Vizualizacija strukturiranih podataka u Pythonu

Pera, Fabio

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Fabio Pera

Vizualizacija strukturiranih podataka u Pythonu

ZAVRŠNI RAD

Varaždin, 2019.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Fabio Pera

Matični broj: 44569/16–R

Studij: Informacijski sustavi

Vizualizacija strukturiranih podataka u Pythonu

ZAVRŠNI RAD

Mentor:

Dr. sc. Miran Zlatović

Varaždin, rujan 2019.

Fabio Pera

Izjava o izvornosti

Izjavljujem da je moj završni rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

Tema rada će biti usporedba strukturiranih i nestrukturiranih podataka s naglaskom na njihove prednosti i nedostatke. Biti će objašnjen način njihova pohranjivanja u memoriju (i u čemu se razlikuju). Prikazani će biti modeli podataka kao primjer strukturiranih podataka. Kao dio strukturiranih podataka biti će definirani polustrukturirani podaci (JSON, XML, CSV). U praktičnom dijelu biti će prikazana obrada i vizualizacija strukturiranih podataka pomoću programskog jezika Python i Jupyter Notebook okruženja uz korištenje nekih specijaliziranih biblioteka kao što su Pandas, Matplotlib, JSON, CSV i XML.

Ključne riječi: python, strukturirani podaci, vizualizacija podataka

Sadržaj

Sadržaj iii
1. Uvod 1
2. Metode i tehnike rada 2
3. Strukturirani podaci 4
3.1. Modeliranje podataka 4
3.1.1. Tipovi modela podataka 6
3.2. Rad s relacijskim modelom podataka 7
3.2.1. Vizualizacija relacijskog modela podataka 11
3.3. Nestrukturirani podaci 12
4. Polustrukturirani podaci 14
4.1. XML 14
4.1.1. Struktura XML zapisa 15
4.1.2. Parsing XML zapisa 16
4.1.2.1. Opasnosti prilikom XML parsinga 18
4.1.3. Vizualizacija XML zapisa 18
4.2. JSON 23
4.2.1. Usporedba JSON i XML 24
4.2.2. Parsing JSON zapisa 25
4.2.3. Vizualizacija JSON zapisa 26
4.2.4. JSON i YAML 30
4.3. CSV 30
4.3.1. Parsing CSV zapisa 31
4.3.2. Vizualizacija CSV zapisa 32
5. Zaključak 35
Popis literature 37
Popis slika 40

1. Uvod

Tema ovog završnog rada je vizualizacija strukturiranih podataka unutar programskog jezika Python. Podaci i njihovo prikupljanje, strukturiranje i vizualiziranje su tema koja postaje sve aktualnija zbog eksponencijalnog rasta prikupljenih podataka u svrhe analitike tržišta.

S obzirom na velik porast prikupljenih podataka javlja se sve veća potreba za njegovim strukturiranjem zbog olakšavanja kasnije obrade. Vizualizacija podataka danas se ne može više smatrati prezentacijskim alatom već alatom analize podataka.

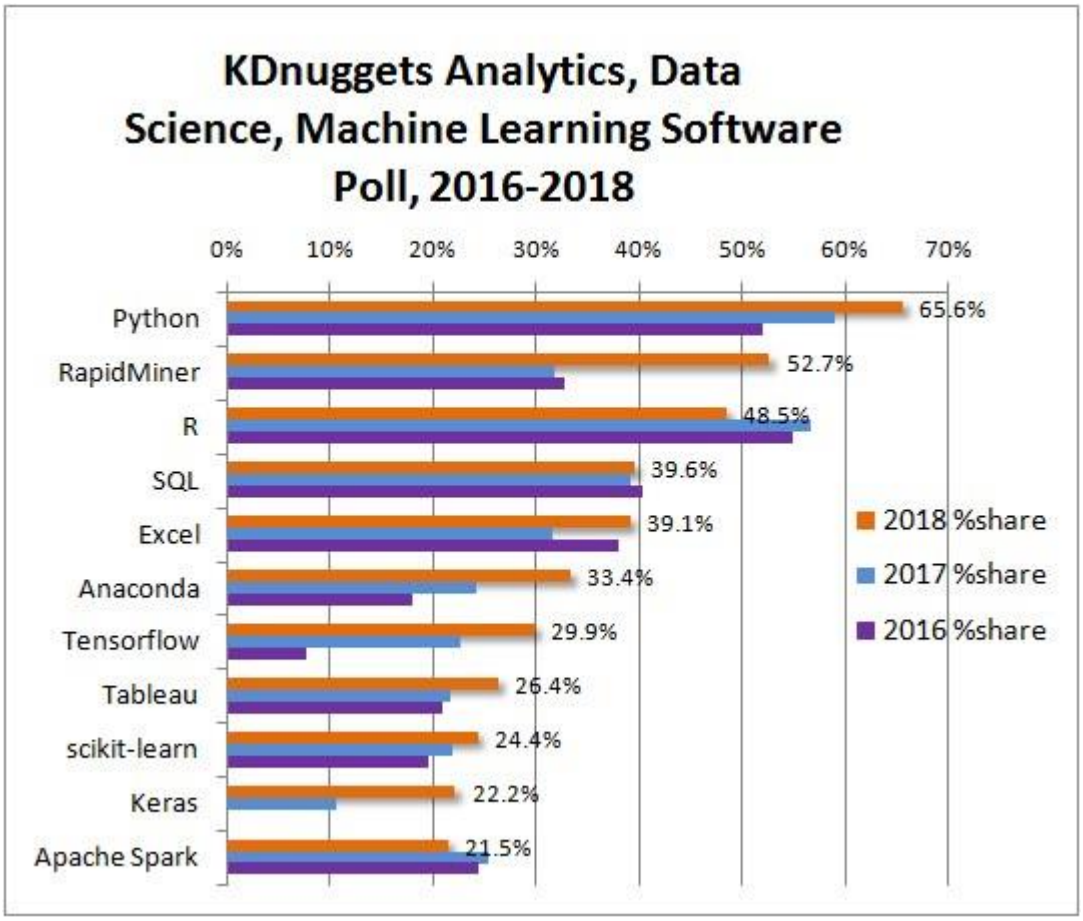
Pojam vizualizacije podataka posebno je eksponiran nakon sve šire primjene pojma velikih podataka (eng. *Big Data)* među analitičarima i programerima. *Big Data* je pojam koji podacima daje veći značaj od onog dodijeljenog kao entitetu baze podataka te mu koristeći neke naprednije tehnike obrade (u novije vrijeme sve više umjetna inteligencija) daje interpretaciju. Kada govorimo o Big Data skupu podataka govorimo o veličini od minimalno nekoliko TB podataka pa sve do nekoliko PB (petabyte) s kojima raspolažu neki servisi najvećih kompanija. Podatke možemo podijeliti u tri grupe: strukturirane podatke, polustrukturirane (ili pseudostrukturirane) i nestrukturirane podatke.[1]

Vizualizacija podataka je također značajno napredovala u zadnjih nekoliko godina te se više ne svodi na stupčaste ili tortne dijagrame karakteristične za Microsoft Excel, nego na veći broj uže specijaliziranih dijagrama pogodnih za interpretaciju konkretnog problema. U svrhu obrade sve veće količine podataka obrada se prebacuje na programske jezike koji se nadograđuju posebnim modulima za vizualizaciju podataka. Takvi moduli omogućavaju lakšu komunikaciju sloja obrade i prezentacije pa stoga ne čudi kako posljednjih godina sve veći broj analitičara i znanstvenika koji nisu dio programske struke se izučava za rad u tim alatima.

Slika 1: Eksponencijalni rast podataka u posljednjih nekoliko godina (Izvor: IDC, 2014)

2. Metode i tehnike rada

Pri razradi teme vizualizacije strukturiranih podataka koristit ću programski jezik Python. Python je interpreterski programski jezik čija je popularnost naglo porasla zbog velikog broja specijaliziranih biblioteka za razna područja primjene, kao i zbog jednostavne sintakse koja olakšava početnicima lakši proces učenja. Kako Python ima širok spektar biblioteka postoje brojne koje se bave problematikom obrade i vizualizacije podataka (te su biblioteke ujedno i najpoznatije). Zbog tih razloga Python je danas među vodećim jezicima unutar domene podatkovne obrade uz neke jezike koji se specijaliziraju za statističku obradu (R, MatLab) i naravno sveprisutni SQL (podatci se i dalje uglavnom spremaju u relacijske baze podataka). [2]



Slika 2: Tržišni udjel softwarea za obradu podataka (Izvor: Gregory Piatetsky, 2018)

Biblioteke koje se široko rasprostranjene u ovoj domeni su SciPy i NumPy kao predvodnici skupine biblioteka koje koristimo za napredne matematičke funkcije (posebno učestalo kod problema umjetne inteligencije jer uvelike olakšava matematičke operacije nad matricama). Kod same vizualizacije podataka najviše se koristi biblioteka Matplotlib koja

omogućava prikazivanje podataka kroz mnogo različitih grafova i drugih grafičkih pomagala. Kod obrade podataka korisna će nam biti biblioteka Pandas koja je biblioteka visokih performansi za obradu i analizu podataka. Također kako ćemo uglavnom raditi sa strukturiranim podacima (ili polustrukturiranim) od pomoći će nam biti biblioteke specijalizirane za određene tipovi zapisa (postoje biblioteke koje se bave s CSV, XML ili JSON zapisima).

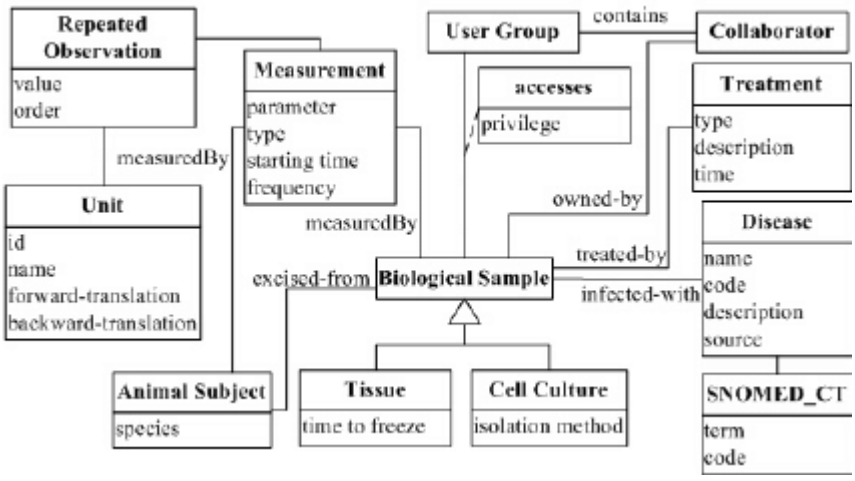
Prezentacijski dio obrade teme biti će obrađen kroz razvojno okružje Jupyter Notebooka zbog njegove jednostavnosti i prilagođenosti mogućnostima prezentiranja. Najveća prednost koju imamo koristeći Jupyter u odnosu na, npr. standardni Python IDLE, što možemo paralelno pregledavati više različitih ideja vizualizacije i uspoređivati ih. Također s obzirom da ćemo koristiti velik broj biblioteka koje nisu standardni dio Python 3 jezika koristit ćemo Anaconda razvojnu platformu kako bi olakšali povezivanje i dodavanje dodatnih biblioteka preko Anacondinog naredbenog retka (značajka posebno važna za korisnike Windows operacijskih sustava).

3. Strukturirani podaci

Strukturirani podaci su strogo definirana organizacija elemenata koja definira njihov zapis i ponašanje. Kada pričamo o strukturiranim podacima najčešće ćemo spominjati pojam model podataka. On definira logičku strukturu baze podataka koju ćemo koristiti. Termin modela podataka se odnosi na apstrakciju i formalizaciju entiteta koji se nalaze u problemskoj domeni. Glavna zadaća modela podataka je pružanje podrške informacijskom sustavu kroz definiranje formata zapisa. Važna je konzistentnost podataka kako bi aplikacija lakše mogla pristupati i dijeliti podatke s drugim aplikacijama. Model podataka se često grafički prikazuje jezikom modeliranja kao što je UML. [3] Dobro definiran model podataka je danas preduvjet za bilo kakvu aplikaciju. Također, treba razumjeti da je model podataka uvijek podrška, a ne svrha same aplikacije. Kada pričamo o primjeni modela podataka za aplikacije informacijskih sustava treba naglasiti da su logička struktura kao i samo podaci preslika stvarnosti i nisu svrha sami sebi.

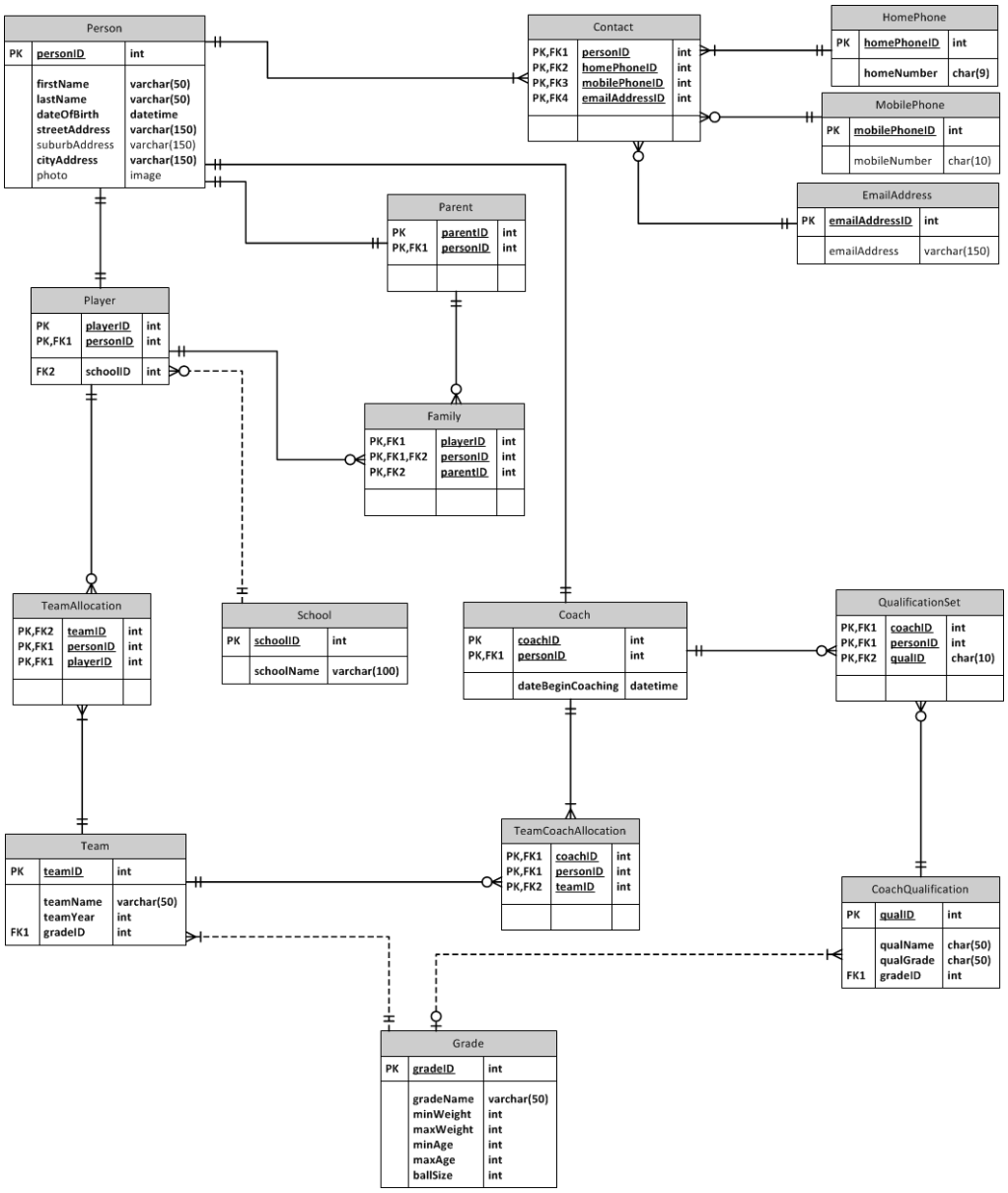
3.1. Modeliranje podataka

Proces modeliranja podataka je proces koji započima analizu specifikacija projekta, a rezultira gotovim modelom podataka. Početne zahtjeve projekta (zahtjeve korisnika) zapisujemo kao konceptualni model podataka. Konceptualni model podataka je skup zahtjeva neovisan o tehnologijama implementacije koji će služiti za daljnji razvoj projekta. On ne sadrži detalje implementacije (tipovi podataka, memorijske potrebe) već služi kao grubi prikaz korisničkih specifikacija. [4] Shema koju smo dobili s konceptualnim modeliranjem nam je kasnije korisna jer prikazuje potrebne entitete baze podataka.



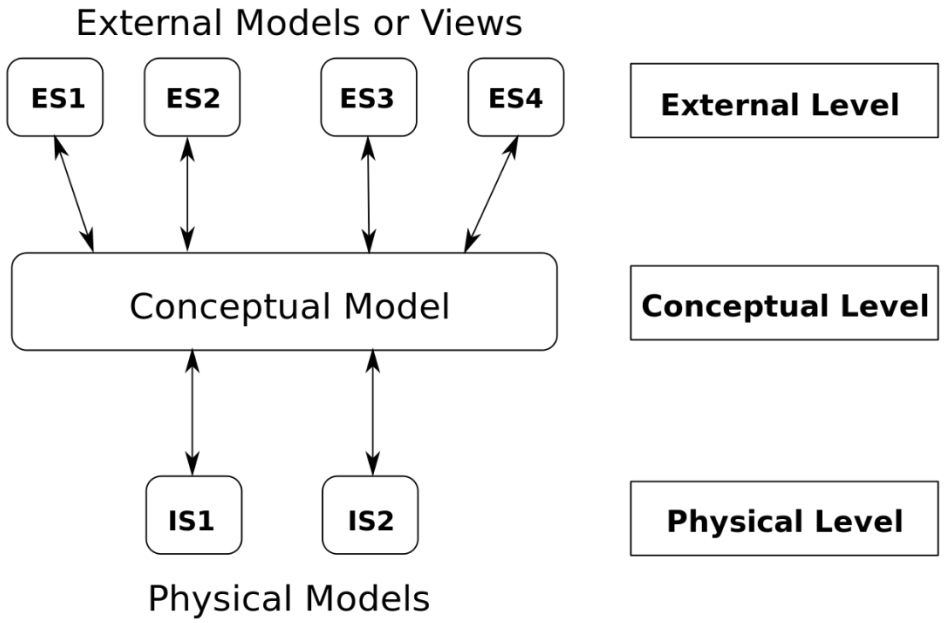
Slika 3: Konceptualni model podataka (Izvor: Thodoros Topaloglou, 2007)

Sljedeći korak je pretvaranje konceptualnog modela u logički. Ova dva modela su jako povezana te se na nekim manjim projektima lako može dogoditi da se ova dva koraka odrade simultano. Logički model dodaje odnose između entiteta (kardinalnost veza) i definira tipove podataka za implementaciju. Finalni produkt logičkog modela je model koji ne treba nadograđivati nego ga je potrebno „prevesti“ u bazu podataka u nekom od sustava za upravljanje bazom podataka. Logički model ulazi u područje tehnologije potrebne za implementaciju. Važnost logičkog modela je osim u implementacijskom dijelu i u dokumentaciji. Naime ukoliko na jednom projektu radi više programera (i osoba zaduženih za baze podataka) važno je da je logički model dovoljno jasan i jednoznačan kako ne bi izazvao konfuziju jer je upravo on referentna točka za implementaciju.



Slika 4: Logički model podataka (Izvor: Rob Atfield, 2012)

Fizički model podataka je ništa drugo nego implementacija logičkog modela u nekom od alata za rad s bazom podataka. Ukoliko je logički model bio ispravan, implementacija fizičkog nebi trebala biti problem jer bi on trebao se odnositi s logičkim 1:1 (poklapanje svih elemenata).



Slika 5: Slojevi aplikacije (Izvor: M.West i J.Fowler, 1999)

3.1.1. Tipovi modela podataka

Tipovi modela podataka određuje strukturu baze podataka kao i njenu logiku. Različiti tipovi modela podataka očituju i razvoj različitih modela podataka kroz vrijeme kao i njihov napredak. Iako su neki modeli podataka danas u toliko širokoj upotrebi da bi ih mogli nazvati i standardnim uvijek postoje slučajevi kada je neki drugi model efikasniji. Koji model podataka će se koristiti ovisi o tipu podataka koji koristimo, ali i o znanju pojedinca. Također tipovi modela podataka su uvjetovani i sustavima za upravljanje bazama podataka koji najčešće podržavaju jedan tip modela.

Jedan od najosnovnijih tipova modela podataka je hijerarhijski. Riječ je o jednostavnoj strukturi koja organizira podatke u strukturu stabla gdje je gornji čvor (list) nadređen donjem. Ovakav jednostavan prikaz je pogodan za neke jednostavnije probleme, a danas se koristi u zapisima kao što su XML i JSON (iako različiti oba imaju hijerarhijsku strukturu). [6]

Nadogradnju na osnovni hijerarhijski tip donosi mrežni tip. Mrežni tip dohvaća podatke pomoću pokazivača tvoreći tako strukturu usmjerenog grafa. Shema ovakvog prikaza se sastoji od „set“ tipova podataka. Set tipovi su entiteti povezani vezama odnosa. [7] Sličan princip ovome ima i relacijski model, ali on diže to na višu razinu. U logičkom modeliranju

ovakav tip podataka je često prikazan Bachmannovim dijagramom (prema tvorcu samog koncepta). Koncept Bachmannovog dijagrama je da su u pravokutnicima označeni entiteti koje povezujemo strelicama koje označavaju kardinalnost veze (predstavljaju veze jedan na više jer se teži izbjegavanju veza više na više). [8]

Relacijske baze podataka su i danas najzastupljenije iako je od njihovog začetka prošlo skoro 50 godina (1970. E.Codd). Baziraju se na relacijskoj algebri. Osnovni koncept relacijskih baza podataka su tablice u koje spremamo podatke o entitetima, a povezujemo ih s drugim entitetima vezama različite kardinalnosti. Svaka tablica sadrži atribute, a od posebnog su značaja atributi primarnog i vanjskog ključa koji utječu na integritet baze podataka. To znači da kako bismo povezali dvije tablice u relaciju moramo povezati primarni ključ s vanjskim ključem druge tablice koji se moraju podudarati u tipu zapisa (ne možemo spajati brojčane vrijednosti s tekstualnim). Primarni ključ tablice mora jednoznačno označavati tablicu, a ukoliko on nije preslika stvarnog svijeta (npr. serijski broj šasije na automobilu) možemo ga postaviti proizvoljno (najčešće kao inkrementirajuću brojčanu vrijednost). Za rad s relacijskim bazama podataka koristi se SQL upitni jezik. [9]

Graf baze podataka su novije od relacijskih i ne koriste SQL kao upitni jezik što ih svrstava u NoSQL grupu modela. Za razliku od relacijskih baza podataka koje kao temelj koriste relacijsku algebru, kod graf baza je to teorija grafova. Razliku u pristupu ovakvog modela podataka u odnosu na prethodne je u tome da se fokus umjesto na podatke stavlja na veze među podacima (entitetima). Zbog toga su ovakve baze brže za pretraživanje. Pohranjivanje podataka u ovakvim bazama može varirati. Dok neke po uzoru na relacijske baze podatke spremaju u tablice, druge se vode konceptom ključ-vrijednost ili dokumentnom orijentiranošću. Kod pisanja upita ili pretraživanja ovakvih baza češće ćemo se koristiti programskim jezicima (preko API-ja) nego specijaliziranim upitnim jezicima za NoSQL baze. [10]

Rastom popularnosti OOP razvoj tipova za modeliranje podataka je krenuo istim smjerom. Uvedene su kao koncept objektno orijentirani modeli podataka koji služe kako bi olakšali komunikaciju između implementacije i podataka. [11] Objektno orijentirane baze podataka nikad nisu dosegle veliku popularnost, dijelom zbog toga što su njihovu svrhu zamijenili razvojni okviri kao što je Entity Framework.

3.2. Rad s relacijskim modelom podataka

Relacijske baze podataka, kao što je prethodno spomenuto, su najzastupljeniji tipovi modela podataka. U praksi često susrećemo brojne sustave za upravljanje relacijskim bazama podataka kao što su MySQL, PostgreSQL, Microsoft Access, MS SQL Server, Oracle koji se

služe infrastrukturom klijent-server. [12] Takve sustave je pogodno koristiti ukoliko radimo na projektima većeg opsega koji imaju više podataka kako bi razdvajanje podatkovnog i logičkog sloja lakše upravljali aplikacijom. No ukoliko radimo na manjim projektima odvajanje može biti nepotrebno i nespretno. U takvim slučajevima je brže da su unutar programskog koda odvija i rad s bazom podataka i logički dio programa. Jedan od sustava koji nam može u tom slučaju pomoći je sqlite. Sqlite je sustav za relacijske baze podataka koji se koristi SQL-om kao upitnim jezikom, a nalazi se unutar samog programa. Ima razvijenu podršku za sve popularnije programske jezike, a u Pythonu je biblioteka za sqlite dio standardnog paketa. Smatra se danas najraširenijim sustavom zbog primjene kod web preglednika i mobilnih uređaja. [13]

Kada radimo s sqliteom prvo što trebamo definirati je konekcija. Konekciju se može definirati na dva općenita načina. Prvi je da podatke iz baze podataka spremamo u radnu memoriju i to se radi tako da se proslijedi parametar :memory: funkciji *connect*, a drugi je da funkciji *connect* proslijedimo putanju do datoteke na koju se treba spojiti (ili je kreirati ukoliko ne postoji). Iduće što trebamo napraviti je kreirati tablice pomoću funkcije execute koju pozivamo nad *cursor* objektom za našu bazu podataka. U našem primjeru ćemo naredbe za kreiranje tablica staviti u *try-catch* kako bismo izbjegli pojavljivanja grešaka prilikom drugog pokretanja programa (tablica već kreirana). Kada bismo radili na pravom projektu odvojili bi skriptu za kreiranje tablica kako se nebi morala svaki put izvršavati. Tu pišemo čisti SQL samo treba paziti na tipove podataka ukoliko smo navikli na rad s drugim sustavima.

Python type	SQLite type
`None`	`NULL`
`int`	`INTEGER`
`float`	`REAL`
`str`	`TEXT`
`bytes`	`BLOB`

Slika 6: Tipovi podataka u Pythonu u usporedbi sa sqlite-om (Python.org, 2019)

Naš primjer će biti jednostavna baza koja sadrži jednu vezu više na više. Ta veza je između tablica student i kolegij, a kako bi tu vezu mogli izvesti koristit ćemo i pomoćnu tablicu pohađa koja će sadržavati primarne ključeve druge dvije tablice. Entitete koje ćemo koristiti (u našem slučaju student i kolegij) ćemo definirati kao klase kako bi nam kasniji rad bio lakši (za

veće projekte se koriste ORM-ovi koji to rade automatski). Klasama pridružujemo *property* vrijednosti i reprezentaciju (izgled ispisa) jer nam metode klase u ovom slučaju neće trebati.

```
class Student:
    def __init__(self, ime, prezime, JMBAG):
        self.ime = ime
        self.prezime = prezime
        self.JMBAG = JMBAG

    def __repr__(self):
        return     "Student('{}',     '{}',     {})".format(self.ime,
self.prezime, self.JMBAG)
class Kolegij:
    def __init__(self, naziv):
        self.naziv = naziv

    def __repr__(self):
        return "Kolegij('{}' )".format(self.naziv)

conn                                                                =
sqlite3.connect('C:\\Users\\Fabio\\Desktop\\zavrsni\\studenti.db')

c = conn.cursor()

try:
    c.execute("""CREATE TABLE studenti (
            id_student INTEGER PRIMARY KEY AUTOINCREMENT,
            ime text,
            prezime text,
            JMBAG integer
            )""")

    c.execute("""CREATE TABLE kolegij (
            id_kolegij INTEGER PRIMARY KEY AUTOINCREMENT,
            naziv text
            )""")

    c.execute("""CREATE TABLE pohadja (
            kolegij_id INTEGER,
            student_id INTEGER,
```

```
                FOREIGN KEY(kolegij_id) REFERENCES kolegij(id_kolegij),
                FOREIGN KEY(student_id) REFERENCES studenti(id_student)
                )""")
    except Exception as e:
        print(e)
```

Kada imamo kreirane klase i tablice možemo pokazati upravljanje s podacima u relacijskom modelu na ovom primjeru. Operacije koje ćemo prikazati su standardne CRUD operacije. Operacije koje ćemo koristiti je potrebno odvojiti u zasebne funkcije kako bi mogli nesmetano ih koristiti. Važno je naglasiti kod rada s bazom podataka koristit ćemo tzv. parametizirane izraze. Parametizirani izrazi su analogija pripremljenih izraza koja se koristi u Pythonu, a koriste se kako bi se baza zaštitila od napada SQL injekcije. [14]

```
    def insert_stud(student):
        with conn:
            c.execute("INSERT INTO studenti VALUES (:id_student,:ime,
:prezime, :JMBAG)", {'id_student':None,'ime': student.ime, 'prezime':
student.prezime, 'JMBAG': student.JMBAG})
    def select_studenti():
        c.execute("SELECT * FROM studenti")
        return c.fetchall()


    def update_jmbag(student, jmbag):
        with conn:
            c.execute("""UPDATE studenti SET JMBAG = :jmbag
                        WHERE ime = :ime AND prezime = :prezime""",
                      {'ime': student.ime, 'prezime': student.prezime,
'jmbag': jmbag})


    def delete_student(student):
        with conn:
            c.execute("DELETE from studenti WHERE JMBAG = :jmbag",
                      {'jmbag': student.JMBAG})
```

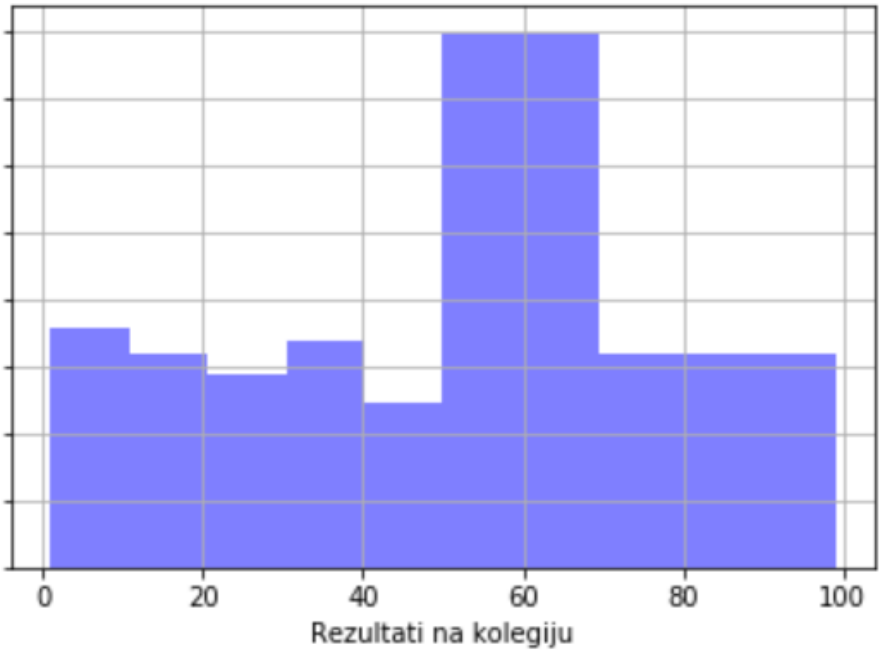
Za ostale dvije tablice koristit ćemo analogne funkcije samo s drugačijim izrazima (osim male razlike kod tablice veza za koju nemamo definiramo klasu pa je popunjavamo vrijednostima primarnih ključeva iz ostale dvije tablice).

3.2.1. Vizualizacija relacijskog modela podataka

Kada pričamo o vizualizaciji relacijskog modela podataka trebamo naglasiti da ćemo vizualizirati podatke, a ne model. To znači da nećemo se baviti standardnom modela kao što su ERA dijagrami. ERA dijagrami su jako korisni dijagrami za vizualizaciju logičkog modela za relacijski model podataka. [15] Takvi dijagrami su korisni za veće projekte, a kako ovdje radimo s sqlite-om nećemo se susretati s većim projektima. Također, ukoliko bi se i bavili većim projektima puno je jednostavniji i spretnije koristiti neki od alata za rad s bazama podataka koji imaju implementiranu mogućnost vizualizacije (npr. MySQL Workbench). Ukoliko bismo baš morali zbog nekog razloga vizualizirati ERA model vrijedi pogledati biblioteke ERAlchemy i SQLAlchemy koje su prilično jednostavne za koristiti i efikasne (iako im se može zamjeriti nekonzistentnost s obzirom da su u ranom razvoju). [16]

Koristeći bazu podataka koju smo prethodno kreirali (uz dodatak atributa bodovi kod kolegija, jer nam je taj atribut potreban za vizualizaciju) napravit ćemo histogram koji će nam prikazivati distribuciju bodova po studentima. Dohvaćanje podataka je vrlo jednostavno. Možemo učiniti to na dva načina. Prva opcija je da SQL naredbom vratimo sve vrijednosti bodova te ih spremimo u listu. Drugi način je da sve vrijednosti koje dobijemo iz tablice kolegij spremimo u memoriju kao matricu. Zatim iteracijom nad matricom dohvatimo vrijednosti bodova na kolegiju. Kada imamo listu bodova onda možemo lagano prikazati histogram.

```
n = 2
ocjene = [x[n] for x in select_kolegij()]
n, bins, patches = plt.hist(ocjene, 10, density=True, facecolor='b',
alpha=0.5)
```



Slika 7: Histogram za relacijski model

3.3. Nestrukturirani podaci

Kako smo strukturirane podatke definirali kroz model podataka koji se implementira pomoću baza podataka tako nestrukturirane podatke možemo definirati kao podatke koje nije moguće efikasno prikazati u bazi podataka. Prema procjenama nekih analitičara danas je oko četiri petine svih podataka nestrukturirano. [17]

Tipovi zapisa koji su najčešće nestrukturirani su tekstualni zapisi (knjige npr.) kao i razni tekstualni i audio zapisi, e-mail poruke, web stranice itd. Nestrukturirani podaci ne moraju nužno biti bez ikakve strukture. Oni uglavnom imaju svoju internu strukturu, ali nam ona ne omogućava strukturiranje tih zapisa pomoću zapisivanja u bazu podataka ili neki od polustrukturiranih zapisa. [17] Zbog toga je s takvim podacima rad otežan. Pretraživanje takvih zapisa je često nespretno i memorijski zahtjevno (u usporedni sa SQL-om i pretraživanja u tabličnim kalkulatorima).

Ukoliko bismo željeli strukturirati nestrukturirani zapis moramo pribjeći tehnikama pronalazaka uzoraka među podacima (ukoliko oni postoje). Tehnike koje se za to služe se najčešće nazivaju tehnikama rudarenja podataka (eng. *data mining).* Rudarenje podataka je u velikoj ekspanziji posljednjih godina jer je zbog sve većih količina prikupljenih podataka postala nužnost valjano ih strukturirati kao bi imali ikakve koristi od njih. Osim rudarenja podataka za strukturiranje se mogu još koristiti i algoritmi *Natural Language Processinga*, i to za audio zapise. [18]

Situacija odnosa između teorije i prakse je kod nestrukturiranih podataka puno kompliciranija nego kod strukturiranih ili polustrukturiranih. Uzmimo za primjer HTML zapis. Ukoliko bismo gledali strukturu zapisa onda je HTML čisti primjer polustrukturiranog zapisa. Naime, jedna od karakteristika polustrukturiranih zapisa je da se podaci nalaze unutar oznaka, a HTML je strogo određen oznakama. Ako na trenutak zanemarimo kao vizualno izgleda HTML zapis i promotrimo podatke koje sadrži tada možemo pomisliti kako se radi o nestrukturiranom zapisu jer podaci unutar oznaka se teško strukturiraju bez korištenja tehnika web struganja. To je zato što, za razliku od XML-a, HTML je jezik koji služi za renderiranje web preglednicima te kao takav nije namijenjen korisnikovom korištenju.

Kako bi pojasnili prethodni paragraf prikazat ćemo na primjeru. Primjer će nam biti stranica koja prikazuje popis zemalja i njihov BDP (izvor: NationMaster.com). Podaci koje očekujemo saznati bili bi ime države i vrijednost BDP-a. Pogledamo li izvorni HTML dokument u njemu možemo pronaći te podatke. Nalaze se unutar zasebnog HTML spremnika koji nam ne sugerira da se radi o podatku koji tražimo. Također isti se spremnik koristi i za druge podatke koji nemaju veze s traženim (podaci vezani uz prikaz stranice) . Zato bi za HTML

mogli reći da ima strukturu koja je interna, ali ako bi gledali s aspekta strukture podataka morali bi ga svrstati u nestrukturirane (iako ga je moguće strukturirati za željeni problem). Kako bismo strukturirali zapis iz primjera koristit ćemo popularnu biblioteku za struganje BeautifulSoup. Također da ubrzamo struganje i olakšamo strukturiranje koristit ćemo biblioteku Pandas. Iako to nije potrebno pogotovo za ovakav problem, ali zašto ne iskoristiti ponuđeno. Kao zapis u koji ćemo strukturirati odabrali smo CSV.

```
    res = requests.get("https://www.nationmaster.com/country-info/stats/Economy/GDP")
    soup = BeautifulSoup(res.content,'lxml')
    table = soup.find_all('table')[0]
    df = pd.read_html(str(table))[0]
    countries = df["COUNTRY"].tolist()
    gdp = df["AMOUNT"].tolist()

    with open('drzave.csv', 'w') as csvfile:
        filewriter = csv.writer(csvfile, delimiter=';',
                                quotechar='|', quoting=csv.QUOTE_MINIMAL)
        filewriter.writerow(['Drzava', 'Iznos BDP'])
        for i in range(len(countries)):
            row = []
            row += [countries[i]]
            row += [gdp[i]]
            filewriter.writerow(row)
```

Finalni rezultat koji dobivamo je strukturiran na sljedeći način:

Drzava;Iznos BDP

European Union;$16.63 trillion

United States;$15.68 trillion

China;$8.36 trillion...

4. Polustrukturirani podaci

Polustrukturirani podaci su svi oni podaci koji ne podliježu strukturama punokrvnih strukturiranih podataka (model podataka i relacijske baze podataka), ali i dalje imaju striktno određenu strukturu koja se očituje kroz grupiranje elemenata pomoću oznaka i sl. Najveća primjena ovakvih oblika podataka je kod web aplikacija. Primjena ovakvih tipova zapisa je učestala zato što programerima uvelike olakšavaju rad. Za njih postoje brojne biblioteke i funkcije, a i lakši je prijenos podataka između nekih bazičnih tipova podataka (liste, rječnici) i polustrukturiranih podataka u odnosu na strukturirane. Također trenutno se smatra da se većina podataka sprema u razne polustrukturirane zapise, a ta će brojka višestruko rast zbog porasta područja znanosti o podacima (eng. *Data Science)* i umjetne inteligencije u kojima se također koriste polustrukturirani podaci. [19]

Ako bismo morali svrstati polustrukturirane podatke u neku skupinu oni bi pripadali strukturiranim podacima. Polustrukturirani podaci su više podskupina strukturiranih podataka nego zasebna grupa. To možemo reći zato što nemaju nikakve dodirne točke s nestrukturiranim zapisima (ukoliko ćemo nestrukturirane zapise promatrati čisto teoretski, jer u praksi i oni imaju nekakvu strukturu), a u velikom broju stvari su slični strukturnim podacima. Podatke u oba vrste zapisa pronalazimo pomoću njihove strukture, a iteriranje je strogo određeno i nefleksibilno.

Iako su polustrukturirani podaci bliži strukturiranim podacima često možemo naići na to da se oni smatraju zasebnom grupom. Najveća razlika je u tome što su strukturirani podaci strogo proizišli iz relacijskih baza podataka dok nastanak polustrukturiranih datoteka vezujemo uz neke zapisa kao npr.XML.

4.1. XML

XML (kratica *extensible markup language)* je proširivi jezik za strukturiranje dokumenata i podataka u tekstualnom zapisu najčešće korišten unutar Web servisa. Važno je naglasiti da XML, kao ni HTML, ne možemo smatrati programskim jezicima. Dizajniran je za prijenos i slanje podataka koji bi trebali biti čitljivi i programu i čovjeku.[20]

```
<XMLprimjer>
      <Fakultet>
            <Fakultet1>Fakultet Organizacije i Informatike</Fakultet1>
```

```
            <Fakultet2>Fakultet Elektrotehnike i Računarstva</Fakultet2>
            <Fakultet3>Fakultet Strojarstva i Brodogradnje</Fakultet3>
      </Fakultet>

      <Gradovi>
            <Grad1>Varazdin</Grad1>
            <Grad2>Zagreb</Grad2>
            <Grad3>Split</Grad3>
            <Grad4>Zadar</Grad4>
      </Gradovi>
</XMLprimjer>
```

Gledajući gore navedeni primjer XML zapisa odmah nam se nameće paralela s HTML zapisom. Oba zapisa sadrže podatke unutar oznaka unutar zagrada i koriste isti način otvaranja i zatvaranje oznake. Iako strukturno slično izgledaju po primjeni i korištenju nemaju prevelike sličnosti. Najočitija razlika je ta da HTML i XML rade na različitim slojevima aplikacije. XML je jezik podatkovnog sloja, a HTML je jezik prezentacijskog. HTML je zamišljen kao prezentacijski jezik kod kojeg je fokus prebačen na to kako će podaci izgledati u konačnom prikazu (web stranici), dok kod XML je fokus na tome kakvi su podaci jer je dizajniran u svrhu prenošenja, a ne prikaza podataka. Također iako se oznake u oba jezika čini identičnima u XML one nisu predefinirane kao u HTML. HTML ima predefinirane oznake u svrhu da preglednik može temeljem njih prikazati sadržaj stranice, dok XML može imati proizvoljne oznake koje predstavljaju tip podataka koji se nalazi unutar njih, a namijenjen je korisniku.[20]

Ekstenzivnost XML dokumenta koja se spominje i u njegovom nazivu očituje se u tome da dokumentu možemo dodavati nove podatke unutar novih oznaka bez da to utječe na izvedbu. Također isto se odnosi i na brisanje zapisa iz XML datoteke. [21]

4.1.1. Struktura XML zapisa

XML zapis spada u polustrukturirane zapise. To možemo lako uočiti zato što, iako ne poštuje stroga ograničenja modela podataka, sadrži oznake koje semantički odvajaju elemente zapisa. Oznake su glavna značajka strukturiranosti XML zapisa jer određuju hijerarhiju zapisa u dokumentima.

Struktura XML zapisa se najčešće definira kao struktura stabla. Početnu oznaku unutar koje se nalaze svi zapisi unutar dokumenta naziva korijenom stabla (eng. *Root*), a oznake koje se nalaze unutar njega nazivaju se listovima. Također nadređena oznaka se smatra roditeljem podređene, dok se dvije oznake na istoj hijerarhijskoj razini nazivaju bratskim (eng. *sibling)*. [22]

Kada bismo uzeli kao primjer prethodni XML zapis onda bi *root* oznaka bila <XMLPrimjer> i ta oznaka je obavezna jer XML zapis nije valjan ukoliko nema *root* oznaku. Nadalje, imamo dva lista stabla: <Fakultet> i <Grad>. Te dvije oznake su u međusobno bratskom odnosu s obzirom da su na istoj hijerarhijskoj razini. Oznaka <Fakultet> je roditelj oznakama <Fakultet1>, <Fakultet2> i <Fakultet3> koje su međusobno bratske oznake. Isto tako, oznaka <Gradovi> je roditelj oznakama <Grad1>, <Grad2>, <Grad3> i <Grad4> koje su međusobno bratske oznake. Također valja napomenuti da su imena oznaka iz primjera dana radi pojašnjavanja odnosa oznaka unutar zapisa te da bi u praksi ipak susreli nadređenu oznaku <Grad> i podređene <Gradovi> umjesto da svaka ima svoj broj (isto vrijedi i za fakultete).

Osim standardnih oznaka koje sadrže podatke imamo i oznaku koja se naziva XML Prolog. Ta oznaka se mora nalaziti na početku dokumenta i sadrži atribute (atributi u XML dokumentima su neobavezni dodaci oznakama koji sadrže vrijednost koja je obavezno u navodnicima i daje informaciju o podatku unutar oznake) s vrijednostima verzije XML-a i *encodinga* koji će se koristiti (skup definiranih znakova).

4.1.2. Parsing XML zapisa

Parsing XML zapisa (ili raščlanjivanje) se svodi na prepoznavanje oznaka unutar kojih se podaci nalaze. S obzirom da ćemo koristiti programski jezik Python, njegove specijalizirane biblioteke će nam omogućiti efikasnije pretraživanje od onog da pretvorimo zapis u string, pa da pretražujemo po znakovima stringa dok ne naiđemo na oznaku.

Za obradu XML zapisa koristit ćemo biblioteku XML, preciznije njen pod modul xml.etree.ElementTree. Taj pod modul se ponaša kao jednostavni XML procesor. Osnovna funkcija pod modula je funkcija parse koja zadanu datoteku (datoteka se zadaje ili relativnom putanjom ili kao string). Do elemenata XML zapis je onda prilično trivijalno doći. Ukoliko želimo doći do korijena stabla pozivamo funkciju tree.getroot(). Elementi XML zapisa kojima pristupamo se sastoje od naziva (oznake) i atributa koji su zapisani unutar rječnika. [23]

Na prethodnom primjeru ćemo pokazati jednostavnost raščlanjivanja XML zapisa unutar programskog jezika Python. Želja nam je kao rezultat dobiti imena svih gradova iz zapisa. Prvo moramo učitati XML zapis, a zatim dobiti korijenski član preko kojeg pristupamo oznaci gradova. Unutar oznake gradova pretražujemo sve podatke unutar oznake Grad i zapisujemo vrijednosti kao listu objekata. Listu ispisujemo preko implementirane funkcije za pristupanje vrijednosti objekta (string vrijednost).

```
import xml.etree.ElementTree as ET
tree = ET.parse('C:\\Users\\Fabio\\Desktop\\zavrsni\\primjerXML.xml')
root = tree.getroot()
imena = []
for gradovi in root.findall('Gradovi'):
    try:
        imena = gradovi.findall('Grad')
        print([i.text for i in imena])
    except:
        print("Nije pronadjen zapis")
```

Osim pristupanja postojećim podacima i oznakama unutar XML dokumenta ovaj pod modul nam omogućava dodavanje novih i brisanje postojećih zapisa, kao i kreiranje XML dokumenta pomoću oznaka.

Mijenjanje se izvodi na prijašnjem principu pristupanje podacima XML dokumenta i nakon što pronađemo željeni izmijenimo ga. Nakon toga potrebno je dodati atribut *updated* (opcionalno ukoliko želimo znati koji smo element mijenjali) i pozvati funkciju tree.write.

```
for gradovi in root.iter('Grad'):

    if gradovi.text == "Split":

        novi_grad = "Dubrovnik"

        gradovi.text = novi_grad

        gradovi.set('updated','promijenjen')

tree.write('C:\\Users\\Fabio\\Desktop\\zavrsni\\promijenjeni.xml')
```

Brisanje se izvodi na sličan način, samo se poziva funkcija remove nakon što pronađemo element koji želimo obrisati.

```
for gradovi in root.findall('Gradovi'):

    grad_lista = gradovi.findall('Grad')

    for grad in grad_lista:

        if grad.text == "Split":

            gradovi.remove(grad)

tree.write('C:\\Users\\Fabio\\Desktop\\zavrsni\\promijenjeni.xml')
```

4.1.2.1. Opasnosti prilikom XML parsinga

Iako XML Python biblioteka se čini prilično trivijalnom također postoje neki dijelovi na koje valja pripaziti. Prethodno opisani pod modul koji koristimo kao XML procesor ne sadrži sve potrebne sigurnosne provjere koje nam potencijalno mogu zatrebati.

Jedan od takvih sigurnosnih propusta je i takozvani „Billion laughs attack“. Taj napad (koji je relativno čest kod XML dokumenata) svrstavamo u DoS (Denial of service) napade. To znači da je njegova zadaća stvoriti opterećenje sustavu (najčešće poslužitelju) te ga tako onemogućiti za obavljanje njegovih primarnih zadaća. [24]

Cilj ovog napada je XML parser, što znači da se unutar samog XML-a nalazi problem. Napad se može simulirati tako da u XML dokument stavimo deset entiteta (može naravno i više, ali ovo se smatra standardnim primjerom) koji su definirani kao deseterostruka referenca na prijašnji entitet (osim prvog kojem često u primjerima označavamo s vrijednosti „lol“, odakle i potječe naziv napada). Takvim eksponencijalnim povećavanje podataka entiteta dolazimo do toga da ovaj jednostavni XML dokument zauzima više od 3GB memorije. Povećanjem broja referenci ili entiteta lako možemo povećati taj broj te tako zauzeti još veći memorijski prostor. Pod modul kojeg smo koristili u prethodnim primjerima nema razvijeni sustav prepoznavanja ovakvih napada što može rezultirati prekidom rada računala na kojem se nalazi XML parser.

Rješenje za siguran rad s XML datotekom bez briga o ovakvim i sličnim napadima je Python biblioteka defusedxml koja se posebno koncentrira na sigurnosni aspekt rada s XML datotekama ostavljajući nepromijenjen dio koji se odnosi na rad s XML datotekom u odnosu na elementTree. Često možemo vidjeti da se defusedxml poziva samo prilikom parsinga početnog XML-a (jer se tu događaju napadi), a u nastavku rada možemo nesmetano koristiti elementtree. [25]

4.1.3. Vizualizacija XML zapisa

Kod vizualizacije XML zapisa pomoći će nam pyplot pod modul matplotlib biblioteke. Taj pod modul je specijaliziran za grafički prikaz podataka. Rad s takvim modulom će naravno uključivati prethodno spomenute mogućnosti parsiranja XML zapisa, te njihovo spremanje u radnu memoriju kroz neki od standardnih tipova podataka u Pythonu.

Generalno, do podataka koji su nam potrebni za vizualizaciju možemo doći na dva načina. Prvi način se svodi na čitanje vrijednosti atributa koji je definiran uz neku oznaku i prikaz odnosa tih vrijednosti. Drugi način, koji je učestaliji, se svodi na interpretaciju i vizualizaciju podataka unutar XML oznaka.

Kao primjer prvog načina prikazat ćemo XML zapis koji sadrži popis studenata. Unutar oznake nalazi se ime i prezime studenta (u ovom slučaju ime je redni broj, a prezime Student) dok se u vrijednosti atributa smjer nalazi smjer na kojem student studira. Kako nam podaci o imenu i prezimenu studenta nisu od prevelikog značaja za interpretaciju, baziramo se na podacima koji se nalaze kao vrijednost atributa. Prikazat ćemo grafički udio pojedinog smjera među odabranim studentima. To ćemo učiniti koristeći tortni dijagram, jedan od poznatijih i jednostavnijih dijagrama koji se baš koristi za prikaz udjela.

```
<XMLprimjer>
   <Studenti>
         <Student smjer="IPS">Prvi Student</Student>
         <Student smjer="Ekonomika poduzetnistva">Drugi Student</Student>
         <Student smjer="PITUP">Treci Student</Student>
         <Student smjer="IPS">Cetvrti Student</Student>
         <Student smjer="Ekonomika poduzetnistva">Peti Student</Student>
         <Student smjer="PITUP">Sesti Student</Student>
         <Student smjer="IPS">Sedmi Student</Student>
         <Student smjer="IPS">Osmi Student</Student>
         <Student smjer="Ekonomika poduzetnistva">Deveti
Student</Student>
   </Studenti>
</XMLprimjer>
```

Prvo što trebamo napraviti je parsing XML zapisa. To radimo tako da učitamo cijeli zapis i zatim navigiramo do oznake koju želimo. S obzirom da se podaci koji su nam važni nalaze kao vrijednosti atributa provjeravamo vrijednosti atributa te inkrementiramo brojač koji nam označava broj studenata određenog smjera.

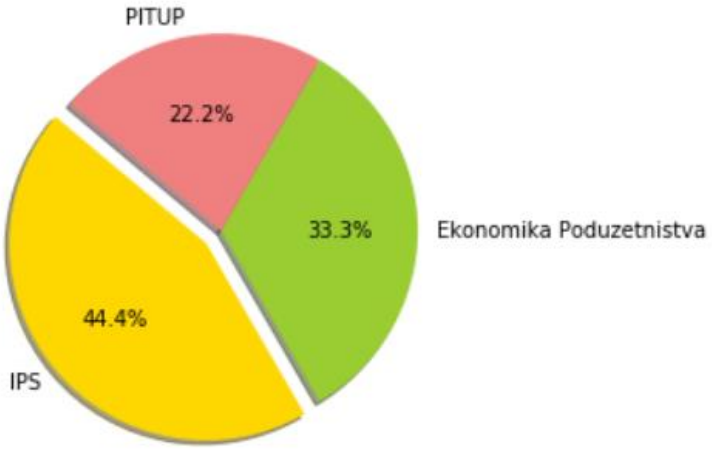
Nakon toga ostaje nam definirati neka svojstva prikaza. Prvo definiramo labele (tekst uz pojedinu vrijednost na grafu). Zatim moramo definirati pojedine udjele na grafu. To se radi tako da se vrijednosti proslijede u listu iz koje svaki element predstavlja jedan udio na dijagramu. Isto napravimo i s bojama te pozovemo funkciju iz biblioteke koja služi za crtanje ovog tipa grafa (funkciji proslijedimo prethodne parametre) i na posljetku prikažemo dijagram.

```
tree = ET.parse('C:\\Users\\Fabio\\Desktop\\zavrsni\\xmlzagraf.xml')
root = tree.getroot()
ips = 0
ep = 0
pitup = 0
```

```
for child in root:
    for value in child:
        if 'IPS' in value.attrib.values(): ips += 1
        if 'Ekonomika poduzetnistva' in value.attrib.values(): ep += 1
        if 'PITUP' in value.attrib.values(): pitup += 1

labels = 'IPS', 'Ekonomika Poduzetnistva', 'PITUP'
sizes = [ips, ep, pitup]
colors = ['gold', 'yellowgreen', 'lightcoral']
explode = (0.1, 0, 0)
plt.pie(sizes, explode=explode, labels=labels, colors=colors,
autopct='%1.1f%%', shadow=True, startangle=140)
plt.axis('equal')
plt.show()
```

Dijagram koji smo dobili prikazuje koliko je postotno studenata nekog smjera uz ime tog smjera.



Slika 8: Tortni dijagram – prikaz udjela studenata po smjerovima

Druga opcija kod uzimanja podataka iz XML zapisa za potrebe vizualizacije je da te podatke uzmemo iz podataka koji se nalaze unutar oznaka u zapisu. Ovakav način pronalaženja podataka za vizualizaciju prikazat ćemo na primjeru XML zapisa koji sadrži popis državnih praznika u Republici Hrvatskoj u 2019. godini. Zapis se sastoji Praznika koji sadrže oznake Naziva i Datuma. Datumi su zapisani u obliku YYYY-MM-DD što nam olakšava kasniji rad.

```
<DrzavniPraznici>
   <Praznik>
         <Naziv>Nova godina</Naziv>
         <Datum>2019-01-01</Datum>
   </Praznik>
   <Praznik>
         <Naziv>Bogojavljanje ili Sveta tri kralja</Naziv>
         <Datum>2019-01-06</Datum>
```

```
	</Praznik>
	<Praznik>
		<Naziv>Uskrs</Naziv>
		<Datum>2019-04-21</Datum>
	</Praznik>
	<Praznik>
		<Naziv>Uskršnji ponedjeljak</Naziv>
		<Datum>2019-04-22</Datum>
	</Praznik>
	<Praznik>
		<Naziv>Praznik rada</Naziv>
		<Datum>2019-05-01</Datum>
	</Praznik>
	<Praznik>
		<Naziv>Tijelovo</Naziv>
		<Datum>2019-06-20</Datum>
	</Praznik>
	<Praznik>
		<Naziv>Dan antifašističke borbe</Naziv>
		<Datum>2019-06-22</Datum>
	</Praznik>
	<Praznik>
		<Naziv>Dan pobjede i domovinske zahvalnosti i Dan hrvatskih
branitelja</Naziv>
		<Datum>2019-08-05</Datum>
	</Praznik>
	<Praznik>
		<Naziv>Velika Gospa</Naziv>
		<Datum>2019-08-15</Datum>
	</Praznik>
	<Praznik>
		<Naziv>Dan neovisnosti</Naziv>
		<Datum>2019-10-08</Datum>
	</Praznik>
	<Praznik>
		<Naziv>Dan svih svetih</Naziv>
		<Datum>2019-11-01</Datum>
	</Praznik>
	<Praznik>
		<Naziv>Božić</Naziv>
		<Datum>2019-12-25</Datum>
	</Praznik>

</DrzavniPraznici>
```

Kako bi vizualizirali podatke trebamo prvo ih izvući iz zapisa. To radimo na sličan način kao i u prethodnim primjerima, samo smo se ovdje odlučili kako je potrebna zasebna funkcija koja će to raditi, kako ne bi dolazilo da miješanja prikupljanja podataka i vizualizacije.

Sami prikaz podataka će se svoditi na graf koji će pokazivati frekventnost praznika u ovisnosti s vremenom. Također koristit će se i labele koje će označavati o kojem prazniku je riječ. Praznici će biti povezani na liniju koja se naziva spines i standardni je dio matplotlib biblioteke. [26]

```
def parse_xml(path):
```

```
        tree = ET.parse(path)
        root = tree.getroot()
        datumi = []
        nazivi = []

        for child in root:
            for value in child:

                if value.tag == "Naziv": nazivi.append(value.text)
                if value.tag == "Datum": datumi.append(value.text)

        return nazivi,datumi

    datumi = []
    nazivi = []
    nazivi,datumi=parse_xml('C:\\Users\\Fabio\\Desktop\\zavrsni\\praznici.xm
l')
    datumi = [datetime.strptime(d, "%Y-%m-%d") for d in datumi]

    levels = np.tile([-5, 5, -3, 3, -1, 1],
                     int(np.ceil(len(datumi)/6)))[:len(datumi)]

    fig, ax = plt.subplots(figsize=(8.8, 4), constrained_layout=True)
    ax.set(title="Državni praznici")

    markerline, stemline, baseline = ax.stem(datumi, levels,
                                             linefmt="C3-", basefmt="k-",
                                             use_line_collection=True)

    plt.setp(markerline, mec="k", mfc="w", zorder=3)
    markerline.set_ydata(np.zeros(len(datumi)))

    vert = np.array(['top', 'bottom'])[(levels > 0).astype(int)]
    for d, l, r, va in zip(datumi, levels, nazivi, vert):
        ax.annotate(r, xy=(d, l), xytext=(-1, np.sign(l)*1),
                    textcoords="offset points", va=va, ha="right")

    ax.get_xaxis().set_major_locator(mdates.MonthLocator(interval=1))
    ax.get_xaxis().set_major_formatter(mdates.DateFormatter("%b %Y"))
    plt.setp(ax.get_xticklabels(), rotation=30, ha="right")

    ax.get_yaxis().set_visible(False)
    for spine in ["left", "top", "right"]:
        ax.spines[spine].set_visible(False)

    ax.margins(y=0.1)
    plt.show()
```

Slika 9: Spine graf s prikazom frekvencijom praznika u kalendarskoj godini

4.2. JSON

JSON ili JavaScript Object Notation je polustrukturirani zapis koji je nastao za potrebe komunikacije između pretraživača (browsera) i poslužitelja (servera). Riječ je o čovjeku čitljivom zapisu kojeg karakterizira korištenje sintakse iz JavaScript jezika. Iako je kao format nastao iz sintakse JavaScripta pohranjuje se kao tekst te je tako dostupan i jednako lagan za korištenje bilo kojem drugom programskom jeziku. [27]

Kada govorimo o JSON formatu strukturiranja podataka tada zapravo govorimo o JavaScript notaciji za objekte. Objekti se nalaze unutar vitičastih zagrada, a podaci se prikazuju formatom ime-vrijednost i međusobno se odvajaju zarezima. Imena se pišu unutar navodnika (dvostrukih) dok se vrijednosti pišu kako god to njihov tip zahtijeva. Na primjer, string vrijednost se obavezno piše unutar navodnika, dok se liste nalaze unutar uglatih zagrada. Imena će nam koristiti za pristupanje podacima (vrijednostima) tijekom obrade i vizualizacije podataka, analogno XML-ovim oznakama i podacima unutar oznaka. [28]

```
{
  "ime": "John",
  "prezime": "Smith",
  "starost": 25,
  "adresa": {
    "ulica": "21 2nd Street",
    "grad": "New York",
    "drzava": "NY",
    "postanskiBroj": "10021"
  },
```

```
  "brojeviTelefona": [
    {
      "vrsta": "kucni",
      "broj": "212 555-1234"
    },
    {
      "vrsta": "fax",
      "broj": "646 555-4567"
    }
  ],
  "spol": {
    "vrsta": "muski"
  }
}
```

Vidimo na primjeru JSON zapisa da ima brojne sličnosti s XML zapisom. Jedna od sličnosti je i sama „logika“ zapisa. Naime vidimo da JSON zapis, kao i XML, hijerarhijski sprema podatke. Dok se kod XML zapisa hijerarhija očitovala kroz sadržavanje jedna oznake unutar druge kod JSON podređena oznaka nalazi se unutar vitičaste zagrade nadređene oznake.

4.2.1. Usporedba JSON i XML

XML i JSON su u mnogočemu slični zapisi. Oba se koriste kod web aplikacija za prijenos podataka. Logički gledano, podaci su jednako zapisani između ta dva zapisa. U oba slučaja logika zapisa se svodi na prikaz hijerarhije između tih zapisa (podređeni zapis je unutar nadređenog). Također oba zapisa su kreirana da budu čitljivi i korisniku, a ne samo aplikacijama.

Razlika između dva zapisa je najočiglednija u sintaksi. Dok XML koristi oznake (eng. *Tagove)* nalik onima iz HTML, JSON koristi sintaksu JavaScripta što rezultira u značajno kraćem zapisu. Jedan od razloga zašto JSON ima kraći zapis je u tome što ne koristi oznake za kraj zapisa. Jedina značajnija razlika između ovih zapisa koja se ne svodi na vizualni aspekt je u tome što JSON podržava liste kao strukturu podataka. [29]

Najveća razlika ova dva zapisa je u tome što je aplikacijama korištenje JSON-a značajno brže.

4.2.2. Parsing JSON zapisa

Kako bi pripremili podatke za obradu i vizualizaciju potrebno ih je preuzeti iz JSON zapisa. Iako je ova radnja trivijalna za JavaScript programski jezik ni u drugim jezicima ne predstavlja veći problem. U radu s JSON dokumentima uglavnom ćemo koristiti json biblioteku za Python. Prva stvar koja je potrebna je dekodirati JSON dokument. To možemo učiniti na dva osnovna načina. Prvi način je da koristimo funkciju load iz biblioteke json. Funkcija load kao argument prima relativnu putanju do json dokumenta (ukoliko je dokument preuzet lokalno) i dekodira ga te sprema kao rječnik (eng. *dictionary*). Drugi način dekodiranja je funkcija loads koja radi s ulaznim podacima tekstualnog oblika, kao npr. string. Korištenje ovog načina nije preporučljivo ukoliko radimo s većim dokumentima.

Ove dvije prije spomenute funkcije rade samostalno i daju nam željeni rezultat za lokalno spremljene dokumente. No što je s dokumentima na poslužitelju? Prilikom rada s JSON dokumentima često ćemo se susreti s dokumentima koji se nalaze na poslužitelju jer je JSON i razvijan za potrebe web aplikacija. Za rad s dokumentima koji nisu spremljeni lokalno trebat će nam biblioteka urllib.request. Pomoću te biblioteke ćemo dohvatiti dokument koji ćemo kasnije trebati dekodirati.

```
import urllib.request, json
with urllib.request.urlopen(url) as url:
    data = json.loads(url.read().decode())
```

Rad s podacima nakon dekodiranja se svodi na rad s rječnicima u Pythonu. Objektima pristupamo preko imena u paru ime-vrijednost. Na primjer, uzmemo li prijašnji JSON zapis, imenu osobe pristupamo kao data['ime']. Ukoliko imamo veći zapis i želimo pregledati sva imena preko kojih možemo pristupiti podacima to možemo učiniti preko jednostavne foreach petlje kroz podatke (sintaksa bi bila *for imena in podaci*).

Isto tako, kako bi mogli podatke koje obrađujemo unutar programa spremati unutar JSON dokumenta postoje funkcije enkodiranja. Enkodiranje je postupak suprotan dekodiranju i izveden je sintaktički na sličan način dekodiranju. I tu postoje dvije opcije koje ćemo koristiti. Prva opcija je funkcija dumps. Funkcija dumps kao argument prima rječnik koji smo definirali u našem programskom kodu i pretvara ga u JSON zapis. Druga opcija je funkcija dump koju koristimo kada želimo dodati JSON zapis unutar postojećeg dokumenta (dokument može, ali i ne mora biti s ekstenzijom *.json, jednako radi i s tekstualnim dokumentima). [30]

4.2.3. Vizualizacija JSON zapisa

Vizualizacija podataka iz JSON zapisa se ne razlikuje u mnogočemu u odnosu na XML zapise. Najveća razlika je vidljiva prilikom dohvaćanja samih podataka (rad s rječnikom koji nije svojstven XML).

Kod vizualizacije podataka opet ćemo imat slične korake. Prvi korak je onaj dohvaćanja podataka iz zapisa koji će biti još i jednostavniji u odnosu na XML u nekim pogledima. Drugi korak će se svoditi na rad s matplotlib bibliotekom i specificiranjem vizualizacije podataka.

Prvi primjer će nam biti na tragu onog s XML datotekama. Također ćemo koristiti jedan široko poznati dijagram s kojim smo se više-manje svi susreli, a to je stupčasti dijagram. U ovom slučaju ćemo stupčastim dijagramom prikazati prolaznost studenata po seminarskim grupama na nekom imaginarnom kolegiju. Lijevi stupac uz labelu koja označava grupu će prikazivati studente koji su položili taj kolegij, a desni sve studente koji su upisali.

```
"grupe": [
   {
     "naziv": "G11",
     "godina": "3",
     "smjer": "IPS",
     "prolaznost": {
       "prosli":  17,
       "ukupno": 35
     }
   },
   {
     "naziv": "G12",
     "godina": "3",
     "smjer": "IPS",
     "prolaznost": {
       "prosli":  11,
       "ukupno": 35
     }
   },...
```

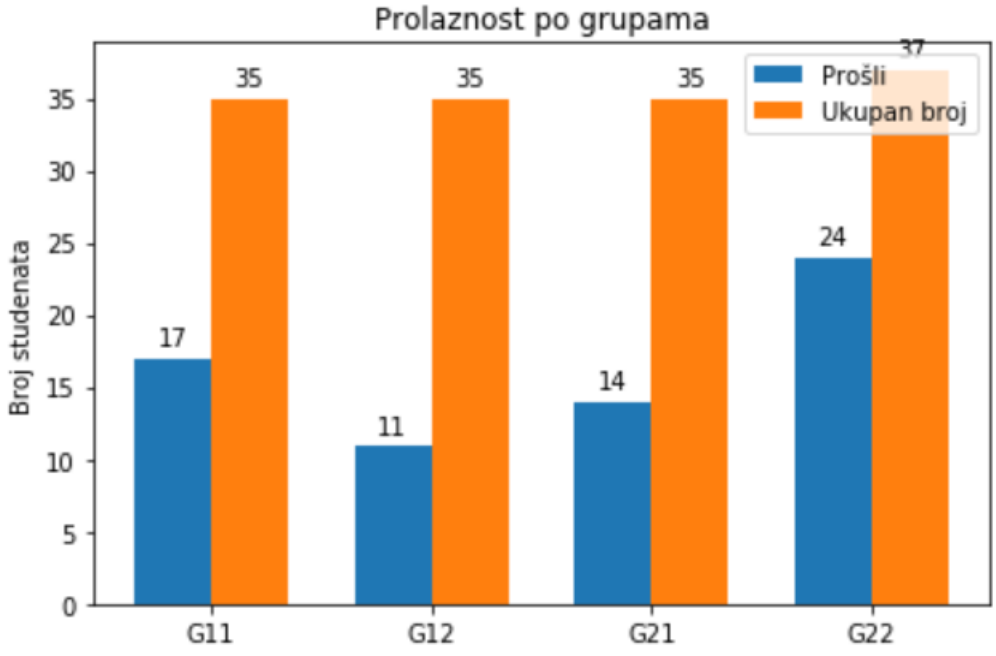
Parsing dokumenta ćemo raditi u odvojenoj funkciji kako ne bismo miješali dohvaćanje i prikazivanje podataka. Nakon što funkcijom load učitamo traženi dokument kao Python objekt (rječnik) lako preko vrijednosti ključeva pristupamo vrijednostima koji zatim spremamo u listu. Vrijednosti tri liste koje nam trebaju vraćamo kao rezultat funkcije.

```
def dohvati_podatke(path):
    with open(path) as jsonfile:
        data = json.load(jsonfile)
    rjecnik = {}
    labele = []
    prolazni = []
    svi = []
    for grupe in data['grupe']:
        rjecnik = grupe
        prolazni += [rjecnik["prolaznost"]["prosli"]]
        svi += [rjecnik["prolaznost"]["ukupno"]]
        labele += [rjecnik["naziv"]]
    return labele,prolazni,svi
```

Preostaje nam samo definirati parametre ispisa za graf. Prvo definiramo labele (oznake). Prema njihovom broju određujemo širinu, te definiramo nazive X i Y koordinata. Postavljamo natpise uz stupce (legendu) kako bi korisniku interpretacija vrijednosti bila odmah jasna. Definiramo zatim vrijednosti za pojedine stupce, za što nam trebaju prethodno dohvaćeni podaci. Ostaje nam na kraju samo dodati funkciju s kojom ćemo ispisivati vrijednost uz svaki stupac kako bi osim grafičke interpretacije korisnik vidio i stvarne podatke (ova funkcija je opcionalna, ali korisna). Za te potrebe koristimo gotovu funkciju koja se nalazi u dokumentaciji biblioteke.

```
x = np.arange(len(labele))
width = 0.
fig, ax = plt.subplots()
rects1 = ax.bar(x - width/2, prolazni, width, label='Prošli')
rects2 = ax.bar(x + width/2, svi, width, label='Ukupan broj')
ax.set_ylabel('Broj studenata')
ax.set_title('Prolaznost po grupama')
ax.set_xticks(x)
ax.set_xticklabels(labele)
ax.legend()
def autolabel(rects):
    for rect in rects:
        height = rect.get_height()
        ax.annotate('{}'.format(height),
                    xy=(rect.get_x() + rect.get_width() / 2, height),
                    xytext=(0, 3),
                    textcoords="offset points",
                    ha='center', va='bottom')

autolabel(rects1)
autolabel(rects2)
fig.tight_layout()
plt.show()
```



Slika 10: Stupčasti dijagram prolaznosti studenata

U drugom primjeru ćemo vizualizirati podatke iz JSON zapisa u trodimenzionalni graf. Dosad nismo imali trodimenzionalne grafove u primjerima pa bi stoga trebalo pojasniti na što se misli kad se priča o njima. Trodimenzionalni graf je bilo koji graf koji sadrži osim X i Y osi i Z os. Koriste se za vizualizaciju podataka u kojima treba pokazati korelaciju između tri skupine podataka. U našem primjeru radit će se u očitanim vrijednostima temperature u razdoblju od mjesec dana i odnosu s jačinom vjetra te vlagom. U ovom primjeru koristit ćemo standardni scatter plot (samo 3D) kojim ćemo s točkom označiti mjesto presijecanja tri vrijednosti na grafu.

```
"vrijeme": [
  {
    "temperaturaC": "24",
    "vlaznostPct": "79",
    "vjetarKpH": "11"
  },
  {
    "temperaturaC": "23",
    "vlaznostPct": "82",
    "vjetarKpH": "11"
  },...
```

Sam postupak dohvaćanja podataka je analogan onome iz prethodnog primjera te kao takvog ga nije potrebno ponovno naglašavati (uz očitu razliku različitih vrijednosti ključeva).

Nakon toga potrebno je definirati graf i njegove podatke. Prvo što radimo je definiramo da se radi o 3d *subplotu* i postavljamo standardne vrijednosti od 111 (što označava da nema pomaka na grafu). Nakon toga definiramo *scatter* (prikaz točaka) i proslijedimo mu vrijednosti koje treba prikazati na grafu (treba paziti da prilikom parsinga vrijednosti nam budu pretvorene u float), odaberemo boju i oblik (boja se odabire slovom koje označava tu boju, npr. r za crvenu). Samo nam ostaje dati nazive našim osima i možemo prikazati graf.

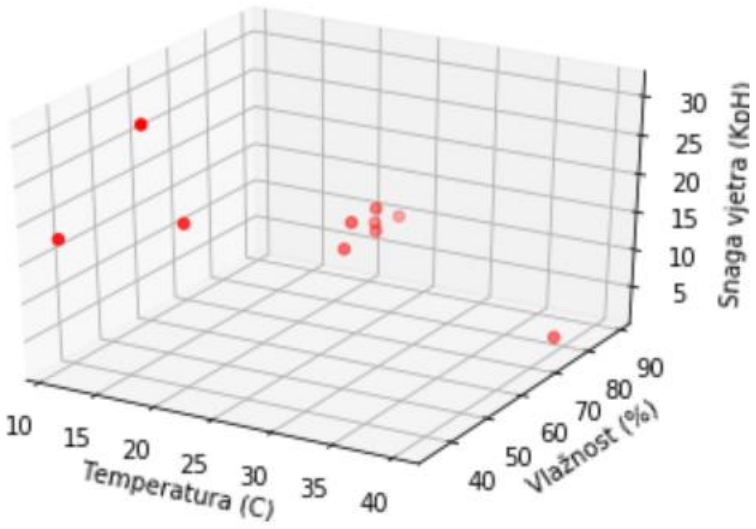
```
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')


ax.scatter(temperature, vlaznost, jacina_vjetra, c='r', marker='o')


ax.set_xlabel('Temperatura (C)')
```

```
ax.set_ylabel('Vlažnost (%)')
ax.set_zlabel('Snaga vjetra (KpH)')

plt.show()
```



Slika 11: 3D scatter prikaz podataka o vremenu

4.2.4. JSON i YAML

YAML (rekurzivni akronim od YAML ain't markup language) je jezik za serijalizaciju podataka, po primjeni istovjetan XML i JSON-u. YAML je nastao kao podset JSON-a, a na popularnosti je najviše dobio kod Python developera zbog korištenja sintakse iz tog jezika. To se očituje u korištenju bijelih prostora (eng. *whitespace*) kao hijerarhijskog uvlačenja i korištenja uglatih i vitičastih zagrada za definiranje liste, odnosno rječnika. U teorijskom smislu smatra se čitljivim za korisnika iako to ne mora uvijek biti slučaj. [31]

```
---
ime:     John
prezime:  Smith
starost:  25


adresa:
    - ulica:   21 2nd Street
      grad:   New York
      drzava:    NY
      postanskiBroj:  10021

brojeviTelefona:
      - vrsta: kucni
        broj: 212 555-1234
      - vrsta: fax
        broj: 646 555-4567
spol:
      -vrsta: muski
```

4.3. CSV

CSV ili comma separated value je tip dokumenta koji zapise sprema unutar redova koji su odvojeni (najčešće zarezom ili točka-zarezom) separatorom. Zarezi su idejno osnovna oznaka za odvajanje podataka, no kako se unutar samih podataka lako može pojaviti zarez (npr. decimalni broj) češće se koristi točka zarez koja nema toliku primjenu u opisivanju svakodnevnih stvari. Podaci uopće ne moraju u ovakvom tipu zapisa biti odvojeni nekim posebnim znakom. Naime, sasvim je validno kao separator koristiti razmak ili tabularni razmak i takav dokument spremiti kao .csv. Ovakvi problemi (nestandardiziranost separatora) predstavljaju problem kod korištenja ovakvih tipova zapisa za rad aplikacija, stoga neke aplikacije strogo traže jedan tip separatora da se koristi (najčešće točka zarez). [32] Jedan od razloga široke rasprostranjenosti ovog formata je njegova kompatibilnost s programima za tablične kalkulacije (Calc, Excel) u kojima lako možemo spremiti i učitati ovakav tip datoteke.

CSV datoteka se zbog velike jednostavnosti često koristi reprezentaciju podataka iz baze podataka. Svaki red predstavlja slog baze podataka, a važno je napomenuti kako je

potrebno definirati u prvom redu nazive stupaca kako bi kasnije lakše mogli pristupati podacima. Nedostatak CSV datoteka je u tome da zbog velike jednostavnosti samog zapisa nisu pogodne za složenije tipove podataka (liste, strukture, rječnike) već samo za osnovne (alfanumeričke). Također vjerojatnost greške je veća jer zbog logike strukture (nema hijerarhije, zapis ispod zapisa) postoji mogućnost da u nekom od redaka ne ispoštujemo polja zadana u retku zaglavlja što dovodi do nevaljanosti podataka.

```
Ime;Prezime;DatumRodjenja;SmjerStudija;GodinaStudija;JMBAG
Ana;Anic;1995-01-01;IPI;4;123456789
Branko;Brankovic;1998-01-01;EP;2;123456788
Marta;Martic;1998-01-01;IS;2;123456787
Josip;Josipovic;1999-01-01;IPS;1;123456786
Ivan;Ivanovic;1991-01-01;BPBZ;5;123456785
Luka;Lukovic;1994-01-01;IPI;4;123456799
Ante;Antic;1993-01-01;PS;3;12345674
```

Strukturiranost zapisa proizlazi iz toga što su podaci jasno odvojeni jedni od drugih (ukoliko je zapis ispravan i nije se potkralo da se separator koristi unutar podataka) i što su podijeljeni u stupce i retke (stupce definira prvi red zaglavlja).

4.3.1. Parsing CSV zapisa

Prilikom parsinga CSV zapisa koristit ćemo biblioteku pandas. Pandas je biblioteka specijalizirana za znanost o podacima, a nama će služiti za pretvaranje CSV zapisa u Dataframe objekt s kojim možemo lakše raditi u programskom jeziku. [32]

Na primjeru CSV zapisa s podacima studenata ćemo pokazati kako se pomoću biblioteke Pandas može jednostavno učitati, obrađivati i pretraživati CSV datoteka. Prva stvar koju trebamo učiniti je učitati CSV datoteku. To činimo tako da proslijedimo putanju prema CSV datoteci koju smo spremili lokalno i definiramo separator koji se koristi u datoteci kao argumente funkcije pandas.read_csv. Ukoliko smo učinili sve kako treba (definirali ispravan separator) prilikom ispisa bi nam se trebao pojaviti strukturiran ispis (nalik tablici) u kojem su stupci odvojeni razmacima umjesto separatorima korištenima u originalnom CSV-u. Ukoliko smo pogriješili prilikom definiranja separatora (ili smo ga propustili deklarirati, a nismo koristili standardni zarez kao separator u dokumentu) dobit ćemo ispis jednak originalnom dokumentu.

Jedna od stvari koju možemo vrlo jednostavno napraviti kada koristimo biblioteku pandas je sortirati vrijednosti u zapisu kojeg smo dobili iz CSV-a. Pozovemo funkciju sort_values nad učitanim popisom i kao argumente proslijedimo obavezni argument *by* koji definira po kojem stupcu će se vrijednosti sortirati. Drugo što možemo definirati je smjer sortiranja (uzlazno/silazno), ali to nije obavezno (ako se ne definira standardno je uzlazno).

Pretraživanje zapisa je u mnogim stvarima slično JSON-u. Možemo dobiti sve vrijednosti nekog stupca ukoliko proslijedimo u indeks popisa ime stupca u dvostrukim

navodnicima (popis[ime]). Ukoliko želimo pretražiti cijeli zapis u cilju pronalaska specifične vrijednosti (u našem primjeru ime i prezime studenta s JMBAG-om kojeg pretražujemo) korisna će nam biti funkcija iloc. Iloc označava index of location i olakšava pretraživanje jer kao vrijednost indeksa lokacije prosljeđujemo iterator dok je druga vrijednost traženi stupac (ili može bez druge vrijednosti ako nam treba cijeli stupac).

```
import pandas as pd

popis = pd.read_csv("C:/Users/Fabio/Desktop/zavrsni/popis.csv", sep =
";")
sort_popis = popis.sort_values(by='Ime', ascending=False)
for i in range (len(sort_popis)):
    print(sort_popis.iloc[i]['JMBAG'])
    if sort_popis.iloc[i]['JMBAG'] == 123456789:
        print(f"{sort_popis.iloc[i]['Ime']}
{sort_popis.iloc[i]['Prezime']}" )
```

4.3.2. Vizualizacija CSV zapisa

Vizualizacija CSV zapisa se koristi jednako kao i vizualizacija prethodno spomenutih zapisa. Za uspješnu vizualizaciju potrebno je prije svega doći do podataka, a nakon toga uspješno koristiti alate (biblioteke) za vizualizaciju.

Vizualizaciju CSV datoteka ćemo prikazati na primjeru podataka o trošku zdravstva za zemlje s privatnim zdravstvom (Island, Norveška, Švicarska, SAD i Slovenija). Želja nam je prikazati kako u zadnjih nekoliko godina troškovi zdravstva značajno rastu, ali i uz to kako se kreću troškovi pojedinih zemalja. [33]

Kako bi to učinili koristit ćemo stupčasti dijagram, ali uz par dodataka. Kako bi uspješno prikazali dva sloja podataka sa stupčastim dijagramom (podaci o trošku po zemljama i podaci o trošku po godinama) koristit ćemo dijagram na kojem će jedan stupac predstavljati godinu za koju prikazujemo podatke, a on će se sastojati od nekoliko boja koje prikazuju pojedinu zemlju. Također s obzirom da je riječ o velikoj količini podataka za prikazati što bi moglo otežati interpretaciju koristit ćemo oblik dijagrama koji će sadržavati i tablicu s vrijednostima ispod. Poveznica dijagrama i tablice će biti u zajednički nazivima stupaca.

Prva stvar koju ćemo učiniti je dohvatiti podatke i pridružiti vrijednosti podataka u tri segmenta. Prvi je podaci za rad koje spremamo u varijablu *data,* drugi je podaci koji će predstavljati retke, a treći su podaci koji predstavljaju stupce. Koristeći prethodno definirane korake za parsing CSV zapisa i neke zgodne mogućnosti što nam dopušta Python (npr. set za dobivanje unikatne liste) dolazimo do podataka.

```
def dohvati_podatke(putanja):
    popis = pd.read_csv(putanja, sep = ";")
    drzave = list(set(popis['Zemlja']))
    troskovi = []
    for drzava in drzave:
        trosak = []
```

```
            troskovi.append(trosak)
            for i in range(len(popis)):
                if popis.iloc[i]['Zemlja'] == drzava:
                    trosak += [popis.iloc[i]['Trosak']]
        return troskovi,drzave

    columns = ['%d. godina' % x for x in (2013, 2014, 2015, 2016, 2017)]
    data,rows                                                          =
dohvati_podatke("C:/Users/Fabio/Desktop/zavrsni/troskovi.csv")
    values = np.arange(0, 30000, 5000)
```

Nadalje moramo definirati prikaz. Kako imamo više stupaca moramo paziti na njihovu širinu, udaljenost jednog od drugog i sl. Također moramo definirati boje što možemo učiniti s definiranim paletama. Nakon toga pozivamo iteraciju s kojom ćemo proći kroz podatke i nacrtati stupce. Također definirat ćemo listu i u nju spremati podatke koje ćemo koristiti za prikaz tablice.

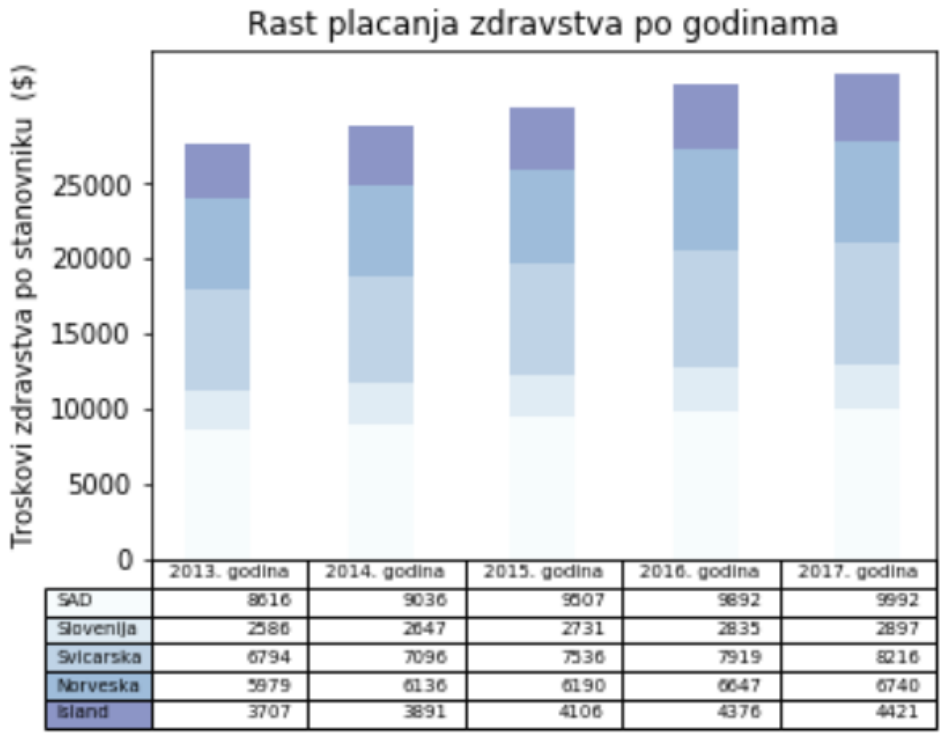
```
    columns = ['%d. godina' % x for x in (2013, 2014, 2015, 2016, 2017)]
    data,rows                                                          =
dohvati_podatke("C:/Users/Fabio/Desktop/zavrsni/troskovi.csv")
    values = np.arange(0, 30000, 5000)
    colors = plt.cm.BuPu(np.linspace(0, 0.5, len(rows)))
    n_rows = len(data)
    index = np.arange(len(columns)) + 0.3
    bar_width = 0.4
    y_offset = np.zeros(len(columns))
    cell_text = []

    for row in range(n_rows):
        plt.bar(index,     data[row],     bar_width,     bottom=y_offset,
color=colors[row])
        y_offset = y_offset + data[row]
        cell_text.append(data[row])
```

Na kraju ostaje nam samo nacrtati tablicu s već definiranim podacima i dodati neke dodatne dijelove koje olakšavaju interpretaciju kao npr. labele.

```
    the_table = plt.table(cellText=cell_text,
                          rowLabels=rows,
                          rowColours=colors,
                          colLabels=columns,
                          loc='bottom')
plt.subplots_adjust(left=0.2, bottom=0.2)
plt.ylabel("Troskovi zdravstva po stanovniku  ($)")
plt.yticks(values , ['%d' % val for val in values])
plt.xticks([])
plt.title('Rast placanja zdravstva po godinama')


plt.show()
```



	2013. godina	2014. godina	2015. godina	2016. godina	2017. godina
SAD	8616	9036	9507	9892	9992
Slovenija	2586	2647	2731	2835	2897
Svicarska	6794	7096	7536	7919	8216
Norveska	5979	6136	6190	6647	6740
Island	3707	3891	4106	4376	4421

Slika 12: Prikaz rasta troškova zdravstva kroz stupčasti dijagram s tablicom

5. Zaključak

Kako je bilo i najavljeno u uvodu u ovom radu smo se bazirali na definiranje pojmova strukturiranih, polustrukturiranih i nestrukturiranih podataka i utvrđivanjem razlika među njima. Prikazani su najvažniji predstavnici tih grupa te je dana teorijska osnova prema kojoj određujemo njihov stupanj strukturiranosti.

U praktičnim primjerima pokazali smo kako se radi s kojim tipom podataka, kako ih strukturiramo te naposljetku vizualiziramo. Fokus je dan na sam rad s podacima kako bismo dobili kvalitetan skup podataka koji ćemo kasnije koristiti za vizualizaciju. Stavljena je važnost na dobivanje podataka iz zapisa baš zato što je to najbitniji preduvjet za vizualizaciju. Jer vizualizacija, iako jako bitna, je i dalje pomoćna radnja kod obrade podataka i neizvediva bez prethodno pripremljenih podataka.

Još jedna bitna značajka je bila prikazati kako se Python „snalazi“ u svijetu rada s podacima. U to svrhu smo koristili brojne biblioteke koji se koriste za različite svrhe kako bi pokazali širinu spektra i specijaliziranost za svaki pojedini zapis koju nudi Python. Baš zbog toga su i spomenute neke biblioteke bez kojih smo mogli prikazati sve što smo željeli u primjerima, ali su jako korisne jer olakšavaju posao. Također kada bismo radili veći projekt neke od tih biblioteke bile bi neizbježne za koristiti pa ih valja spomenuti (Pandas i BeautifulSoup ponajviše).

Sama obrada vizualizacije možda nije prikazana kroz prizmu nekih „spektakularnih“ dijagrama koji plijene pažnju već je bazirana na temeljne primjere koji na jednostavan način prikazuje mogućnosti koji kasnije možemo nadograđivati kada nam priroda projekta to bude zahtijevala.

Glavni zaključak što bi se dao izvući je da je strukturiranje zapisa apsolutno nužno za bilo kakav daljnji rad s podacima. Polustrukturirani zapisi su također vrlo jednostavni za korištenje i postoje brojne biblioteke koje olakšavaju rad s njima tako da će se oni zasigurno i dalje u velikoj mjeri koristiti. Isto tako možemo reći s obzirom koliko mogućnosti imamo za korištenje polustrukturiranih zapisa da su oni i jednostavniji za korištenje od strukturiranih (iako njihova primjena nipošto nije istovjetna). Python kao programski jezik nudi nepregledan broj biblioteka za vizualizaciju i strukturiranje raznih zapisa te je kao takav idealan za korištenje. Razlozima za popularnost Pythona u ovom području mogli smo svjedočiti u prethodnim primjerima, kada se pokazalo kako za korištenje većine biblioteka ne treba puno vremena da ih savladamo (jednostavna sintaksa), a nude široke mogućnosti.

Popis literature

[1] W3 Training school (bez dat.), *Structured, Semi-Structured And Unstructured Data.* Dostupno: https://www.w3trainingschool.com/structured-semi-structured-unstructured-data [pristupano 15.07.2019].

[2] Gregory Piatetsky, *R, Python Duel As Top Analytics, Data Science software,* 2016. Dostupno: https://www.kdnuggets.com/2016/06/r-python-top-analytics-data-mining-data-science-software.html [pristupano 17.07.2019.]

[3] SQLdbm, *Main features of sqldbm modeling tool*, 2017. Dostupno: http://blog.sqldbm.com/main-features-of-sqldbm-modeling-tool/ [pristupano 19.07.2019.]

[4] Rishab Sharma, *Data modelling,* 2017. Dostupno :https://medium.com/@rishusharma.sharma7/data-modelling-b63ffc6a4b64. pristupano[21.07.2019.]

[5] George Tillman, *Bilding a Logical Data Model,*1995. Dostupno: https://web.archive.org/web/20080509063521/http://www.dbmsmag.com/9506d16.html, pristupano [21.07.2019.] .

[6] MariaDB (bez dat.), *Understanding The Hierarchial Database Model,* https://mariadb.com/kb/en/library/understanding-the-hierarchical-database-model/ [pristupano 21.07.2019.]

[7] Vishal Sharma (bez dat.), *Big data storage and old dbms technique comparison* https://www.analyticbridge.datasciencecentral.com/profiles/blogs/big-data-storage-and-old-dbms-te8chnique-comparison, pristupano [21.07.2019.]

[8] IRS (bez dat.), *Database design techniques,*Dostupno: https://www.irs.gov/irm/part2/irm_02-005-013, pristupano [21.07.2019.]

[9] How stuff works (bez dat.), *What are relational databases,* Dostupno: https://computer.howstuffworks.com/question599.htm, pristupano[22.07.2019.]

[10] Neo4j (bez dat.), *Graph database,* Dostupno: https://neo4j.com/developer/graph-database/, pristupano [22.07.2019.]

[11]Techopedia (bez dat.), *Object oriented database,* Dostupno: https://www.techopedia.com/definition/8639/object-oriented-database, pristupano[22.07.2019.]

[12] Techterms (bez dat.), *DBMS,* Dostupno: https://techterms.com/definition/dbms, pristupano[23.07.2019.]

[13] Python.org (bez dat.), *Sqlite3,* Dostupno: https://docs.python.org/3.6/library/sqlite3.html, pristupano[25.07.2019.]

[14] Corey Schafer,(18.04.2017.) „Python SQLite Tutorial: Complete Overview - Creating a Database, Table, and Running Queries“, Youtube [video datoteka], Dostupno: https://www.youtube.com/watch?v=pd-0G0MigUA, pristupano[25.07.2019.]

[15] Smartdraw (bez dat.), *Entity relationship diagram,* Dostupno: https://www.smartdraw.com/entity-relationship-diagram/, pristupano[27.07.2019.]

[16] PyPi.org (bez dat.), *ERAlchemy,* Dostupno: https://pypi.org/project/ERAlchemy/, pristupno [27.07.2019.]

[17] Mary Shacklett, *Unstructured data,* 2017. Dostupno: https://www.techrepublic.com/article/unstructured-data-the-smart-persons-guide/, pristupano [28.07.2019.]

[18] Expertsystem, *Natural language processing and text mining,* 2016. Dostupno: https://www.expertsystem.com/natural-language-processing-and-text-mining/, pristupano[29.07.2019.]

[19] Geeks for geeks (bez dat.), *Difference between structured, semi-structured and unstructured data,* Dostupno : https://www.geeksforgeeks.org/difference-between-structured-semi-structured-and-unstructured-data/. Pristupano [01.08.2019.]

[20] W3 Schools (bez dat.), *What is XML,* Dostupno: https://www.w3schools.com/XML/xml_whatis.asp. Pristupano [01.08.2019.]

[21] Inc.com (bez dat.), *Why companies should use XML,* Dostupno: https://www.inc.com/guides/2010/04/why-companies-should-use-xml.html. Pristupano [01.08.2019.]

[22] W3 Schools (bez dat.), *XML tree,* Dostupno: https://www.w3schools.com/xml/xml_tree.asp. Pristupano [01.08.2019.]

[23] Python.org (bez dat.), *XML.etree.elementtree* , Dostupno: https://docs.python.org/3.6/library/xml.etree.elementtree.html. Pristupano [02.08.2019.]

[24] Amrita Mitra, *What is billion laughs attack,* 2017. Dostupno: https://www.thesecuritybuddy.com/dos-ddos-prevention/what-is-billion-laughs-attack/. Pristupano [04.08.2019.]

[25] Christian Heimes, *Defusedxml,* 2019. Dostupno: https://pypi.org/project/defusedxml/. Pristupano [04.08.2019.]

[26] Matplotlib (bez dat.), *Spines.* Dostupno: https://matplotlib.org/3.1.1/api/spines_api.html.

Pristupano [07.08.2019.]

[27] W3 Schools (bez dat.), *JSON intro,* Dostupno: https://www.w3schools.com/js/js_json_intro.asp. Pristupano [10.08.2019.]

[28] W3 Schools (bez dat.), *JSON syntax,* Dostupno: https://www.w3schools.com/js/js_json_syntax.asp. Pristupano [10.08.2019.]

[29] W3 Schools (bez dat.), *JSON vs XML,* Dostupno: https://www.w3schools.com/js/js_json_xml.asp. Pristupano [11.08.2019.]

[30] Python.org (bez dat.), *JSON,* Dostupno: https://docs.python.org/3.6/library/json.html. Pristupano[12.08.2019.]

[31] Ansible.com (bez dat.), *YAML,* Dostupno: https://docs.ansible.com/ansible/latest/reference_appendices/YAMLSyntax.html. Pristupano [14.08.2019.]

[32] Python.org (bez dat.), *CSV,* Dostupno: https://docs.python.org/3.6/library/csv.html. Pristupano[16.08.2019.]

[33] Pandas (bez dat.), *pandas.read_csv,* Dostupno: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html. Pristupano[17.08.2019.]

[34] OECD.org (bez dat.), *Health expenditures and resources* . Dostupno: https://stats.oecd.org/Index.aspx?DataSetCode=SHA 2019. Pristupano [20.08.2019.]

Popis slika

Slika 1: Eksponencijalni rast podataka u posljednjih nekoliko godina 1
Slika 2: Tržišni udjel softwarea za obradu podataka 2
Slika 3: Konceptualni model podataka 4
Slika 4: Logički model podataka 5
Slika 5: Slojevi aplikacije 6
Slika 6: Tipovi podataka u Pythonu u usporedbi sa sqlite-om 8
Slika 7: Histogram za relacijski model 11
Slika 8: Tortni dijagram – prikaz udjela studenata po smjerovima 20
Slika 9: Spine graf s prikazom frekvencijom praznika u kalendarskoj godini 23
Slika 10: Stupčasti dijagram prolaznosti studenata 27
Slika 11: 3D scatter prikaz podataka o vremenu 29
Slika 12: Prikaz rasta troškova zdravstva kroz stupčasti dijagram s tablicom 34