteke bomo odpirali s pomočjo vgrajene funkcije `readFile(path[, options], callback)`, ki je del Node.js modula `fs`. Funkcija `readFile` nam vrne HTML dokument kot instanco razreda `Buffer`, ki ga v nadaljevanju pretvorimo v niz (*string*) s pomočjo funkcije `toString()` in podamo kot vhodni argument funkciji `batch(html, dictionary [, callback])`. Funkcija `batch(...)` je del html-to-json[3] paketa, ki nam omogoča enostavno pretvorbo niza znakov v HTML obliki v Javascript objekt. Funkcija `batch(...)` pričakuje kot drugi argument

[3] `https://github.com/prolificinteractive/node-html-to-json`

(`dictionary`) Javascript objekt s ključi (*key*) in pripadajočimi razčlenjevalniki. Zato moramo za vsak ključ ustvariti razčlenjevalnik, kar pa ni nič drugega kot definirati jQuery[4] selektorje. Končni rezultat te funkcije bo Javascript objekt, ki bo v enaki obliki, kot smo ga definirali v drugem argumentu.

```
htmlToJson.batch(html.toString(), {
    chapters: htmlToJson.createParser(['h1:has(>span)', {
        'title': function ($chapter) {
            return textProcessor.removeWhiteSpaces(
                $chapter.clone().children().remove().end().text()
            )
        },
        'number': function ($chapter) {
            return textProcessor.removeWhiteSpaces(
                $chapter.find('.header-section-number').text()
            )
        },
        'intro': function ($chapter) {
            return textProcessor.processContent(
                $chapter.nextUntil('div.section.level2').text()
            )
        },
        'rawContent': function ($chapter) {
            return textProcessor.removeWhiteSpaces(
                $chapter.nextUntil('div.section.level2').text()
            )
        },
        'chapterId': function ($chapter) {
            return textProcessor.removeWhiteSpaces(
                $chapter.find('.header-section-number').text()
            ).match(/\d+/).join('')
        },
        'images': htmlToJson.createMethod(['div.figure', {
            'src': function ($figure) {
                return $figure.find('img').attr('src');
            },
            'alt': function ($figure) {
                return textProcessor.removeWhiteSpaces(
                    $figure.find('img').attr('alt')
```

[4] https://jquery.com/

```
35                  );
36              },
37              'imageId': function ($figure) {
38                  return $figure.find('span').attr('id');
39              },
40              'caption': function ($figure) {
41                  return textProcessor.removeWhiteSpaces(
42                      $figure.find('p.caption').text()
43                  );
44              }
45          }]),
46          'fileName': docName
47      }]),
48      ...
```

Izvorna koda 5.6: Funkcija za razčlenjevanje poglavja.

Kot lahko vidimo na izseku kode 5.6, uporabljamo različne funkcije paketa `textProcessor`. Te funkcije vključujejo odstranjevanje nepotrebnih presledkov in posebnih znakov ter procesiranje besedila, ki vključuje iskanje ključnih besed, njihovo štetje in lematizacijo ter odstranjevanje stop besed. Funkcije `processContent(text)` v poglavju 5.2 nismo eksplicitno omenili, saj združuje funkcije `removeWhiteSpaces`, `removeStopWords`, `removePunctuations` in `removeNumbers`, ki so gnezdene ena znotraj druge.

5.4 Pridobivanje podatkov o študentih

Podatke o študentih bomo pridobivali s pomočjo Googlovih preglednic, saj nam takšen način omogoča, da lahko administrator (nosilec predmeta) enostavno ureja podatke, ki se jih nato avtomatsko shranjuje v podatkovno bazo.

Preglednica je zaščitena s pomočjo uporabniškega imena in gesla in se je ne more javno deliti, zato bomo do nje programsko dostopali s pomočjo Google API in Google Auth Library. Za uspešno avtentikacijo in avtorizacijo moramo omenjenima knjižnicama podati client secret, client id in redirect uri. Te podatke si zgeneriramo na nadzorni plošči Googlovega razvijalskega portala na spletnem naslovu `https://console.developers.google.com/`

`apis/credentials`. Naš Node.js odjemalec bo te podatke črpal preko okoljskih spremenljivk (angl. *environment variables*), ki jih podamo preko `docker-compose.yml` datoteke. V procesu avtentikacije nam knjižnica Google Auth Library zgradi unikaten URL naslov, ki ga moramo ročno obiskati; se tam prijaviti z računom, ki ima dostop do želene preglednice; avtorizirati, da bomo s tem uporabniškim računom dovolili aplikaciji (v našem primeru je to pametni asistent) dostop do preglednice; in skopirati zgenerirano kodo v našo aplikacijo preko standardnega vhoda. Ob uspešnem koraku avtentikacije uporabnika z aplikacijo se ustvari nov OAuth2 objekt, s katerim lahko v nadaljevanju dostopamo do preglednice. Ob vsakem dostopu do preglednice se s pomočjo tega objekta avtoriziramo in nazaj dobimo preglednico v obliki Javascript tabele (*Array*). Zaradi lažjega in predvsem občutno hitrejšega shranjevanja v podatkovno bazo podatke ločimo s podpičjem (`;`) in vrstico po vrstico zapisujemo v CSV datoteko, ki jo lokalno shranimo. Podatke iz datoteke uvozimo v Neo4j podatkovno bazo s pomočjo enostavnega insert stavka, saj podatkovna baza omogoča enostavno in hitro delo z uvažanjem podatkov iz CSV datotek.

5.5 Podatkovni model

Za shranjevanje podatkov bomo uporabljali dve podatkovni bazi, ena bo relacijska (MySQL) in ena nerelacijska (Neo4j). V relacijski podatkovni bazi bomo shranjevali podatke o uporabnikih in vprašanja, ki jih bodo zastavljali pametnemu asistentu, za kar bomo ustvarili dve tabeli. Prva tabela se bo imenovala `users` in bo vsebovala naslednje stolpce: `id`, `slack_channel_id`, `slack_user_id`, `slack_email` ter časovna žiga `created_at` in `updated_at`. Stolpec `slack_channel_id` nam bo v pomoč, če bomo želeli ugotoviti, ali je uporabnik poslal sporočilo v javni kanal (*general*) ali kot privatno sporočilo pametnemu asistentu. Stolpec `slack_user_id` bomo uporabljali izključno za lažjo analizo podatkov, medtem ko bo stolpec `slack_email` služil le kot referenca, ali je uporabnik uporabil e-mail naslov, ki se ujema z njegovim za-

pisom v podatkih o študentih. Časovna žiga nam bosta prišla prav v procesu analize podatkov, ko bomo skušali ugotoviti vedenjski vzorec uporabnikov. Druga tabela bo `user_queries`, v katero bomo shranjevali vse podatke o zastavljenih vprašanjih in bo imela naslednje stolpce: `id`; `slack_channel_id` in `slack_user_id`, ki se bosta navezovala na tabelo `users`; `slack_event`, ki nam bo povedal, ali je bilo sporočilo naslovljeno direktno ali s pomočjo sklicevanja (*mention*); `controller_command` nam bo povedal, kateri rokovalnik (*controller*) je procesiral ukaz; stolpec `raw_text`, ki bo vseboval neobdelano vprašanje; `parsed_text`, ki bo vseboval obdelano vprašanje, kot ga bo razumel pametni asistent; stolpec `has_successfully_replied` nam bo povedal, ali je rokovalnik uspešno odgovoril na vprašanje ali pa je naletel na napako; stolpec `reply_text` bo vseboval odgovor, ki ga je pametni asistent poslal uporabniku; stolpec `error_dump`, v katerem bo zapisan tekst morebitne napake, v kolikor je pametni asistent naletel na napako in ni mogel smiselno odgovoriti na zastavljeno vprašanje; ter časovna žiga `created_at` in `updated_at`. Diagram entiteta–razmerje omenjenih tabel je prikazan na sliki 5.2. Za lažje izvajanje CRUD operacij na podatkovno bazo bomo uporabljali Node.js ORM knjižnico Sequelize,[5] ki nam omogoča mapiranje med vrsticami v podatkovni bazi in Javascript objekti. Z drugimi besedami to pomeni, da se kot razvijalci ne zavedamo tega, da delamo z vrsticami v podatkovni bazi, ampak da delamo z običajnimi objekti, kot smo vajeni v objektno orientiranem programiranju. Vse, kar moramo narediti, je, da ustvarimo nov objekt s povezavo do podatkovne baze in modele, nad katerimi bomo izvajali CRUD operacije. Tako bomo za vrstice v `users` tabeli ustvarili objekt `User`:

```
1 var _sequelize      = require('../Mysql.js');
2 var Sequelize       = require('sequelize');
3
4 var User = _sequelize.define('users', {
5     slack_channel_id:   Sequelize.STRING,
6     slack_user_id:      Sequelize.STRING,
7     slack_email:        Sequelize.STRING
```

[5] http://sequelize.readthedocs.io/en/v3/

```
8 }, { charset: 'utf8mb4' });
```

Izvorna koda 5.7: Definicija objekta `User`.

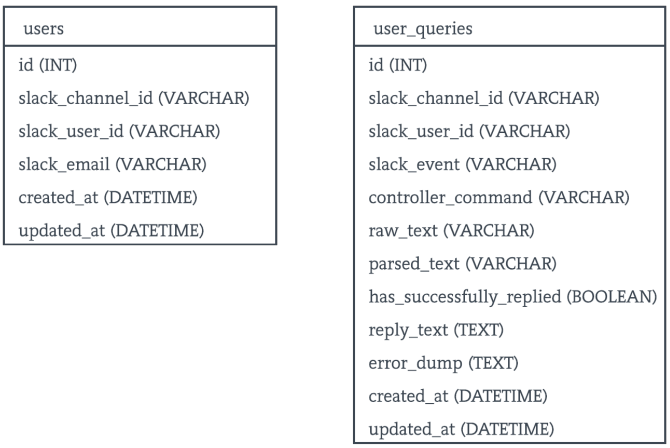
Za vrstice v tabeli `user_queries` pa bomo ustvarili objekt `UserQuery`:

```
1  var _sequelize      = require('../Mysql.js');
2  var Sequelize       = require('sequelize');
3
4  var UserQuery = _sequelize.define('user_queries', {
5      slack_channel_id:           Sequelize.STRING,
6      slack_user_id:              Sequelize.STRING,
7      slack_event:                Sequelize.STRING,
8      controller_command:         Sequelize.STRING,
9      raw_text:                   Sequelize.STRING,
10     parsed_text:                Sequelize.STRING,
11     has_successfully_replied:   Sequelize.BOOLEAN,
12     reply_text:                 Sequelize.TEXT,
13     error_dump:                 Sequelize.TEXT
14 }, { charset: 'utf8mb4' });
```

Izvorna koda 5.8: Definicija objekta `UserQuery`.

V spremenljivki `_sequelize` hranimo povezavo do podatkovne baze, ki jo definiramo v datoteki `Mysql.js`.

Razčlenjeno gradivo in podatke o študentih bomo shranjevali v NoSQL podatkovni bazi (Neo4j). Vozlišča bomo ločili v dva sklopa: prvi sklop bodo vozlišča s podatki o študentih, drugi sklop pa bodo vozlišča, ki bodo hranila podatke o gradivu. Vsako vozlišče bo imelo svojo oznako (angl. *label*) in vsak eno poimenovano povezavo (angl. *named relationship*). V Neo4j terminologiji se takšna oblika grafa imenuje *labeled property graph data model*. Vozlišče z oznako `Student` bo hranilo lastnosti posameznega študenta, kot so npr. ime, priimek, vpisna številka, e-mail naslov itd. Vsako vozlišče `Student` bo imelo tri izhodne povezave: ena povezava se bo imenovala `SUBMITTED` in bo povezana z vozlišči tipa `Homework`, ki bo hranilo podatke o doseženih točkah pri posamezni oddani domači nalogi; druga povezava se bo imenovala `COMPLETED` in bo povezovala vozlišča `Quiz`, ki bodo hranila podatke o doseženih točkah na posameznem preverjanju znanja. Tretji tip povezave iz vozlišča `Student`



Slika 5.2: ER diagram relacijske podatkovne baze.

se bo imenovalo `ATTENDED` in bo vhodna povezava v vozlišča tipa `Lecture`. Ta vozlišča bodo hranila podatke o prisotnosti na predavanjih, ki jo bo nosilec predmeta izvajal z odgovarjanjem na vprašanja s pomočjo portala Kahoot.[6] Kot lahko vidimo, je struktura vozlišč in medsebojnih povezav za podatke o študentih trivialna. Na drugi strani imamo kompleksnejšo strukturo vozlišč in povezav za gradivo. Korensko vozlišče se imenuje `Document` in ima izhodne povezave na vozlišča:

- `Acronym` s pripadajočo povezavo `HAS_ACRONYM`, ki se nanaša na kratice, uporabljane pri predmetu;

- `Reference` s povezavo `HAS_REFERENCE`, ki opisujejo podatke o literaturi za predmet;

[6] `https://kahoot.com`

- `Chapter` s povezavo `HAS_CHAPTER`, ki bo opisovalo posamezno poglavje gradiva.

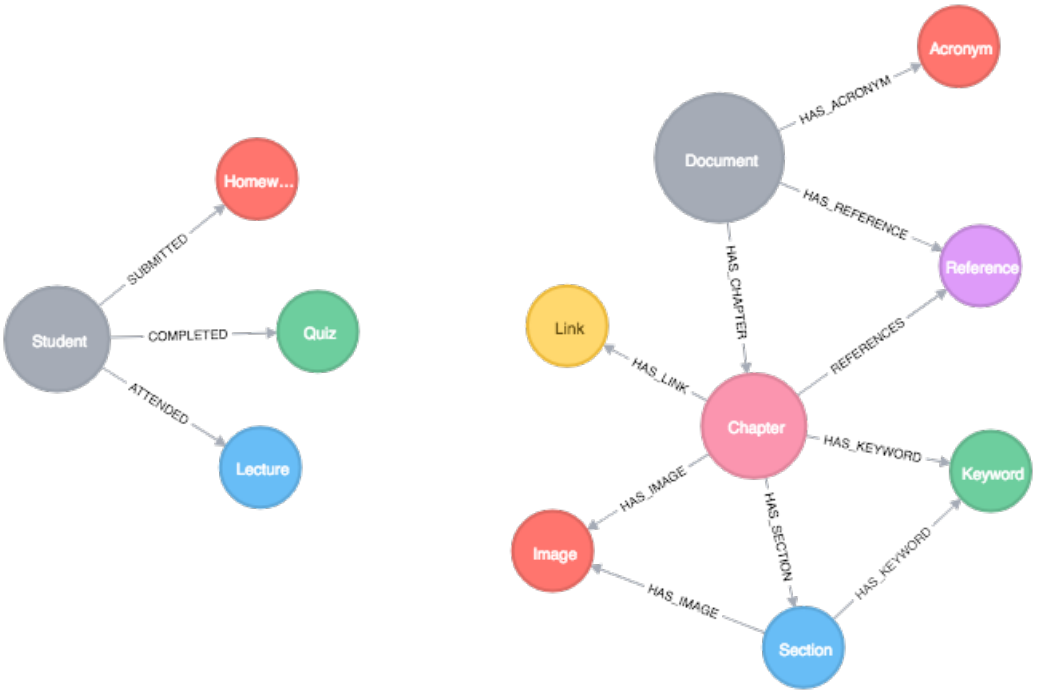
Posamezno poglavje (oznaka `Chapter`) bomo nadalje razdelili na podpoglavja z oznako vozlišča `Section` in povezavo `HAS_SECTION`, ki bo hranila vsebino posameznega podpoglavja. Vozlišče `Chapter` bo lahko imelo dva izhodna tipa povezav do vozlišč `Image` in `Keyword` z oznakama `HAS_IMAGE` in `HAS_KEYWORD`, ki sta lahko povezani tudi z vozliščem `Chapter`. Zadnja izmed izhodnih povezav iz vozlišča `Chapter` je povezava `HAS_LINK` do vozlišča `Link`. Vozlišče bo opisovalo povezavo med predavanjem in vajami, kar lahko pokažemo na primeru: „P1.1 → V1“. To pomeni, da se predavanje 1 navezuje na vaje 1 in omogoča študentom lažje priprave na vaje. Struktura vozlišč in medsebojnih povezav je prikazana na sliki 5.3. Tako kot uporabljamo Sequelize za CRUD operacije nad SQL podatkovno bazo, bomo tudi za operacije nad NoSQL podatkovno bazo uporabljali knjižnico z razliko, da ta knjižnica ne bo tipa ORM, kar pomeni, da ne bo direktnih preslikav med Javascript objekti ter vrsticami, ampak bo služila zgolj za povezavo do podatkovne baze ter izvajanje domorodnih (angl. *native*) poizvedb. Uporabljali bomo knjižnico neo4j,[7] ki ni uradna knjižnica podjetja Neo Technology, ampak so jo razvili pri podjetju FiftyThree[8] za lastne potrebe. Za uporabo te knjižnice namesto uradne smo se odločili zato, ker knjižnica podjetja FiftyThree podpira paketno obdelavo poizvedb, kar bo omogočalo hitrejše izvajanje operacij vstavljanja v procesu razčlenjevanja gradiva.

5.6 Prepoznavanje ukazov

Ogrodje Botkit, ki ga bomo uporabljali za implementacijo pametnega asistenta, ima zelo dobro zasnovano logiko za prepoznavanje ukazov. Za ta namen nam ponuja funkcijo `hears(patterns, types, [middleware|callback,] [callback])`, ki nam omogoča implementacijo rokovalnikov, spe-

[7] `https://github.com/thingdom/node-neo4j/tree/v2`

[8] `https://www.fiftythree.com/`



Slika 5.3: Diagram nerelacijske podatkovne baze.

cifičnih za določene ključne besede, fraze ali tekst ter tipe sporočil, kot so npr. `direct_mention`, `direct_message` itd. Kot prvi argument funkciji `hears(...)` tako podamo seznam (*array*) regularnih izrazov, ki naj jih določeni rokovalnik prepozna in obdela. Funkcijo, ki delegira, kateri rokovalnik mora prevzeti in sprocesirati ukaz, se lahko implementira in uporablja na dva načina. Prvi način je, da se funkcija poda kot tretji argument posameznemu rokovalniku kot vmesna logika (angl. *middleware*). Ta način bi lahko uporabljali, če ne bi imeli veliko rokovalnikov ali pa če bi rabili več različnih rokovalnikov. Drugi način je (kot ga bomo uporabili) implementacija funkcije `changeEars(callback)`, ki popolnoma nadomesti vse vmesne funkcije vseh rokovalnikov. Ta način je bolj primeren za naš primer, saj se izognemo ponavljanju kode, ker se uporabi ena globalna funkcija za vse rokovalnike.

Kot smo omenili, funkcija `hears(...)` kot prvi argument zahteva seznam regularnih izrazov. Te bomo za vsak rokovalnik definirali posebej v datoteki

`Constants.js` zaradi boljše preglednosti kode. Ustvarili bomo objekt, ki bo vseboval ime rokovalnika, seznam regularnih izrazov, poizvedbo za bazo in funkcijo, ki bo vračala rezultat poizvedbe. Ostale podatke objekta bomo uporabljali pri nadaljnjem procesiranju v rokovalniku. Izsek kode 5.9 prikazuje, kako izgleda seznam regularnih izrazov za rokovalnik, ki procesira uporabnikovo zahtevo za prikaz točk na določenem preverjanju znanja:

```
1  studentTestScoreSpecific: {
2      ...
3      hears: [
4          'koliko sem dosegel na pz(.*)',
5          'koliko sem dosegel na preverjanju znanja (.*)',
6          'tocke pz(.*)',
7          'tocke preverjanja znanja (.*)',
8          'tocke pz'
9      ],
10     ...
11 }
```

Izvorna koda 5.9: Seznam regularnih izrazov za prikaz točk na določenem preverjanju znanja.

Funkcijo `changeEars(...)` bomo zasnovali tako, da bomo najprej iterirali čez vse vnaprej definirane regularne izraze in poskusili najti popolno ujemanje med podanim regularnim izrazom ter uporabnikovo zahtevo (sporočilom). V kolikor ne bomo našli ujemanja, bomo uporabili algoritem Jaro Winkler, s katerim bomo poskušali najti regularni izraz, ki je čim bolj podoben uporabnikovi zahtevi. Lahko se zgodi, da bomo našli več ujemanj, ki so nad našo določeno mejo. V tem primeru bomo sortirali vsa ujemanja po vrednosti ujemanja (in abecedi) in vrnili ukaz, ki se najbolj ujema z vnaprej definiranimi ukazi. Mejo ujemanja smo hevristično določili na podlagi testiranj. Izkazalo se je, da je najbolj primerna vrednost podobnosti pri $0,89$, kar pomeni, da se morata niza ujemati vsaj 89 %. V primeru, da ni niti popolnega niti približnega ujemanja z vsaj enim izmed regularnih izrazov, se na uporabnikovo sporočilo ne odzovemo. Implementacija funkcije je prikazana na 5.10.

```
controller.changeEars(function (patterns, message) {
    var similarPatterns = [];
    message.text = message.text.toLowerCase();

    for (var i = 0; i < patterns.length; i++) {
        if (message.text) {
            var test = new RegExp(patterns[i], 'i');
            var match = message.text.match(test);
            var similarity = natural.JaroWinklerDistance(patterns[i],
                message.text);

            if (match) {
                message.match = match;
                return true;
            } else if (similarity >= 0.89) {
                message.test = textProcessor.removeWhiteSpaces(
                    textProcessor.removePunctuations(message.text)
                );
                match = textProcessor.extractNumber(message.text);

                if (match.trim() === '') {
                    match = textProcessor.stringDifference(patterns[i],
                        message.text);
                }

                similarPatterns.push({
                    similarity: similarity,
                    match: [message.text, match, patterns[i]]
                });
            }
        }
    }

    if (similarPatterns.length) {
        var similarMatch = _.last(
            _.sortBy(similarPatterns, ['similarity'])
        );
        message.match = similarMatch.match;
        return true;
    }

```

```
40      return false;
41  });
```

Izvorna koda 5.10: Funkcija `changeEars`.

5.7 Procesiranje ukazov

Kot smo opisali v poglavju 5.6, bomo uporabljali funkcijo `hears(...)` za procesiranje zahtevkov in funkcijo `changeEars(...)` za delegiranje zahtevka posameznemu rokovalniku. Odzivali se bomo na tri tipe Slack dogodkov, in sicer:

- na dogodke tipa `direct_message`, ki se bodo uporabljali pri rokovalnikih za vprašanja o študentih, saj bomo s to omejitvijo onemogočili, da bi uporabniki po nesreči zahtevali odgovor na osebne podatke znotraj skupnega kanala;
- na dogodke tipa `direct_message` in `direct_mention`, ki se bodo uporabljali pri rokovalnikih za vprašanja glede gradiva;
- na dogodek `team_join`, ki se sproži, ko se nov uporabnik pridruži ekipi.

5.7.1 Odzivanje na Slack dogodke

Dogodek `team_join` bomo uporabljali za to, da zabeležimo novega uporabnika v podatkovno bazo in mu pošljemo pozdravno sporočilo ter osnovne napotke za uporabo. To bomo implementirali tako, da bomo najprej naredili API klic na končno točko (angl. *endpoint*) `https://api.slack.com/methods/users.info` z argumentom `user`, ki ga pridobimo iz sporočila. S tem klicem bomo pridobili podatke o uporabniku, natančneje njegov e-mail naslov, ki ga bomo v nadaljevanju uporabljali za avtorizacijo pri rokovalnikih za podatke o študentih. Ko se API klic uspešno razreši, odpremo direktno povezavo za pošiljanje sporočil med pametnim asistentom in na novo pridruženim uporabnikom. To storimo z API klicem na končno točko

`https://api.slack.com/methods/im.open` z istim argumentom, kot smo ga uporabili za pridobivanje podatkov o uporabniku. Ob uspešno razrešenem API klicu se bo odprla nova povezava in vrnil se bo ID te povezave (kanala). Tako e-mail naslov kot ID povezave bomo shranili v podatkovno bazo, saj bomo te podatke potrebovali za statistično obdelavo. Na koncu obeh API klicev bomo poslali direktno sporočilo uporabniku z novim API klicem na končno točko `https://api.slack.com/methods/chat.postMessage` z argumenti ID kanala, sporočilom ter logično vrednostjo (angl. *boolean*) `true` za argument `as_user`.

```
1  controller.on('team_join', function(bot, message) {
2      var userId = message.user.id;
3
4      slackApiHelper.fetchUserInfo(bot, userId).then(function (user) {
5          var _User = {
6              slack_user_id: userId,
7              slack_email: user.profile.email
8          };
9
10         slackApiHelper.openIm(bot, userId).then(function (channel) {
11             _User.slack_channel_id = channel.id;
12             User.create(_User);
13
14             slackApiHelper.postMessage(bot, channel.id,
                   constants.botMessages.welcome);
15             setTimeout(function () {
16                 slackApiHelper.postMessage(bot, channel.id,
                       constants.botMessages.help) }
17             , 200);
18         });
19     });
20 });
```

Izvorna koda 5.11: Rokovalnik za dogodek `team_join`.

5.7.2 Odzivanje na vprašanja v povezavi s podatki o študentih

Rokovalniki, ki bodo zadolženi za procesiranje vprašanj v povezavi s podatki o študentih, bodo izgledali približno enako, edina razlika med njimi bo, v kolikor bodo upoštevali tudi parametre znotraj vprašanja ali ne. Vseh skupaj jih bo 15 in bodo obsegali tako odgovore na osnovne podatke, na število doseženih točk pri posamezni obveznosti, kot tudi na to, ali je študent opravil predmet itd. Parametri, ki jih bodo lahko funkcije sprejemale, bodo le številskega tipa in bodo označevale zaporedne številke preverjanj znanj, domačih nalog in predavanj. Rokovalniki, ki bodo sprejemali dodatne parametre, bodo najprej pretvorili niz (angl. *string*), ki bo vseboval zaporedno številko, v celo število (angl. *integer*) s pomočjo funkcije `parseInt(value)`. V naslednjem koraku bomo pretvorjeno število testirali, če je res število, s pomočjo funkcije `isNan(value)` iz knjižnice Lodash. Funkcija `isNan(value)` nam vrne logično vrednost `true`, v kolikor je podani parameter `value` tipa `NaN` in `false` sicer. V kolikor bo funkcija vrnila vrednost `true`, bomo uporabnika opozorili s sporočilom, da podani parameter ni številska vrednost in v podatkovno bazo zabeležili, da smo pri procesiranju vprašanja naleteli na napako. Proces procesiranja v rokovalnikih je za tem korakom povsem enak. Najprej avtoriziramo uporabnika z API klicem na končno točko `https://api.slack.com/methods/users.info` z argumentom `user`, ki ga pridobimo iz sporočila. V primeru, da se klic neuspešno razreši, uporabniku odgovorimo s sporočilom, da je pametni asistent naletel na napako in da naj napako sporoči administratorju ekipe. Ko se klic uspešno razreši, izvedemo poizvedbo nad našo nerelacijsko podatkovno bazo z argumentom e-mail uporabnika, ki nam ga vrne uspešno razrešen Slack API klic. To nam zagotavlja, da uporabnik lahko vidi samo svoje podatke in na nikakršen način ne more videti podatkov o ostalih uporabnikih pametnega asistenta.

```
1 MATCH (student:Student {studisEmail: {email}})
2       -[:COMPLETED]->
3       (quiz:Quiz {number: {number}})
```

```
4  RETURN
5      'Preverjanje znanja ' + quiz.number  + '\n' +
6      'Dosezene tocke: ' + quiz.score  + '\n' +
7      'Odbitek: ' + quiz.deduction + '\n' +
8      'Skupaj: ' + toFloat(quiz.score - quiz.deduction)
9  AS result
```

Izvorna koda 5.12: Poizvedba za iskanje točk določenega preverjanja znanja.

Ko se poizvedba uspešno izvede, uporabimo funkcijo `data(neo4jResponse)`, ki za vsak rokovalnik izlušči podatke za posredovanje uporabniku. Implementacija funkcije za luščenje podatkov iz rezultata poizvedbe je za rokovalnike, ki odgovarjajo na vprašanja, povezana s podatki o študentih, zelo preprosta:

```
1  var data = function (neo4jResponse) {
2      return neo4jResponse[0].result
3  };
```

Izvorna koda 5.13: Funkcija za vračanje podatkov iz rezultata poizvedbe.

Vse poizvedbe za rokovalnike tega tipa nam bodo vračale seznam JSON objektov dolžine 1, mi pa bomo potrebovali samo vrednost `result`, kot smo jo poimenovali v poizvedbi v stavku `RETURN AS`. V primeru, da se poizvedba neuspešno konča, je to lahko bodisi zaradi tega, da podatki ne obstajajo v naši podatkovni bazi (v tem primeru primerno obvestimo uporabnika), bodisi zaradi prekinjene povezave (v tem primeru se celotna aplikacija ponovno zažene s pomočjo paketa Forever[9] in pametni asistent ponovno začne s procesiranjem ukaza).

```
1  controller.hears(constants.botHearPatterns.studentInfo.hears,
2      'direct_message', function (bot, message) {
3
4      var userId = message.user;
5
6      var _UserQuery = {
7          slack_channel_id: message.channel,
8          slack_user_id: userId,
9          slack_event: message.event,
```

[9] https://github.com/foreverjs/forever

```
10          controller_command: constants.botHearPatterns.studentInfo.name,
11          raw_text: message.text,
12          parsed_text: message.match[0]
13      };
14
15
16      slackApiHelper.fetchUserInfo(bot, userId)
17          .then(function (user) {
18              neo4j.executeCypher(
                    constants.botHearPatterns.studentInfo.cypher, { email:
                    user.profile.email })
19                  .then(function (result) {
20                      var reply = constants.botHearPatterns.studentInfo
21                          .data(result);
22
23                      bot.reply(message, reply);
24
25                      _UserQuery.has_successfully_replied = true;
26                      _UserQuery.reply_text = reply;
27                      UserQuery.create(_UserQuery);
28                  })
29                  .catch(function (error) {
30                      bot.reply(message,
                            constants.botMessages.errors.userNotFound);
31
32                      _UserQuery.has_successfully_replied = false;
33                      _UserQuery.reply_text =
                            constants.botMessages.errors.userNotFound;
34                      _UserQuery.error_dump = error.message;
35                      UserQuery.create(_UserQuery);
36                  });
37          })
38          .catch(function (error) {
39              bot.reply(message, constants.botMessages.errors.botError);
40
41              _UserQuery.has_successfully_replied = false;
42              _UserQuery.reply_text =
                    constants.botMessages.errors.botError;
43              _UserQuery.error_dump = error.message;
44              UserQuery.create(_UserQuery);
45          });
```

```
46 });
```

Izvorna koda 5.14: Rokovalnik za vprašanje o osnovnih podatkih študenta.

5.7.3 Odzivanje na vprašanja v povezavi z gradivom pri predmetu

V poglavju 5.7.2 smo omenili, da so si rokovalniki po strukturi in procesiranju zelo podobni in da imamo samo dva različna načina procesiranja: tiste brez parametra in tiste s parametri. Pri rokovalnikih, ki odgovarjajo na vprašanja v povezavi z gradivom, je drugače. Razlikujejo se predvsem v pripravi in procesiranju parametrov, ki jih uporabljamo pri izvajanju poizvedb na NoSQL podatkovni bazi. Ti rokovalniki prav tako ne vsebujejo procesa avtorizacije (s klici na Slack API). Postopek izvajanja poizvedb in vračanja odgovorov uporabniku pa je povsem enak. Za vsakega od rokovalnikov bomo predstavili, kako se ti parametri pripravijo in kakšne poizvedbe uporabljajo.

Začeli bomo z rokovalnikom, ki išče razlago kratice. Ta rokovalnik je dokaj enostaven, saj prejme parameter kar iz vmesne funkcije (opisali smo jo v poglavju 5.6) in do njega dostopamo s selektorjem objekta `message.match[1]`. Izvorna koda 5.15 prikazuje poizvedbo, ki poišče razlago kratice. Poizvedba je oblikovana kot seznam nizov, ki jih združimo v enega s pomočjo funkcije `join(separator)` in se nahaja znotraj objekta, kot smo opisali v poglavju 5.6. Zaradi lažje berljivosti je na 5.15 prikazana v *cypher* obliki.

```
1  WITH {word} AS word
2  OPTIONAL MATCH (a:Acronym)
3  WHERE toLower(a.acronymSl) = word OR toLower(a.acronymEn) = word
4  RETURN CASE
5      WHEN exists(a.descriptionSl) AND exists(a.descriptionEn)
6          THEN 'Kratica ' + toUpper(word) +
7              ' v slovenscini pomeni ' + a.descriptionSl +
8              ', v anglescini pa ' + a.descriptionEn + '.'
9      WHEN exists(a.descriptionSl)
10         THEN 'Kratica ' + toUpper(word) +
11             ' v slovenscini pomeni ' + a.descriptionSl + '.'
12     WHEN exists(a.descriptionEn)
```

```
13          THEN 'Kratica ' + toUpper(word) +
14              ' v anglescini pomeni ' + a.descriptionEn + '.'
15      ELSE 'Kratica ' + toUpper(word) +
16          ' ni v slovarju.'
17  END AS result
```

Izvorna koda 5.15: Poizvedba za iskanje razlage kratice.

Naslednji izmed rokovalnikov je rokovalnik, ki odgovarja na vprašanja, kot so npr. „Kje lahko najdem kaj več o X“, „Kaj pomeni X“, „V katerem poglavju je X“ itd. Ta rokovalnik uporablja funkcije iz paketa `textProcessor`, kot smo uporabljali pri razčlenjevanju gradiva. V prvem koraku najprej izvlečemo ključne besede iz parametra `message.match[1]`, ki nam ga vrača vmesna funkcija `changeEars(...)`. Pri tem uporabimo funkcijo `extractKeywords(...)`, kot smo jo opisali v poglavju 5.2. V kolikor nam bo funkcija vrnila več kot eno ključno besedo, bomo v poizvedbi uporabljali iskanje po množici vseh ključnih besed. Znotraj poizvedbe bomo uporabili statistiko *tf-idf* za iskanje dokumenta znotraj korpusa, kjer ima ključna beseda največjo težo. Poizvedba je prikazana na 5.16.

```
1  MATCH (:Section)
2  WITH COUNT(*) AS numberOfDocuments
3  MATCH (k:Keyword {word: {word}})<-[r:HAS_KEYWORD]-(s:Section)<-[:
       HAS_SECTION]-(c:Chapter)
4  WITH numberOfDocuments, k, s, c, toFloat(r.count) / toFloat(
       s.numberOfTerms) AS tf
5  MATCH (k)<-[:HAS_KEYWORD]-(allSectionsWithTerm)
6  WITH s AS section, c AS chapter, tf, s.numberOfTerms AS numberOfTerms,
       log(toFloat(numberOfDocuments) / toFloat(COUNT(allSectionsWithTerm)))
        AS idf
7  WITH section, chapter, toFloat(idf) * toFloat(tf) AS score
8  ORDER BY score DESC
9  RETURN section, chapter
10 LIMIT 1
```

Izvorna koda 5.16: Poizvedba za iskanje poglavja.

V kolikor nam poizvedba vrne rezultat, uporabimo funkcijo `data(neo4jResponse, keyword)`, ki se bistveno razlikuje od tiste, ki smo jo opisali v

poglavju 5.7.2. Implementacija funkcije `data(neo4jResponse, keyword)` je prikazana na 5.18. V tej funkciji bomo poskušali tudi poiskati poved, ki bi najbolj odgovorila na uporabnikovo vprašanje. Pri tem bomo prav tako uporabili statistiko *tf-idf*. V prvem koraku bomo uporabili funkcijo `splitTextIntoSentences(text)` znotraj paketa `textProcessor`, v kateri uporabljamo paket Sentence Tokenizer,[10] ki nam omogoča enostavno razbitje niza na posamezne povedi, čigar rezultat bomo uporabili pri nadaljnjem procesiranju:

```
1 var Tokenizer = require('sentence-tokenizer');
2
3 var splitTextIntoSentences = function (text) {
4     var tokenizer = new Tokenizer('Chuck');
5     tokenizer.setEntry(text);
6
7     return tokenizer.getSentences();
8 };
```

Izvorna koda 5.17: Funkcija za razbitje niza na posamezne povedi.

V nadaljevanju bomo za *tf-idf* procesiranje uporabili paket Natural,[11] ki omogoča enostavno delo s procesiranjem naravnega jezika. S pomočjo tega paketa bomo določili, katera poved se najbolj ujema s ključno besedo. Pri tem smo prav tako hevristično določili mejo za to, da lahko povemo, da je poved relevantna. Tu smo to mejo nastavili na $0,75$. V kolikor smo našli poved, ki bi lahko odgovorila na vprašanje, jo vrnemo, v ostalih primerih pa vrnemo vrednost `null`. Poleg odgovora vrnemo tudi sporočilo, v katerem poglavju lahko uporabnik prebere kaj več o želeni stvari in tudi URL povezavo do tega poglavja.

```
1 ...
2 data: function (neo4jResponse, keyword) {
3     if (neo4jResponse.length <= 0) throw Error('No results.');
4
5     var section = neo4jResponse[0]['section'].properties;
6     var chapter = neo4jResponse[0]['chapter'].properties;
```

[10] https://github.com/parmentf/node-sentence-tokenizer
[11] https://github.com/NaturalNode/natural

```
7       var readMoreTemplate = _.template('Vec o *<%= keyword %>* si lahko
            preberete v poglavju <%= number %> - <%= title %>');
8
9       var sentences = textProcessor.splitTextIntoSentences(
            section.rawContent);
10      var tfidf = new natural.TfIdf();
11      _.forEach(sentences, function (sentence) {
12          tfidf.addDocument(sentence);
13      });
14      var topMatch = {
15          i: -1,
16          measure: -1
17      };
18      tfidf.tfidfs(keyword, function(i, measure) {
19          if (measure > topMatch.measure) {
20              topMatch.i = i;
21              topMatch.measure = measure;
22          }
23      });
24      var answer = null;
25      if (topMatch.i >= 0 && topMatch.measure >= 0.75) {
26          answer = sentences[topMatch.i];
27      }
28
29      return {
30          answer: answer,
31          message: readMoreTemplate({
32              'keyword': keyword,
33              'number': section.number,
34              'title': section.title
35          }),
36          url: docsUrl + chapter.fileName + '#' + section.sectionId
37      };
38  },
39  ...
```

Izvorna koda 5.18: Funkcija za procesiranje rezultata poizvedbe za iskanje poglavja.

Kot zadnji izmed rokovalnikov je rokovalnik, ki procesira iskanje slik. Ta rokovalnik skuša poiskati sliko na podlagi ujemanja med ključnimi besedami

iz uporabnikovega vnosa ter opisom slike (HTML atribut `alt`). Najprej si zgradimo regularni izraz, ki ga bomo uporabili v poizvedbi, ki je prikazana na 5.16:

```
var regex =
    '.*' +
    message.match[1].split(' ').join('.*') +
    '.*';
```

Izvorna koda 5.19: Regularni izraz za iskanje slik.

```
MATCH (i:Image)
WHERE toLower(i.alt) =~ '!_REGEX'
RETURN i AS image
```

Izvorna koda 5.20: Poizvedba za iskanje slike.

Preden se poizvedba izvede, naredimo zamenjavo parametra `!_REGEX` z regularnim izrazom, ki se nahaja v spremenljivki `regex`, s pomočjo funkcije `replace(searchValue, replaceValue)`. V kolikor se poizvedba uspešno izvede, pričnemo z nadaljnjim procesiranjem rezultatov. V primeru, da smo našli samo eno sliko, vrnemo samo ta rezultat, sicer pa uporabimo *tf-idf* za iskanje najprimernejše izmed slik. Postopek iskanja najprimernejše slike je enak kot iskanje najprimernejše povedi za odgovor. V kolikor ne najdemo primerne slike izmed množice vseh, vrnemo kar prvo izmed vseh. Implementacija funkcije je prikazana na sliki 5.21.

```
...
data: function (neo4jResponse, keyword) {
    if (neo4jResponse.length <= 0) throw Error('No results.');

    var imageTemplate = _.template('*<%= alt %>*');

    if (neo4jResponse.length === 1) {
        return {
            message: imageTemplate({
                alt: neo4jResponse[0]['image'].properties['alt']
            }),
            url: docsUrl + neo4jResponse[0]['image'].properties['src']
        }
```

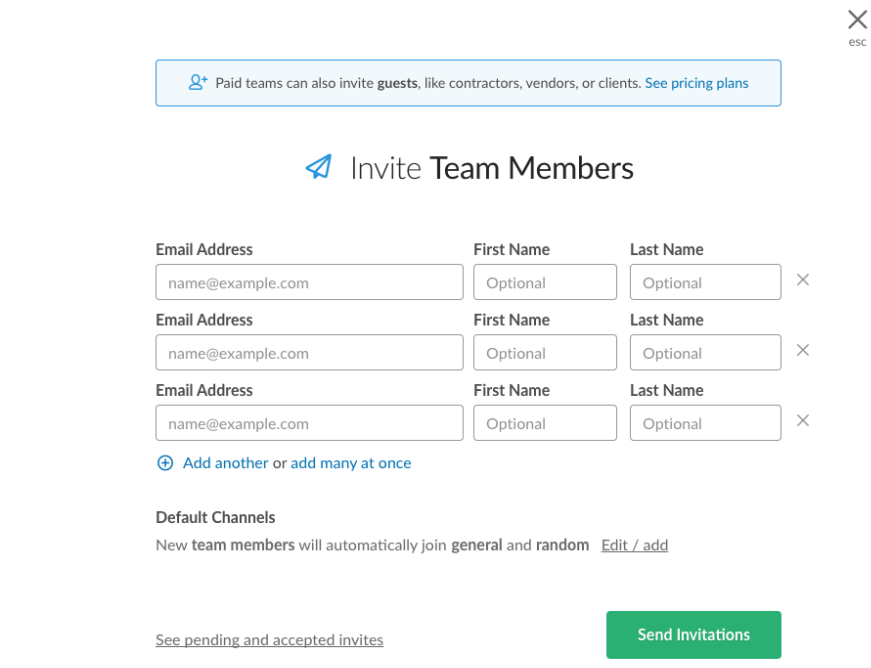
```
14      } else {
15          var tfidf = new natural.TfIdf();
16
17          for (var i = 0; i < neo4jResponse.length; i++) {
18              tfidf.addDocument(
19                  neo4jResponse[i]['image'].properties['alt']
20              );
21          }
22
23          var topMatch = {
24              i: 0,
25              measure: 0
26          };
27
28          tfidf.tfidfs(keyword, function(i, measure) {
29              if (measure > topMatch.measure) {
30                  topMatch.i = i;
31                  topMatch.measure = measure;
32              }
33          });
34
35          if (topMatch.i >= 0 && topMatch.measure >= 0.75) {
36              return {
37                  message: imageTemplate({
38                      alt: neo4jResponse[topMatch.i]['image']
39                          .properties['alt']
40                  }),
41                  url: docsUrl + neo4jResponse[topMatch.i]['image'].
                        properties['src']
42              }
43          } else {
44              throw Error('No results.');
45          }
46      }
47  },
48  ...
```

Izvorna koda 5.21: Funkcija za procesiranje rezultata poizvedbe za iskanje slike.

5.8 Pošiljanje povabil uporabnikom

Zadnji korak v procesu razvoja pametnega asistenta bo avtomatsko pošiljanje povabil uporabnikom za pridružitev Slack ekipi. Slack omogoča tri načine pošiljanja povabil, in sicer:

- ročno pošiljanje povabil znotraj aplikacije, kot je prikazano na sliki 5.4;
- pošiljanje univerzalne povezave za pridružitev ekipi, ki jo zgenerirajo na zahtevo na Slack podporni deski (angl. *support*);
- pošiljanje povabil z uporabo Slack API klicev.



Slika 5.4: Obrazec za pošiljanje povabil znotraj aplikacije Slack.

Pametni asistent bo uporabljal slednjega, saj lahko naenkrat hitro in enostavno pošljemo povabila vsem uporabnikom. Za ta namen bomo implementirali funkcijo, ki bo najprej iz nerelacijske podatkovne baze s pomočjo

poizvedbe pridobila seznam vseh študentov in v naslednjem koraku znotraj zanke pošiljala povabila uporabnikom s pomočjo API klica tipa `POST` na končno točko `https://TEAM.slack.com/api/users.admin.invite?t=TIMESTAMP`, pri čemer se `TEAM` in `TIMESTAMP` ustrezno zamenjata z imenom naše ekipe ter trenutnim časovnim žigom. V telesu `POST` zahtevka podamo dva parametra, in sicer `email` uporabnika, ki mu želimo poslati povabilo, ter `token`, ki je identifikator našega pametnega asistenta na platformi Slack.

```
1  var q       = require('q');
2  var request = require('request');
3
4  var sendInvite = function(email, team, token){
5      var deferred = q.defer();
6      var data = {
7          email: email,
8          token: token
9      };
10     var time = new Date().getTime();
11     var endpoint =
12         'https://' +
13         team +
14         '.slack.com/api/users.admin.invite?t=' +
15         time;
16
17     request.post(endpoint, { form: data }, function (error, response) {
18         error !== null ? deferred.reject(error) : deferred.resolve(
               response);
19     });
20
21     return deferred.promise;
22 }
```

Izvorna koda 5.22: Funckija za avtomatsko pošiljanje Slack povabil.

Poglavje 6

Testiranje

Izdelano rešitev smo dali v testiranje študentom, ki so bili vpisani v predmet Osnove informacijskih sistemov v študijskem letu 2016/2017, teh je bilo 221. Analizo testiranja bomo opravili s pomočjo zapisov v podatkovni bazi, kot smo opisali v poglavju 5.5.

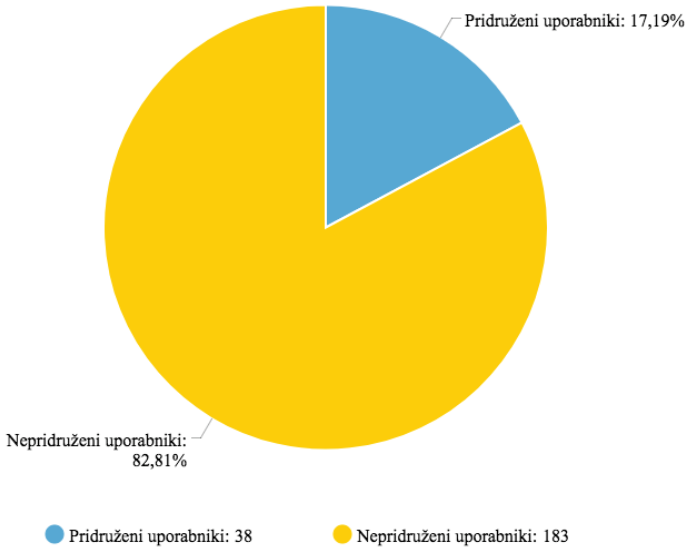
Fazo testiranja smo začeli 8. 6. 2017, ko smo programsko poslali povabila študentom, in končali 20. 7. 2017 (skupaj 43 dni). Ob tem smo naleteli na omejitev aplikacije Slack, saj ima omejitev števila poslanih povabil. Ta omejitev se nanaša na scenarij, ko smo naenkrat poslali mnogo povabil in pri tem je velika večina povabil ostala odprta, tj. uporabniki se niso odzvali na njih. Tako smo lahko naenkrat poslali le 86 povabil in smo se zaradi te omejitve morali obrniti na pomoč, ki jo nudi Slack. Tam so nam ustvarili univerzalno povezavo za pridružitev ekipi, ki jo je nosilec predmeta posredoval študentom preko foruma na spletni učilnici. Poudarimo lahko še to, da se v fazi testiranja aplikacija ni niti enkrat sesula ali bila nedosegljiva in neodzivna.

6.1 Analiza zbranih podatkov

V nadaljevanju bomo predstavili podrobnejšo analizo zbranih podatkov po posameznih tipih.

6.1.1 Število uporabnikov

Kot smo že omenili v uvodu tega poglavja, je skupno 221 študentov vpisanih v predmet Osnove informacijskih sistemov. Od teh 221 študentov smo 86 študentom poslali povabilo na njihov e-mail naslov, ostali so pa imeli možnost pridružitve ekipi in testiranja pametnega asistenta preko univerzalnega povabila, ki je bilo objavljeno na forumu na spletni učilnici. Na povabila, poslana na e-mail naslov, se je odzvalo 18 študentov in naknadno preko univerzalnega povabila še 20, kar pomeni, da se je ekipi pridružilo 17, 19 % študentov, vpisanih v predmet. Tortni diagram deležev je prikazan na sliki 6.1.



Slika 6.1: Delež pridruženih in nepridruženih uporabnikov ekipi.

6.1.2 Aktivnost uporabnikov

Od 38 pridruženih uporabnikov je pametnega asistenta uporabljalo (zastavilo vsaj eno vprašanje) 30 uporabnikov, kar pomeni, da je pametnega asistenta uporabljalo 78, 95 % pridruženih uporabnikov. To se na prvi pogled zdi visoka številka, vendar če pogledamo absolutno število uporabnikov, vidimo, da je

30 uporabnikov zelo malo v primerjavi z maksimalnim možnim potencialom 221 uporabnikov (le $13,57$ %).

Uporabniki so skupaj zastavili 184 vprašanj. V tabeli 6.1 lahko vidimo število zastavljenih vprašanj posameznega uporabnika. Vidimo lahko, da je samo 5 uporabnikov zastavilo 10 ali več vprašanj, ostali so pa bolj kot ne le odprli aplikacijo Slack in nekaj poskušali vprašati (v podpoglavju 6.1.3 bomo natančneje analizirali vprašanja). Iz tabele 6.1 lahko izračunamo naslednje statistične vrednosti (upoštevamo samo tiste uporabnike, ki so uporabljali pametnega asistenta):

- povprečno število vprašanj na uporabnika: $\bar{x} = \frac{||v||}{||u||} = \frac{184}{30} = \underline{\underline{6,13}}$
- mediana: $Me = \frac{5+5}{2} = \underline{\underline{5}}$
- varianca: $\sigma^2 = \frac{\sum_{i=1}^{N}(x_i - \bar{x})^2}{N} = \frac{529,47}{184} = \underline{\underline{2,89}}$
- standardni odklon: $\sigma = \sqrt{\sigma^2} = \sqrt{2,89} = \underline{\underline{1,70}}$

V tabeli 6.2 vidimo, da je $83,33$ % uporabnikov uporabljalo pametnega asistenta samo prvi dan. Iz tega lahko sklepamo dve stvari: prva je ta, da so samo želeli poskusiti, kaj zmore pametni asistent in niso videli dodane vrednosti pri učenju ali pridobivanju splošnih informacij; druga pa je ta, da se jim je zdel pametni asistent zelo omejen v tem smislu, da so imeli prevelika pričakovanja in da jih pametni asistent ni prepričal v to smer, da bi ga uporabljali še naprej. Razlog za neuporabo pametnega asistenta lahko mogoče najdemo tudi v tem, da se je testiranje začelo na koncu semestra, saj se je semester končal 9. 6. 2017, kar je en dan po začetku testiranja. Uporabnika z identifikatorjema `U5QJGSBCM` in `U5R5X5A91`, ki sta pametnega asistenta uporabljala po 18 in 28 dneh, sta se na dan začetka obdobja testiranja samo pridružila ekipi in šele po 18 in 28 dneh prvič zastavljala vprašanja pametnemu asistentu. Oba uporabnika sta zastavljala vprašanja v obdobju treh minut in po treh minutah nikoli več nista uporabljala pametnega asistenta. Le uporabniki z identifikatorji `U5R0WLRK6`, `U5PUL7SE5` in `U5R3C018T` so pametnega asistenta uporabljali več kot en dan.

6.1.3 Zastavljena vprašanja

Pametni asistent je vsa vprašanja, pri katerih ni naletel na napako, pravilno delegiral rokovalnikom. Pred začetkom testiranja smo upali, da bo večina vprašanj povezanih s snovjo pri predmetu, vendar se je po končanem testiranju izkazalo, da se je pametni asistent uporabljal predvsem za povpraševanje po podatkih o študentih. Takšnih vprašanj je bilo vsega skupaj 109 (59,24 %); vprašanj, povezanih s snovjo pri predmetu, je bilo 59 (44,02 %); 16 vprašanj pa se je nanašalo na zahtevo po prikazu navodil za uporabo pametnega asistenta. To, da je pametni asistent obdelal več zahtevkov za povpraševanje po podatkih o študentih, lahko pripišemo temu, da se je testiranje začelo ob koncu semestra, ko študente bolj zanimajo rezultati njihovega dela čez semester. V tabeli 6.3 so prikazani podatki o številu vprašanj, ki jih je obdelal posamezni rokovalnik.

Med 184 zastavljenimi vprašanji je pametni asistent naletel na napako 21-krat, kar je 11,41 %. V teh 21 primerih je prišlo 12-krat do napake zaradi tega, ker se uporabniki niso pridružili ekipi z dogovorjenim e-mail naslovom in zaradi tega pametni asistent ni našel podatkov o uporabnikih. 2-krat so uporabniki zastavili vprašanje, ki ni bilo v povezavi s predmetom (primer: „Kaj je Scala?“), 7-krat pa so uporabniki zastavili nerelevantna vprašanja, iz česar lahko sklepamo, da so želeli preizkusiti odpornost pametnega asistenta na taka vprašanja.

ID uporabnika	N
U5R8H39U4	22
U5QSU1TTM	13
U5R0WLRK6	11
U5R2VN71B	10
U5RSEKEP9	10
U5R0LC3A7	9
U5VFBHATC	9
U5S87D0P9	8
U5QEH32JX	8
U5R5X5A91	7
U5RDN896Z	6
U5PUL7SE5	6
U5UP3K4T1	6
U68N3UCJK	6
U5RDN9727	5
U5TGBCJTY	5
U5QJGSBCM	5
U5R599DQB	5
U5PV7JH4H	4
U5S704E2W	4
U5S0J5E9L	4
U5R19LUAH	3
U5RF9CGEP	3
U5R3C018T	3
U5R85RWDQ	3
U5RATCG9G	2
U5RTFQMM0	2
U5S5YFG2W	2
U5QN991V2	2
U5R4EMG67	1

Tabela 6.1: Število zastavljanih vprašanj posameznega uporabnika.

ID uporabnika	N
U5RTFQMM0	0
U5S0J5E9L	0
U5S5YFG2W	0
U5PV7JH4H	0
U5R4EMG67	0
U5S704E2W	0
U5QEH32JX	0
U5R8H39U4	0
U5TGBCJTY	0
U5R19LUAH	0
U5R599DQB	0
U5UP3K4T1	0
U5R85RWDQ	0
U5VFBHATC	0
U5QSU1TTM	0
U5RDN896Z	0
U5R2VN71B	0
U5RATCG9G	0
U5QN991V2	0
U5S87D0P9	0
U68N3UCJK	0
U5RSEKEP9	0
U5RDN9727	0
U5R0LC3A7	0
U5RF9CGEP	0
U5R0WLRK6	0,73
U5PUL7SE5	1
U5R3C018T	5,33
U5QJGSBCM	18
U5R5X5A91	28

Tabela 6.2: Povprečno število pretečenih dni od pridružitve ekipi in zastavljanju vprašanj.

Naloga	**N**
Iskanje poglavja	38
Podatki o uporabniku	22
Razlaga kratice	18
Končna ocena predmeta	16
Pomoč	16
Točke sodelovanja	13
Ali je študent uspešno opravil predmet	12
Informacije glede prijave na izpit	9
Ali ima študent priznan pisni izpit	7
Povprečje domačih nalog	6
Dosežene točke na določeni domači nalogi	6
Povprečje preverjanj znanj	5
Število predlaganih popravkov	4
Ali je študent opravil obveznosti preverjanj znanj	4
Iskanje slike	3
Ali je študent opravil obveznosti domačih nalog	2
Ali je študent opravil pisni izpit	2
Dosežene točke na določenem preverjanju znanja	1

Tabela 6.3: Število obdelanih zahtevkov po posameznih tipih rokovalnika.

Poglavje 7

Sklepne ugotovitve

Cilj diplomskega dela je bil razviti prototip pametnega asistenta, ki bi ga uporabljali študenti pri določenem predmetu kot pripomoček za pomoč pri pridobivanju osnovnih informacij o predmetu in ocenah, ki so jih dosegli, ter iskanje informacij iz gradiva. To bi pomenilo razbremenitev za izvajalce predmeta, saj bi študenti najprej povprašali pametnega asistenta in šele ob nezadostnem odgovoru pametnega asistenta nato kontaktirali izvajalce.

V uvodnem poglavju smo predstavili dva slovenska pametna asistenta, povzetek raziskave o pametnih asistentih, ki ga je izvedlo britansko podjetje Ubisend, vpeljavo pametnega asistenta pri letalskem podjetju KLM in problem v procesu izobraževanja, ki bi ga lahko (delno) rešili s pomočjo pametnih asistentov.

Za tem je sledilo teoretično poglavje o pametnih asistentih, v katerem smo najprej definirali, kaj je pametni asistent, in za tem predstavili zgodovinsko ozadje le-teh. Predstavili smo Turingov test in dva primera pametnih asistentov, ELIZA in A.L.I.C.E. Na koncu tega poglavja smo opisali uporabo pametnih asistentov v praksi in navedli povzetke ostalih raziskav vpeljave pametnih asistentov v proces izobraževanja.

V tretjem poglavju smo navedli in na kratko opisali tehnologije in knjižnice, ki so nam pomagale pri implementaciji prototipa.

Temu poglavju je sledilo poglavje, v katerem smo predstavili teoretično

ozadje implementacije, kot so vrsta in viri vhodnih podatkov ter algoritmi in postopki za obdelavo teh podatkov. Uporablili smo algoritma *Term frequency-inverse document frequency* in razdaljo *Jaro-Winkler* ter postopek obdelave besed dokumenta, lematizacija.

Osrednji del diplomskega dela je bilo poglavje s podrobnostmi implementacije. V tem poglavju smo najprej predstavili razvojno in produkcijsko okolje (na osnovi Dockerja), za tem je sledila predstavitev lastno razvitega paketa za procesiranje teksta `textProcessor`. V naslednjem podpoglavju smo predstavili postopek razčlenjevanja podatkov. Za tem smo opisali način, kako smo dostopali do podatkov o študentih in kako smo definirali podatkovni model. Tu smo podrobneje predstavili tudi, za kaj bomo shranjevali in uporabljali posamezni stolpec v relacijski podatkovni bazi in posamezni atribut v podatkovnem modelu v obliki grafa znotraj nerelacijske podatkovne baze. Temu je sledila predstavitev implementacije algoritma *Jaro-Winkler* znotraj funkcije za prepoznavanje in delegiranje ukazov posameznemu rokovalniku. V zadnjem delu tega poglavja smo podrobneje predstavili, kako smo procesirali različne tipe ukazov in na kakšen način smo izluščili parametre vprašanja ter iskali po domeni.

V naslednjem poglavju smo predstavili način testiranja razvitega prototipa in analizirali zbrane podatke. Ugotovili smo, da je bila uporaba pametnega asistenta nizka, kar bi lahko bila posledica tega, da se je testiranje začelo en dan pred koncem obdobja pedagoškega dela (semestra).

Menimo, da bi lahko že trenutni prototip bil v veliko pomoč izvajalcem predmetov, da bi jih delno razbremenil. Predpostavljamo, da bi tovrstna rešitev predstavljala tudi večje zanimanje študentov za sodelovanje pri predmetu, saj bi imeli občutek, da izvajalci sledijo trendom na področju komunikacije in da ne uporabljajo samo „zastarelih“ načinov komunikacije, kot sta npr. e-mail dopisovanje in diskusija na forumu.

7.1 Predlogi za izboljšave

Trenutni prototip pametnega asistenta ima še veliko prostora za izboljšave. Nekatere izmed izboljšav bomo navedli in predstavili v nadaljevanju:

- **izboljšava razčlenjevalnika za gradivo:** razčlenjevalnik bi dodelali do te mere, da bi bolje prepoznal tipe besedila (izsek kode, definicije, letnice itd.);
- **dodelava poizvedbe za iskanje po gradivu:** v primeru iskanja po več ključnih besedah bi izbrali poglavje, ki ima večji presek množic ključnih besed (ključne besede poglavja in ključne besede uporabnikovega vprašanja);
- **osveževanje podatkov na zahtevo:** namesto ponovnega zagona celotne aplikacije bi naredili rokovalnik, ki bi sprožil ponovno razčlenjevanje podatkov, do katerega bi dostopale samo avtorizirane osebe;
- **spletna aplikacija:** s pomočjo spletne aplikacije bi lahko nosilci predmeta lažje urejali podatke o gradivu in študentih in imeli vpogled v statistiko vprašanj;
- **vrednotenje vrnjenih odgovorov:** implementirali bi mehanizem, ki bi služil za ocenjevanje (uporabniki) točnosti vrnjenega odgovora;
- **strojno učenje:** pametnemu asistentu bi dodali logiko, ki bi mu omogočala učenje iz dosedanjih odgovorov (na podlagi ocen uporabnikov), da bi izboljšali bodoče odgovore na podobna vprašanja;
- **optično prepoznavanje besedila:** dodali bi OCR logiko za prepoznavanje teksta na slikah, ki bi nam pomagal pri natančnejšem iskanju slik.

Slike

Tabele

Izvorne kode

Literatura

[1] Luciana Benotti, María Cecilia Martínez, and Fernando Schapachnik. Engaging high school students using chatbots. In *Proceedings of the 2014 conference on Innovation & technology in computer science education*, pages 63–68. ACM, 2014.

[2] Botkit dokumentacija. Dosegljivo: `https://github.com/howdyai/botkit/blob/master/readme.md`, 2017. [Dostopano: 19. 5. 2017].

[3] Docker. Dosegljivo: `https://www.docker.com/what-docker`, 2017. [Dostopano: 19. 5. 2017].

[4] Sašo Dzeroski and Tomaz Erjavec. Strojno učenje lematizacije neznanih slovenskih besed. In *v zborniku Jezikovne tehnologije: zbornik konference, Institut Jozef Stefan, Ljubljana*, pages 14–30, 2000.

[5] LK Fryer and Rollo Carpenter. Bots as language learning tools. *Language Learning & Technology*, 2006.

[6] Bob Heller, Mike Procter, Dean Mah, Lisa Jewell, and Billy Cheung. Freudbot: An investigation of chatbot technology in distance education. In *Proceedings of the World Conference on Multimedia, Hypermedia and Telecommunications*, pages 3913–3918, 2005.

[7] Alice Kerly, Phil Hall, and Susan Bull. Bringing chatbots into education: Towards natural language negotiation of open learner models. *Knowledge-Based Systems*, 20(2):177–185, 2007.

[8] KLM. Dosegljivo: `https://messenger.klm.com`, 2017. [Dostopano: 22. 5. 2017].

[9] Christopher D Manning, Prabhakar Raghavan, Hinrich Schütze, et al. *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge, 2008.

[10] MySQL. Dosegljivo: `https://dev.mysql.com/doc/refman/5.7/en`, 2017. [Dostopano: 19. 5. 2017].

[11] Neo4j. Dosegljivo: `https://neo4j.com/product`, 2017. [Dostopano: 19. 5. 2017].

[12] Node.js. Dosegljivo: `https://nodejs.org/en/about`, 2017. [Dostopano: 19. 5. 2017].

[13] Anand Rajaraman and Jeffrey David Ullman. *Mining of Massive Datasets*. Cambridge University Press, 2011.

[14] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph databases*. "O'Reilly Media, Inc.", 2013.

[15] Bayan Abu Shawar and Eric Atwell. Chatbots: are they really useful? In *LDV Forum*, volume 22, pages 29–49, 2007.

[16] Slack. Dosegljivo: `https://slack.com`, 2017. [Dostopano: 19. 5. 2017].

[17] Troljo chatbot. Dosegljivo: `http://troljo.si`, 2017. [Dostopano: 22. 5. 2017].

[18] Alan M Turing. Computing machinery and intelligence. *Mind*, 59(236):433–460, 1950.

[19] James Turnbull. *The Docker Book: Containerization is the new virtualization*. James Turnbull, 2014.

[20] Raziskava podjetja ubisend o mobilni komunikaciji s podjetji. Dosegljivo: http://assets.ubisend.com/insights/ubisend_2016_Mobile_Messaging_Report.pdf, 2017. [Dostopano: 22. 5. 2017].

[21] VIDA Virtualna davčna asistentka. Dosegljivo: http://www.eracunovodstvo.org/blog/pripomocki/vida-virtualna-davcna-asistentka, 12 2009. [Dostopano: 22. 5. 2017].

[22] Richard S Wallace. The anatomy of alice. In *Parsing the Turing Test*, pages 181–210. Springer, 2009.

[23] Joseph Weizenbaum. Eliza—a computer program for the study of natural language communication between man and machine. *Communications of the ACM*, 9(1):36–45, 1966.

[24] Wikipedia. Chatbot — wikipedia, the free encyclopedia. Dosegljivo: https://en.wikipedia.org/w/index.php?title=Chatbot&oldid=791131230, 2017. [Dostopano 31. 7. 2017].

[25] Wikipedia. Jaro–winkler distance — wikipedia, the free encyclopedia. Dosegljivo: https://en.wikipedia.org/w/index.php?title=Jaro\OT1\textendashWinkler_distance&oldid=786702122, 2017. [Dostopano: 22. 6. 2017].

[26] Wikipedija. Lematizacija — wikipedija, prosta enciklopedija. Dosegljivo: https://sl.wikipedia.org/w/index.php?title=Lematizacija&oldid=3902158, 2017. [Dostopano: 22. 6. 2017].

[27] William E Winkler. Overview of record linkage and current research directions. In *Bureau of the Census*. Citeseer, 2006.