UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Anže Vavpetič

Uporaba ontologij kot predznanja v podatkovnem rudarjenju

DIPLOMSKO DELO
NA UNIVERZITETNEM ŠTUDIJU

Mentor: prof. dr. Igor Kononenko
Somentorica: prof. dr. Nada Lavrač

Ljubljana, 2011

Univerza
v Ljubljani

Fakulteta za računalništvo
in informatiko

Tržaška 25
1000 Ljubljana, Slovenija
telefon: 01 476 84 11
faks: 01 426 46 47
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 01752/2011
Datum: 01.04.2011

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **ANŽE VAVPETIČ**

Naslov: **UPORABA ONTOLOGIJ KOT PREDZNANJA V PODATKOVNEM RUDARJENJU**

THE USE OF ONTOLOGIES AS BACKGROUND KNOWLEDGE IN DATA MINING

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

Cilj diplomskega dela je razvoj metodologije in prilagoditev obstoječega sistema za podatkovno rudarjenje SEGS, ki omogoča analizo rangiranih podatkov z uporabo predznanja v obliki ontologij. Poleg funkcionalnosti sistema SEGS naj v diplomski nalogi razviti sistem omogoča tudi delo z labeliranimi (dvorazrednimi) podatki, predznanje sistema pa naj bodo namesto treh bioloških ontologij lahko tudi poljubne druge ontologije. Jezik za opis odkritih podskupin naj bodo pravila, katerih konjunkti so izrazi, ki opisujejo koncepte v izbranih ontologijah. Sistem naj bo implementiran kot delotok (workflow) spletnih servisov, ki ga bodo uporabniki lahko uporabljali v okviru sistema za rudarjenje podatkov Orange4WS. Diplomsko delo naj pokaže ustrezen način formulacije naloge tudi v ILP sistemu Aleph.

Mentor:

prof. dr. Igor Kononenko

Dekan:

prof. dr. Nikolaj Zimic

Somentor:

prof. dr. Nada Lavrač

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

IZJAVA O AVTORSTVU

diplomskega dela

Spodaj podpisani Anže Vavpetič,

z vpisno številko 63060306,

sem avtor diplomskega dela z naslovom:

Uporaba ontologij kot predznanja v podatkovnem rudarjenju

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom
  prof. dr. Igorja Kononenka
  in somentorstvom
  prof. dr. Nade Lavrač
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 22.6.2011 Podpis avtorja:

Zahvala

Zahvaljujem se mentorju prof. dr. Igorju Kononenku in somentorici prof. dr. Nadi Lavrač za vse nasvete in pozorno oko pri izdelavi diplomske naloge.

Za nasvete in pomoč se zahvaljujem še dr. Petri Kralj Novak, Vidu Podpečanu in dr. Vitorju Santos Costi.

Za podporo, uspešno lovljenje vejic in slogovne komentarje se zahvaljujem Kaji Bucik.

Kazalo

Seznam uporabljenih kratic in simbolov

DM - *Data Mining*
GO - *Gene Ontology*
ILP - *Inductive Logic Programming*
KDD - *Knowledge Discovery in Databases*
OWL - *Web Ontology Language*
RDF - *Resource Description Framework*
SEGS - *Search for Enriched Gene Sets*
SOAP - *Simple Object Access Protocol*
WSDL - *Web Services Description Language*

Seznam prevedenih izrazov

Accuracy - *točnost*
Binding - *vezava*
Coverage - *pokritost*
Data mining - *rudarjenje podatkov*
Inheritance network - *dedovalna mreža*
Knowledge Discovery in Databases (KDD) - *odkrivanje znanja iz podatkovnih baz*
Microarray - *mikromreža*
Namespace - *imenski prostor*
Pattern discovery - *odkrivanje vzorcev*
Precision - *natančnost*
Refinement operator - *ostrilni operator*
Relational rule learning - *relacijsko učenje pravil*
Semantic web - *semantični splet*
Stub - *štrcelj*
Subgroup discovery - *odkrivanje podskupin*
Workflow - *delotok*

Povzetek

V diplomskem delu je opisan razvoj aplikacije za odkrivanje podskupin g-SEGS, ki omogoča uporabo ontologij kot predznanja. Aplikacija ontološke koncepte uporablja kot člene v pravilih, ki opisujejo podskupine primerov. Razviti sistem je posplošitev sistema SEGS, ki je bil že uspešno uporabljen na področju genomike, vendar ne omogoča aplikacije na druge domene. Sistem g-SEGS je bil implementiran v obliki spletnega servisa, ki ga je mogoče uvoziti v različne aplikacije za delo s spletnimi servisi. Naloga pokaže uporabo sistema kot gradnika v okviru okolja za podatkovno rudarjenje Orange in njegove nadgradnje Orange4WS; za to okolje, ki omogoča vizualno programiranje, je bil razvit tudi poseben uporabniški vmesnik. Diplomsko delo nato opiše, kako nalogo iskanja podskupin, kjer želimo za predznanje uporabiti ontologije, formuliramo tudi za sistem induktivnega logičnega programiranja Aleph. Na koncu diplomsko delo še eksperimentalno oceni delovanje obeh pristopov na umetni domeni in dveh resničnih bioloških domenah ter navede ideje za nadaljnje delo.

Ključne besede:

odkrivanje podskupin, relacijsko učenje, induktivno logično programiranje, spletni servisi, ontologije

Abstract

The thesis describes the development of an application for subgroup discovery called g-SEGS, which supports the use of ontologies as background knowledge. The application can use ontological concepts as terms in rules, which describe subgroups of examples. The system is a generalization of an existing system SEGS, which was successfully used in the field of genomics, but it cannot be applied to other fields. The system g-SEGS was implemented as a web services and can thus be imported into various applications which support the use of web services. The thesis also presents the use of g-SEGS as a widget in the Orange data mining environment for visual programming and its extension Orange4WS; an additional, easy-to-use user interface was implemented for this purpose. Next, the thesis describes how to formulate the problem of using ontologies in background knowledge for subgroup discovery in the inductive logic programming system Aleph. Both approaches were experimentally evaluated on a toy domain and on two real-life biological domains. Lastly, the thesis provides some ideas for future work.

Key words:

subgroup discovery, relational learning, inductive logic programming, web services, ontologies

Poglavje 1

Uvod

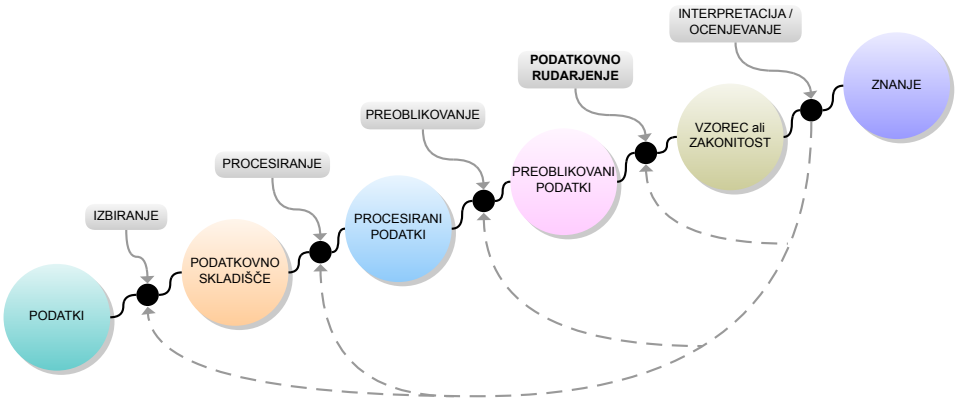
V vsakdanjem življenju ljudje neprestano uporabljamo vzorce. Na podlagi preteklih izkušenj in trenutnih informacij se odločimo, kaj bo naša naslednja poteza - poteza, ki jo narekuje na izkušnjah naučeni vzorec. Če nekdo zakliče naše ime in glas prepoznamo kot glas svojega znanca, se bomo ozrli v smeri glasu. Seveda pričakujemo, da bomo zagledali znan obraz. V primeru, da ga ne, se znajdemo v stanju nelagodja, saj nam tega naše pretekle izkušnje (podobna situacija se nam je tekom življenja verjetno ponovila že velikokrat) in naše zaznave niso napovedale. Vzorci predstavljajo red in konsistenco in nam omogočajo razlikovati znano od neznanega.

Vzorce lahko iščemo tudi v podatkih. Ena od nalog, ki si jih zadajata statistika in področje računalništva, imenovano podatkovno rudarjenje, je ugotavljanje, kaj lahko povemo o sistemu, o katerem smo zbrali nekaj meritev njegovih lastnosti.

Podatkovno rudarjenje (*ang. data mining*, s kratico DM) je osrednji del procesa odkrivanja znanja iz podatkovnih baz (*ang. knowledge discovery in databases*, s kratico KDD). Da bi razumeli bistvo podatkovnega rudarjenja, se moramo najprej ustaviti pri definiciji procesa odkrivanja znanja iz podatkovnih baz.

Odkrivanje znanja iz podatkovnih baz je po [1] netrivialen proces odkrivanja veljavnih, novih, potencialno uporabnih in razumljivih vzorcev iz podatkov.

Če se opremo na to opredelitev, je podatkovno rudarjenje korak procesa KDD, ki se ukvarja z uporabo računskih tehnik za iskanje vzorcev ali zakonitosti, značilnih za dane podatke [2, poglavje 1]. Podatkovno rudarjenje ima osrednjo vlogo v procesu KDD. Preostali koraki služijo pripravljanju podatkov na rudarjenje ali pa gre za intepretacijo in ocenjevanje ugotovljenih vzorcev.



Slika 1.1: Shema procesa odkrivanja znanja iz podatkovnih baz.

Vlogo podatkovnega rudarjenja v kontekstu procesa KDD prikazuje slika 1.1.

Vzorci služijo odkrivanju posameznih zakonitosti v opisnem učenju (*ang. descriptive learning*). Poleg vzorcev pa predstavljajo velik del strojnega učenja in podatkovnega rudarjenja tudi modeli, ki služijo napovedovanju v napovednem učenju (*ang. predictive learning*).

Vzorce in modele najpogosteje gradimo nad dvo-dimenzionalnimi tabelami, kjer imamo v vrsticah posamezne primere, v stolpcih atribute, elementi tabele pa so vrednosti atributov. Taki predstavitvi rečemo propozicionalna predstavitev. Ni presenetljivo, da se pojavlja potreba po iskanju vzorcev tudi v različnih drugih (bolj kompleksnih) virih podatkov. Navedimo nekaj primerov:

- več povezanih tabel (relacij),
- grafi,
- besedila,
- slike, itd.

V takih primerih je pogost pristop, da podatke iz prvotne (bolj kompleksne) oblike prepišemo v (bolj obvladljivo) obliko ene tabele, z določenim številom stolpcev. Problem take pretvorbe je lahko nepopolna predstavitev podatkov, saj se moramo zaradi določenega števila stolpcev zateči k atributom, ki zgolj povzamejo določene lastnosti kompleksnejše strukture (npr. povprečno število vhodnih povezav v vozlišča v danem grafu).

Poleg oblike zapisa vhodnih podatkov je pogosto pomembno tudi, kakšne oblike *vzorcev* oz. *modelov* nam dani algoritem lahko poišče - glede tega pa

se pogosto odločamo različno od problema do problema. Navedimo nekaj primerov modelov in vzorcev [2, poglavje 1]:

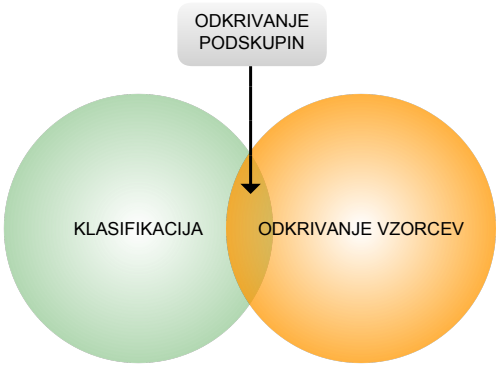
- klasifikacijska in regresijska drevesa (modeli),
- enačbe (modeli),
- asociacijska, klasifikacijska in regresijska pravila (vzorci),
- pravila, ki opisujejo podskupine primerov (*ang. subgroups*) (vzorci).

Za to delo je relevantna predvsem kombinacija *odkrivanja opisov podskupin* v *več tabelah*. Naloga odkrivanja podskupin je bila v [3, 4] definirana na naslednji način. Dano imamo populacijo objektov ali osebkov in eno izmed njihovih lastnosti, ki nas zanima. Poišči podskupine te populacije, ki so statistično najbolj zanimive. Zanimiva podskupina je lahko npr. taka, ki je čim večja in (v smislu distribucije) statistično najbolj nenavadna glede na lastnost, ki smo si jo izbrali.

Po svojih značilnostih je naloga odkrivanja podskupin nekje v preseku med klasifikacijo in klasičnim odkrivanjem vzorcev (*ang. pattern discovery*). Idejo ponazarja slika 1.2. Nalogi klasifikacije je podobna, ker se algoritmi učijo pravil (s pravili tu mislimo simbolične opise odkritih podskupin), ki opisujejo skupine primerov iz označenih podatkov (vsak primer pripada nekemu razredu - vrednosti ciljne spremenljivke). Podobnost z nalogo odkrivanja vzorcev pa velja zaradi simbolične narave odkritih vzorcev (pravil), ki so namenjeni človeški interpretaciji, ter zato, ker so cilj odkrivanja posamezni vzorci (pravila) in ne množica pravil, ki sestavlja klasifikacijski model.

Nabor pravil, ki jih dana metoda za odkrivanje podskupin izlušči, se lahko uporabi tudi za klasifikacijo novih, še ne videnih primerov, četudi klasifikacija ni glavni cilj odkrivanja podskupin. Tak seznam pravil je običajno neurejen (*ang. unordered*) - to pomeni, da nima določenega vrstnega reda, po katerem bi se dana pravila brala ali prožila (v nasprotju z npr. klasifikacijskimi pravili (*ang. classification rules*)). Za klasifikacijski model poleg seznama pravil potrebujemo še klasifikacijsko shemo. Klasifikacijska shema iz množice pravil določi razred novega, vhodnega primera. Primer enostavne klasifikacijske sheme je glasovanje, kjer vsako pravilo glasuje za določen razred; za razred novega primera pa se vzame razred z največ glasovi.

Običajno se bo klasifikacijski model, izdelan iz opisov podskupin, odrezal slabše, saj posamezna pravila ne optimizirajo klasifikacijske točnosti, temveč sklepajo kompromis med pokritostjo primerov (*ang. coverage*) in natančnostjo (*ang. precision*).



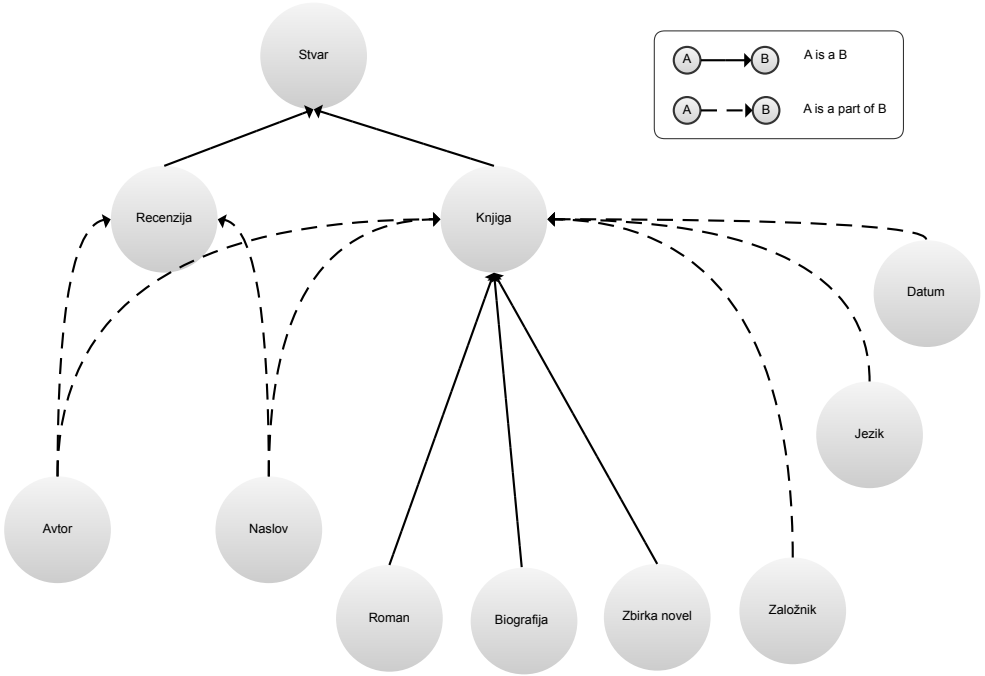
Slika 1.2: Odkrivanje podskupin v relaciji s klasifikacijo in klasičnim odkrivanjem vzorcev.

Za to delo so koristni tudi pristopi s področja induktivnega logičnega programiranja (*ang. Inductive Logic Programming*, s kratico ILP). Na tem mestu le na kratko povzemamo nalogo ILP (podrobnejši opis logičnega programiranja in ILP lahko najdemo v [2, poglavje 3]).

Najpogostejša naloga, ki jo obravnava ILP, je naloga učenja logičnih definicij relacij iz danih vhodnih primerov (n-terke z oznako, ali pripadajo ciljni relaciji ali ne). Sistem ILP iz učnih primerov nato sestavi logični program (definicijo predikata) iz drugih relacij, ki so podane kot predznanje. Jezik hipotez (definicij ciljnega predikata) so Hornovi stavki - podmnožica logike prvega reda. Jezik Hornovih stavkov je močnejši od Boolove algebre, ki je osnova za učenje pravil za atributne probleme.

V okolju, razvitem v okviru tega dela, za predznanje pri učenju uporabljamo ontologije. Pojem ontologije izhaja iz filozofije. Veda, ontologija, se ukvarja z vprašanji kot npr., katere entitete obstajajo in kako jih lahko združujemo v okviru hierarhije glede na njihove podobnosti in razlike.

V računalništvu se je pojem ontologija pojavil v okviru umetne inteligence z namenom ustvarjanja skupnega besednjaka za komunikacijo med različnimi sistemi. V računalništvu je ontologija definirana kot eksplicitna specifikacija konceptualizacije [5]. Ontologija je sestavljena iz množice konceptov iz dane domene, razmerij med temi koncepti in aksiomov, ki določajo možne interpretacije tega znanja. Primer majhne ontologije iz domene knjig je prikazan na sliki 1.3. Slika prikazuje koncepte (knjiga, recenzija, avtor, naslov, itd.) in relacije med temi koncepti. Primer prikazuje dva tipa relacij ('is-a' in 'part-of'), a v splošnem ima ontologija lahko poljubno veliko različnih relacij.



Slika 1.3: Primer ontologije iz domene knjig.

1.1 Opis problema

Problem, ki ga obravnava diplomska naloga, je razvoj okolja za uporabo ontologij kot predznanja v podatkovnem rudarjenju, bolj natančno: za odkrivanje statistično nenavadnih podskupin. Jedro okolja tvori posplošen agoritem sistema SEGS (Search for Enriched Gene Sets) [6]. Posplošitev imenovana g-SEGS je bila potrebna zato, da bo lahko naš sistem deloval na poljubni domeni. Sistem g-SEGS je implementiran deloma v programskem jeziku C (osnovni sistem SEGS), deloma v jeziku Java (napredne operacije nad ontologijami v zapisu OWL (Web Ontology Language)), deloma pa v jeziku Python (koda, ki skrbi za komunikacijo prek jezika SOAP ter uporabniški vmesnik).

Da bi bil sistem g-SEGS uporabniku čim bolj prijazen, smo se odločili okolje vključiti v sistem za strojno učenje, ki omogoča uporabniku izvajanje eksperimentov v obliki delotokov (*ang. workflows*). Uporabnik lahko tako s povezovanjem različnih gradnikov na enostaven način sestavi in požene eksperiment ter pregleduje dobljene rezultate. Problemi, ki jih rešuje razvito okolje, so lahko računsko (časovno in prostorsko) precej potratni, zato je bilo smiselno (tudi zaradi relativno dobre podpore porazdeljenemu računanju v okoljih za strojno učenje) gradnike tega okolja razviti v obliki spletnih servisov. Odločili

smo se za sistem Orange4WS [7], ki temelji na okolju za strojno učenje Orange [8]. Orange4WS nudi možnost uvoza spletnih servisov in njihovo uporabo v delotokih. Nudi tudi knjižnico, ki razvijalcu olajša izdelavo spletnih servisov.

Med drugim v diplomski nalogi tudi pokažemo, kako lahko uporabimo ontologije kot predznanje za učenje s pomočjo sistema ILP Aleph in nato primerjamo razultate, dobljene v obeh pristopih.

Preostanek tega poglavja se osredotoči na motivacijo za razvoj okolja g-SEGS in povzame sorodne obstoječe tehnologije. V drugem poglavju so opisane metode in orodja, uporabljena pri razvoju našega sistema. Tretje poglavje predstavi razvito okolje in načine uporabe. V četrtem poglavju opišemo sistem ILP Aleph in postopek formulacije problema uporabe ontologij kot predznanja za ta sistem. V petem poglavju primerjamo naše okolje s sistemom Aleph; najprej na izmišljenem primeru, nato pa na dveh resničnih bioloških primerih. V zaključku povzamemo naše ugotovitve in navedemo ideje za nadaljnje delo.

1.2 Motivacija

Že kar nekaj časa so poznani algoritmi, ki znajo direktno izkoriščati izrazno moč raznih formalizmov (npr. ILP) in se s tem ne omejujejo več zgolj na eno tabelo (*ang. single-table assumption*), ki predstavlja "klasični" zapis vhodnih podatkov za naloge strojnega učenja in podatkovnega rudarjenja.

Tako poznamo tudi algoritme, ki so zmožni dela nad podatki, zapisanimi v več-relacijskih podatkovnih bazah (npr. sistem MIDOS [9] za več-relacijsko odkrivanje podskupin). Tabelam v okviru podatkovnih baz in logike prvega reda rečemo relacije; tudi v tem delu bomo od zdaj naprej govorili o relacijah. V uvodu omenjeni pristop ILP je primer orodja, ki ga lahko direktno apliciramo nad več-relacijskimi problemi, saj nanj lahko gledamo tudi kot na relacijsko učenje pravil (*ang. relational rule learning*) [2, poglavje 4].

Teoretično sicer lahko katerokoli več-relacijsko podatkovno bazo zapišemo kot eno tabelo. V takih primerih je problematična redundantnost zapisov ali pa težavno vzdrževanje podatkov, če en predmet našega preučevanja ne ustreza le eni vrstici [2, poglavje 4]. Pogosto algoritmi predpostavijo nasprotno - da enemu predmetu proučevanja ustreza natanko ena vrstica.

Na tem mestu omenimo, da za posebne primere, kjer so si relacije v bazi v odnosu ena-proti-mnogo (*ang. one-to-many*), poznamo postopek propozicionalizacije preko značilk logike prvega reda, opisan v [10]. S pomočjo tega postopka lahko učinkovito, skoraj brez izgube izrazne moči, več-relacijski problem pretvorimo v problem z eno relacijo.

Seveda se v praksi pojavljajo naloge za več-relacijsko učenje. Skoraj vsi informacijski sistemi imajo dandanes za osnovo več-relacijsko podatkovno bazo; pogosta naloga podatkovnega rudarjenja je tudi analiza anket, ki so lahko precej kompleksne in potrebujejo več-relacijsko predstavitev; znane so tudi aplikacije v kemiji, kjer je več-relacijska predstavitev zelo naravna za zapis spojin [2, poglavje 4]. Velika izrazna moč pa nam bo za opise podskupin koristila tudi v tem delu.

Ugotovitve, ki so še posebej motivirale razvoj okolja, obravnavanega v tem delu, se nanašajo predvsem na biološko domeno genomike. V [5] je opisan sistem SEGS, ki na podlagi meritev, zajetih s pomočjo mikromrež (*ang. microarrays*), iz vzorca tkiva generira opise skupin genov, ki so diferenčno izraženi.

Vsaka celica organizma ima enak nabor genov, vendar je v določeni celici aktivna le določena podmnožica genov. Nabor izraženih genov določa edinstvene lastnosti posameznih celic. Izraženost genov je dinamičen proces in se spreminja glede na vplive okolja in glede na celične lastne potrebe [11]. Z opazovanjem diferenčno izraženih genov v tkivih, ki so izpostavljeni različnim pogojem (npr. primerjava zdravega in okuženega tkiva), lahko biologi bolje razumejo biološke procese, ki se odvijajo v celicah [5].

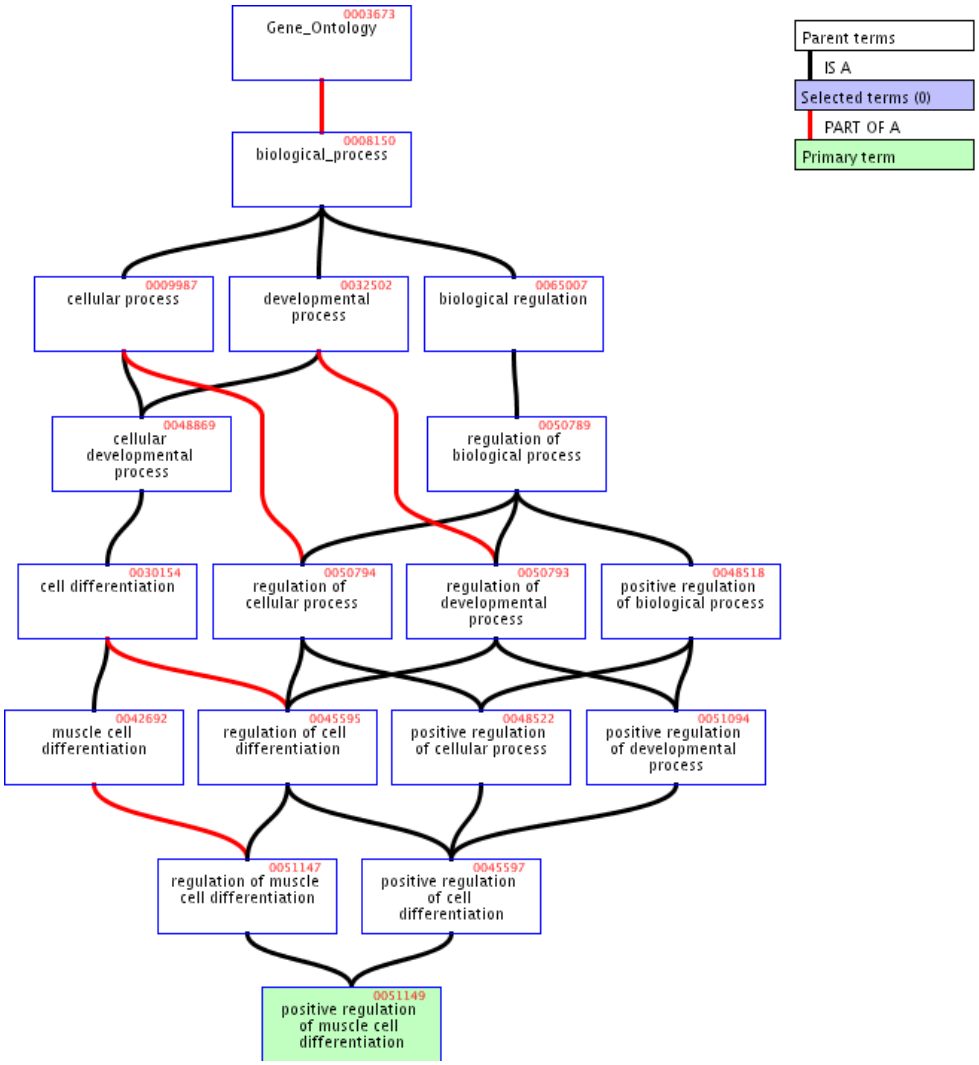
Ročna analiza zbranih bioloških meritev zahteva od eksperta znanje o mnogo genih. Cilj take analize je razbrati, kateri globalni celični proces se odvija v danem eksperimentu. Takšna analiza je težavna zaradi količine in raznolikosti genov. Na srečo je veliko ključnega znanja na voljo v obliki javno dostopnih ontologij genov, kar omogoča tudi algoritmične pristope k reševanju problema. Primer take ontologije je Gene Ontology (GO) [12], ki predstavlja besednjak molekularne biologije in vsebuje koncepte in relacije med njimi za različne molekularne funkcije, celične komponente in biološke procese. Slika 1.4 prikazuje del ontologije GO.

Omenjeni sistem SEGS s pomočjo besednjaka iz ontologije GO in znanih povezav med geni in njihovimi funkcijami generira opise diferenčno izraženih genov. Opisi diferenčno izraženih genov so predstavljeni s konjunkcijami izrazov iz vhodne ontologije, poleg tega pa lahko tak vpis vsebuje tudi binarno relacijo interakcije med dvema skupinama genov. Predstavitev takšnih opisov (bioloških hipotez) je več-relacijska in Boolova algebra ni več zadostna.

Ontologije niso tehnologija, ki bi bila vezana le na biologijo. Dandanes se v obliki prosto dostopnih ontologij konceptualizira precejšnja množica domen (znaten faktor, ki vpliva na to, je tudi rastoča iniciativa Linked Data[1]).

Uporaba ontologij ima lahko več prednosti. Prva prednost je natančneje

[1] `http://linkeddata.org/`



Slika 1.4: Del ontologije Gene Ontology.

opisana v tem delu. Gre za formalno zapisano, dogovorjeno znanje o neki domeni, ki ga lahko neposredno vključimo v proces odkrivanja znanja v podatkih. Drugič, podatki, ki so opisani s koncepti iz ontologije, imajo natančno predpisan pomen. To se lahko izkoristi za samodejno in ponovljivo predprocesiranje. Zamislimo si lahko nabor podatkov, kjer je en izmed vnosov čas. Če s pomočjo koncepta iz ontologije vemo, da ta podatek predstavlja čas, ga lahko program brez pomoči analitika na nek način predprocesira (npr. izvede diskretizacijo). Navedimo še en primer: na voljo imamo tabelo strank, za katere hranimo tudi njihovo lokacijo. Poleg tega imamo na voljo še ontologijo lokacij. Pravila, ki jih odkrije učni sistem s pomočjo te ontologije, so lahko bolj splošna glede na kraj bivanja strank, saj lahko npr. opisujejo stranke iz Nemčije, ker v ontologiji obstaja koncept Nemčija. Običajni sistemi, ki iščejo pravila, bi v pravila vključili le lokacije, ki so prisotne v vhodnih podatkih. Taka pravila so seveda bolj specifična, saj lahko govorijo npr. le o strankah iz konkretnih nemških mest. V duhu semantičnega spleta pa lahko, preko ontologije, dane podatke povežemo z neko tretjo podatkovno bazo, ki ima z istim ontološkim konceptom označene podatke.

Zadnja, ključna motivacija za razvoj tega sistema pa je porazdeljeno izvajanje algoritmov strojnega učenja in podatkovnega rudarjenja - to pomeni, da za algoritme ni nujno, da tečejo na lokalnem računalniku, ampak lahko tečejo neodvisno na več oddaljenih računalnikih. To paradigmo podpira vse več sistemov za podatkovno rudarjenje (npr. Weka4WS, Orange4WS, Padme [13] ter splošno-namenski sistem Taverna). Odločili smo se, da naše okolje vključimo v Orange4WS [7], sistem za strojno učenje, ki podpira porazdeljeno izvajanje algoritmov.

1.3 Pregled obstoječih tehnologij

Ideja o uporabi hierarhij v predznanju za posploševanje izrazov v pravilih ni nova. To je predlagal Michalski že leta 1983 [14], kjer je opisal metodologijo, v katero lahko v obliki predznanja zakodiramo različne informacije o deskriptorjih (predikat, spremenljivka ali funkcija). Za vsak deskriptor lahko določimo:

- domeno in tip,
- katere operatorje lahko uporabimo nad njim,
- razmerja in omejitve, ki veljajo med danim deskriptorjem in drugimi,
- za numerične deskriptorje: povprečje, varianco, distribucijo vrednosti,

- razred deskriptorjev, ki vsebuje dani deskriptor - starš vozlišča, ki predstavlja gledani deskriptor, v generalizacijski hieararhiji (*ang. generalization hierarchy*), itd.

Znane so tudi novejše raziskave, ki taksonomije ali ontologije izkoriščajo v obliki predznanja o dani domeni [15, 16, 6].

V [16] opisujejo uporabo taksonomij na paleontoloških podatkih, kjer listi v taksonomiji ustrezajo atributom vhodnih podatkov, vozliščem višje v taksonomiji pa določimo vrednost kot agregirano vrednost svojih naslednikov preko operatorja `ALI`. S pomočjo takšne taksonomije so lahko tekom učenja modela generirali pravila s členi, ki predstavljajo združeno vrednost večih prvotnih atributov. Izkaže se, da lahko nad določenimi vrstami podatkov to privede do točnejšega klasifikacijskega modela.

V [15] je opisan sistem KBRL (Knowledge Based Rule Learner), ki omogoča zapis predznanja v obliki formalizma dedovalnih mrež (*ang. inheritance networks*) in to znanje uporabi za iskanje pravil, ki zadoščajo uporabnikovim kriterijem. Prostor pravil preiskuje v smeri od bolj splošnih proti bolj specifičnim, kot to narekuje vhodna dedovalna mreža.

V [6] je opisan že v prejšnjem poglavju omenjeni sistem SEGS, ki uporablja ontologijo o molekularni biologiji kot besednjak za opise diferenčno izraženih genov.

Glavne razlike opisanih sistemov od sistema, opisanega v tem delu, vključujejo:

- lokalno izvajanje [15, 16],
- nestandarden formalizem [16, 6],
- omejenost na eno domeno [16, 6],
- obliko odkritih vzorcev [16].

Poglavje 2

Uporabljene metode in orodja

V tem poglavju opišemo vsa orodja in knjižnice, ki smo jih uporabili pri razvoja sistema g-SEGS. Različna orodja so nam koristila predvsem pri razvoju uporabniškega vmesnika in pri definiciji spletnih servisov; uporabili pa smo tudi prosto dostopne knjižnice za lažjo implementacijo delov, ki skrbijo za komunikacijo med strežnikom in odjemalcem preko jezika SOAP in za manipulacijo ontologij.

2.1 Uporabljene tehnologije pri razvoju spletnih servisov

Razvoj spletnega servisa lahko razdelimo v dve fazi. V prvi fazi načrtujemo in definiramo spletni servis v obliki datoteke WSDL. V drugi fazi tudi dejansko napišemo program, ki zna komunicirati z odjemalci in opravlja zamišljene naloge; v tem smislu tudi predstavljamo uporabljena orodja. Prvo podpoglavje opisuje orodja, ki so bila uporabljena pri definiciji spletnih servisov, drugo podpoglavje pa tehnologijo, uporabljeno pri sami implementaciji.

2.1.1 Definicija spletnih servisov

Vsak spletni servis, ki ga lahko uvozimo v različna okolja za delo s spletnimi servisi, ima svoj opis zapisan v obliki datoteke WSDL (*ang. Web Services Definition Language*). Datoteka WSDL je v svoji osnovi zgolj datoteka XML z naborom elementov, ki omogočajo opis spletnega servisa. Vsaka datoteka WSDL določa lokacijo spletnega servisa in operacije, ki jih le-ta nudi. Za vsako operacijo lahko povemo njene zahtevane vhode in tipe podatkov na vhodu.

Poleg operacij lahko za vsak spletni servis definiramo tudi tipe sporočil in komunikacijski protokol.

Navedimo nekaj pomembnejših elementov, ki jih definira shema WSDL XML:

- `binding` - vezava (*ang. binding*), določi protokol za prenos sporočil,
- `service` - definira ime, lokacijo, uporabljeno vezavo in dokumentacijo servisa,
- `operation` - definira novo operacijo,
- `message` - definicija sporočila, ki ga sprejme ali odpošlje spletni servis preko določene operacije,
- `part` - definira del sporočila in njegov podatkovni tip.

Slika 2.1 prikazuje enostaven primer definicije spletnega servisa z datoteko WSDL. Primer prikazuje opis enostavnega spletnega servisa "Hello, world!", ki definira operacijo `greet`, ki sprejme en parameter tipa `string` in vrne sporočilo tipa `string`. Vidimo, da je datoteka WSDL že za zelo enostaven servis lahko precej kompleksna.

Za lažji razvoj in boljši nadzor smo v tej diplomski nalogi uporabili Web Tools Platform[1] (WTP) za razvojno okolje Eclipse. WTP razširi Eclipse tako, da vanj doda različna orodja za razvoj in testiranje spletnih in aplikacij Java Enterprise Edition. Za nas je bila koristna predvsem komponenta WSDL Editor za grafično načrtovanje spletnih servisov. WSDL Editor nudi intuitiven vmesnik za definicijo spletnega servisa v jeziku WSDL. Preko grafičnega vmesnika dostopamo do vseh elementov sheme WSDL; celotno sliko pa nam sproti prikazuje v obliki diagrama, ki spominja na diagrame jezika UML (Unified Modeling Language).

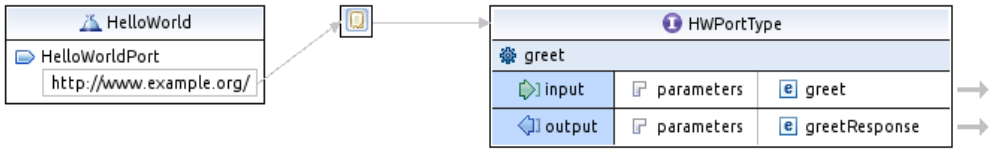
Slika 2.2 prikazuje grafični pogled na WSDL s slike 2.1. Levi gradnik predstavlja definicijo in lokacijo servisa, srednji gradnik predstavlja vezavo, ki definira protokol SOAP, desni gradnik pa prikazuje operacijo in tip vhodnega in izhodnega sporočila (`greet` in `greetResponse`). S klikom na puščico posameznega tipa sporočila se nam odpre nov pogled, kjer vidimo podrobnosti o tem sporočilu. Slika 2.3 prikazuje prikaz sporočila `greetResponse` in del sporočila - element tipa `string` z imenom `message`.

[1] `http://www.eclipse.org/webtools/`

Primer datoteke WSDL

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="demo"
    targetNamespace="http://www.example.org/demo/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://www.example.org/demo/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">
    <wsdl:types>
    <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://www.example.org/demo/">
        <xsd:element name="greet">
                <xsd:complexType>
                        <xsd:sequence>
                                <xsd:element name="name" type="xsd:string"></xsd:element>
                        </xsd:sequence>
                </xsd:complexType>
        </xsd:element>
        <xsd:element name="greetResponse">
                <xsd:complexType>
                        <xsd:sequence>
                                <xsd:element name="message" type="xsd:string"></xsd:element>
                        </xsd:sequence>
                </xsd:complexType>
        </xsd:element></xsd:schema></wsdl:types>
    <wsdl:message name="greetRequest">
            <wsdl:part name="parameters" element="tns:greet"></wsdl:part>
    </wsdl:message>
    <wsdl:message name="greetResponse">
            <wsdl:part name="parameters" element="tns:greetResponse"></wsdl:part>
    </wsdl:message>
    <wsdl:portType name="HWPortType">
            <wsdl:operation name="greet">
                    <wsdl:input message="tns:greetRequest"></wsdl:input>
                    <wsdl:output message="tns:greetResponse"></wsdl:output>
            </wsdl:operation>
    </wsdl:portType>
    <wsdl:binding name="HWBinding" type="tns:HWPortType">
            <soap:binding style="document"
                    transport="http://schemas.xmlsoap.org/soap/http" />
            <wsdl:operation name="greet">
                    <soap:operation
                            soapAction="http://www.example.org/demo/greet" />
                    <wsdl:input>
                            <soap:body use="literal" />
                    </wsdl:input>
                    <wsdl:output>
                            <soap:body use="literal" />
                    </wsdl:output>
            </wsdl:operation>
    </wsdl:binding>
    <wsdl:service name="HelloWorld">
                <wsdl:port name="HelloWorldPort" binding="tns:HWBinding"">
                        <soap:address location="http://www.example.org/" />
                </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

Slika 2.1: Primer datoteke WSDL.



Slika 2.2: Grafični pogled na WSDL s slike 2.1.

Slika 2.3: Grafični prikaz elementov enega tipa sporočil.

Vidimo, da lahko s tem orodjem na enostaven način gradimo tudi zelo kompleksne spletne servise s kompleksnejšimi podatkovnimi tipi in večjim številom operacij. Na tem mestu še omenimo občasne probleme tega orodja na Eclipse Helios na 64-bitnem sistemu Linux Kubuntu, saj v primeru, da imamo v Eclipse odprt tudi vsaj en projekt drugih tipov (npr. Javanski projekt), uporaba urejevalnika WSDL lahko povzroči zrušitev celotne aplikacije Eclipse.

2.1.2 Implementacija spletnih servisov

Potem, ko definiramo spletni servis z datoteko WSDL, se lahko lotimo implementacije samega programa (strežnika in odjemalca). Strežniški del bo sprejemal zahtevke odjemalcev, pošiljal nazaj rezultate, odjemalčev del pa bo generiral nove zahtevke in sprejemal rezultate; vsi ti zahtevki pa se morajo pošiljati v obliki sporočil SOAP (*ang. Simple Object Access Protocol*) prek protokola HTTP (*ang. Hypertext Transfer Protocol*).

SOAP

Protokol SOAP določa način za prenašanje strukturiranih podatkov v implementacijah spletnih servisov preko računalniških omrežij. Sporočila SOAP so zapisana v jeziku XML, po omrežju pa se prenašajo preko protokolov na aplikacijskem sloju (npr. Remote Procedure Call (RPC) in Hypertext Transfer Protocol (HTTP)). SOAP je pogosto uporabljan protokol za zapis sporočil v t.i. skladu protokolov za spletne servise (*ang. Web services protocol stack*).

Sklad protokolov za spletne servise je sestavljen iz štirih slojev:

- protokoli za prenašanje sporočil (*ang. Service Transport Protocol*), ki so odgovorni za prenašanje sporočil med aplikacijami. Primeri: že omenjeni HTTP, SMTP (*ang. Simple Mail Transfer Protocol*), FTP (*ang. File Transfer Protocol*),
- protokoli, ki definirajo obliko sporočil (*ang. XML Messaging Protocol*). Med te protokole spada SOAP,
- protokoli za opis spletnih servisov (*ang. Service Description Protocol*). Med te protokole spada WSDL,
- protokoli za odkrivanje spletnih servisov (*ang. Service Discovery Protocol*), nudijo centralni register spletnih servisov, objavljenih na spletu (npr. Universal Description Discovery and Integration (UDDI)).

Slika 2.4 prikazuje ogrodje sporočila SOAP. Kot prikazuje slika, so sporočila SOAP zgrajena iz treh osnovnih delov:

- iz ovojnice (*ang. envelope*), ki vsebuje vse ostale elemente in pove, da gre za sporočilo SOAP XML,
- iz (neobvezne) glave (*ang. header*), ki vsebuje specifične informacije za dano aplikacijo; tipično so to kontrolne informacije (*kako* obdelati vsebino sporočila),
- iz telesa (*ang. body*), ki vsebuje ključno vsebino sporočila.

Ogrodje sporočila SOAP

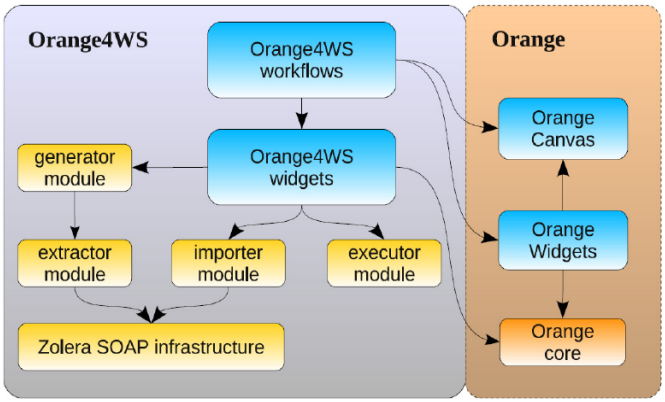
```
<?xml version="1.0"?>
  <soap:Envelope
    xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
    soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
    <soap:Header>
    ...
    </soap:Header>
    <soap:Body>
    ...
  </soap:Body>
</soap:Envelope>
```

Slika 2.4: Ogrodje sporočila SOAP.

V tej diplomski nalogi smo uporabili SOAP kot protokol za zapis sporočil za komunikacijo s spletnimi servisi preko protokola HTTP. Uporabo protokola SOAP v našem okolju omogoča knjižnica ZSI (Zolera Soap Infrastructure) iz projekta Python for Web Services[2].

ZSI in Orange4WS

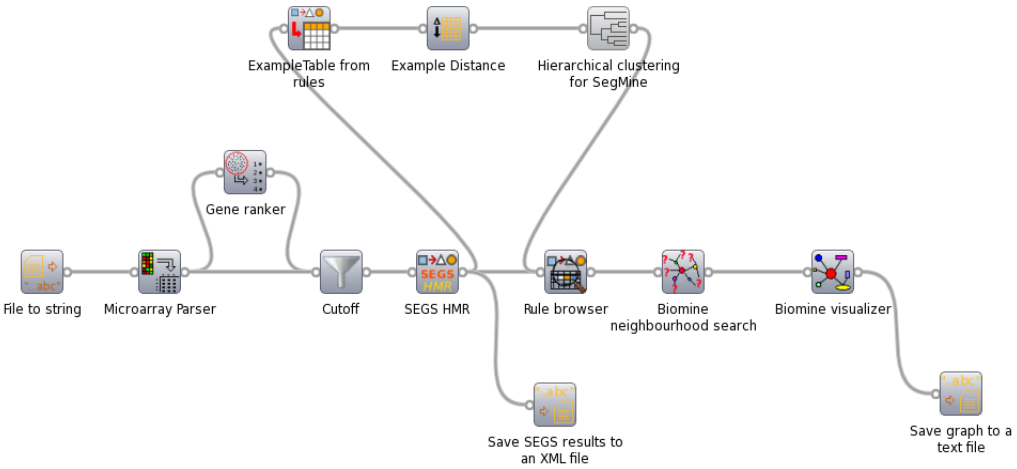
Ključno orodje oziroma knjižnica za implementacijo naši spletnih servisov je Orange4WS in (posredno) Zolera SOAP Infrastructure (ZSI) iz projekta Python for Web Services. Orange4WS je razširitev okolja za strojno učenje Orange, ki omogoča vizualno programiranje s povezovanjem različnih gradnikov (*ang. widgets*) v delotoke. Slika 2.5 prikazuje shemo arhitekture sistema Orange4WS.



Slika 2.5: Arhitektura sistema Orange4WS in njegova integracija v Orange.

S pomočjo knjižnice ZSI omogoča Orange4WS v sistemu Orange uvoz spletnih servisov, ki se nato lahko uporabljajo v povezavi z ostalimi gradniki. Uporabnik potrebuje le naslov spletnega servisa, aplikacija pa nato iz njegove datoteke WSDL, ki jo aplikaciji zna poslati strežniški del spletnega servisa, razbere podrobnosti, ki so potrebne za komunikacijo s tem servisom. Vsaka operacija spletnega servisa se preslika v en gradnik, vhodi in izhodi pa se preslikajo v vhode in izhode tega gradnika. Tak gradnik je nato pripravljen na uporabo z ostalimi (lokalnimi) gradniki sistema Orange, uporabniku pa je oddaljeno izvajanje skrito in lahko s temi gradniki ravna enako kot z lokalnimi, saj je vsa komunikacija s spletnimi servisi, ki se dogaja v ozadju, uporabniku skrita.

[2]http://pywebsvcs.sourceforge.net/



Slika 2.6: Primer delotoka v sistemu Orange4WS.

Slika 2.6 prikazuje primer delotoka v sistemu Orange4WS. Delotok vsebuje gradnike, ki se izvajajo lokalno (npr. `Microarray Parser`) in gradnike, ki se izvajajo na oddaljenem računalniku (npr. sistem za odkrivanje podskupin `SEGS HMR`).

Poleg nadgradnje sistema Orange nudi Orange4WS tudi nabor razredov za implementacijo spletnih servisov v jeziku Python. Ta del je osnovan na knjižnici ZSI, ki implementira protokol SOAP 1.1. ZSI samodejno pretvarja med sporočili SOAP in osnovnimi podatkovnimi tipi v Pythonu. Programer na ta način gradi in prebira sporočila SOAP v sintaksi jezika Python. Knjižnica ZSI nudi tudi vmesnik za branje datotek WSDL ter orodje `wsdl2py` za generiranje ogrodja (*ang. skeleton*) strežniškega in odjemalčevega dela programa na podlagi datoteke WSDL.

V ukazni lupini program `wsdl2py` poženemo kot:

wsdl2py

```
1 $ wsdl2py <pot do datoteke WSDL>
```

Če skripto `wsdl2py` poženemo nad primerom 2.1, dobimo tri Python-module:

- `HelloService_client.py`, ki vsebuje definicije razredov, ki jih potrebuje odjemalec za komunikacijo s spletnim servisom (glej sliko 2.7),
- `HelloService_server.py`, ki vsebuje ogrodje za strežniški del aplikacije, ki jo mora programer dopolniti tako, da spletni servis opravlja predpisano nalogo (glej sliko 2.8) in

- `HelloService_types.py`, ki definira morebitne kompleksnejše tipe, definirane v datoteki WSDL. V našem primeru "Hello, World!" je ta datoteka prazna, saj uporabljamo zgolj primitivne podatkovne tipe.

Orange4WS uporabniku ta del še bolj olajša in generiranje ustreznih razredov (oz. štrcljev (*ang. stubs*)) skrije v klic ene funkcije. Recimo, da je naš primer datoteke WSDL iz slike 2.1 shranjen v datoteki `hello_world.wsdl`, potem do ustreznih štrcljev dostopamo tako kot prikazuje naslednji odsek kode:

generiranje ogrodja v Orange4WS

```
1 import webServices
2 stubs = webServices.stubImporter.importZSIstubsFromURI('hello_world.wsdl')
3 client, server, types, WSDL = stubs.client, stubs.server, stubs.types, stubs.WSDL
```

Objekti `client`, `server`, `types` in `WSDL` so reference na module, ki jih v ozadju generira `wsdl2py`. Uporabnik nato postopa podobno, kot če bi module generiral s programom `wsdl2py`.

2.2 Uporabljene tehnologije pri razvoju grafičnega uporabniškega vmesnika

Za enostavnejšo uporabo naših spletnih servisov smo se odločili, da namesto samodejno generiranih vmesnikov razvijemo svoj grafični vmesnik, saj je tudi ta način dobro podprt v Orange oziroma v Orange4WS.

Orange Canvas, kot se imenuje del sistema Orange, ki implementira grafični vmesnik, ponuja osnovne gradnike za razvoj lastnih uporabniških vmesnikov za morebitne nove gradnike, ki jih želimo vgraditi. Celoten grafični vmesnik sistema Orange je zgrajen na osnovi knjižnice PyQt[3], ki je vezava knjižnice Qt[4] za C++ za jezik Python.

Orange Canvas definira osnovni razred `OWBaseWidget`, iz katerega izpeljemo razred za naš nov gradnik. Da bi se izognili ročnemu postavljanju elementov uporabniškega vmesnika, smo uporabili orodje Qt Designer, ki omogoča vizualno gradnjo uporabniških vmesnikov z uporabo knjižnice Qt.

[3] http://www.riverbankcomputing.co.uk/software/pyqt/download
[4] http://qt.nokia.com/products/

Odjemalčev del servisa Hello_World

```
import urlparse, types
from ZSI.TCcompound import ComplexType, Struct
from ZSI import client
from ZSI.schema import GED, GTD
import ZSI

# Locator
class Hello_ServiceLocator:
    Hello_Port_address = "http://www.examples.com/SayHello/"
    def getHello_PortAddress(self):
        return Hello_ServiceLocator.Hello_Port_address
    def getHello_Port(self, url=None, **kw):
        return Hello_BindingSOAP(url or Hello_ServiceLocator.Hello_Port_address, **kw)

# Methods
class Hello_BindingSOAP:
    def __init__(self, url, **kw):
        kw.setdefault("readerclass", None)
        kw.setdefault("writerclass", None)
        # no resource properties
        self.binding = client.Binding(url=url, **kw)
        # no ws-addressing

    # op: sayHello
    def sayHello(self, request, **kw):
        if isinstance(request, SayHelloRequest) is False:
            raise TypeError, "%s incorrect request type" % (request.__class__)
        # no input wsaction
        self.binding.Send(None, None, request, soapaction="sayHello", \
  encodingStyle="http://schemas.xmlsoap.org/soap/encoding/", **kw)
        # no output wsaction
        typecode = Struct(pname=None, ofwhat=SayHelloResponse.typecode.ofwhat, \
  pyclass=SayHelloResponse.typecode.pyclass)
        response = self.binding.Receive(typecode)
        return response

class SayHelloRequest:
    def __init__(self, **kw):
        """Keyword parameters:
        firstName -- part firstName
        """
        self._firstName =  kw.get("firstName")
SayHelloRequest.typecode = Struct(pname=("urn:examples:helloservice","sayHello"), \
  ofwhat=[ZSI.TC.String(pname="firstName", aname="_firstName", typed=False, \
  encoded=None, minOccurs=1, maxOccurs=1, nillable=True)], pyclass=SayHelloRequest, \
  encoded="urn:examples:helloservice")

class SayHelloResponse:
    def __init__(self, **kw):
        """Keyword parameters:
        greeting -- part greeting
        """
        self._greeting =  kw.get("greeting")
SayHelloResponse.typecode = Struct(pname=("urn:examples:helloservice","sayHelloResponse"), \
  ofwhat=[ZSI.TC.String(pname="greeting", aname="_greeting", typed=False, encoded=None, \
  minOccurs=1, maxOccurs=1, nillable=True)], pyclass=SayHelloResponse, \
  encoded="urn:examples:helloservice")
```

Slika 2.7: Odjemalčev del servisa “Hello, world!”.

Strežniški del servisa Hello_World

```
from ZSI.schema import GED, GTD
from ZSI.TCcompound import ComplexType, Struct
from HelloService_types import *
from ZSI.ServiceContainer import ServiceSOAPBinding

# Messages
class SayHelloRequest:
    def __init__(self, **kw):
        """Keyword parameters:
        firstName -- part firstName
        """
        self._firstName =  kw.get("firstName")
SayHelloRequest.typecode = Struct(pname=("urn:examples:helloservice","sayHello"), \
  ofwhat=[ZSI.TC.String(pname="firstName", aname="_firstName", typed=False, encoded=None,\
  minOccurs=1, maxOccurs=1, nillable=True)], pyclass=SayHelloRequest, \
  encoded="urn:examples:helloservice")

class SayHelloResponse:
    def __init__(self, **kw):
        """Keyword parameters:
        greeting -- part greeting
        """
        self._greeting =  kw.get("greeting")
SayHelloResponse.typecode = Struct(pname=("urn:examples:helloservice","sayHelloResponse"), \
  ofwhat=[ZSI.TC.String(pname="greeting", aname="_greeting", typed=False, encoded=None, \
  minOccurs=1, maxOccurs=1, nillable=True)], pyclass=SayHelloResponse, \
  encoded="urn:examples:helloservice")


# Service Skeletons
class Hello_Service(ServiceSOAPBinding):
    soapAction = {}
    root = {}

    def __init__(self, post='/SayHello/', **kw):
        ServiceSOAPBinding.__init__(self, post)

    def soap_sayHello(self, ps, **kw):
        request = ps.Parse(SayHelloRequest.typecode)
        return request,SayHelloResponse()

    soapAction['sayHello'] = 'soap_sayHello'
    root[(SayHelloRequest.typecode.nspname,SayHelloRequest.typecode.pname)] = 'soap_sayHello'
```

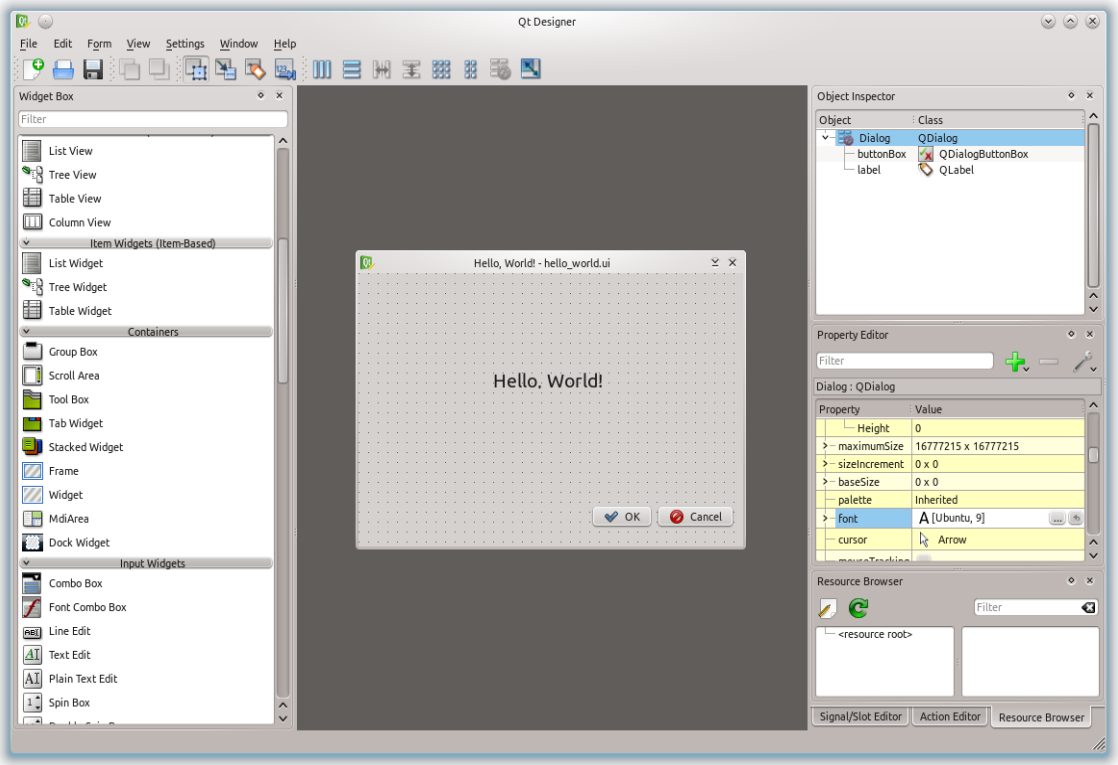
Slika 2.8: Strežniški del servisa “Hello, world!”.

Slika 2.9 prikazuje zaslonsko masko orodja Qt Designer. Na levem delu slike vidimo različne gradnike, ki jih zlagamo skupaj na srednjem delu slike. Na desnem delu slike pa vidimo seznam, s pomočjo katerega pregledujemo in nastavljamo željene lastnosti posameznih elementov (npr. ime objekta, s katerim nato v samem programu referenciramo željeni element). Ko smo z vmesnikom zadovoljni, ga shranimo v posebno vmesno datoteko s končnico `.ui`. To datoteko nato lahko prevedemo s priloženim programom `pyuic4` na naslednji način:

pyuic4

```
1 pyuic4 -o hello_world.py hello_world.ui
```

Dobljeni modul lahko nato enostavno uporabljamo v lastnih programih, napisanih v Pythonu.



Slika 2.9: Zaslonska maska orodja Qt Designer.

2.3 Ostale uporabljene tehnologije

2.3.1 OWL

Kot formalizem za zapis ontologij v našem sistemu smo si izbrali Web Ontology Language[5] (OWL). Jezik OWL je bil ustvarjen za lažje objavljanje in deljenje ontologij na svetovnem spletu. Dokumenti OWL so dokumenti XML napisani po shemi OWL. OWL razširja shemo RDF (Resource Description Framework; gre za enostaven podatkovni model, kjer vse podatke hranimo v obliki trojčkov `subject-predicate-object`) z novimi elementi. V OWL lahko, za razliko od RDF, povemo tudi kaj o lastnostih same relacije: npr. ali je tranzitivna, simetrična, funkcijska in ali je inverzna kateri drugi relaciji.

V praksi se uporabnik odloči za enega izmed treh podmnožic jezika OWL, ki se razlikujejo po svojih izraznih močeh:

- *OWL Lite* je najosnovnejši in najenostavnejši nabor funkcionalnosti jezika OWL,
- *OWL DL* ponuja izrazno moč in zaželjene računske lastnosti (npr. odločljivost) t.i. deskriptivne logike (*ang. Descriptive Logic*, s kratico DL) in
- *OWL Full*, ki ponuja celoten nabor funkcionalnosti, vendar krši omejitve deskriptivne logike in zato ni odločljiv jezik.

Zaradi preglednosti primer cele datoteke OWL prilagamo v dodatku A. Primer prikazuje predstavitev ontologije s slike 1.3 v jeziku OWL/XML. Vsi nadaljni odseki kode so vzeti iz tega primera.

Opis ontologije se začne z elementom `ontology` in z definicijo imenskih prostorov (*ang. namespaces*), ki jih lahko uporabljamo znotraj dokumenta:

```
1 <Ontology xmlns="http://www.w3.org/2002/07/owl#"
2      xml:base="http://www.semanticweb.org/ontologies/2011/4/knjige.owl"
3      xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
4      xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
5      xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
6      xmlns:xml="http://www.w3.org/XML/1998/namespace"
7      ontologyIRI="http://www.semanticweb.org/ontologies/2011/4/knjige.owl">
8 ...
9 </Ontology>
```

[5] http://www.w3.org/TR/owl-ref/

Na primeru vidimo, da imenski prostor oz. shemo OWL predpišemo z identifikatorjem (*ang. Uniform Resource Identifier oz. Internationalized Resource Identifier*, s kratico URI oz. IRI) `http://www.w3.org/2002/07/owl#`. Poleg OWL pa na tem mestu definiramo še identifikator ontologije, ki jo opisujemo[6] in identifikator morebitnih drugih shem (npr. RDFS in RDF).

Z elementom `prefix` definiramo predpono, ki jo uporabljamo za določen imenski prostor in s tem skrajšamo zapis identifikatorjev in izboljšamo preglednost dokumenta. Navedimo primer:

```
1 <Prefix name="owl" IRI="http://www.w3.org/2002/07/owl#"/>
```

Na ta način lahko v nadaljevanju dokumenta uporabimo zapis, kot je uporabljen v naslednji definiciji razreda z elementom `Class`:

```
1 <Class abbreviatedIRI="owl:Thing"/>
```

Zapis `owl:Thing` interpretiramo kot `http://www.w3.org/2002/07/owl#Thing`. Na ta način definirane razrede lahko povežemo v hierarhijo z relacijo `SubClassOf` (v literaturi se pogosto uporablja tudi ime `is_a`, a tega imena shema OWL ne podpira), ki pove, da je določen razred podrazred drugega razreda - interpretacija je enaka kot v objektnem programiranju:

$$A \text{ SubClassOf } B \Leftrightarrow \forall x \in A : x \in B$$

Naslednji odsek dokumenta OWL pove, da vsi objekti, ki pripadajo razredu `Knjiga`, pripadajo tudi razredu `Thing`:

```
1 <SubClassOf>
2     <Class IRI="#Knjiga"/>
3     <Class abbreviatedIRI="owl:Thing"/>
4 </SubClassOf>
```

Jezik omogoča tudi definicijo novih relacij med razredi, ki se v jeziku OWL imenujejo lastnosti (*ang. object properties*). To dosežemo z uporabo elementa `ObjectProperty`. Definicijsko območje (*ang. domain*) in zalogo vrednosti (*ang. range*) pa definiramo z elementoma `ObjectPropertyDomain` in `ObjectPropertyRange`:

[6] http://www.semanticweb.org/ontologies/2011/4/knjige.owl

```
<Declaration>
    <ObjectProperty IRI="#partOf"/>
</Declaration>
<ObjectPropertyDomain>
    <ObjectProperty IRI="#partOf"/>
    <Class abbreviatedIRI="owl:Thing"/>
</ObjectPropertyDomain>
<ObjectPropertyRange>
    <ObjectProperty IRI="#partOf"/>
    <Class abbreviatedIRI="owl:Thing"/>
</ObjectPropertyRange>
```

S pomočjo elementa `AnnotationAssertion` lahko razredom in lastnostim dodamo tudi različne komentarje ali oznake, namenjene ljudem. Naslednji odsek prikazuje definicijo oznake za razred `Knjiga`:

```
<AnnotationAssertion>
    <AnnotationProperty abbreviatedIRI="rdfs:label"/>
    <IRI>#Knjiga</IRI>
    <Literal datatypeIRI="&rdf;PlainLiteral">Knjiga</Literal>
</AnnotationAssertion>
```

2.3.2 Jena

Za manipulacijo ontologij v jeziku OWL smo uporabili odprto kodno knjižnico Jena[7] za programski jezik Java, namenjeno splošni uporabi v aplikacijah za semantični splet. Omogoča nabor funkcionalnosti za delo s podatki RDF in OWL. Vsebuje:

- vmesnik API za RDF in OWL,
- branje in pisanje datotek RDF v zapisih RDF/XML, N3 in N-Triples,
- implementacijo jezika SPARQL (SPARQL Protocol and RDF Query Language) za poizvedbe nad podatki RDF.

Slika 2.10 prikazuje ilustrativen primer uporabe knjižnice Jena. Priloženi program na standardni izhod izpiše hierarhijo razredov dane ontologije OWL.

V vrsticah 11 in 12 preberemo ontologijo z diska v spomin s pomočjo razreda `OntModel`, ki predstavlja abstrakcijo našega modela. V vrstici 15 nato prikličemo referenco na korenski razred `Thing`, ki ga podamo metodi

[7] http://jena.sourceforge.net/

`print(...)`, ki rekurzivno izpiše vsa imena podrazredov razreda, ki ji ga podamo kot parameter. Metoda `getLabel(...)` razreda `OntClass` (vrstica 29) vrne oznako danega razreda.

V tej diplomski nalogi smo Jeno uporabili predvsem za branje strukture ontologij, da smo le-to lahko prepisali v drug format in ustvarili pomožne strukture.

Jena

```
1  import java.util.Iterator;
2
3  import com.hp.hpl.jena.ontology.*;
4  import com.hp.hpl.jena.rdf.model.ModelFactory;
5  import com.hp.hpl.jena.vocabulary.OWL;
6
7  public class JenaExample
8  {
9      public static void main(String[] args)
10     {
11         OntModel model = ModelFactory.createOntologyModel(OntModelSpec.OWL_MEM, null);
12         model.read("file:/home/anzev/programiranje/diploma/diploma_tex/knjige.owl");
13
14         // Retrieve the root class
15         OntClass cl = model.getOntResource(OWL.Thing).asClass();
16         print(cl, 0);
17     }
18
19     private static void print(OntClass base, int d)
20     {
21         Iterator<OntClass> it = base.listSubClasses(true);
22
23         while (it.hasNext()) {
24             OntClass cl = it.next();
25
26             for (int i = 0; i < d; i++)
27                 System.out.print('\t');
28
29             System.out.println(cl.getLabel(null));
30
31             // Recursively print the subclasses of this class
32             print(cl, d+1);
33         }
34     }
35 }
```

Slika 2.10: Primer uporabe knjižnice Jena.

2.3.3 Programski jeziki in urejevalniki

Za izdelavo diplomske naloge smo uporabili več programskih jezikov in razvojnih okolij. Kot smo omenili v prvem poglavju tega dela, smo večino sistema in vse skripte, napisane za poganjanje in pripravo eksperimentov (glej poglavje 5), napisali v jeziku Python (verzija 2.6). Preostali del je napisan v jezikih Java (verzija 1.6.0_24) in C (za prevajanje smo uporabili prevajalnik gcc (GNU C Compiler) verzije 4.4.5). Za urejanje kode Python smo uporabili razvojno okolje Wingware[8], za urejanje Javanske kode razvojno okolje Eclipse, za vse ostalo urejanje (C in Prolog) pa splošno-namenski urejevalnik Kate[9] (KDE Advanced Text Editor).

[8] http://www.wingware.com/
[9] http://kate-editor.org/

Poglavje 3

Okolje za uporabo ontologij v podatkovnem rudarjenju

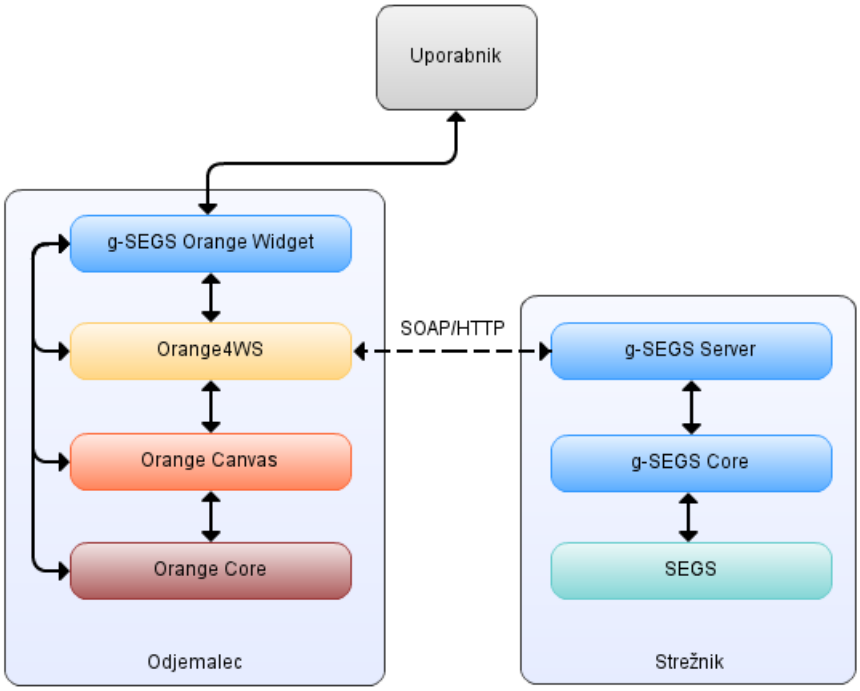
V tem poglavju opišemo okolje za uporabo ontologij v podatkovnem rudarjenju g-SEGS. V prvem podpoglavju najprej opišemo arhitekturo sistema, v drugem podpoglavju opišemo spletni servis, ki izvaja osrednje operacije našega sistema, v tretjem podpoglavju opišemo razvoj strežniškega dela, v četrtem podpoglavju sledi opis uporabniškega dela aplikacije, na koncu pa prikažemo še tipično uporabo razvitega sistema.

3.1 Arhitektura sistema

Shemo arhitekture razvitega sistema prikazuje slika 3.1: uporabnik sistem uporablja preko grafičnega uporabniškega vmesnika, gradnika `g-SEGS` v okviru sistema Orange. S pomočjo tega gradnika in že obstoječih gradnikov okolja Orange uporabnik definira učni problem, pregleduje podatke, izbere ontologije, nastavi ustrezne parametre, itn. Ko je uporabnik zadovoljen z izdelanim delotokom, lahko izda ukaz za začetek izvajanja.

Izdelani `g-SEGS Orange Widget` zna preko funkcionalnosti Orange4WS komunicirati s spletnim servisom našega okolja, ki teče na nekem spletnem strežniku. Vsa komunikacija teče preko sporočil SOAP, ki jih zna `g-SEGS Orange Widget` iz uporabnikovih zahtev ustrezno oblikovati in jih, ko so rezultati dobljeni, tudi na ustrezen način prebrati in posredovati uporabniku. Ta komponenta sistema uporablja tudi funkcionalnosti modulov `Orange Canvas` in `Orange Core`.

Komponenta `Orange Canvas` skrbi za funkcionalnosti, povezane z grafičnim vmesnikom okolja Orange. Pod `Orange Core` pa pojmujemo jedro okolja



Slika 3.1: Arhitektura okolja g-SEGS.

Orange. Gre za različne podatkovne strukture in algoritme strojnega učenja, predprocesiranja, vizualizacije ipd., napisane v programskem jeziku C++.

Na strežniški strani `g-SEGS Server` sprejema zahtevke od odjemalcev, jih ustrezno obravnava in pošlje delu aplikacije `g-SEGS Core`, ki opravi ustrezne transformacije in posreduje podatke aplikaciji SEGS, ki nato odkrite podskupine (skupaj z ustreznimi statistikami) posreduje nazaj.

Razvoj sistema smo začeli od spodaj navzgor (*ang. bottom up*). Začeli smo z modifikacijo osnovnega sistema SEGS. Nato smo razvili modul g-SEGS, ki ustrezno ovije sistem SEGS. Za tem smo na podlagi napisanega modula za g-SEGS definirali spletni servis. Z dano definicijo smo nato lahko razvili strežniški modul in za tem še gradnik za sistem Orange, ki služi kot odjemalec za razviti spletni servis in kot grafični uporabniški vmesnik do sistema.

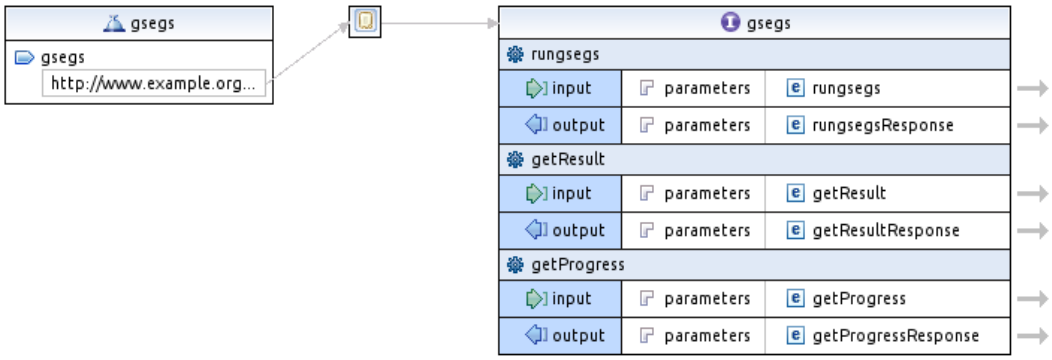
Omenjeni moduli in spletni servis so podrobneje opisani v nadaljevanju tega poglavja.

3.2 Opis spletnega servisa

V okviru razvoja sistema g-SEGS smo se lotili zasnove spletnega servisa, ki izvaja kritične operacije našega gradnika v delotokih. Prednost takšne zasnove je predvsem porazdeljeno računanje, saj so problemi, ki jih rešuje sistem g-SEGS, lahko precej časovno in prostorsko potratni in je izvajanje na lokalnem, uporabnikovem računalniku nezaželjeno. Zaradi porazdeljenega izvajanja se lahko operacije različnih gradnikov delotoka izvajajo celo vzporedno.

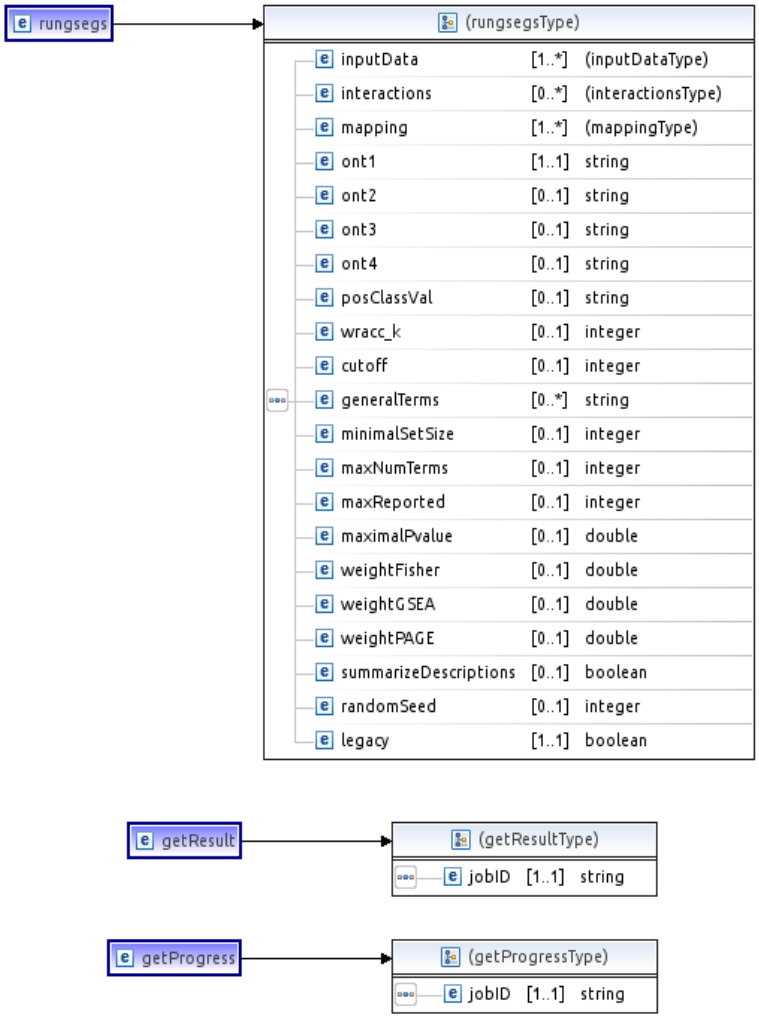
Slika 3.2 prikazuje operacije, ki jih ponuja spletni servis, skupaj z imeni vhodnih in izhodnih sporočil. Kot je razvidno s slike, ponuja spletni servis tri operacije:

- `rungsegs` - požene sistem za odkrivanje podskupin,
- `getResult` - vrne rezultat - odkrite podskupine skupaj z različnimi merami,
- `getProgress` - vrne trenutno stanje izvajanja.



Slika 3.2: Spletni servis g-SEGS.

Vhod operacije `rungsegs` je definiran s sporočilom `rungsegs`, katerega podrobnosti prikazuje zgornji del slike 3.3. Pomeni posameznih elementov tega sporočila so enaki parametrom pri uporabi sistema g-SEGS v skriptnem načinu, ki je opisan v nadaljevanju.



Slika 3.3: Vhodna sporočila spletnega servisa g-SEGS.

Slika v prvem stolpcu prikazuje imena elementov, v srednjem stolpcu prikazuje številnost posameznega elementa, v desnem stolpcu pa podatkovne tipe. Na tem mestu hitro razložimo pomene posameznih oznak številnosti:

- `[0..1]` - element ni obvezen, vsebuje pa *največ* en element danega tipa,
- `[1..1]` - element je obvezen, vsebuje pa *natanko* en element danega tipa,
- `[0..*]` - element ni obvezen, vsebuje pa lahko *poljubno* število elementov danega tipa,

- `[1..*]` - element je obvezen, vsebuje pa lahko *poljubno* število elementov danega tipa.

Za večino parametrov zadostujejo primitivni podatkovni tipi (`string`, `integer`, `double`, `boolean`), medtem ko za kompleksnejše vhode definiramo lastne anonimne podatkovne tipe. To velja za elemente:

- `inputData`, kjer smo definirali podatkovni tip - par `(id, oznaka)` tipov `integer` in `string`. To dosežemo z naslednjim delom kode v datoteki WSDL (vrstice 7 do 16):

```
1  <xsd:element name="inputData" maxOccurs="unbounded" minOccurs="1">
2          <xsd:annotation>
3                  <xsd:documentation>
4                          This parameter is a list of pairs (id, rank).
5                  </xsd:documentation>
6          </xsd:annotation>
7          <xsd:complexType>
8                  <xsd:sequence>
9                          <xsd:element name="id" type="xsd:integer"
10                                 maxOccurs="1" minOccurs="1">
11                         </xsd:element>
12                         <xsd:element name="rank_or_label"
13                                 type="xsd:string" maxOccurs="1" minOccurs="1">
14                         </xsd:element>
15                 </xsd:sequence>
16         </xsd:complexType>
17 </xsd:element>
```

- `interactions`, kjer smo definirali podatkovni tip - par `(id1, id2)` tipov `integer` in `integer`. To dosežemo z naslednjim delom kode v datoteki WSDL (vrstice 7 do 16):

```
1  <xsd:element name="interactions" maxOccurs="unbounded" minOccurs="0">
2          <xsd:annotation>
3                  <xsd:documentation>
4                          This parameter is a list of pairs (id, id).
5                  </xsd:documentation>
6          </xsd:annotation>
7          <xsd:complexType>
8                  <xsd:sequence>
9                          <xsd:element name="id1" type="xsd:integer"
10                                 maxOccurs="1" minOccurs="1">
11                         </xsd:element>
12                         <xsd:element name="id2" type="xsd:integer"
13                                 maxOccurs="1" minOccurs="1">
14                         </xsd:element>
15                 </xsd:sequence>
16         </xsd:complexType>
17 </xsd:element>
```

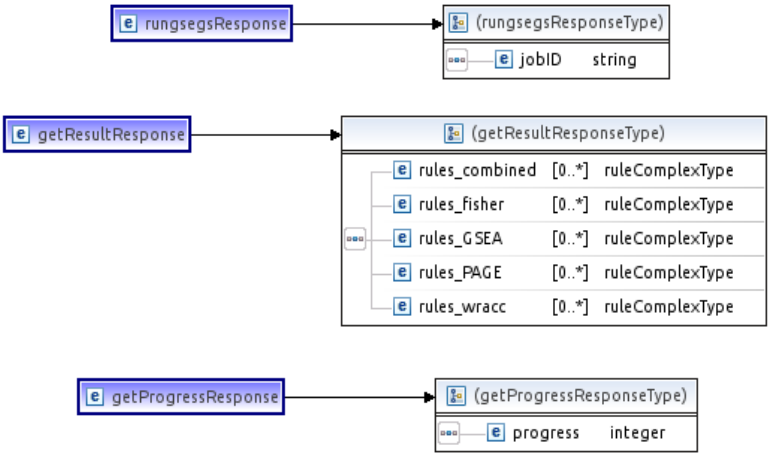
- `mapping`, kjer smo definirali podatkovni tip - par identifikatorja in (neomejenega) seznama identifikatorjev konceptov `(id, [uri1, uri2, ...])`. To dosežemo z naslednjim delom kode v datoteki WSDL (vrstice 8 do 17):

```
1  <xsd:element name="mapping" maxOccurs="unbounded" minOccurs="1">
2        <xsd:annotation>
3                  <xsd:documentation>
4                          This parameter is a list of pairs
5                          [..., (id_i, [URI1, URI2, ...]), ...].
6                  </xsd:documentation>
7          </xsd:annotation>
8          <xsd:complexType>
9                  <xsd:sequence>
10                         <xsd:element name="id" type="xsd:integer"
11                                 maxOccurs="1" minOccurs="1">
12                         </xsd:element>
13                         <xsd:element name="uri" type="xsd:string"
14                                 maxOccurs="unbounded" minOccurs="1">
15                         </xsd:element>
16                 </xsd:sequence>
17         </xsd:complexType>
18 </xsd:element>
```

Kot je razvidno iz primerov, lahko razvijalec nov tip definira z elementom `complexType`. V našem primeru smo znotraj tega elementa uporabili element `sequence`, ki pove, da je tip sestavljen iz več elementov. Znotraj elementa `sequence` nato definiramo elemente tega tipa.

Takšna tipizacija v okviru datoteke WSDL je pomembna, če želimo, da je spletni servis takoj uporaben v različnih okoljih za delo s spletnimi servisi. Sama struktura podatkov je na ta način definirana že v opisu servisa. V nasprotnem primeru, če bi namesto novega podatkovnega tipa uporabili kar tip `string`, ki bi predstavljal znakovni zapis vhodnih podatkov, bi moral uporabnik spletnega servisa poznati sintakso takega zapisa in lastnoročno poskrbeti (npr. z lastnim vmesnim gradnikom), da se podatki pravilno pretvarjajo med to obliko in obliko s primitivnimi podatkovnimi tipi. Tak scenarij ni zaželjen in se mu lahko na prej opisan način povsem izognemo.

Izhodno sporočilo operacije `rungsegs` prikazuje slika 3.4. Sporočilo ima eno polje, `jobID` tipa `string`, ki predstavlja identifikator za to opravilo (*ang. job*). Spletni servis je zasnovan tako, da je komunikacija med odjemalcem in strežnikom *nepovezavna* (*ang. connectionless*). Seveda je za izvedbo ene operacije komunikacija *povezavna* (*ang. connection-oriented*) preko protokola HTTP, vendar pa odjemalec za izvedbo enega opravila kliče več operacij. Brez identifikatorja `jobID` strežnik ne more vedeti, kateri klici operacij spadajo istemu opravilu.



Slika 3.4: Izhodna sporočila spletnega servisa g-SEGS.

Operacijo `getProgress` odjemalec kliče po tem, ko je že izstavil zahtevo za začetek računanja s klicem `rungsegs`. Operacija pričakuje sporočilo tipa `getProgress`, ki vsebuje polje z identifikatorjem `jobID` in vrne sporočilo tipa `getProgressResponse` s poljem `progress` tipa `integer`, ki odraža trenutno stanje izvajanja v procentih.

Tretjo operacijo, `getResult`, odjemalec kliče, ko ugotovi, da je računanje zaključeno. Kliče jo seveda lahko kadarkoli, vendar, če opravilo še ni končano, spletni servis vrne sporočilo z opisom napake (npr. "Opravilo s tem identifikatorjem še ni zaključeno."). Operacija sprejme sporočilo tipa `getResult` s poljem `jobID`, vrne pa sporočilo tipa `getResultResponse`.

Sporočilo `getResultResponse` vsebuje štiri polja tipa `ruleComplexType`, urejena po dani meri za oceno kvalitete podskupine (vsako polje ustreza eni meri). Zaradi preglednosti definicijo tega tipa prilagamo v dodatku B. Tip `ruleComplexType` predstavlja eno odkrito pravilo oz. podskupino. Vsako pravilo sestavlja opis, seznam vseh pokritih primerov, seznam pozitivnih pokritih primerov, ki pripadajo pozitivnemu razredu v primeru označenih primerov, oz. so med prvimi `cutoff` (glej nadaljevanje) rangiranimi primeri, ter vrednosti različnih mer za oceno te podskupine.

Izvedba enega opravila je običajno sestavljena iz naslednjega zaporedja operacij:

1. odjemalec pošlje strežniku sporočilo `rungsegs`, ki vsebuje vhodne podatke,

2. strežnik da opravilo v izvajanje in odjemalcu pošlje identifikator opravila `jobID`,

3. odjemalec pošilja sporočila `getProgress` za dani `jobID`, strežnik pa mu odgovarja z `getProgressResponse`, dokler ni `progress` enak 100,

4. odjemalec nato lahko pošlje sporočilo `getResult` za dani `jobID`, strežnik pa mu pošlje sporočilo z rezultati.

Servis bi lahko realizirali tudi tako, da bi operacija `rungsegs` takoj vrnila rezultate, vendar nam to onemogoča obveščanje uporabnika o trenutnem stanju izvajanja. Prikazovanje trenutnega stanja izvajanja je pri dolgotrajnih eksperimentih seveda zaželjeno.

3.3 Strežniški del aplikacije

Strežniški del aplikacije je tisti, ki ga ni treba (nujno) izvajati na uporabnikovem računalniku, ampak na nekem oddaljenem računalniku. Seveda lahko oba dela aplikacije v principu tečeta na istem računalniku (tako smo naše okolje v času razvoja tudi testirali), vendar bo komunikacija med odjemalcem in strežnikom še vedno tekla preko protokolov SOAP/HTTP po omrežju - to pa ni cilj uporabe spletnih servisov.

Kot prikazuje slika 3.1 (glej poglavje 3.1), lahko strežniški del razdelimo na tri dele:

- strežniški modul (`g-SEGS Server`),
- modul g-SEGS (`g-SEGS Core`),
- (dopolnjeni) modul SEGS (`SEGS`).

Vsakega izmed modulov smo implementirali v obliki modulov za Python, z izjemo zadnjega modula, ki je napisan v jeziku C, vendar ga je preko vmesnika Python za C možno direktno uvoziti tudi v programe, napisane v Pythonu. V nadaljevanju podrobneje opišemo vsakega izmed modulov.

3.3.1 Strežniški modul

Osnovna naloga strežniškega modula je, da "streže" zahtevkom, ki želijo usluge spletnega servisa. Sprejete zahtevke interpretira, obdela vsebino paketov

SOAP in se nanje ustrezno odzove (npr. požene sistem g-SEGS ali pa pošlje nazaj trenutno stanje računanja).

Strežniški modul je implementiran v dveh modulih, napisanih v Pythonu. Naloga prvega modula (`server_gsegs.py`) je, da požene spletni strežnik, medtem ko drugi modul (`service.py`) implementira dejansko obravnavanje zahtevkov.

Spletni strežnik

V sistemu g-SEGS za spletni strežnik uporabimo razred `Server`, ki ga nudi Orange4WS. Ta razred ponuja enostaven način za realizacijo spletnih strežnikov za lastne spletne servise. Ko ima uporabnik implementiran strežniški del ogrodja (opisan v drugem poglavju; v primeru g-SEGS implementiran v modulu `service.py`), lahko s pomočjo razreda `Server` takoj postavi strežnik. Strežniški modul sistema g-SEGS obsega le nekaj vrstic kode v Pythonu:

server_gsegs.py

```
1  from webServices.serverBase import Server
2  from webServices.common import cmdlServer
3
4  import service as gsegs
5
6  if __name__ == "__main__":
7      logFname, port = cmdlServer()
8
9      SERVICE_MODULES = [gsegs]
10     SERVICE_LIST = [x.getService(newPort=port) for x in SERVICE_MODULES]
11     srv = Server(SERVICE_LIST, logFname, port)
12     srv.serveForever()
```

Strežnik nato poženemo na naslednji način:

```
1  $ python server_gsegs.py --port=<številka vrat> --log=<ime datoteke za beleženje>
```

Strežnik, dokler ga ne prekinemo, čaka na zahtevke na podanih vratih in beleži dogodke v željeni datoteki.

Obravnavanje zahtevkov

Omenili smo že, da za obravanje zahtevkov spletnega servisa g-SEGS skrbi modul `service.py`. Ta modul definira dva razreda:

- `gsegsService`, izpeljan iz razreda, ki je generiran na način, opisan v drugem poglavju; implementira operacije spletnega servisa, opisane v poglavju 3.2 in

- `gsegsProcess`, izpeljan iz razreda `GenericWebServiceProcess` sistema Orange4WS. Naloga tega razreda je, da se vsako opravilo izvaja v ločenem procesu, definira pa tudi metode, ki so potrebne za generiranje strukture sporočila `getResultResponse`, ki vsebuje najdene podskupine.

Spodnji odsek prikazuje metode razreda `gsegsService`:

Ogrodje razreda gsegsService

```
class gsegsService(server.gsegs):
    """
    This class handles all requests and responses for the gsegs service.
    """
    def __init__(self, **kwargs):
        ...

    def __del__(self):
        ...

    def soap_rungsegs(self, ps):
        ...

    def soap_getResult(self, ps):
        ...

    def soap_getProgress(self, ps):
        ...
```

Vidimo, da se s pomočjo knjižnice ZSI operacije, definirane v datoteki WSDL, preslikajo v metode razreda, napisanega v Pythonu. Preostali dve metodi (`__init__` in `__del__`) skrbita za inicializacijo in za varno brisanje instanc tega razreda. Parameter `ps`, ki ga sprejmejo omenjene metode, predstavlja sporočilo SOAP v obliki objekta v Pythonu. Naslednji del kode je del metode `soap_rungsegs`, ki ponazarja idejo branja in gradnje sporočil SOAP:

Metoda soap_rungsegs

```
    def soap_rungsegs(self, ps):
        request = ps.Parse(services.rungsegsRequest.typecode)

        # Convert the input data to a list of tuples
        inputData_tmp = request.get_element_inputData()
        inputData = []
        for e in inputData_tmp:
            ex_id = e.get_element_id()
            ex_rank_or_label = e.get_element_rank_or_label() if posClassVal else \
                float(e.get_element_rank_or_label())
            inputData.append((ex_id, ex_rank_or_label))

        # Read all other parameters
        ...

        # Create and start a gsegs process with the given parameters
        proc = gsegsProcess(kwargs=args)
        proc.start()
        try:
            self.procPool.addProcess(proc, jobid=jobID)
```

```
21          except Exception, e:
22              raise Fault(Fault.Server, 'Internal error: %s' % str(e))
23
24          # Build the response object
25          response = services.rungsegsResponse()
26          response.set_element_jobID(jobID)
27          return request, response
```

Prvi korak, ki ga izvede vsaka izmed metod, je, da sporočilo pretvori v objekt, ki predstavlja zahtevek `request`, ki nudi metode za dostop do elementov sporočila SOAP (vrstica 2). Ko imamo objekt `request`, lahko začnemo z branjem posameznih polj sporočila.

Vrstice od 5 do 11 prikazujejo primer branja polj kompleksnega podatkovnega tipa naših vhodnih podatkov `inputData` (glej poglavje 3.2). Metode, ki jih ustvari ZSI za dostop do polj sporočila, so oblike `get_element_X()`, kjer je `X` ime elementa, do katerega želimo dostopati. Primer dostopa do seznama vhodnih podatkov prikazuje vrstica 5. Dobljeni objekt se obnaša kot Pythonov seznam. Izrezani del metode se ukvarja predvsem s pretvorbo vhodnih parametrov v obliko, ki jo sprejme modul g-SEGS, opisan v naslednjem poglavju.

Vrstice od 16 do 22 ustvarijo nov proces g-SEGS z danimi parametri, ki jih zapakiramo v objekt `args`. Proces se doda v nabor že (morebitnih) obstoječih procesov (*ang. process pool*).

Vrstice od 25 do 27 prikazujejo še način gradnje sporočila SOAP. Objekt `services` je ustvarila knjižnica ZSI in nam ponuja konstruktorje za vse tipe sporočil, ki jih definira datoteka WSDL; gre za sporočilo `rungsegsResponse`, ki ga ustvarimo s klicem konstruktorja, kot prikazuje vrstica 25. Elemente sporočil nastavljamo na podoben način, kot jih beremo; to prikazuje vrstica 26. Na koncu metoda vrne terko, ki vsebuje objekt zahtevka `request` in odziva `response`.

Na podoben način smo definirali tudi vse ostale operacije spletnega servisa.

3.3.2 Modul g-SEGS

Modul `g-SEGS` je osrednji modul razvite aplikacije. Napisan je v obliki enega modula v Pythonu in smo ga zaradi večje splošnosti in morebitne uporabe v skriptah razvili tako, da ga je mogoče enostavno uporabljati tudi kot knjižnico.

V modulu je definiran osnovni razred `gsegs`, ki vsebuje metodo `run(...)`, s katero poženemo postopek odkrivanja podskupin, imena različnih parametrov algoritma in prednastavljene vrednosti parametrov. Spodnji del kode prikazuje možen način uporabe sistema kot knjižnice v Pythonovih skriptah:

Primer uporage modula g-SEGS

```
from gsegs import gsegs as gSEGS

runner = gSEGS()
result = runner.run(
            'foo.txt',
            inputData,
            interactions,
            mapping,
            open('ontology1.owl').read(),
            ont2=open('ontology2.owl').read(),
            ont3=open('ontology3.owl').read(),
            posClassVal='Yes'
         )
```

Metoda `run(...)` sprejme naslednje parametre:

- `progressFname` - pot do datoteke za zapis trenutnega napredka računanja,
- `inputData` - vhodni podatki, zapisani s seznamom parov oblike

  $$[(id_1, value_1), \ldots, (id_i, value_i), \ldots, (id_n, value_n)],$$

  kjer je id_i identifikator i-tega vhodnega objekta oz. instance (npr. gen ali oseba), $value_i$ pa je rang ali razred (*ang. class*) - odvisno od problema, ki ga rešujemo,
- `interactions` - seznam parov vhodnih instanc oz. objektov, ki so na nek način povezani med sabo, predstavljeni so s seznamom oblike

  $$[(id_1, id_{11}), \ldots, (id_i, id_{ij}), \ldots, (id_n, id_{nk})],$$

  kjer je objekt id_i povezan z objektom id_{ij}. Če interakcij ni ali jih ne poznamo, potem lahko podamo prazen seznam `[]`,
- `mapping` - preslikava med identifikatorji vhodnih objektov in koncepti v danih ontologijah; preslikavo podamo s seznamom oblike

  $$[(id_1, \mathrm{URI}_{11}, \ldots), \ldots, (id_i, \mathrm{URI}_{i1}, \mathrm{URI}_{i2}, \ldots), \ldots, (id_n, \mathrm{URI}_{n1}, \ldots)],$$

  kjer velja, da je objekt id_i označen (*ang. annotated*) s koncepti z identifikatorji $\mathrm{URI}_{i1}, \mathrm{URI}_{i2}, \ldots$ iz ontologij `ont1` do `ont4`,
- `ont1` - serializirana (v obliki zaporedja znakov) ontologija OWL.

Vsi navedeni parametri morajo biti nujno podani s strani uporabnika, medtem ko so nadaljni parametri neobvezni.

- `ont2`, `ont3` in `ont4` - dodatne serializirane ontologije OWL; privzeto `None`,
- `generalTerms` - seznam oznak konceptov, ki so preveč splošni, da bi bili uporabljeni v pravilih; privzeto `[]`,
- `legacy` - zastavica, v primeru, da je postavljena na `True`, omogoča uporabo prvotnih formatov sistema SEGS (vpliva na parameter `ont1`); privzeto `False`,
- `posClassVal` - oznaka razreda, ki smo si ga izbrali kot pozitivnega - podskupine se iščejo za ta razred; privzeto `None` - v tem primeru se privzame problem rangiranja,
- `cutoff` - v primeru problema rangiranja si izberemo prvih `cutoff` objektov, nad katerimi se potem odkriva podskupine; privzeto `None`, vendar mora biti v tem primeru podan `posClassVal`,
- `wracc_k` - parameter k mere za ocenjevanje kvalitete podskupin Weighted Relative Accuracy (WRACC) (opisane v nadaljevanju); privzeto 5,
- `minimalSetSize` - podpora (*ang. support*), najmanjše število objektov, ki jih lahko opisuje pravilo; privzeto 5,
- `maxNumTerms` - največje število termov v pravilu, navzgor omejeno s številom ontologij, ki smo jih podali; privzeto 3,
- `maxReported` - število podskupin, ki jih sistem vrne; privzeto 100.

Zaradi združljivosti z delotoki, v katerih nastopa prvotni sistem SEGS, smo obdržali naslednje parametre, ki so smiselni zgolj za biološke domene:

- `summarizeDescriptions` - če je ta zastavica postavljena, sistem poskuša izločiti pravila, ki opisujejo iste objekte; privzeto `False`,
- `maximalPvalue` - podskupine, za katere je verjetnost, da so naključne, večja od `maximalPvalue`, izločimo (velja le za problem rangiranja); privzeto 0.05,
- `weightFisher`, `weightGSEA`, `weightPAGE` - uteži za različne mere, s katerimi jih kombiniramo v eno mero; privzeto 1.0, 0.0 in 0.0,
- `randomSeed` - seme za generator naključnih števil; privzeto 10.

Naloga tega modula je, da ovije sistem SEGS na način, da lahko le-ta sprejme poljubne ontologije in podatke. Da to dosežemo, moramo vse vhodne podatke prepisati v format, ki ga sprejema SEGS in ustrezno nastaviti parametre glede na tip problema, ki ga rešujemo, saj je ena izmed novosti sistema g-SEGS v primerjavi s sistemom SEGS odkrivanje podskupin na označenih podatkih in ne le na rangiranih podatkih.

Pretvorba ontologij

Pretvorba vhodnih podatkov in interakcij med objekti je trivialna, medtem ko smo za pretvorbo ontologij implementirali ločeno orodje (imenovano OWL2X) za pretvorbo ontologij iz formata OWL v druge formate. Trenutno podpira pretvorbo v format SEGS in v predikate v Prologu. Morebitne druge formate, za katere bi se pojavila potreba v prihodnosti, pa lahko enostavno dodajamo. To orodje je napisano s pomočjo knjižnice Jena v jeziku Java.

Orodje poženemo v ukazni lupini na naslednji način:

Primer uporage OWL2X

```
1 java -jar owl2x.jar (segs|prolog) (long|short) <outDir> <mapfile> <ontology1>.owl [...]
```

Prvi parameter (`segs` ali `prolog`) predstavlja format, v katerega pretvarjamo, drugi parameter (`long` ali `short`) pove ali želimo za predikate uporabljati kratka ali dolga imena (celotni URI koncepta) posameznega koncepta v ontologiji. Parameter `outDir` pove pot do direktorija, kamor naj orodje shrani izhodne datoteke, parameter `mapFile` pa predstavlja pot do datoteke, ki definira preslikavo med vhodnimi objekti in koncepti v ontologijah. Na koncu še navedemo poljubno število ontologij (oz. poti do teh ontologij), razen v primeru, ko pretvarjamo v format SEGS, največ 4.

Za lažjo uporabo v Pythonu smo za OWL2X razvili tudi modul za Python, ki ovije program v Javi tako, da orodje v našem sistemu uporabljamo preko razreda `OWL2X` in njegovih metod. Če želimo pretvoriti tri vhodne ontologije OWL v format za SEGS, potem lahko OWL2X v Pythonu uporabimo na naslednji način:

Primer uporage OWL2X

```
1 ont, g2ont = OWL2X.get_segs_input([ont1, ont2, ont3], mapping)
```

Metoda vrne dva objekta - `ont` in `g2ont`. Oba sta v primernem zapisu, da ju lahko takoj posredujemo kot parameter modulu `SEGS`.

Sistem SEGS ima definiran svoj format za vhodne ontologije. Vhodna datoteka, ki vsebuje vse ontologije, je naslednje oblike:

Format SEGS

```
1 [term_ID, [branch_of_ontology, is_a_list, part_of_list]]
2 ...
```

kjer je `term_ID` indentifikator koncepta s predpono `GO:`, `branch_of_ontology` je ime ontologije, ki ji pripada koncept, `is_a_list` ter `part_of_list` pa sta seznama identifikatorjev konceptov, s katerimi je obravnavani koncept v relaciji (`is_a` oz. `part_of`).

Zaradi specifičnosti tega zapisa se pojavi omejitev na največ štiri vhodne ontologije, saj lahko `branch_of_ontology` zavzame le točno določene štiri vrednosti:

- `molecular_function`,
- `cellular_component`,
- `biological_process` ali
- `KEGG_pathway`.

Orodje OWL2X poskrbi, da se vsaka vhodna ontologija preslika v eno izmed teh vrednosti, za ustrezno številčenje identifikatorjev in da ostane struktura enaka tisti, ki je določena v zapisu OWL. Sama pretvorba ni zahtevna, ker SEGS upošteva le relacijo `is_a`. Gre le za enostaven rekurzivni sprehod po strukturi ontologije z iskanjem v globino, s časovno kompleksnostjo $O(n)$, kjer je n število konceptov v ontologiji.

3.3.3 Dopolnjeni modul SEGS

Osnovna metoda za odkrivanje podskupin v našem sistemu temelji na implementaciji sistema SEGS, predstavljenega v [5]. Kljub temu, da je srž odkrivanja še vedno enaka, smo vseeno morali poseči v jedro tega sistema.

Gradnja pravil

Na tem mestu opišimo osnovni postopek (glej sliko 3.5 za psevdo kodo), ki ga SEGS (in s tem tudi g-SEGS) uporablja za gradnjo podskupin. Algoritem je sledeč. Recimo, da imamo tri vhodne ontologije: $A = \{a_1, a_2, \ldots\}, B = \{b_1, b_2, \ldots\}$ in $C = \{c_1, c_2, \ldots\}$, kjer med koncepti velja:

$$a_1 \succ a_2 \succ \ldots,$$

$$b_1 \succ b_2 \succ \ldots,$$

$$c_1 \succ b_2 \succ \ldots,$$

kjer $x \succ y$ pomeni, da je koncept x bolj splošen od y. Velja tudi:

$$x \succ y$$

$$\equiv$$

$$y \text{ subClassOf } x.$$

$$\equiv$$

$$y \text{ is\_a } x.$$

Vsak koncept si lahko predstavljamo kot množico objektov, ki mu pripadajo. Zaradi lastnosti relacije `is_a` objekt pripada konceptu, če je označen z njim ali s katerim izmed njegovih potomcev, saj je množica objektov koncepta enaka uniji množice objektov njegovih potomcev.

Gradnja pravil se začne z dodajanjem umetnih (*ang. dummy*) konceptov v vhodne ontologije. Resničnemu korenskemu konceptu vsake ontologije algoritem doda novega starša tako, da so s temi koncepti označeni vsi vhodni objekti oz. primeri. Ti umetni koncepti naj bodo a_0, b_0 in c_0. Velja:

$$a_0 \succ a_1,$$

$$b_0 \succ b_1,$$

$$c_0 \succ c_1.$$

Naj bo $target(A)$ ciljni predikat, ki pove, ali objekt A pripada ciljnemu razredu ali ne. Velja še, da $target(A) \leftarrow a_0 \wedge b_0 \wedge c_0$ ustreza praznemu pravilu $target(A) \leftarrow true$, ki pokriva vse primere.

Učni algoritem začne s praznim pravilom $target(A) \leftarrow true$, ki mu poskuša dodati nov koncept oz. člen (*ang. term*) a_0 (koncept igra vlogo konjunktivnega člena v pravilu). Na tem mestu preveri, ali takšno, novo pravilo ustreza kriteriju za najmanjšo velikost (parameter `MIN_SIZE`) in če ustreza, še preveri ali pravilo ne vsebuje preveč (parameter `MAX_TERMS`) ali premalo členov. Pri izračunu števila členov se umetni členi ne upoštevajo (funkcija `clean` vrne pravilo brez umetnih členov), zato se lahko zgodi, da ima pravilo nič členov.

V naslednjem koraku algoritem preveri, ali pravilu še lahko konjunktivno doda kak nov člen in če lahko, potem rekurzivno pokliče proceduro za gradnjo z umetnim konceptom naslednje ontologije. Za tem rekurzivno kliče proceduro še za vsakega izmed otrok trenutnega koncepta. V primeru a_0 je to a_1.

Za vsak obravnavani koncept oz. člen preveri še množico objektov, s katerimi so v *neki interakciji* objekti, ki pripadajo obravnavanemu konceptu

(vrstice od 28 dalje). Naj bo $X = \{o_1, o_2, \ldots\}$ nek koncept iz ontologije, $o_1, o_2, \ldots$ pa objekti, ki pripadajo temu konceptu. Potrebujemo še definicijo relacije $interacts(a, b)$:

$$interacts(a, b) \Leftrightarrow \text{objekt } a \text{ je v neki interakciji z objektom } b. \tag{3.1}$$

Relacija *interacts* je simetrična.

V psevdo-kodi navajamo še *funkcijo* `interacts`, ki pa je definirana nad množico. Množico objektov Y, s katerimi so v interakciji objekti množice X, poiščemo na naslednji način:

$$Y = interacts(X) = \{y | \exists x \in X, interacts(x, y)\}. \tag{3.2}$$

S to funkcijo pa sistem SEGS uvaja še novo vrsto členov pri generiranju pravil. Recimo, če algoritem v nekem koraku gleda pravilo (dodaja koncept C_j):

$$target(A) \leftarrow C_i \wedge C_j,$$

potem hkrati poskusi generirati še pravilo:

$$target(A) \leftarrow C_i \wedge interacts(C_j).$$

To pravilo interpretiramo kot: vsi objekti A, ki pripadajo konceptu C_i in so v interakciji z objekti, ki pripadajo konceptu C_j.

Z rekurzivnimi klici procedure za gradnjo pravil SEGS zgradi seznam pravil, ki ustrezajo danim omejitvam glede velikosti. Takih pravil je kljub postavljenim omejitvam lahko precej, zato koraku gradnje pravil sledi še korak, v katerem sistem oceni dobljena pravila, da lahko potem uporabnik izlušči zgolj najboljša.

V najslabšem primeru bo algoritem zgradil vsa možna pravila. V najslabšem primeru imamo 4 ontologije in največ n konceptov v vsaki izmed njih. Če se omejimo na pravila dolžine 4 in ker SEGS v pravilu uporabi največ en konjunkt iz vsake ontologije, je možnih pravil

$$\left(\begin{array}{c} 4n \\ 4 \end{array} \right) = \frac{(4n)!}{4!(4n-4)!} = \frac{4n(4n-1)(4n-2)(4n-3)}{4!}$$

to pa je reda $O(n^4)$ pravil. V praksi je tak scenarij malo verjeten, saj se algoritem s pomočjo parametra o minimalni velikosti podskupin izogne nepotrebnemu preiskovanju pravil, ki pokrivajo premalo primerov. To je možno zaradi narave relacije `is_a` oz. $\succ$. Če ontološka koncepta C_i in C_j, za katera velja

$C_i \succ \ldots \succ C_j$, interpretiramo kot množico primerov, označenih z njima, potem velja:

$$|C_i| \geq |C_j|$$

oziroma

$$C_i \supseteq C_j$$

Na kratko: dani koncept vedno pokriva vsaj toliko primerov kot katerikoli izmed njegovih potomcev. Idejo lahko ponazorimo na primeru. Če v nekem trenutku algoritem poskusi specializirati pravilo $target(A) \leftarrow C_i \wedge C_j \wedge C_k$ tako, da namesto koncepta C_k v pravilu uporabi enega izmed njegovih potomcev C'_k in če ugotovi, da pravilo pokriva premalo primerov, tega pravila ne zgenerira; izpusti pa tudi vsa druga pravila, ki bi jih dobil s specializacijo člena/koncepta C'_k, saj se lahko število primerov, ki jih pravilo pokriva, kvečjemu zmanjša.

Ocenjevanje zgrajenih pravil

V prejšnjem poglavju je opisan postopek generiranja pravil. Zgrajenih pravil je lahko precej, lahko so nezanimiva in lahko se med seboj prekrivajo. Da bo algoritem lahko skonstruiral zares pomembna pravila, ki bi uporabniku lahko povedala kaj novega, sistem SEGS nudi različne mere za ocenjevanje kvalitete generiranih pravil.

Osnovna metoda SEGS podpira različne mere za ocenjevanje podskupin:

- Fisherjev test,
- PAGE (Parametric Analysis of Gene Set Enrichment) in
- GSEA (Gene Set Enrichment Analysis).

a so le-te neprimerne za splošno uporabo, saj so (predvsem zadnji dve) namenjene predvsem analizi skupin genov [6]. Fisherjev test je sicer dovolj splošen za uporabo tudi na drugih domenah, a je namenjen predvsem ocenjevanju podskupin v domenah z malo primeri.

Odločili smo se, da v okviru sistema g-SEGS implementiramo tudi bolj splošno mero za ocenjevanje podskupin, imenovano $WRAcc$ (Weighted Relative Accuracy) [17]. V sistemu uporabljamo dve izvedenki te mere. Prvo, $wWRAcc$ (Weighted Relative Accuracy with example weights), uporabljamo za filtriranje generiranih pravil. Drugo, osnovno $WRAcc$, pa nato izračunamo nad že filtriranimi pravili in podaja oceno, ki jo lahko interpretiramo. V nadaljevanju bomo uporabljali naslednje oznake:

- $p(\cdot)$ - verjetnost dogodka,

Psevdo-koda postopka

```
function construct (rule, term, k):
   # rule - trenutno pravilo, ki jo želimo podaljšati
   # term - člen/koncept s katerim želimo podaljšati pravilo
   # k - dodajamo term iz k-te ontologije

   # Množica, ki jo opisuje novo pravilo
   newSet = intersection(set(rule), set(term))

   # Ali je množica dovolj velika?
   if newSet.size > MIN_SIZE then
     rule.add(term)
     if clean(rule).size < MAX_TERMS and clean(rule).size > 0 then
       rules.add(rule)
     end if

     # Ali pravilo lahko še podaljšamo?
     if rule.size < max(MAX_TERMS, MAX_ONTOLOGIES) then
       construct(rule, ontologies[k+1], k+1)
       rule.remove(term)

       # Pravilo podaljšamo še z vsemi nasledniki
       for each child in children(term) do
          construct(rule, child, k)
       end for
     end if
   end if

   # Hkrati pogledamo še množico, ki je v interakciji z
   # objekti, ki jih opisuje trenutni predikat
   interactingSet = intersection(set(rule), interacts(set(term)))
   if interactingSet.size > MIN_SIZE then
     rule.add('interacts(' term ')')
     if clean(rule).size < MAX_TERMS then
       rules.add(clean(rule))
     end if
   end if
   return rules
```

Slika 3.5: Postopek za gradnjo podskupin sistemov g-SEGS in SEGS.

- Cnd - pogojni del pravila (*ang. antecedent*),
- $Class$ - razred oz. posledica pravila (*ang. consequent*).

Mera $WRAcc$ je sestavljena iz dveh osnovnih delov:

1. *splošnost pravila* (*ang. generality*) $p(Cnd)$, ki označuje velikost podskupine glede na celotno število primerov in
2. *relativna točnost* (*ang. relative accuracy*), ki podaja razliko med točnostjo pravila $p(Class|Cnd)$ in točnostjo praznega pravila $p(Class)$.

Relativna točnost nam pove, koliko je gledano pravilo boljše od praznega pravila - torej, če vse primere razvrstimo v razred $Class$. Ker so pravila z visoko relativno točnostjo lahko preveč specifična, je v meri relativna točnost še utežena s splošnostjo pravila. Mera $WRAcc$ je definirana na naslednji način:

$$WRAcc(Class \leftarrow Cnd) = p(Cnd) \cdot (p(Class|Cnd) - p(Class)). \quad (3.3)$$

V naši implementaciji verjetnosti ocenjujemo z relativno frekvenco, lahko pa bi za ocenjevanje verjetnosti uporabili tudi m-oceno ali Laplace oceno. $WRAcc$ izračunamo z relativnimi frekvencami na naslednji način:

$$WRAcc(Class \leftarrow Cnd) = \frac{n(Cnd)}{N} \cdot \left(\frac{n(Class \wedge Cnd)}{n(Cnd)} - \frac{n(Class)}{N} \right). \quad (3.4)$$

Mera, ki smo jo uporabili za filtriranje pravil, se imenuje $wWRAcc$ in izhaja iz osnovne mere $WRAcc$. To mero smo uporabili kot hevristiko za izbiro najboljših pravil, podobno kot so avtorji predlagali v [18], saj na ta način pri izračunu kvalitete pravila upoštevamo še, *katere* primere pravilo pokriva. Lahko imamo dve zelo podobni pravili, ki sta enako dobro ocenjeni glede na neko mero, a za uporabnika zaradi svoje podobnosti nista zanimivi, saj pokrivata podoben prostor primerov. Z izbranimi pravili želimo imeti pokrit celoten prostor primerov (vse primere) in želimo, da je čimveč izbranih pravil za uporabnika zanimivih.

Da z mero dosežemo opisani učinek, vsakemu primeru priredimo utež $w(e_i) = \frac{1}{m_i}$, kjer $m_i < k$, pove koliko pravil že pokriva primer e_i, k pa je parameter, ki pove, največ kolikokrat je primer lahko pokrit preden ga odstranimo iz množice primerov. To utež upoštevamo pri izračunu relativnih frekvenc v meri $wWRAcc$ tako, da primer, ki je bil pokrit večkrat, prispeva k oceni manj kot bi sicer. Mera $wWRAcc$ je definirana na naslednji način:

$$wWRAcc(Class \leftarrow Cnd) = \frac{n'(Cnd)}{N'} \cdot \left(\frac{n'(Cnd \wedge Class)}{n'(Cnd)} - \frac{n'(Class)}{N'} \right), \quad (3.5)$$

kjer N', $n'(Cnd)$ in $n'(Cnd \wedge Class)$ izračunamo na naslednji način (E je množica vseh primerov):

$$N' = \sum_{e_i \in E} w(e_i), \tag{3.6}$$

$$n'(Cnd) = \sum_{e_i \in E'} w(e_i), \tag{3.7}$$

$$n'(Cnd \wedge Class) = \sum_{e_i \in E''} w(e_i) \tag{3.8}$$

in kjer sta E' in E'':

$$E' = \{e_i | e_i \in E, \text{ za } e_i \text{ velja pogoj } Cnd\}, \tag{3.9}$$

$$E'' = \{e_i | e_i \in E, \text{ za } e_i \text{ velja pogoj } Cnd \text{ in pripada razredu } Class\}. \tag{3.10}$$

Opisani meri v sistemu SEGS uporabljamo na naslednji način. Najprej po prej opisani metodi za gradnjo pravil zgradimo vsa pravila, ki zadoščajo pogojem glede omejitev velikosti. Nato izmed teh pravil s pomočjo mere $wWRAcc$ iterativno izbiramo pravila. V vsakem koraku za vsa pravila izračunamo oceno, izberemo tisto z največjo in ga odstranimo iz množice pravil. Po vsaki iteraciji ustrezno popravimo uteži pokritih primerov. Te korake ponavljamo, dokler obstaja še kak primer oz. pravilo. Idejo prikazuje psevdo koda na sliki 3.6.

3.4 Uporabniški del aplikacije

Uporabniški del aplikacije je tisti, ki vedno teče na uporabnikovem računalniku. V sistemu g-SEGS skrbi za komunikacijo s strežnikom in nudi grafični vmesnik do funkcionalnosti sistema g-SEGS.

3.4.1 Grafični vmesnik

Sistem Orange omogoča razvijalcu vključevanje svojih gradnikov v Orange Canvas tako, da definira svoj t.i. *widget* (v nadaljevanju mu rečemo gradnik). Vsak gradnik v sistemu Orange ima vhodne in izhodne signale določenih tipov; ob dvojnem kliku na gradnik pa se odprejo še dodatne možnosti (npr. dodatni parametri), ki določajo izvajanje operacije gradnika.

Sistem Orange4WS uporabniku omogoča, da v Orange vključuje tudi spletne servise tako, da sistem na podlagi definicije WSDL samodejno ustvari gradnik za vsako operacijo spletnega servisa. Take gradnike lahko uporabnik nato povezuje v delotoke z že obstoječimi.

Psevdo-koda izbiranja pravil

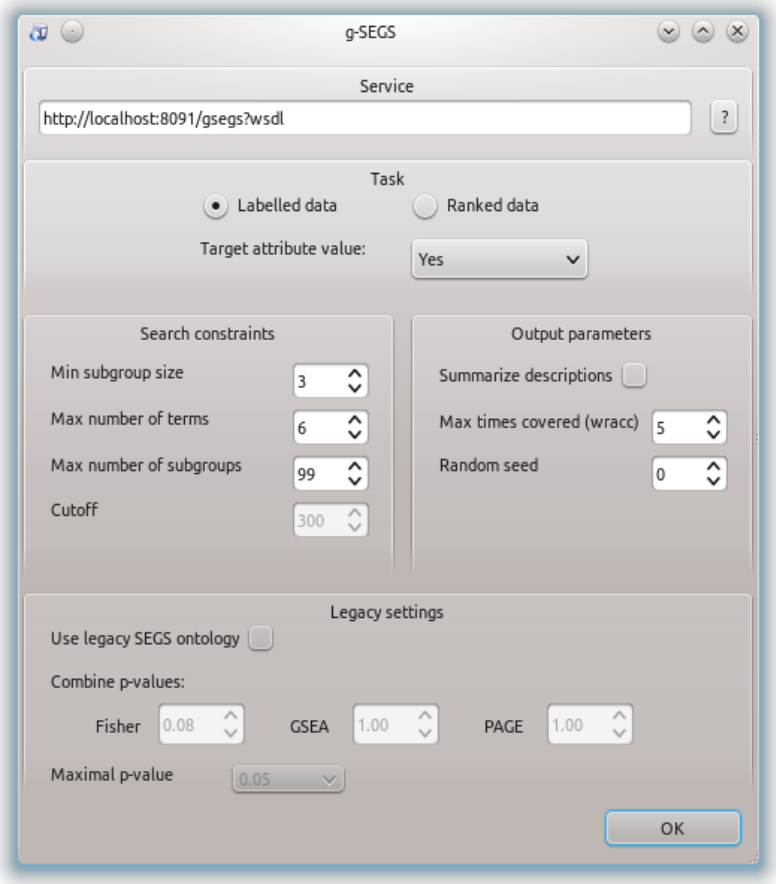
```
function ruleSelection(examples, k):
   # examples - množica primerov
   # k - največ koliko pravil lahko pokriva isti primer

   # Pokličemo metodo za gradnjo pravil
   ruleSet = construct ([], ontologies[0], 0)
   resultSet = []
   repeat
     # Funkcija bestRule za vsako pravilo izračuna wWRAcc in vrne
     # najboljše pravilo
     rule = bestRule(ruleSet)
     resultSet.add(rule)
     # Pokritim primerom zmanjšamo utež in odstranimo tiste, ki so
     # bili pokriti ze k-krat
     decreaseWeights(examples, rule, k)
   until examples == [] or ruleSet == []

   # Za vsako pravilo izračuna še osnovno mero WRAcc in
   # ignoriramo uteži primerov
   for each rule in resultSet:
      rule.score = WRAcc(rule)
   end for
   return resultSet
```

Slika 3.6: Postopek izbiranja pravil.

Tak samodejno zgrajen gradnik se nam za servis g-SEGS ni zdel smiseln, saj se izkaže, da je težaven za uporabo. Razlog je v tem, da se pri avtomatski gradnji gradnikov vsaka operacija preslika v en gradnik, vsak vhodni parameter se preslika v svoj vhodni signal, izhodi posamezne operacije spletnega servisa pa se preslikajo v izhodne signale gradnika. Taka rešitev je sicer dovolj dobra, če gre za spletne servise z malo parametri, v našem primeru pa ima servis 21 vhodnih parametrov, kar naredi gradnik neprimeren za uporabo in lahko odvrne uporabnika. Zaradi tega smo se odločili za naš spletni servis razviti prilagojen grafični vmesnik.

Grafični vmesnik, ki se uporabniku prikaže po dvojnem kliku na gradnik g-SEGS, prikazuje slika 3.7, vhodne in izhodne signale pa sliki 3.8 ter 3.9.

Preko signalov uporabnik poveže vhodne podatke (v obliki seznama parov ali v obliki Orangeove podatkovne strukture `ExampleTable` za predstavitev



Slika 3.7: Grafični vmesnik sistema g-SEGS.

tabel), ontologije, splošne terme, podatke o interakcijah in preslikavo med vhodnimi primeri in koncepti v ontologijah.

Na izhodnih signalih dobi uporabnik sezname pravil (oz. podskupin) sortirane po različnih merah. Dobljene podskupine lahko pošlje v gradnik za vizualizacijo ali jih shrani v datoteko (lahko tudi v zapisu XML preko signala `resultsAsXML`).

Ko uporabnik gradnik ustrezno poveže v delotok, lahko z dvojnim klikom nanj izbere še dodatne možnosti. V tem pogledu lahko izbere naslov spletnega servisa, ciljni razred, če gre za označene podatke in različne lastnosti, ki naj jih imajo izhodne podskupine in ki vplivajo na iskanje. Če želi uporabnik podati podatke v formatu osnovnega sistema SEGS, lahko to stori z označitvijo postavke `Use legacy SEGS ontology` in nato nastavlja tudi različne parametre prvotnega sistema, ki so namenjeni analizi skupin genov.

Svoj gradnik v sistem Orange dodamo tako, da iz razreda `OWBaseWidget` (iz katerega so izpeljani vsi gradniki v Orange) izpeljemo nov razred; v našem primeru se imenuje `OWgsegs`. Dani razred igra v našem sistemu dve vlogi. Prva vloga je implementacija grafičnega vmesnika, druga vloga pa komunikacija s strežnikom spletnega servisa, kar opisujemo v naslednjem poglavju.

Z definicijo ustreznih razrednih spremenljivk definiramo vhodne in izhodne signale in metode, ki naj obravnavajo dogodek, ko uporabnik kaj poveže na gledani signal. Spodnji odsek kode prikazuje primer definicije enega izmed vhodnih signalov (ostale zaradi preglednosti izpuščamo):

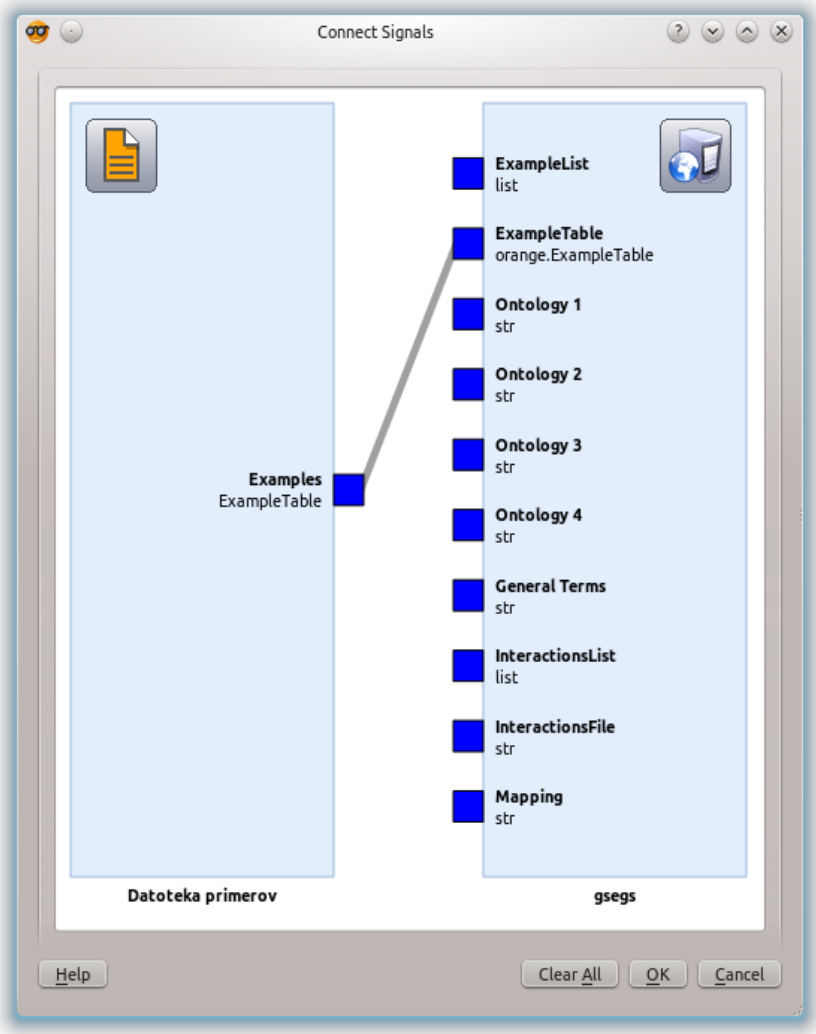
```
1 self.inputs = [("ExampleTable", orange.ExampleTable, self.receiveDataset), ...]
```

Seznam `self.inputs` je seznam trojk. V vsaki trojki navedemo ime signala (v tem primeru `ExampleTable`, lahko pa je poljubno), tip (`orange.ExampleTable`) in metodo (`self.receiveDataset`), ki bo skrbela za dogodke v zvezi s tem signalom. Na podoben način definiramo izhodne signale, s to razliko, da nam pri le-teh ni potrebno navesti metode za obravnavanje dogodkov.

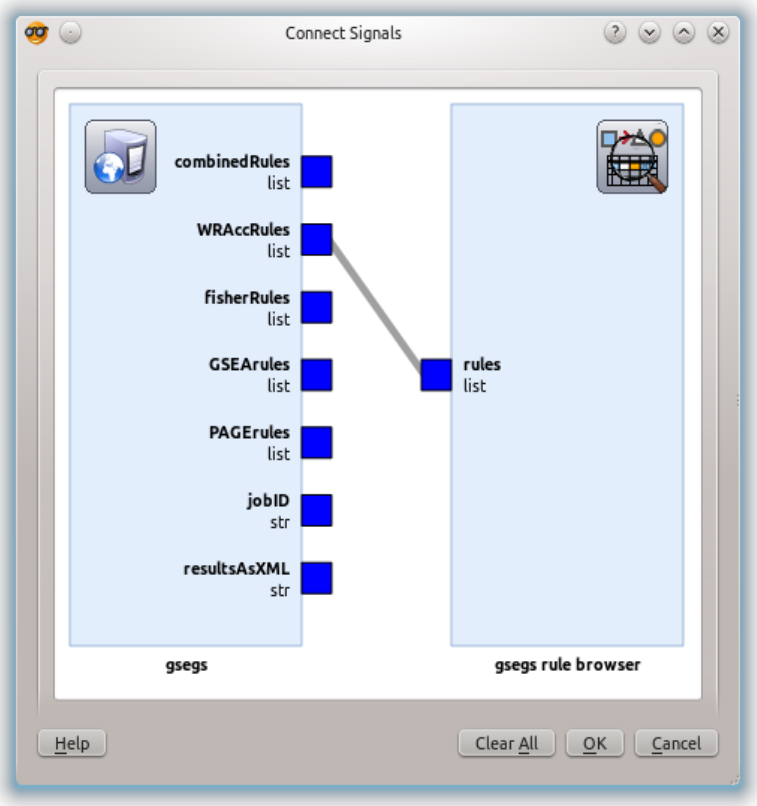
Metoda `receiveDataset` dobi kot parameter objekt, ki ga uporabnik postavi na vhodni signal. Podpis metode je naslednji:

```
1 def receiveDataset(self, data):
2     ...
```

Parameter `data` torej predstavlja objekt, ki ga je uporabnik postavil na ta signal. Naloga metode je potem, da ustrezno obravnava prejete podatke. V našem primeru vhodne podatke pretvori v strukturo, ki jo definira sporočilo SOAP. Podobno naredimo za vse signale.



Slika 3.8: Vhodni signali gradnika g-SEGS.



Slika 3.9: Izhodni signali gradnika g-SEGS.

Zanimiva je tudi uporaba uporabniškega vmesnika, ki smo ga definirali s pomočjo orodja Qt Designer (opisano v drugem poglavju). Ko uporabnik dobi definicijo vmesnika v obliki modula v Pythonu (`ui_gsegs.py`), lahko vmesnik uporabi na naslednji način:

```
1 self.ui = ui_gsegs.Ui_Form()
2 self.ui.setupUi(self)
```

Metode za obravnavanje dogodkov za posamezne komponente vmesnika specificiramo na naslednji način:

```
1 self.ui.okButton.clicked.connect(self.startProcessing)
```

V tem primeru gre za nastavitev metode `startProcessing` za dogodek, ko uporabnik klikne na gumb za potrditev. Metoda je ob dogodku poklicana in dobi trenutno stanje elementa (npr. če gre za element izbiranja (*ang. checkbox*), dobi vrednost `True` ali `False`). S temi principi smo razvili celotno funkcionalnost uporabniškega vmesnika.

3.4.2 Komunikacija s strežnikom

Druga vloga, ki jo ima v našem sistemu razred `OWgsegs`, je še komunikacija s strežnikom spletnega servisa. Tudi pri tem si pomagamo s sistemom Orange4WS, ki tako kot na strežniški strani preko klica funkcije vrne štrclje na podlagi vhodne definicije WSDL. Preko naslova spletnega servisa, ki ga poda uporabnik, sistem najprej preveri, ali je dan naslov veljaven - če je spletni servis v resnici tudi na voljo. Če je, potem na podlagi definicije WSDL, ki jo dobi z danega naslova, generira štrclje na enak način kot na strežniški strani:

```
1 self.serviceURI = str(self.ui.serviceLineEdit.text())
2 stubs = importZSIstubsFromURI(self.serviceURI)
```

Ko uporabnik izda ukaz za izvajanje, je za komunikacijo s strežnikom potrebnih več korakov. Izvorna koda na sliki 3.10 prikazuje glavne potrebne korake:

1. generiranje zahtevka za pričetek iskanja opisov podskupin (vrstice od 6 do 12),
2. pošiljanje zahtevka in odgovor - identifikator opravila (vrstice od 16 do 21),

3. povpraševanje po stanju izvajanja, ki ga posredujemo uporabniku (vrstice od 25 do 31),
4. pošiljanje zahtevka za rezultat (vrstice od 34 do 36) in
5. pošiljanje rezultata na izhodne signale gradnika (vrstica 40).

Komunikacija s strežnikom

```
1  def execute(self):
2      # Povezava na spletni servis
3      locator = self.services.gsegsLocator()
4      port = locator.getgsegs()
5
6      # Sporočilo tipa rungsegsRequest
7      request = self.services.rungsegsRequest()
8
9      # Nastavimo elemente sporočila
10     request.set_element_inputData(self.dataset)
11     request.set_element_generalTerms(self.genTerms)
12     request.set_element_interactions(self.interactions)
13     request.set_element_mapping(self.mapping)
14     ...
15
16     # Pošljemo sporočilo in preberemo odgovor
17     try:
18         response = port.rungsegs(request)
19     except Exception, e:
20         self.error('gsegs service error:\n%s' % str(e))
21         return
22     jobid = response.get_element_jobID()
23     ...
24
25     # Sporočilo tipa getProgressRequest
26     request = self.services.getProgressRequest()
27     request.set_element_jobID(jobid)
28
29     # V zanki povprašujemo po stanju
30     ...
31         newProgress = port.getProgress(request).get_element_progress()
32     ...
33
34     # Ko je postopek končan pošljemo sporočilo getResultRequest
35     request = self.services.getResultRequest()
36     request.set_element_jobID(jobid)
37     segsResult = port.getResult(request)
38     self.progressBar.finish()
39
40     # Elemente rezultata pošljemo na izhodne signale
41     self.send('WRAccRules', segsResult.get_element_rules_wracc())
42     ...
```

Slika 3.10: Komunikacija s strežnikom v g-SEGS. Zaradi preglednosti vmesne korake izpuščamo.

3.5 Primer uporabe

V tem poglavju navajamo tipičen postopek uporabe sistema g-SEGS. Recimo, da ima uporabnik na voljo podatke v enem izmed običajnih zapisov vhodnih podatkov v strojnem učenju, ki jih podpira Orange (npr. tab ali arff), ter tri ontologije v zapisu OWL. Za ustrezno delovanje mora uporabnik pridobiti še datoteko, ki definira preslikavo med vhodnimi primeri in koncepti v ontologijah; zaenkrat še ne poznamo orodja, ki bi pomagal definirati tako datoteko.

Za prikaz uporabe bomo uporabili izmišljeni primer. Recimo, da imamo na voljo podatke, ki jih v dodatku prikazuje tabela D.1. Gre za domeno potrošnikov. Predstavljamo si lahko, da je dane podatke zbrala banka, ki ima o svojih strankah podatke o poklicu, prebivališču, različnih storitvah, ki jih uporabljajo, in o tem ali so zapravljivci ali ne (stolpec `big_spender` - razredna spremenljivka problema). Podrobnosti problema za prikaz uporabe niso pomembne in jih navajamo v petem poglavju.

Recimo, da ima uporabnik še neko dodatno znanje o tej domeni v obliki treh ontologij. Na voljo ima ontologijo poklicev, ontologijo bančnih storitev in ontologijo lokacij. Primere ontologij prilagamo v dodatku C.

Glede na vhodne podatke lahko primere označimo z različnimi koncepti iz ontologij. Primer: če vemo, da je obravnavana oseba po poklicu zdravnik (`occupation=Doctor`), potem ga lahko označimo z ontološkim konceptom poklica `Doctor`. Naš primer je sestavljen tako, da lahko za vsako vrednost atributa v ontologijah najdemo ustrezen koncept. Če je vrednost katerega izmed atributov `No`, potem koncepta za ta atribut ne navedemo. Vhodna datoteka, ki preslika vhodne primere v ontološke koncepte, ima naslednji format:

Format preslikave

```
...
[i, [URI1, URI2, ...]]
...
```

Prva vrednost, `i`, je vedno zaporedna številka primera iz tabele primerov. Sledijo ji z vejicami ločeni identifikatorji konceptov iz vhodnih ontologij. Primer datoteke za prvih pet primerov iz tabele D.1 (zaradi preglednosti navajamo le označitvi za prva dva atributa):

Primer preslikave

```
[0, [http://kt.ijs.si/ontologies/occupation.owl#Doctor, http://kt.ijs.si/ontologies/geography.owl#Milan, ...]]
[1, [http://kt.ijs.si/ontologies/occupation.owl#Doctor, http://kt.ijs.si/ontologies/geography.owl#Krakow, ...]]
[2, [http://kt.ijs.si/ontologies/occupation.owl#Military, http://kt.ijs.si/ontologies/geography.owl#Munich, ...]]
[3, [http://kt.ijs.si/ontologies/occupation.owl#Doctor, http://kt.ijs.si/ontologies/geography.owl#Catanzaro, ...]]
[4, [http://kt.ijs.si/ontologies/occupation.owl#Energy, http://kt.ijs.si/ontologies/geography.owl#Poznan, ...]]
...
```

Uporabnik lahko uporabi tudi podatke o interakcijah med vhodnimi primeri, ni pa to obvezno. Datoteka mora biti v naslednjem formatu:

Primer interakcij

```
...
[i, [i1, i2, ...]]
...
```

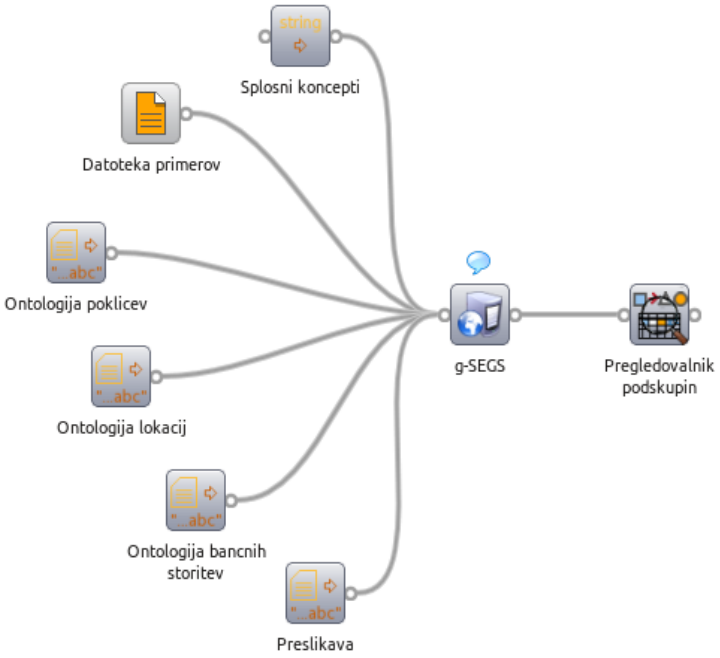
kjer dana vrstica pomeni, da je primer z zaporedno številko `i` v neki interakciji s primeri z zaporednimi številkami `i1, i2, ...` (in obratno).

Ko ima uporabnik pripravljene vse datoteke, lahko prične z gradnjo delotoka. Za gradnjo delotoka smo v našem primeru uporabili naslednje gradnike:

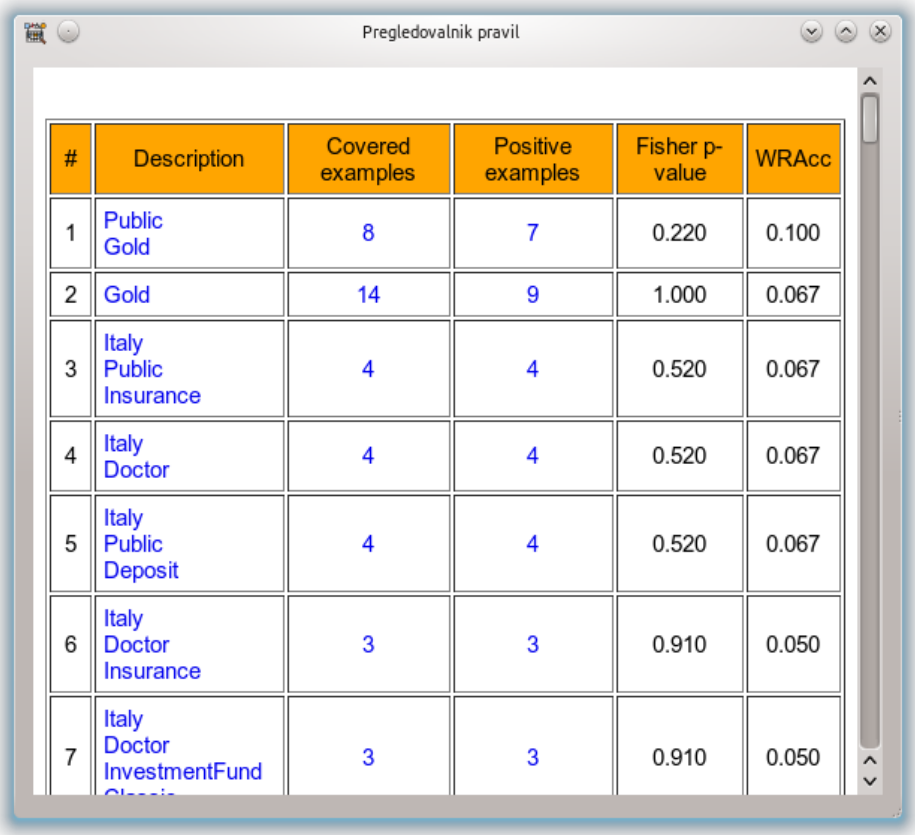
- `File` - prebere datoteko primerov in na izhodni signal pošlje predstavitev v obliki objekta `ExampleTable`,
- `File to String` - prebere vsebino poljubne tekstovne datoteke v znakovno predstavitev; tak gradnik smo uporabili za uvoz ontologij OWL in datoteke s preslikavo,
- `String` - uporabnik v polje vnese poljubno tekstovno vsebino, gradnik pa jo pošlje na izhodni signal; tak gradnik smo uporabili za vnos splošnih konceptov,
- `g-SEGS` - gradnik sistema g-SEGS,
- `Rule Browser` - pregledovalnik pravil, ki jih odkrije sistem g-SEGS.

Ko uporabnik uporabi določen gradnik (npr. prebere datoteko z diska), lahko izhod tistega gradnika poveže na ustrezen vhodni signal gradnika `g-SEGS`. Slika 3.11 prikazuje primer povezanih gradnikov - delotoka.

Uporabnik lahko iskanje dodatno prilagodi z dvojnim klikom na gradnik `g-SEGS`. Preko zaslonske maske (slika 3.7), ki se mu prikaže, lahko izbere ciljni razred (v našem primeru `YES` ali `NO`), najmanjšo velikost podskupin, največje število členov v pravilih in podobno. Ko je uporabnik zadovoljen z nastavitvami, s klikom na `OK` požene izvajanje. Nad ikono gradnika se uporabniku nato izpisuje stanje izvajanja. Ko je iskanje končano, se nad ikono pregledovalnika pojavi oblaček za obvestilo, ki pove, da so rezultati prisotni na signalu. Z dvojnim klikom na ikono pregledovalnika pravil se uporabniku prikaže okno (slika 3.12), kjer lahko pregleduje pravila, število primerov, ki jih to pravilo pokriva, število primerov, ki jih pravilo pokriva in pripadajo ciljnemu razredu ter ocene zgrajenega pravila.



Slika 3.11: Primer delotoka v okolju Orange za uporabo sistema g-SEGS.



#	Description	Covered examples	Positive examples	Fisher p-value	WRAcc
1	Public Gold	8	7	0.220	0.100
2	Gold	14	9	1.000	0.067
3	Italy Public Insurance	4	4	0.520	0.067
4	Italy Doctor	4	4	0.520	0.067
5	Italy Public Deposit	4	4	0.520	0.067
6	Italy Doctor Insurance	3	3	0.910	0.050
7	Italy Doctor InvestmentFund	3	3	0.910	0.050

Slika 3.12: Pregledovalnik pravil.

Poglavje 4

Formulacija problema za sistem Aleph

V diplomskem delu smo problematiko učenja pravil, kjer kot predznanje uporabljamo ontologije, poskusili rešiti tudi s pomočjo sistema za induktivno logično programiranje Aleph. V prvem podpoglavju najprej opišemo uporabo sistema Aleph, v drugem pa predstavimo naš postopek formulacije problema.

4.1 Aleph

Sistem Aleph je prosto-dostopen na spletu[1]. Napisan je v deklarativnem jeziku Prolog in za njegovo uporabo potrebujemo interpreter za Prolog imenovan Yap[2] (Yet Another Prolog) ali SWI Prolog[3]. Aleph je sistem ILP, zato nam ponuja glavni značilnosti takih sistemov: uporabo predikatne logike za zapis hipotez (natančneje: Hornovi stavki) in uporabo predznanja v obliki poljubnih predikatov.

Uporabnik mora za reševanje nekega problema s sistemom Aleph predložiti tri datoteke:

- datoteko s predznanjem (*ang. background knowledge file*),
- datoteko s pozitivnimi primeri (*ang. positive examples file*) in
- datoteko z negativnimi primeri (*ang. negative examples file*).

[1] http://www.comlab.ox.ac.uk/activities/machinelearning/Aleph/aleph.html
[2] http://yap.sourceforge.net/
[3] http://www.swi-prolog.org/

Uporabnik jih nato prebere v spomin s predikatom `read_all/1`. Aleph za iskanje pravil, ki sestavljajo ciljno relacijo, ponuja več ukazov (npr. `induce/0` in `induce_cover/0`), ki ustrezajo različnim načinom iskanja, a oblika rezultatov je v osnovi vedno seznam pravil (*ang. rule list*). Vsako izmed dobljenih pravil lahko interpretiramo tudi kot opis ene podskupine primerov.

4.1.1 Datoteka s predznanjem

V datoteko s predznanjem (s končnico `.b`) uporabnik v programskem jeziku Prolog zakodira znanje, ki je relevantno za dano domeno, in različne nastavitve, ki se tičejo jezika hipotez (npr. največ kolikokrat se lahko določen predikat pojavi v hipotezi) in omejitev glede preiskovanja (npr. najmanjše število primerov, ki jih mora pokrivati hipoteza). Hipoteza je v sistemih ILP logični program, ki definira ciljni predikat s pomočjo predikatov v predznanju.

Uporabnik predznanje zakodira s pomočjo predikatov, ki jih nudi Aleph. Med najpomembnejše sodijo naslednji:

- `modeh(RecallNumber, PredicateMode)` - s pomočjo tega predikata definiramo ciljni predikat. `RecallNumber` pomeni število uspešnih klicev tega predikata (če gre za omejeno nedeterminističen predikat (*ang. bounded non-determinacy*) potem za ta parameter nastavimo `*`), medtem ko `PredicateMode` določi pravilen način za uporabo tega predikata v hipotezah.

- `modeb(RecallNumber, PredicateMode)` - s pomočjo tega predikata definiramo ostale predikate, ki jih ima Aleph na voljo kot predznanje; pomena parametrov sta enaka kot pri `modeh/2`.

- `determination(TargetName/Arity, BackgroundName/Arity)` - s tem predikatom uporabnik določi, kateri predikati `BackgroundName` se lahko pojavijo v telesu ciljnega predikata `TargetName`. Parameter `Arity` označuje mestnost predikata oz. število parametrov, ki jih sprejme.

Parameter `PredicateMode`, ki ga uporabimo v predikatih `modeh` in `modeb`, je oblike `p(ModeType, ModeType, ...)`. Vsak `ModeType` je lahko (a) *enostaven* ali (b) *strukturiran*. Enostavni so oblike `+T` ali `-T`. Prva oblika pove, da naj predikat `p` v hipotezi, v kateri nastopa, dobi *vhodno* spremenljivko tipa `T`. Druga oblika pove, da je spremenljivka tipa `T` *izhodna* spremenljivka predikata `p`. Poznamo še tretjo obliko `#T`, ki pomeni, da naj na tem mestu v predikatu nastopa konstanta tipa `T`.

Strukturiran `ModeType` je oblike `f(...)`, kjer je `f` funkcijski simbol, vsak argument je spet lahko enostaven ali strukturiran `ModeType`. Primer dveh deklaracij predikatov:

```
% Deklaracija predikata z enostavnimi parametri
:- modeb(1, mult(+integer,+integer,-integer)).
% Deklaracija predikata z enostavnim in strukturiranim parametrom
:- modeb(1, mem(+number,[+number|+list])).
```

Poleg teh osnovnih predikatov lahko uporabnik uporabi še različne druge, ki služijo naprednejši rabi sistema. Med te predikate sodijo:

- `prune/1` - definicijo predikata `prune/1`, ki sprejme obravnavano hipotezo kot parameter, napiše uporabnik tako, da je predikat resničen za tiste hipoteze (in njihove specializacije), ki jih lahko sistem pri iskanju preskoči,
- `cost/3` - z definicijo tega predikata lahko uporabnik določi lastno funkcijo za izračun cene dane hipoteze,
- `refine/2` - s pomočjo tega predikata uporabnik določi pravila, kako se določen stavek (*ang. clause*) zostri (*ang. refine*) v drug stavek,
- `set/2` - s predikatom `set/2` uporabnik nastavlja različne nastavitve delovanja sistema.

4.1.2 Datoteki primerov

Vhodne primere ločimo v dve datoteki. V prvi datoteki (s končnico `.f`) navedemo vse pozitivne primere ciljne relacije, v drugi datoteki (s končnico `.n`) pa vse negativne primere ciljne relacije.

Če v datoteki s predznanjem definiramo ciljni predikat `target/1` in instance tipa `obj`:

```
:- modeh(*, target(+obj)).
...
obj(o1).
obj(o2).
obj(o3).
...
```

potem v datoteki navedemo seznam primerov na naslednji način:

```
target(o1).
target(o2).
target(o3).
...
```

kjer za atome `o1`, `o2` in `o3` velja, da so tipa `obj`.

4.2 Postopek formulacije problema

Pri formulaciji predpostavimo naslednjo situacijo: dano imamo neko število ontologij, množico primerov s pripisano oznako oz. rangom ter preslikavo med primeri in koncepti v ontologiji. Dano imamo lahko tudi poljubno število relacij med vhodnimi primeri. Problem formulacije smo rešili tako, da smo vse te vhodne podatke ustrezno pretvorili v predikate jezika Prolog na način, ki ga predpisuje sistem Aleph. V naslednjih podpoglavjih opišemo potrebne korake za pretvorbo vsake izmed vhodnih struktur.

4.2.1 Pretvorba ontologij

Pri pretvorbi ontologij iz jezika OWL upoštevamo osnovno zakodirano znanje: koncepte in relacijo `SubClassOf` med koncepti. Morebitnega kompleksnejšega znanja (npr. hierarhije relacij in lastnosti relacij v OWL Full) opisani postopek ne podpira, saj je prej potrebno še raziskati, kako bi lahko učni algoritem tako znanje sploh uporabil.

Najprej opišimo, kako koncepte, povezane z relacijo `SubClassOf`, definiramo s predikati v Prologu. Vsak koncept v ontologiji preslikamo v unarni predikat, ki je definiran s predikati otrok danega koncepta. Vsak koncept `c` z otroci `c1,c2,...,cm` v Prologu definiramo kot:

```
c(X) :-
    c1(X)
    ;
    c2(X)
    ;
    ...
    ;
    cm(X).
```

Z besedami se zgornja koda prebere: primer `X` pripada konceptu `c`, če pripada vsaj enemu izmed konceptov `c1,c2,..,cm`. Postopek rekurzivno ponovimo za vse `c1,c2,...,cm`. Da zakodiramo celotno ontologijo, postopek začnemo pri korenskem konceptu.

Ker zgornji postopek zgolj definira dane koncepte v obliki Prologovih predikatov, moramo z uporabo predikatov, ki so opisani v poglavju 4.1, določiti še njihovo vlogo in uporabo v okviru sistema Aleph.

Za spremenljivke, ki jih kot vhod dobijo zakodirani koncepti, smo določili splošen tip `instance`, ki predstavlja instanco vhodnega primera. Izbrali smo si tudi splošen ciljni predikat `target/1`, ki ga definiramo s pomočjo predikata `modeh/2`:

```
1 :- modeh(1, target(+instance)).
```

Ukaz pove, da je predikat `target/1` ciljni predikat, ki mora kot vhodno spremenljivko dobiti spremenljivko tipa `instance`.

Za vsak predikat/koncept `c` določimo še:

- `modeh/2` naslednje oblike:

```
1 :- modeb(1, c(+instance)).
```

- `determination/2` naslednje oblike:

```
1 :- determination(target/1, c/1).
```

S tem omogočimo, da se lahko vsak koncept pojavi v telesu pravila.

4.2.2 Pretvorba vhodnih podatkov

Za vsak vhodni primer vemo, s katerimi koncepti iz ontologij je označen, ter razred, ki mu pripada oz. rang, ko gre za rangirane podatke. To znanje za vsak vhodni primer zakodiramo na naslednji način: recimo, da obravnavamo k-ti primer, ki je označen s koncepti `c1, c2, ..., cm`. Ta primer zapišemo kot:

```
1 instance(ik).
2 c1(ik).
3 c2(ik).
4 ...
5 cm(ik).
```

To ponovimo za vsakega izmed vhodnih primerov.

Ker Aleph razlikuje le med pozitivnimi in negativnimi primeri, moramo problem pretvoriti na binarni učni problem. V primeru, da gre za večrazredni problem, primeri označeni z razredom, ki si ga uporabnik izbere kot pozitivnega oz. ciljnega (*ang. target class*), postanejo pozitivni, vsi ostali pa negativni. Glede na to razlikovanje nato posamezni primer ustrezno ločimo v datoteki `.f` in `.n`. Če k-ti primer pripada ciljnemu razredu, potem ga v datoteko `.f` zapišemo kot pozitivni primer predikata `target/1`:

```
1 target(ik).
```

v nasprotnem primeru pa v `.n`, da predstavlja negativni primer predikata `target/1`.

Tudi rangirni problem je potrebno pretvoriti v binarni učni problem. Po zgledu iz [5] rangirane primere najprej uredimo in jih nato glede na nek prag (*ang. threshold*), ki ga izbere uporabnik, razdelimo v dva razreda. Pozitivne primere zatem interpretiramo kot visoko rangirane primere.

4.2.3 Pretvorba dodatnih relacij

Če imamo na voljo še neko tretjo relacijo (kot npr. `interacts`), jo lahko na enostaven način vključimo v domensko predznanje. Ponazorimo pretvorbo na primeru binarne relacije `interacts` med vhodnimi primeri. Dane imamo pare primerov, ki pripadajo tej relaciji. V datoteki s predznanjem ekstenzionalno definiramo (*ang. extensional definition*) relacijo `interacts/2` tako, da navedemo posamezne primere, ki ji pripadajo:

```
1 ...
2 interacts(ik, il).
3 ...
```

Tu sta `ik` in `il` identifikatorja primerov, ki sta v tej relaciji.

Za relacijo moramo definirati še pripadajoči Prologov predikat in določiti, kako ga lahko sistem uporablja v telesu ciljnega predikata. Navesti moramo še:

- `modeh/2` naslednje oblike:

```
1 :- modeb(*, interacts(+instance, -instance)).
```

S tem povemo, da za dani vhodni primer, ki že nastopa nekje v hipotezi, predikat omejeno nedeterministično vrne nek drug vhodni primer.

- `determination/2` naslednje oblike:

```
1 :- determination(target/1, interacts/2).
```

S tem omogočimo, da se predikat lahko pojavi v telesu ciljnega predikata.

4.3 Izboljšava učinkovitosti izvajanja

Med prvimi eksperimenti s sistemom Aleph smo ugotovili, da čas za učenje hitro narašča s številom konceptov v ontologijah. Učenje na primeru ontologije GO zahteva že kar precejšno količino časa (okrog 20 minut za 600 primerov), saj vsebuje okrog 35.000 konceptov. Za problematične se izkažejo predvsem rekurzivne definicije predikatov ontoloških konceptov. Na primer, za ugotavljanje, ali določena instanca pripada danemu predikatu, mora Prologov interpreter poiskati dokaz, ki poteka od (v hierarhiji) najnižjega predikata/koncepta, s katerim je označen obravnavani primer, do obravnavanega predikata.

Opisani problem je moč elegantno rešiti s konceptom *tabeliranja* (*ang. tabling*), ki ga podpira tudi intepreter YAP, če ga prevedemo z zastavico `--enable-tabling`. Osnovna ideja tabeliranja je, da za željene predikate med računanjem shranjujemo njihove odgovore v tabelo tako, da jih lahko uporabimo pri morebitnih naslednjih klicih in s tem preprečimo nepotrebno novo računanje. To se je izkazalo kot primerno za predikate konceptov, saj se tekom preiskovanja izvede veliko enakih klicev rekurzivno definiranih predikatov.

Da tabeliranje lahko uporabimo v YAP Prologu oz. v sistemu Aleph, moramo za vsak predikat, ki ga želimo tabelirati, uporabiti predikat `table/1`. Če želimo tabelirati predikat/koncept `'c'/1`, potem to naredimo na naslednji način:

```
1 :- table c/1.
```

Uporaba tabeliranja sicer povzroči večjo prostorsko kompleksnost, a je zelo koristna, če lahko žrtvujemo prostor zato, da pridobimo na času. Izkaže se, da je čas, ki ga Aleph potrebuje za iskanje pravil na biološki domeni z ontologijo GO, približno 10-krat manjši z uporabo tabeliranja.

Poglavje 5

Eksperimentalni rezultati

V tem poglavju opišemo tri problemske domene in na njih ocenimo uspešnost odkrivanja podskupin sistemov g-SEGS in Aleph. Dosežene rezultate ocenimo z različnimi merami za ocenjevanje podskupin in jih komentiramo.

5.1 Umetna domena: analiza potrošnikov

Umetno domeno smo ustvarili za lažjo ilustracijo tipa problema, ki ga lahko rešimo s sistemom g-SEGS, kot tudi s sistemom Aleph, če uporabimo postopek, predstavljen v poglavju 4.

Za namen ilustracije smo sestavili tri ontologije, ki jih prilagamo v dodatku C, in označene vhodne podatke, ki jih navajamo v dodatku D. Podobne podatke bi lahko zbrala banka, ki jo zanima, kdo izmed njenih strank je zapravljivec (`big_spender = YES`). Recimo, da banka o strankah hrani naslednje podatke:

- poklic (`occupation`),
- kraj bivanja (`location`),
- tip bančnega računa (`account`),
- tip posojila (`loan`),
- tip depozita (`deposit`),
- tip investicijskega fonda (`inv_fund`),
- tip zavarovanja (`insurance`) in

- ali je stranka zapravljiva ali ne (`big_spender`).

Ker ima banka na voljo tudi tri ontologije: ontologijo poklicev, lokacij in bančnih storitev, jih želi uporabiti kot predznanje pri iskanju pravil, ki opisujejo podskupine strank, ki so zapravljive. Primer: z uporabo ontologije lokacij so lahko pravila bolj splošna glede na kraj bivanja strank, saj lahko npr. opisujejo stranke iz Nemčije, ker v ontologiji obstaja koncept Nemčija. Običajni sistemi, ki iščejo pravila, bi v pravila vključili le lokacije, ki so prisotne v vhodnih podatkih. Taka pravila so seveda bolj specifična, saj lahko govorijo npr. le o strankah iz konkretnih nemških mest.

Vhodne primere smo sestavili tako, da smo vanje vključili nekaj zakonitosti, ki jih lahko ustrezno opišemo le s koncepti iz ontologij, ki v našem primeru določajo splošnejše pojme kot vrednosti atributov v vhodni tabeli. Podatki vsebujejo zakonitosti, kot so:

- zapravljivci so zdravniki ali zaposlenci iz storitvenih dejavnosti iz Italije ali Nemčije,
- zapravljivci so stranke z zlatimi bančnimi računi, ki so zaposleni v javnem sektorju,
- bolj verjetno je, da so zapravljivci ljudje z investicijami, depoziti ali zavarovanji.

5.2 Analiza podatkov mikromrež

V tem podpoglavju predstavimo dve domeni bioloških podatkov, ki so bili pridobljenih s tehnologijo mikromrež: *acute lymphoblastic leukemia* (ALL) [19] ter *human scenescence mesenchymal stem cells* (hMSC) [20]. Opišemo tudi, kako so bili podatki predprocesirani, preden smo lahko na njih uporabili sistema g-SEGS in Aleph.

5.2.1 Opis domen

Obe domeni opisujeta, kako se izraženost posameznih genov spreminja glede na različne vzorce mikromrež. Podatki so prosto dostopni na spletni strani orodja za analizo mikromrež SegMine[1].

V primeru domene ALL imamo podatke kliničnega testiranja, kjer gre za opis izraženosti večjega števila genov za različne paciente, ki so razdeljeni v dva

[1] http://segmine.ijs.si/

razreda glede na njihov limfocitni podtip B ali T. V postopku predprocesiranja smo za pozitivni razred vzeli primere s podtipom T. Izraženost je bila izmerjena za 12.625 genov na 128 vzorcih.

V primeru domene hMSC podatki opisujejo izraženost genov treh pacientov v dveh točkah v času - kar nam da dva razreda. Prvi razred so meritve, opravljene v zgodnjem stanju (*ang. early senescent passage*), drugi razred pa meritve, opravljene v poznem stanju (*ang. late senescent passage*). V postopku predprocesiranja smo za pozitivni razred vzeli kasnejše meritve. Izraženost je bila izmerjena za 41.930 genov na skupaj 6 vzorcih (3 zgodnje meritve in 3 pozne meritve).

5.2.2 Predprocesiranje

Opravljeno predprocesiranje je sledilo postopku, ki ga predlaga metodologija SegMine [21]. Gene smo najprej rangirali s pomočjo algoritma ReliefF [22], ki vsak gen postavi v vlogo atributa in oceni njegovo kvaliteto; gene smo nato še filtrirali glede na mero $logFC$ (*ang. logarithm of expression fold change*).

Mero $logFC$ za gen g izračunamo kot:

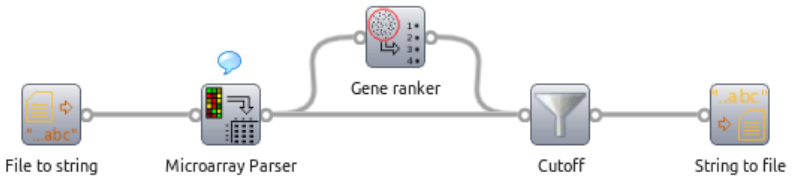
$$logFC(g) = \log_2 \frac{\sum \text{izraženost gena } g \text{ pri negativnem razredu}}{\sum \text{izraženost gena } g \text{ pri pozitivnem razredu}} \tag{5.1}$$

Glede na metodologijo smo izločili vse gene z $|logFC| < 0.3$, saj za take gene velja, da se njihova izraženost premalo spreminja, da bi bili zanimivi za nadaljno analizo.

Predprocesiranje smo opravili v skladu z metodologijo SegMine, implementirano v okolju Orange4WS. Korake predprocesiranja prikažemo z delotokom na sliki 5.1. Podatke smo najprej s pomočjo gradnika `Microarray Parser` pretvorili v primerno obliko in izračunali vrednosti $logFC$. Gene smo nato rangirali z gradnikom `Gene ranker` in jih z uporabo gradnika `Cutoff` filtrirali glede na vrednosti $logFC$. Na koncu smo tako dobljeni seznam rangiranih genov le še zapisali v datoteko za nadaljno uporabo. Predprocesiranje smo opravili za vsako izmed domen posebej. Tako dobljen seznam genov z rangi, skupaj z ontologijo GO in podatki o interakcijah ter s preslikavo med geni in ontološkimi koncepti[2], smo nato uporabili za analizo s pomočjo sistemov g-SEGS in Aleph.

Na primeru ALL nam je po opravljenem predprocesiranju ostalo 9.001 genov, na primeru hMSC pa 20.326.

[2]Datoteke so dostopne na http://www.gomapman.org/import/segs/hsa/current/.



Slika 5.1: Delotok, ki filtrira in rangira gene iz vhodnih podatkov.

5.3 Ocenjevanje in rezultati

V tem podpoglavju najprej predstavimo kriterije, ki smo jih uporabili za ocenjevanje, nato pa še dobljene rezultate.

5.3.1 Kriteriji za ocenjevanje

Vse eksperimente smo opravili na računalniku z 8-jedrnim Core-i7 2.93GHz procesorjem, z 8GB pomnilnika na 64-bitnem sistemu Linux Kubuntu. Sistem g-SEGS smo, tako kot Aleph, uporabljali v skriptnem načinu. Odločili smo se, da v nabor mer za oceno uspešnosti sistemov vključimo tudi čas izvajanja. Pri meritvah časa smo merili čas od izdaje ukaza za pričetek postopka odkrivanja podskupin (čas branja vhodnih podatkov ni vključen) do trenutka, ko je sistem končal z delom.

Za evalvacijo dobljenih podskupin smo se odločili uporabiti mere, ki se osredotočajo na kvaliteto opisnih pravil (*ang. descriptive measures*), ki jih predlagajo avtorji v [17]. Predlagane mere so dovolj splošne, da jih uporabimo na različnih domenah, vendar bi si v praksi za oceno delovanja seveda vseeno želeli pridobiti mnenje domenskega eksperta, kar bo predmet nadaljnjega dela.

Uporabili smo naslednje mere:

- *AvgCov* - povprečen delež pozitivnih primerov, ki jih pokriva pravilo; mero izračunamo z enačbo 5.2,
- *AvgSup* - odstotek pozitivnih primerov, ki jih pokriva celotna množica pravil; mero izračunamo z enačbo 5.3,
- *AvgSig* - povprečna signifikantnost (*ang. significance*) pravil; mero izračunamo po enačbi 5.4,

- *AvgWRAcc* - povprečna nenavadnost (*ang. unusualness*) pravil, ki jo izračunamo po enačbi 5.5 ter
- *AUC* - površina pod krivuljo (*ang. Area Under Curve*), ki povezuje najboljša pravila v prostoru ROC (*ang. Receiver Operating Characteristic*). Uporabili smo metodo, ki jo avtorji v [17] imenujejo AUC-1. V prostor ROC narišemo vsa pravila tako, da je vsako pravilo $Class \leftarrow Cnd$ predstavljeno s točko (FPr, TPr), kjer je $FPr = \frac{n(\overline{Class} \wedge Cnd)}{n(\overline{Class})}$ in $TPr = \frac{n(Class \wedge Cnd)}{n(Class)}$. Točke, ki so najbolj oddaljene od diagonale, nato povežemo (skupaj tvorijo t.i. konveksno lupino (*ang. convex hull*)) in izračunamo površino pod to krivuljo.

$$AvgCov = \frac{1}{n_R} \sum_{i=1}^{n_R} \frac{n(Class \wedge Cnd_i)}{n(Cnd_i)} \tag{5.2}$$

$$AvgSup = \frac{1}{n(Class)} \cdot n(Class \bigvee_{Class \leftarrow Cnd_i} Cnd_i) \tag{5.3}$$

$$AvgSig = 2 \sum_j n(Class_j \wedge Cnd_i) \cdot \log \frac{n(Class_j \wedge Cnd_i)}{n(Class_j) \cdot p(Cnd_i)} \tag{5.4}$$

$$AvgWRAcc = \frac{1}{n_R} \sum_{i=1}^{n_R} WRAcc(R_i) \tag{5.5}$$

V zgornjih enačbah n_R označuje moč množice odkritih pravil, R_i pa i-to pravilo.

5.3.2 Rezultati

Vrednost domene strank je predvsem v tem, da vnaprej vemo, kakšne zakonitosti so zakodirane v vhodnih primerih. Z razliko od obeh bioloških domen lahko na tem mestu analiziramo tudi posamezna odkrita pravila, saj lahko nastopamo v vlogi domenskega eksperta.

Sliki 5.2 in 5.3 prikazujeta nekaj pravil, ki jih sistema odkrijeta na domeni strank. Pravila navajamo v notaciji, ki jo sistema uporabljata za izpis pravil. Vidimo, da v splošnem uspešno odkrijeta zakonitosti, ki smo jih zakodirali v vhodne podatke. Npr. pravilo, ki opisuje stranke iz javnega sektorja (člen oz. konjunkt `Public`) z zlatim računom (`Gold`) se pojavi v obeh primerih. V obeh seznamih najdemo tudi zdravnike iz Italije ter Nemce, ki so zaposleni v storitvenih dejavnostih (`Service`).

```
Class YES:
  Public, Gold.

  Italy, Public, Insurance.

  Italy, Doctor.

  Bavaria, Gold.

  Germany, Service, InvestmentFund.
```

Slika 5.2: Primeri pravil, ki jih odkrije sistem g-SEGS na domeni potrošnikov.

```
target(A) :-
   'Doctor'(A), 'Italy'(A).

target(A) :-
   'Public'(A), 'Gold'(A).

target(A) :-
   'Poland'(A), 'Deposit'(A), 'Gold'(A).

target(A) :-
   'Germany'(A), 'Insurance'(A).

target(A) :-
   'Service'(A), 'Germany'(A).
```

Slika 5.3: Primeri pravil, ki jih odkrije sistem Aleph na domeni potrošnikov.

Tabela 5.1 prikazuje ocene za množici odkritih pravil na domeni potrošnikov. Odebeljene vrednosti pomenijo boljšo izmed obeh vrednosti. Vidimo, da na tej domeni oba algoritma dosežeta enak AUC ter s pravili pokrijeta vse pozitivne primere. Delež primerov, ki jih pokrije posamezno pravilo, je večji pri Alephu. Enako velja tudi za signifikantnost odkritih pravil ter povprečno nenavadnost, vendar pa na tej domeni g-SEGS potrebuje manj časa za izračun.

Sistem	$AvgCov$	$AvgSup$	$AvgSig$	$AvgWRAcc$	AUC	t[s]
g-SEGS	0.149	1.000	3.101	0.049	**0.709**	**0.00**
Aleph	**0.161**	1.000	**3.278**	**0.058**	**0.709**	0.12

Tabela 5.1: Eksperimentalni rezultati domene potrošnikov.

Na obeh bioloških domenah smo za pozitivni razred vzeli zgornjih 300 primerov. V primeru g-SEGS to pomeni, da nastavimo parameter `Cutoff` na 300 (ostale primere sistem obravnava kot negativne). Medtem ko smo pri

Alephu zgornjih 300 genov označili za pozitivne primere, izmed ostalih pa smo naključno izbrali 300 dodatnih primerov, ki smo jih označili kot negativne. Razliko v porazdelitvi primerov med obema sistemoma odraža predvsem mera signifikantnosti. Dolžino pravil smo v obeh sistemih omejili na 4 člene, najmanjše število pokritih primerov na 20, največje število pravil pa na 100.

Tabela 5.2 prikazuje ocene za domeno ALL. Na tej domeni se Aleph precej bolje odreže pri vseh merah, razen pri signifikantnosti in AUC. Večjo signifikantnost pravil sistema g-SEGS lahko razložimo s prej omenjenim dejstvom, da pri učenju upošteva znatno večje število negativnih primerov. Za končnega uporabnika je zanimivo tudi to, da za iskanje Aleph porabi približno 4-krat manj časa.

Sistem	$AvgCov$	$AvgSup$	$AvgSig$	$AvgWRAcc$	AUC	t[min]
g-SEGS	0.024	0.770	**15.214**	0.0012	**0.573**	11.25
Aleph	**0.109**	**0.967**	3.964	**0.0122**	0.549	**2.59**

Tabela 5.2: Eksperimentalni rezultati domene ALL.

Sistem	$AvgCov$	$AvgSup$	$AvgSig$	$AvgWRAcc$	AUC	t[min]
g-SEGS	0.023	0.700	**10.427**	0.0005	0.563	10.75
Aleph	**0.105**	**0.963**	7.391	**0.0157**	**0.596**	**2.72**

Tabela 5.3: Eksperimentalni rezultati domene hMSC.

Tabela 5.3 prikazuje ocene sistemov za domeno hMSC. Tudi na tej domeni dobi Aleph opazno boljše ocene pri vseh merah, razen pri signifikantnosti, kar spet lahko razložimo s prej omenjenim dejstvom.

Pri obeh bioloških domenah lahko opazimo še, da pravilo, ki ga odkrije Aleph, v povprečju pokrije večji delež pozitivnih primerov (na obeh domenah okrog 10%, v primerjavi z dobrimi 2% sistema g-SEGS). Opazimo tudi, da množica pravil sistema Aleph pokrije večji delež vseh pozitivnih primerov. To je razumljivo, saj Aleph potrebuje manj pravil, da pokrije enako število primerov, in lahko se zgodi, da zaradi omejitve števila pravil g-SEGS pokrije manj pozitivnih primerov kot bi jih, če bi ta parameter nastavili na večjo vrednost.

Ilustrativne so tudi slike podskupin v prostoru ROC, ki jih navajamo v dodatku E. Iz narisanih krivulj lahko pridemo do podobnih ugotovitev kot v prejšnjem odstavku. Hitro je razvidno, da posamezna pravila sistema Aleph v povprečju pokrijejo večji prostor primerov, saj imajo večji TPr in so zato

v večini narisana višje v smeri ordinate. Naslednja zanimivost, ki nam jo razkrijejo krivulje ROC je tudi, da g-SEGS najde nekaj pravil (glej krivulji za ALL in hMSC), ki precej odstopajo od povprečja in pokrivajo veliko večji delež pozitivnih primerov - v teh dveh primerih tudi do okrog 30% vseh pozitivnih primerov, vendar je cena za to tudi večje število pokritih negativnih primerov, kar nakazuje to, da taka pravila izstopajo tudi v smeri abscise.

Iz rezultatov lahko povzamemo naslednje:

- g-SEGS zaradi upoštevanja večjega števila negativnih primerov dosega boljše rezultate glede na mero povprečne signifikantnosti odkritih pravil,
- pravilo, ki ga odkrije Aleph, v povprečju pokrije več pozitivnih primerov,
- množica pravil, ki jih odkrije Aleph, pokrije večji delež vseh pozitivnih primerov in doseže večjo povprečno nenavadnost in AUC ter
- Aleph odkrije pravila na obeh domenah približno 4-krat hitreje.

Pozorni moramo biti na dejstvo, da ni nujno, da skupine pravil, ki so bolje ocenjene glede na zgornje mere, v resnici tudi skrivajo koristne in nove informacije za domenskega eksperta. Analiza s pomočjo domenskega eksperta je načrtovana za nadaljnje delo.

Poglavje 6

Sklepne ugotovitve in nadaljnje delo

V diplomskem delu je opisan razvoj aplikacije za odkrivanje podskupin g-SEGS, ki omogoča uporabo ontologij kot predznanja pri iskanju opisov podskupin. Aplikacija dane ontološke koncepte uporablja kot člene v pravilih, ki opisujejo podskupine primerov. Sistem g-SEGS je direktna posplošitev sistema SEGS, ki je bil že uspešno uporabljen na področju analize mikromrež, vendar ima to slabost, da je z uporabo ontologije genov GO omejen zgolj na analizo bioloških podatkov.

Sistem je bil implementiran v obliki spletnega servisa, ki ga lahko uporabnik uvozi v različne aplikacije za delo s spletnimi servisi. V delu je opisana uporaba razvitega sistema v okviru okolja za podatkovno rudarjenje Orange oz. Orange4WS, za katerega je bil razvit tudi poseben uporabniški vmesnik za lažjo uporabo. Gradnik sistema g-SEGS in že obstoječe gradnike okolja Orange4WS lahko uporabnik povezuje v delotoke. S pomočjo teh gradnikov uporabnik naloži podatke, jih predprocesira, nastavi željene parametre in na ta način z vizualnim programiranjem pride do rešitve problema.

Diplomsko delo opiše tudi, kako lahko nalogo iskanja podskupin, kjer želimo za predznanje uporabiti ontologije, formuliramo tudi za sistem induktivnega logičnega programiranja Aleph.

Na koncu diplomsko delo eksperimentalno oceni delovanje obeh pristopov na umetni domeni in dveh resničnih bioloških domenah.

Prispevki te diplomske naloge so relevantni iz več razlogov. Vse več skupnosti se odloča za formalizacijo znanja neke domene v obliki prosto dostopnih ontologij. Ontologija tako predstavlja kos konsenzualnega, preverjenega znanja o neki domeni in je na ta način lahko koristna tudi kot predznanje pri

rudarjenju podatkov. Če znamo podatke povezati z ustreznimi ontološkimi koncepti, potem ti podatki dobijo predpisan pomen in take podatke lahko algoritem samodejno predprocesira ter jih povezuje z drugimi podatkovnimi bazami. V tem smislu je g-SEGS le začetni korak v tej smeri, saj s pomočjo ontoloških konceptov zgolj združuje koncepte, ki jih predstavljajo vrednosti atributov, v višje, splošnejše koncepte in še ne izkorišča vse moči, ki jo nudijo ontologije.

Pomemben prispevek je tudi implementacija v obliki spletnega servisa. Sistem na ta način ne obremenjuje uporabnikovega sistema in omogoča porazdeljeno izvajanje posameznih gradnikov v delotokih. S pomočjo okolja Orange4WS in posebej razvitega uporabniškega vmesnika lahko uporabnik s pomočjo sistema g-SEGS probleme rešuje tudi na enostaven način.

Za nadaljnje delo so pomembne tudi ugotovitve eksperimentalnega dela diplomske naloge. Iz evalvacije sledi, da je sistem Aleph zelo uspešen pri reševanju tovrstnih nalog in se na podlagi različnih mer za ocenjevanje odkritih podskupin izkaže znatno bolje kot g-SEGS. Pri razvoju morebitnega novega in naprednejšega sistema, ki bi znal izkoriščati ontološko znanje, bo vsekakor nujno potrebno raziskati, kateri principi delovanja sistema Aleph so ključni za to uspešnost, jih na primeren način uporabiti in morda tudi izboljšati za ta specifični problem. Za nadaljnje delo načrtujemo še podrobnejšo analizo odkritih pravil na bioloških domenah s pomočjo domenskega eksperta, saj želimo dobiti pogled na kvaliteto odkritih pravil tudi s tega, najpomembnejšega zornega kota. Vredno bi bilo tudi preizkusiti, ali lahko s takim filtriranjem pravil, kot smo ga uporabili pri sistemu g-SEGS, kaj pridobimo tudi pri sistemu Aleph. Med nadaljnje delo sodi še vključitev možnosti uporabe atributov oz. vrednosti atributov, ki jih ne moremo preslikati v ontološke koncepte, saj tega trenutna implementacija ne omogoča.

Diplomska naloga tako predstavlja tudi nekaj relevantnih ugotovitev, ki jih je potrebno upoštevati pri nadaljnjem raziskovalnem delu na temo uporabe ontologij kot predznanja v podatkovnem rudarjenju.

Literatura

[1] U. Fayyad, G. Piatetsky-Shapiro, and P. Smyth, "From data mining to knowledge discovery in databases," *Ai Magazine*, vol. 17, pp. 37–54, 1996.

[2] S. Džeroski and N. Lavrač, eds., *Relational Data Mining*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1st ed., 2001.

[3] W. Klösgen, "Explora: A multipattern and multistrategy discovery assistant," *Advances in Knowledge Discovery and Data Mining*, pp. 249–271, 1996.

[4] S. Wrobel, "An algorithm for multi-relational discovery of subgroups," in *Proceedings of the 1st European Conference on Principles of Data Mining and Knowledge Discovery (PKDD-97)*, pp. 78–87, 1997.

[5] I. Trajkovski, *Functional Interpretation of Gene Expression Data*. PhD thesis, Jožef Stefan International Postgraduate School, 2007.

[6] I. Trajkovski, N. Lavrač, and J. Tolar, "SEGS: Search for enriched gene sets in microarray data," *Journal of Biomedical Informatics*, vol. 41, no. 4, pp. 588–601, 2008.

[7] V. Podpečan, M. Juršič, M. Žakova, and N. Lavrač, "Towards a service-oriented knowledge discovery platform," in *Third-generation data mining: towards service-oriented knowledge discovery* (V. Podpečan and N. Lavrač, eds.), pp. 25–36, 2009.

[8] J. Demšar, B. Zupan, and G. Leban, "Orange: From experimental machine learning to interactive data mining, white paper (www.ailab.si/orange)." Fakulteta za računalništvo in informatiko, Univerza v Ljubljani, 2004.

[9] S. Wrobel, "An algorithm for multi-relational discovery of subgroups," pp. 78–87, Springer, 1997.

[10] F. Železný and N. Lavrač, "Propositionalization-based relational subgroup discovery with RSD," *Machine Learning*, vol. 62, pp. 33–63, 2006.

[11] "Microarrays: Chipping away at the mysteries of science and medicine." Obiskano 11. maja 2011, `http://www.ncbi.nlm.nih.gov/About/primer/microarrays.html`.

[12] T. G. O. Consortium, "Gene ontology: tool for the unification of biology," *Nature Genetics*, May 2000.

[13] J. Kranjc, "Izdelava spletne aplikacije za strojno učenje," 2010. Diplomsko delo, Fakulteta za računalništvo in informatiko, Univerza v Ljubljani.

[14] R. Michalski, "A theory and methodology of inductive learning," in *Machine Learning: An artificial intelligence approach* (R. Michalski, J. Carbonell, and T. Mitchell, eds.), pp. 83–129, Palo Alto: Tioga Publishing Company, 1983.

[15] J. Aronis, F. Provost, and B. Buchanan, "Exploiting background knowledge in automated discovery," in *Proc. of the 2nd International Conference on Knowledge Discovery and Data Mining*, pp. 355–358, 1996.

[16] G. Garriga, A. Ukkonen, and H. Mannila, "Feature selection in taxonomies with applications to paleontology," in *Proceedings of the 11th International Conference on Discovery Science*, DS '08, (Berlin, Heidelberg), pp. 112–123, Springer-Verlag, 2008.

[17] N. Lavrač, B. Kavšek, P. Flach, and L. Todorovski, "Subgroup discovery with CN2-SD," *Journal of Machine Learning Research*, vol. 5, pp. 153–188, 2004.

[18] B. Kavšek and N. Lavrač, "APRIORI-SD: Adapting association rule learning to subgroup discovery," *Applied Artificial Intelligence*, vol. 20, no. 7, pp. 543–583, 2006.

[19] S. Chiaretti, X. Li, R. Gentleman, A. Vitale, M. Vignetti, F. Mandelli, J. Ritz, and R. Foa, "Gene expression profile of adult t-cell acute lymphocytic leukemia identifies distinct subsets of patients with different response to therapy and survival," *Blood*, vol. 103, pp. 2771–2778, 2004.

[20] W. Wagner, P. Horn, M. Castoldi, A. Diehlmann, S. Bork, R. Saffrich, V. Benes, J. Blake, S. Pfister, V. Eckstein, and A. D. Ho, "Replicative

senescence of mesenchymal stem cells: A continuous and organized process," *PLoS ONE*, vol. 3, p. e2213, 05 2008.

[21] V. Podpečan, N. Lavrač, I. Mozetič, P. K. Novak, I. Trajkovski, L. Langohr, K. Kulovesi, H. Toivonen, M. Petek, H. Motaln, and K. Gruden, "SegMine workflows for semantic microarray data analysis in Orange4WS," *v pripravi*.

[22] M. Robnik-Šikonja and I. Kononenko, "Theoretical and empirical analysis of ReliefF and RReliefF," *Machine Learning*, vol. 53, pp. 23–69, October 2003.

Dodatek A

Primer datoteke OWL

V nadaljevanju je prikazana ontologija s slike 1.3 v zapisu OWL/XML.

Primer OWL/XML dokumenta

```
<?xml version="1.0"?>

<!DOCTYPE rdf:RDF [
    <!ENTITY owl "http://www.w3.org/2002/07/owl#" >
    <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
    <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
    <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
    <!ENTITY knjige "http://www.semanticweb.org/ontologies/2011/4/knjige.owl#" >
]>

<rdf:RDF xmlns="http://www.semanticweb.org/ontologies/2011/4/knjige.owl#"
     xml:base="http://www.semanticweb.org/ontologies/2011/4/knjige.owl"
     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
     xmlns:owl="http://www.w3.org/2002/07/owl#"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:knjige="http://www.semanticweb.org/ontologies/2011/4/knjige.owl#">
    <owl:Ontology rdf:about="http://www.semanticweb.org/ontologies/2011/4/knjige.owl"/>

    <owl:AnnotationProperty rdf:about="&rdfs;label"/>

    <owl:ObjectProperty rdf:about="&knjige;partOf">
        <rdfs:label>partOf</rdfs:label>
        <rdfs:domain rdf:resource="&owl;Thing"/>
        <rdfs:range rdf:resource="&owl;Thing"/>
    </owl:ObjectProperty>

    <owl:Class rdf:about="&knjige;Avtor">
        <rdfs:label>Avtor</rdfs:label>
        <rdfs:subClassOf rdf:resource="&owl;Thing"/>
        <rdfs:subClassOf>
            <owl:Restriction>
                <owl:onProperty rdf:resource="&knjige;partOf"/>
                <owl:someValuesFrom>
                    <owl:Class>
                        <owl:unionOf rdf:parseType="Collection">
                            <rdf:Description rdf:about="&knjige;Knjiga"/>
```

```
38                          <rdf:Description rdf:about="&knjige;Recenzija"/>
39                      </owl:unionOf>
40                  </owl:Class>
41              </owl:someValuesFrom>
42          </owl:Restriction>
43      </rdfs:subClassOf>
44  </owl:Class>
45
46  <owl:Class rdf:about="&knjige;Biografija">
47      <rdfs:label>Biografija</rdfs:label>
48      <rdfs:subClassOf rdf:resource="&knjige;Knjiga"/>
49  </owl:Class>
50
51  <owl:Class rdf:about="&knjige;Datum">
52      <rdfs:label>Datum</rdfs:label>
53      <rdfs:subClassOf rdf:resource="&owl;Thing"/>
54      <rdfs:subClassOf>
55          <owl:Restriction>
56              <owl:onProperty rdf:resource="&knjige;partOf"/>
57              <owl:someValuesFrom rdf:resource="&knjige;Knjiga"/>
58          </owl:Restriction>
59      </rdfs:subClassOf>
60  </owl:Class>
61
62  <owl:Class rdf:about="&knjige;Jezik">
63      <rdfs:label>Jezik</rdfs:label>
64      <rdfs:subClassOf rdf:resource="&owl;Thing"/>
65      <rdfs:subClassOf>
66          <owl:Restriction>
67              <owl:onProperty rdf:resource="&knjige;partOf"/>
68              <owl:someValuesFrom rdf:resource="&knjige;Knjiga"/>
69          </owl:Restriction>
70      </rdfs:subClassOf>
71  </owl:Class>
72
73  <owl:Class rdf:about="&knjige;Knjiga">
74      <rdfs:label>Knjiga</rdfs:label>
75      <rdfs:subClassOf rdf:resource="&owl;Thing"/>
76  </owl:Class>
77
78  <owl:Class rdf:about="&knjige;Naslov">
79      <rdfs:label>Naslov</rdfs:label>
80      <rdfs:subClassOf rdf:resource="&owl;Thing"/>
81      <rdfs:subClassOf>
82          <owl:Restriction>
83              <owl:onProperty rdf:resource="&knjige;partOf"/>
84              <owl:someValuesFrom>
85                  <owl:Class>
86                      <owl:unionOf rdf:parseType="Collection">
87                          <rdf:Description rdf:about="&knjige;Knjiga"/>
88                          <rdf:Description rdf:about="&knjige;Recenzija"/>
89                      </owl:unionOf>
90                  </owl:Class>
91              </owl:someValuesFrom>
92          </owl:Restriction>
93      </rdfs:subClassOf>
94  </owl:Class>
95
96  <owl:Class rdf:about="&knjige;Recenzija">
```

```
97          <rdfs:label>Recenzija</rdfs:label>
98          <rdfs:subClassOf rdf:resource="&owl;Thing"/>
99      </owl:Class>
100
101     <owl:Class rdf:about="&knjige;Roman">
102         <rdfs:label>Roman</rdfs:label>
103         <rdfs:subClassOf rdf:resource="&knjige;Knjiga"/>
104     </owl:Class>
105
106     <owl:Class rdf:about="&knjige;Založnik">
107         <rdfs:label>Založnik</rdfs:label>
108         <rdfs:subClassOf rdf:resource="&owl;Thing"/>
109         <rdfs:subClassOf>
110             <owl:Restriction>
111                 <owl:onProperty rdf:resource="&knjige;partOf"/>
112                 <owl:someValuesFrom rdf:resource="&knjige;Knjiga"/>
113             </owl:Restriction>
114         </rdfs:subClassOf>
115     </owl:Class>
116
117     <owl:Class rdf:about="&knjige;ZbirkaNovel">
118         <rdfs:label>ZbirkaNovel</rdfs:label>
119         <rdfs:subClassOf rdf:resource="&knjige;Knjiga"/>
120     </owl:Class>
121
122     <owl:Class rdf:about="&owl;Thing"/>
123
124     <rdf:Description>
125         <rdf:type rdf:resource="&owl;AllDisjointClasses"/>
126         <owl:members rdf:parseType="Collection">
127             <rdf:Description rdf:about="&knjige;Biografija"/>
128             <rdf:Description rdf:about="&knjige;Roman"/>
129             <rdf:Description rdf:about="&knjige;ZbirkaNovel"/>
130         </owl:members>
131     </rdf:Description>
132     <rdf:Description>
133         <rdf:type rdf:resource="&owl;AllDisjointClasses"/>
134         <owl:members rdf:parseType="Collection">
135             <rdf:Description rdf:about="&knjige;Avtor"/>
136             <rdf:Description rdf:about="&knjige;Datum"/>
137             <rdf:Description rdf:about="&knjige;Jezik"/>
138             <rdf:Description rdf:about="&knjige;Knjiga"/>
139             <rdf:Description rdf:about="&knjige;Naslov"/>
140             <rdf:Description rdf:about="&knjige;Recenzija"/>
141             <rdf:Description rdf:about="&knjige;Založnik"/>
142         </owl:members>
143     </rdf:Description>
144 </rdf:RDF>
```

Dodatek B

Definicija izhodnega tipa spletnega servisa

Spodnji odsek datoteke WSDL spletnega servisa g-SEGS prikazuje definicijo kompleksnega podatkovnega tipa `ruleComplexType`, ki je element sporočila `getResultResponse`, ki ga vrne operacija `getResult`.

Primer OWL/XML dokumenta

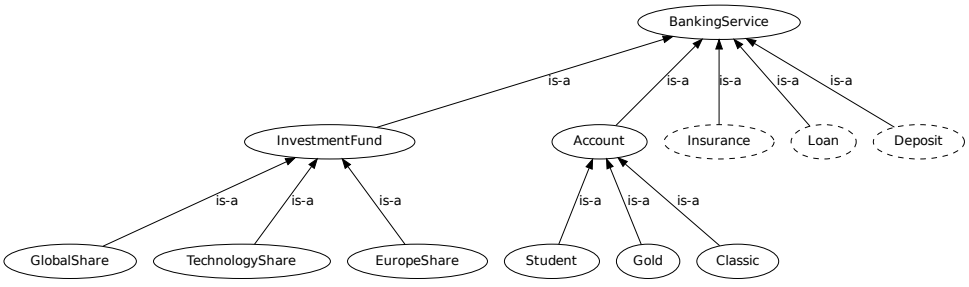
```
<xsd:complexType name="ruleComplexType">
    <xsd:sequence>
        <xsd:element name="description" type="tns:descriptionComplexType" maxOccurs="1"
            minOccurs="1">
                  <xsd:annotation>
                      <xsd:documentation>This is the actual rule. It is composed of two
                      parts: rule conjuncts and interacting terms.</xsd:documentation>
                  </xsd:annotation>
        </xsd:element>
        <xsd:element name="coveredGenes" type="xsd:string" maxOccurs="unbounded"
            minOccurs="1">
            <xsd:annotation>
                <xsd:documentation>This parameter is a list of all genes from the input data
                which are covered by the rule.</xsd:documentation>
            </xsd:annotation>
        </xsd:element>
        <xsd:element name="coveredTopGenes" type="xsd:string" maxOccurs="unbounded"
            minOccurs="1">
            <xsd:annotation>
                <xsd:documentation>This parameter is a list of top ranked genes from the
                input data which are covered by the rule.</xsd:documentation>
            </xsd:annotation>
        </xsd:element>
        <xsd:element name="fisher_pval" type="xsd:double" maxOccurs="1" minOccurs="1">
            <xsd:annotation>
                <xsd:documentation>This is the p-value of the rule by Fisher test.
                </xsd:documentation>
            </xsd:annotation>
        </xsd:element>
        <xsd:element name="GSEA_pval" type="xsd:double" maxOccurs="1" minOccurs="1">
            <xsd:annotation>
```

```
32                <xsd:documentation>This is the p-value of the rule by GSEA test.
33                </xsd:documentation>
34            </xsd:annotation>
35        </xsd:element>
36        <xsd:element name="PAGE_pval" type="xsd:double" maxOccurs="1" minOccurs="1">
37            <xsd:annotation>
38                <xsd:documentation>This is the p-value of the rule by PAGE test.
39                </xsd:documentation>
40            </xsd:annotation>
41        </xsd:element>
42        <xsd:element name="aggregate_pval" type="xsd:double" maxOccurs="1" minOccurs="0">
43            <xsd:annotation>
44                <xsd:documentation>This is the aggregate p-value, computed by using user
45                defined weights on p-values of other tests. This aggregate p-value is not
46                p-value in classical sense but is only used to find genes that have small
47                p-value on several tests. This parameter is present only for combined rules.
48            </xsd:documentation>
49            </xsd:annotation>
50        </xsd:element>
51        <xsd:element name="wracc" type="xsd:double" maxOccurs="1" minOccurs="1">
52            <xsd:annotation>
53                <xsd:documentation>WRAcc score.</xsd:documentation>
54            </xsd:annotation>
55        </xsd:element>
56    </xsd:sequence>
57</xsd:complexType>
58
59<xsd:complexType name="descriptionComplexType">
60    <xsd:sequence>
61        <xsd:element name="terms" type="tns:termComplexType" maxOccurs="unbounded"
62            minOccurs="0">
63        </xsd:element>
64        <xsd:element name="interactingTerms" type="tns:termComplexType" maxOccurs="unbounded"
65            minOccurs="0">
66        </xsd:element>
67    </xsd:sequence>
68</xsd:complexType>
69
70<xsd:complexType name="termComplexType">
71    <xsd:sequence>
72        <xsd:element name="termID" type="xsd:string" maxOccurs="1" minOccurs="1">
73        </xsd:element>
74        <xsd:element name="name" type="xsd:string" maxOccurs="1" minOccurs="1">
75        </xsd:element>
76        <xsd:element name="domain" type="xsd:string" maxOccurs="1" minOccurs="1">
77        </xsd:element>
78    </xsd:sequence>
79</xsd:complexType>
```

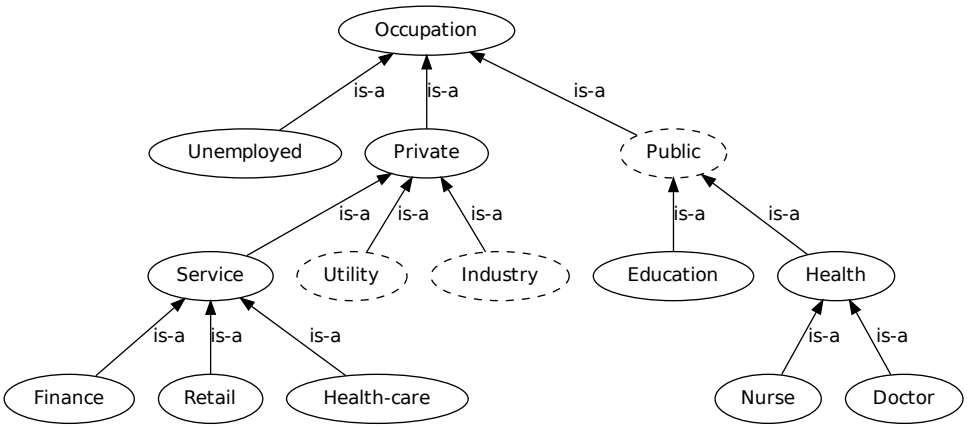
Dodatek C

Primeri ontologij umetne domene

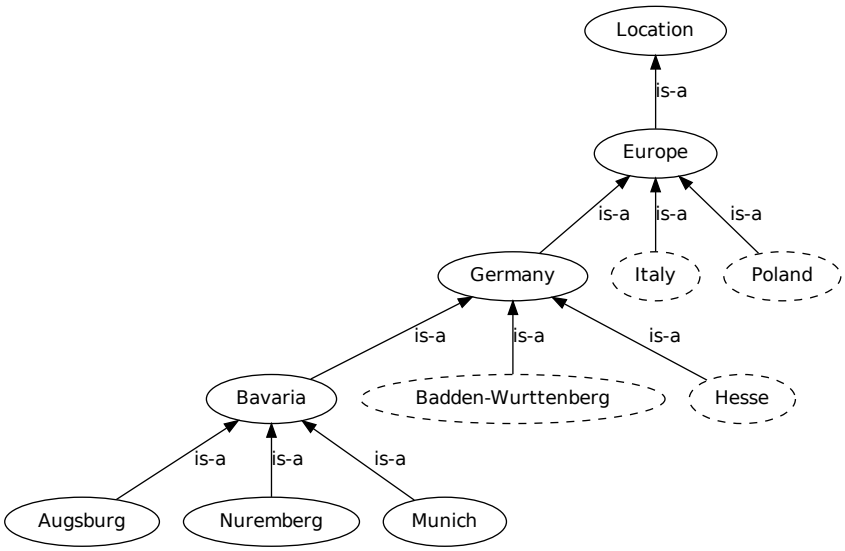
V tem dodatku prilagamo primere ontologij, ki smo jih uporabili v umetni domeni analize potrošnikov. Črtkana vozlišča označujejo vozlišča katerih potomce smo zaradi preglednosti izpustili.



Slika C.1: Ontologija bančnih storitev.



Slika C.2: Ontologija poklicev.



Slika C.3: Ontologija lokacij.

Dodatek D

Tabela podatkov umetne domene

V tem dodatku prilagamo celotno tabelo podatkov z umetne domene potrošnikov.

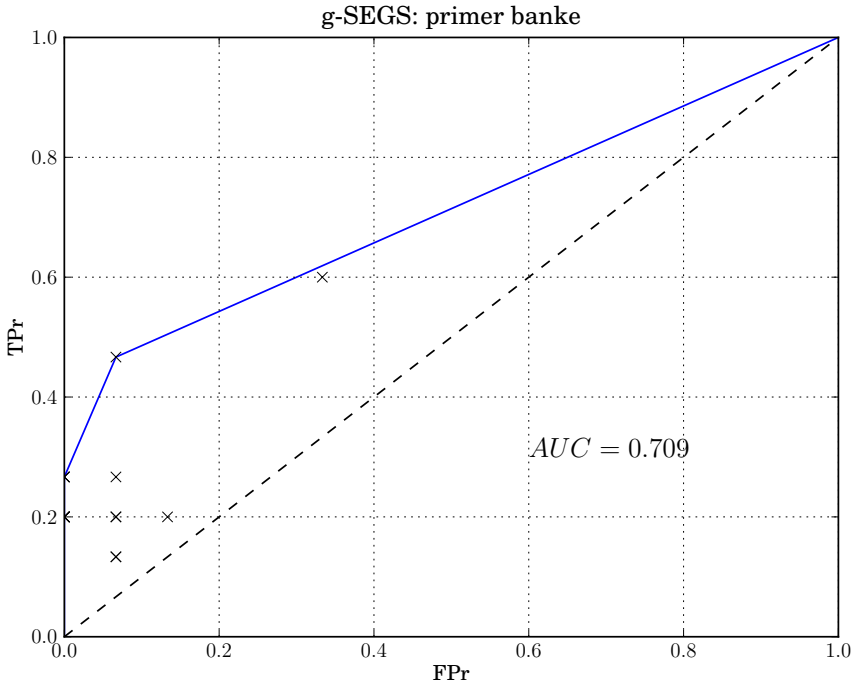
Tabela D.1: Vhodni podatki z domene potrošnikov.

id	occupation	location	account	loan	deposit	inv_fund	insur.	big_spender
1	Doctor	Milan	Classic	No	No	TechShare	Family	YES
2	Doctor	Krakow	Gold	Car	ShortTerm	No	No	YES
3	Military	Munich	Gold	No	No	No	Regular	YES
4	Doctor	Catanzaro	Classic	Car	LongTerm	TechShare	Senior	YES
5	Energy	Poznan	Gold	Apart.	LongTerm	No	No	YES
6	Doctor	Rome	Gold	Apartment	ShortTerm	No	Regular	YES
7	Finance	Bavaria	Gold	No	ShortTerm	GlobalShare	No	YES
8	Health-care	Frankfurt	Classic	Car	No	GlobalShare	Family	YES
9	Military	Warsaw	Gold	No	ShortTerm	No	Regular	YES
10	Education	Latina	Gold	Apartment	ShortTerm	No	Family	YES
11	Health-care	Karlsruhe	Classic	Apartment	No	EuropeShare	No	YES
12	Retail	Munich	Classic	Car	LongTerm	TechnologyShare	Regular	YES
13	Education	Catanzaro	Gold	Car	No	No	No	YES
14	Doctor	Milan	Classic	No	No	EuropeShare	No	YES
15	Police	Munich	Gold	Apartment	No	No	No	YES
16	Retail	Stuttgart	Classic	Car	LongTerm	TechologyShare	No	NO
17	Finance	Brescia	Gold	Apartment	No	EuropeShare	Regular	NO
18	Administration	Tarnow	Classic	Car	No	No	Senior	NO
19	Materials	Freiburg	Gold	Apartment	ShortTerm	GlobalShare	No	NO
20	Doctor	Poznan	Classic	Personal	ShortTerm	EuropeShare	Regular	NO
21	Administration	Cosenza	Classic	Car	No	No	No	NO
22	Unemployed	Munich	Classic	Car	No	No	No	NO
23	Military	Kalisz	Classic	Apartment	ShortTerm	EuropeShare	Regular	NO
24	Manufacturing	Cosenza	Gold	Apartment	LongTerm	No	No	NO
25	Transport	Cosenza	Classic	Car	ShortTerm	No	Family	NO
26	Police	Tarnow	Gold	Apart.	No	No	No	NO
27	Nurse	Radom	Classic	No	No	No	Senior	NO
28	Education	Catanzaro	Classic	Apart.	No	No	No	NO
29	Transport	Warsaw	Gold	Car	ShortTerm	TechShare	Regular	NO
30	Police	Cosenza	Classic	Car	No	No	No	NO

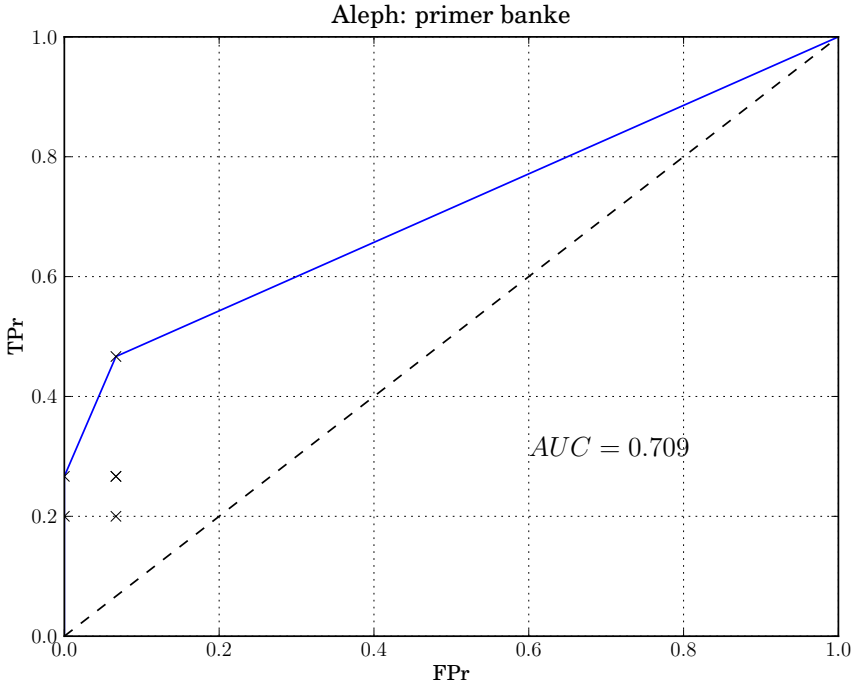
Dodatek E

Krivulje ROC

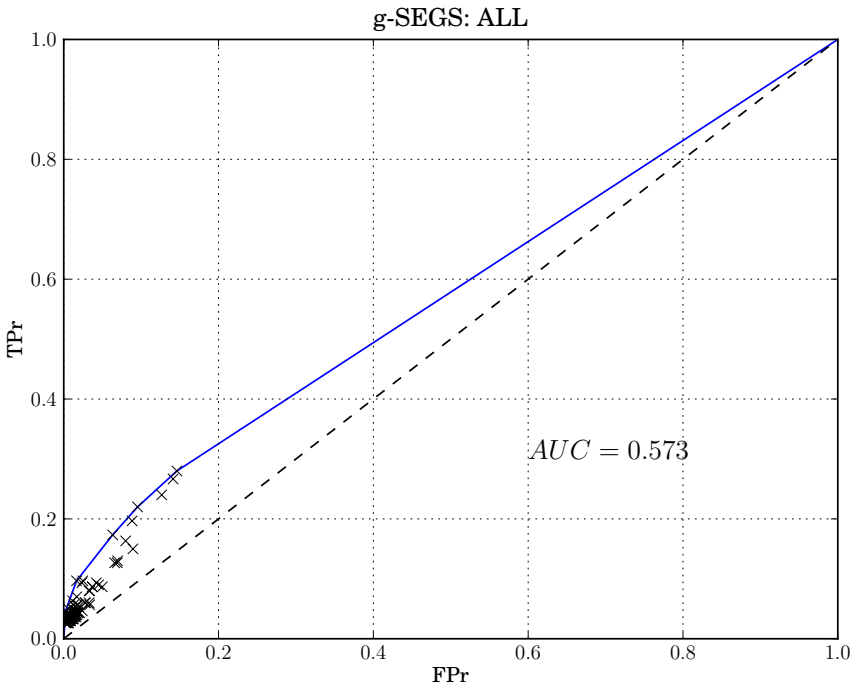
V tem dodatku prilagamo krivulje ROC opisov podskupin, ki jih sistema g-SEGS in Aleph odkrijeta na različnih domenah.



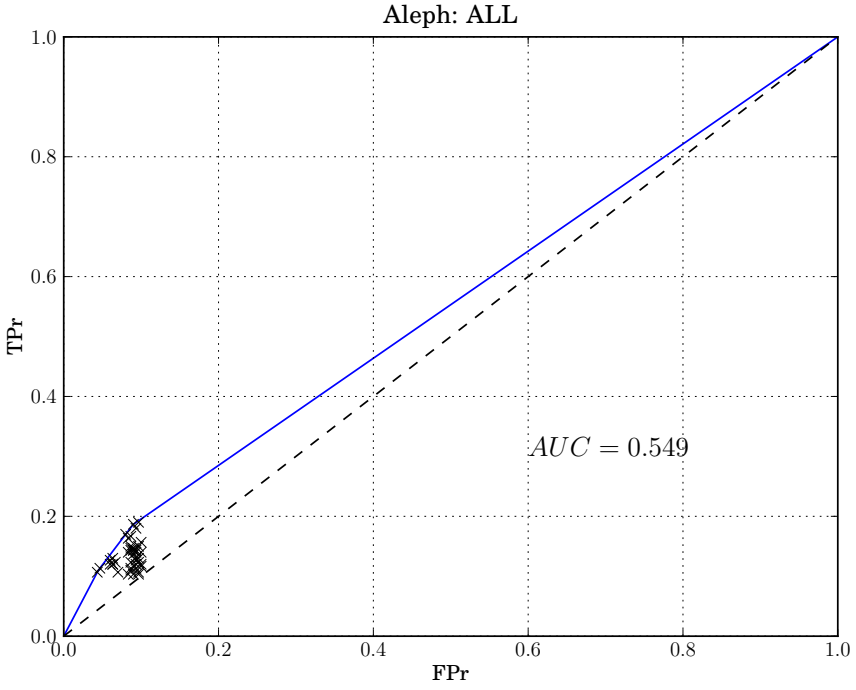
Slika E.1: Krivulja ROC podskupin odkritih s sistemom g-SEGS na domeni potrošnikov.



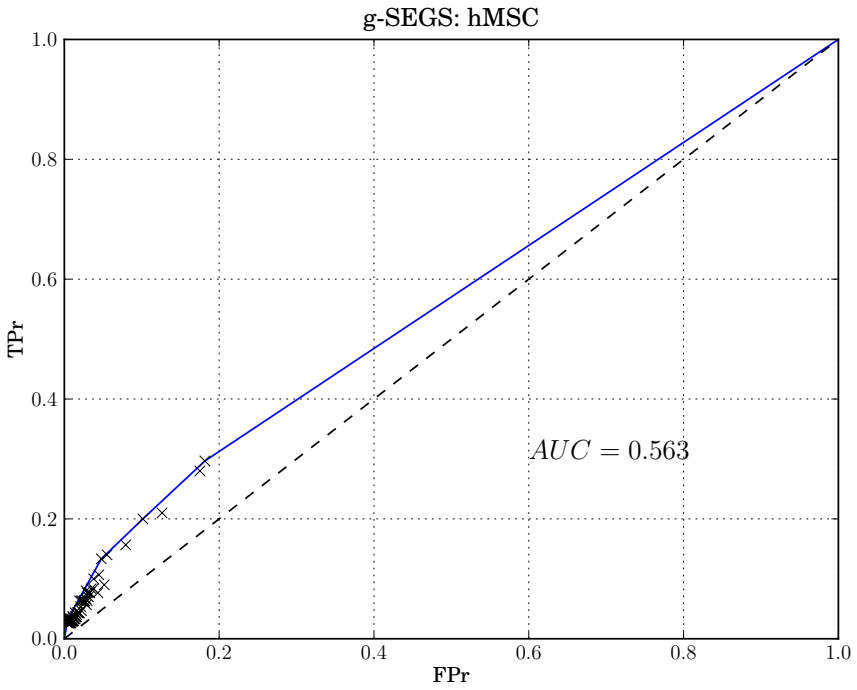
Slika E.2: Krivulja ROC podskupin odkritih s sistemom Aleph na domeni potrošnikov.



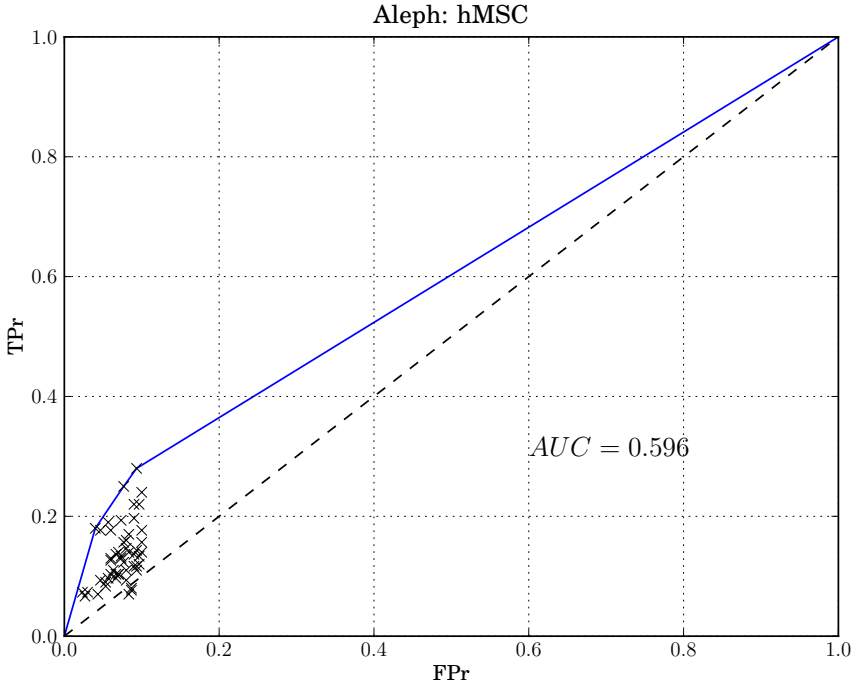
Slika E.3: Krivulja ROC podskupin odkritih s sistemom g-SEGS na domeni ALL.



Slika E.4: Krivulja ROC podskupin odkritih s sistemom Aleph na domeni ALL.



Slika E.5: Krivulja ROC podskupin odkritih s sistemom g-SEGS na domeni hMSC.



Slika E.6: Krivulja ROC podskupin odkritih s sistemom Aleph na domeni hMSC.

Slike

Tabele