UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Koristnostno podatkovno rudarjenje

Vid Podpečan

Ljubljana, 2007

POVZETEK

V diplomskem delu je podan pregled področja koristnostnega podatkovnega rudarjenja. Zgoščeno je opisana najpomembnejša oblika koristnostnega podatkovnega rudarjenja, cenovno občutljivo učenje. Natančno je predstavljeno podpodročje iskanja koristnih n-teric, ki predstavlja teoretično podlago implementiranih algoritmov. V okolju Orange sta implementirana dva najhitrejša algoritma za iskanje vseh koristnih n-teric, ShFSM in DCG. Oba algoritma sta podrobno opisana, podana pa je tudi kritična primerjava med njima ter njune posebnosti, ki so v primeru realizacije v jeziku Python ključnega pomena.

Diplomsko delo obsega tudi kratek opis posebne oblike koristnih n-teric, koristne pogoste n-terice. Analizirane so slabosti prvega obstoječega algoritma TP za iskanje koristnih pogostih n-teric, predstavljen pa je tudi nov, hitrejši in preprostejši algoritem FUFM, razvit v okviru tega diplomskega dela. Kratko je opisana podatkovna struktura razpršeno drevo, ki jo potrebujemo za učinkovito realizacijo vseh opisanih iskalnih algoritmov.

Testiranja implementiranih algoritmov so bila izvedena na umetnih transakcijskih podatkovnih bazah, pridobljenih z generatorjema IBM Quest in LUCS-KDD ARM, ki so bile z implementiranim transformacijskim programom pretvorjene v obliko, uporabno v okolju Orange. Na podlagi rezultatov so bile ugotovljene nekatere dobre in slabe lastnosti implementiranih algoritmov in uporabljenih generatorjev podatkov. Rezultati testiranj kažejo, da so kljub omejitvam, ki jih postavlja izbira programskega jezika, implementirani algoritmi uporabni tudi na večjih podatkovnih bazah, ki so po velikosti že blizu realnim.

SUMMARY

This thesis is a contribution to the area of utility-based data mining. It first presents a review of the area, by focusing on cost-sensitive learning as the most important area of utility-based data mining. A detailed description of high utility itemset mining is then presented, which forms a theoretical basis of implemented algorithms. Two fastest high utility itemset mining algorithms, ShFSM and DCG, were implemented into the Orange data mining framework. Both algorithms are described in detail and a critical comparison between them is given, together with their specifics which are of key importance for their implementation in the Python programming language.

The thesis describes also a specific form of high utility itemsets, utility-frequent itemsets. It analyzes deficiences of the first existing algorithm TP for utility-frequent itemset mining. A new, faster and simpler algorithm FUFM, proposed and implemented in this thesis is then presented. There is also a brief description of the hash tree data structure needed for the efficient realization of all the described data mining algorithms.

The testing of the implemented algorithms was performed on synthetic transactional databases generated with IBM Quest and LUCS-KDD ARM data generators, which were transformed into a form applicable to the Orange environment by the implemented transformation program. Based on the results some positive and negative properties of the implemented algorithms and of the applied database generators were noted. The results of the tests show that inspite of the limits imposed by the choice of the programming language, the implemented algorithms are usefull also on larger databases, which are close to the size of real bases.

Zahvala

Zahvaljujem se somentorici prof. dr. Nadi Lavrač in mentorju prof. dr. Igorju Kononenku za ideje, kritike, pomoč in spodbudo pri izdelavi diplomskega dela.

Zahvaljujem se tudi staršem, bratu, sestri ter prijateljem za potrpežljivost in podporo v času nastajanja te naloge.

Kazalo

1. Uvod

Podatkovno rudarjenje je ključen korak v procesu odkrivanja zakonitosti v podatkih [55, 61] (angl. knowledge discovery in databases – KDD). Opredelimo ga lahko kot uporabo specializiranih algoritmov, s katerimi iz zbrane množice podatkov izluščimo zanimive ter uporabne (koristne) vzorce in modele. Koristnostno podatkovno rudarjenje, predmet obravnave v tej nalogi, pa je podpodročje podatkovnega rudarjenja, katerega končni cilj je posplošena obravnava koristi rezultatov rudarjenja.

V tem diplomskem delu želimo uresničiti dva cilja. Prvi, teoretični cilj, je proučiti in kritično ovrednotiti področje *koristnostnega podatkovnega rudarjenja* [58, 59, 60] (angl. utility-based data mining). Podali bomo pregled metod in podpodročij, ki jih lahko štejemo kot oblike koristnostnega podatkovnega rudarjenja. Drugi, praktični cilj, pa je implementacija algoritmov za iskanje koristnih n-teric v okolju Orange ter testiranje na sintetičnih podatkovnih bazah.

V uvodnem poglavju je kratko predstavljeno področje podatkovnega rudarjenja in koristnostnega podatkovnega rudarjenja ter cilji, ki jih avtorji tega novega pojma želijo doseči, navedene pa so tudi nekatere nedoslednosti pri opredelitvi pojma in njegovem obsegu.

1.1 Podatkovno rudarjenje

Podatkovno rudarjenje je multidisciplinarna znanost [55, 61], saj uporablja in združuje postopke in ideje številnih področij. Na sliki 1.1 lahko vidimo umestitev področja podatkovnega rudarjenja med statistiko ter strojno učenje, umetno inteligenco in prepoznavanje vzorcev. Nekatera druga področja pa zagotavljajo podlago za izvajanje procesov podatkovnega rudarjenja: tehnologija podatkovnih baz, vzporedno računanje in porazdeljeno računanje. Poleg tega lahko opazimo, da podatkovno rudarjenje hitro sprejema ideje iz drugih, bolj teoretično usmerjenih področij: optimizacijske metode, informacijska teorija, evolucijsko računanje ipd.

Slika 1.1: Umestitev področja podatkovnega rudarjenja.

Kot smo povedali, je podatkovno rudarjenje najpomembnejši korak v procesu odkrivanja zakonitosti v podatkih. Ta proces je sestavljen iz (poenostavljena delitev [55]):

1. predobdelave pridobljenih podatkov;
2. uporabe tehnik podatkovnega rudarjenja;
3. poobdelave rezultatov.

Končni produkt procesa KDD je informacija (torej novo znanje), ki jo pridobimo iz rezultatov podatkovnega rudarjenja, med katerimi izberemo najboljše (filtriranje vzorcev) in jih primerno predstavimo (vizualizacija). V praksi seveda dodamo še četrto fazo, v kateri pridobljeno znanje tudi uporabimo.

Naloge podatkovnega rudarjenja po najsplošnejši delitvi razdelimo na *napovedne* in *opisne*.

Cilj prvih je napovedati vrednost izbranega atributa glede na vrednosti ostalih (izbranih) atributov. Atributu, katerega vrednost napovedujemo, pravimo tudi ciljna ali odvisna spremenljivka, pri napovedi uporabljenim atributom pa neodvisne ali pojasnjevalne spremenljivke.

Cilj opisnih nalog podatkovnega rudarjenja pa je pridobiti vzorce (korelacije, trende, skupine, anomalije), ki povzemajo (skrite) povezave in razmerja v podatkih.

1.2 Koristnostno podatkovno rudarjenje

Pojem koristnostnega podatkovnega rudarjenja je dokaj nov [58, 59, 60] in ne opredeljuje novega področja, ampak je krovni pojem, ki združuje določene raziskave na področju podatkovnega rudarjenja in strojnega učenja. Besedo *korist* avtorji tega novega izraza opredeljujejo kot ekonomsko korist [58] (angl. economic utility), torej nas zanima predvsem ekonomičnost in dobičkovnost celotnega procesa oz. posameznih faz.

S prvo težavo pri definiciji tega novega pojma se srečamo pri opredelitvi podatkovnega rudarjenja. Naša dosedanja obravnava je temeljila na uveljavljeni terminologiji [16], po kateri z izrazom podatkovno rudarjenje označujemo najpomembnejšo fazo v procesu odkrivanja znanja. Nekateri raziskovalci, vključno z avtorji izraza koristnostno podatkovno rudarjenje, pa vanj prištevajo še fazo pridobivanja podatkov in fazo uporabe pridobljenega znanja. Takšna opredelitev se po obsegu že enači z omenjeno definicijo procesa odkrivanja zakonitosti v podatkih in lahko povroči nejasnosti. Omenimo, da nekateri avtorji [19] eksplicitno enačijo oba pojma. Takšnih opredelitev sicer ne moremo imeti za napačne, saj sta oba izraza pomensko zelo široka, vendar vnašajo nepotrebno zmedo, ki otežuje strukturirano obravnavo.

Ker naš namen ni uveljavljanje novih izrazov, ampak pregled področja, bo nadaljnja obravnava *odstopala od uveljavljenih opredelitev*, predstavljenih v uvodnih odstavkih, ker želimo zagotoviti skladnost z (neformalno) terminologijo področja koristnostnega podatkovnega rudarjenja. Kadar bomo govorili o podatkovnem rudarjenju, bo to pomenilo definicijo, ki je enakovredna odkrivanju zakonitosti v podatkih. Kot poskus pravilnejšega in pomensko bolje opredeljenega imena lahko navedemo izraz “ekonomično odkrivanje zakonitosti v podatkih”, ki pa ga iz že omenjenega razloga ne bomo uporabljali.

Temelj, iz katerega avtorji izpeljujejo smiselnost uvedbe novega pojma, je dejstvo, da večina raziskav na področju strojnega učenja in napovednega podatkovnega rudarjenja ne upošteva kompleksnih okoliščin in faktorjev, ki vplivajo na gradnjo in uporabo modelov. Kot tipične (nerealne) predpostavke lahko navedemo prisotnost primerne količine popolnih učnih podatkov ter ocenjevanje kvalitete zgrajenega modela zgolj po napovedni točnosti.

Vprašanja, ki si jih zastavlja raziskovalec (ali tudi uporabnik) tehnik koristnostnega podatkovnega rudarjena, so:

- koliko podatkov potrebujemo za gradnjo modela, ki naj bo dovolj točen;

- kakšna je cena teh podatkov;
- kaj storiti z manjkajočimi vrednostmi, za kakšno ceno jih lahko pridobimo, ali je pridobitev cenovno upravičena glede na izboljšanje modela;
- kakšna je cena označitve (klasifikacije) učnih podatkov (primerov), če za to potrebujemo ekspertno znanje (strokovnjaka), koliko takšnih označitev potrebujemo;
- kolikšen čas je na voljo za izvajanje algoritmov, ali je računalniški sistem dovolj zmogljiv;
- kako v procesu gradnje modela upoštevati različne cene oz. pomembnosti posameznih razredov, primerov, vrednosti ter redko zastopane razrede;
- ali je pridobljeno znanje (vzorec, model) koristno ali samo zanimivo (nepričakovano);
- ali so stroški celotnega procesa podatkovnega rudarjenja upravičeni glede na (pričakovane) rezultate.

Druga težava, na katero naletimo pri naši obravnavi, je pomenski obseg izraza koristnostno podatkovno rudarjenje. Očitno je, da je le-ta zelo širok, saj lahko skoraj v vsako od metod podatkovnega rudarjenja vpeljemo še koristnostni faktor, npr. ceno podatkov, čas računanja, korist pri uporabi pridobljenega znanja ipd.

Med dosedanjimi raziskavami lahko v področje koristnostnega podatkovnega rudarjenja pri napovednem podatkovnem rudarjenju prištevamo vse oblike cenovno občutljivega učenja (npr. aktivno učenje, upoštevanje cene napačnih klasifikacij, idr.), pri opisnem pa povezovalna pravila (pogojno), iskanje koristnih n-teric ter v splošnem vse oblike iskanja vzorcev, kjer je upoštevan faktor koristnosti.

Obravnava tako širokega področja na omejenem obsegu diplomskega dela je nehvaležna naloga, ker je skoraj vedno potrebno izpustiti podrobnosti, ki so pomembne, vendar bi njihov opis terjal preveč prostora. Cenovno občutljivo učenje, predstavljeno v drugem poglavju, je tako obravnavano le površno, večinoma brez formalno zapisanih definicij in algoritmov, vendar smo poskušali zajeti vsa najpomembnejša podpodročja in oblike, pri čemer je poudarek na navedbah ustrezne specializirane literature in člankov ter nekaj praktičnih primerih uporabe. Prav tako smo v drugem poglavju deloma izpustili teoretično podlago povezovalnih pravil in formalne definicije različnih mer zanimivosti (koristnosti), podrobno je obravnavano le področje iskanja koristnih n-terk, ker predstavlja podlago, uporabljeno v

praktičnem delu naloge.

Resna težava, omenjena tudi na mednarodni delavnici o koristnostnem podatkovnem rudarjenju [60], je pomanjkanje javno dostopnih, realnih podatkov, ki bi vključevali tudi cene in informacije o koristnosti. Večina raziskovalcev si je tako prisiljena pomagati z generatorji umetnih podatkov, in kar je še bolj pereče, z naključno generiranimi cenami oz. koristnostmi (dobički).

Praktični del te naloge ni izjema, ker je testiranje omejeno na podatke, pridobljene z generatorjema podatkov IBM Quest [23] in LUCS-KDD ARM [36]. Prvi je sicer dokaj kompleksen in upošteva tipične realne privzetke [3], vendar kljub temu ne more nadomestiti pravih, naravnih podatkov. Glede na uporabnost tehnik iskanja najkoristnejših n-terk, npr. pri poslovnih odločitvah glede prodaje in določanja cen, bi izvajanje realiziranih algoritmov na realnih podatkih z realnimi cenami dobilo novo dimenzijo, saj bi pridobili informacijo o tem, katere n-terice produktov so tiste, ki dejansko nosijo največji dobiček. Takšni podatki bi lahko bili ključnega pomena v reviziji cen, prodaje in nabave v nekem trgovskem podjetju. Ker so v testiranjih, opisanih v četrtem poglavju, uporabljene zgolj umetne podatkovne baze, je tudi uporabnost dobljenih rezultatov omejena. Tako je analiza rezultatov omejena na časovne in prostorske primerjave med algoritmi ter splošne lastnosti procesa iskanja koristnih n-teric.

Zaključimo lahko, da je cilj koristnostnega podatkovnega rudarjenja proučevanje vseh koristnostnih faktorjev, še posebej ekonomskih, ki vplivajo na skupno korist znanja, pridobljenega v procesu podatkovnega rudarjenja. Ker lahko te faktorje najdemo v vsaki od faz, je končni cilj združiti obstoječe, bolj specializirane raziskave v enotno ogrodje, ki bo omogočalo maksimizacijo koristi celotnega procesa podatkovnega rudarjenja. Zavedamo se, da je takšna opredelitev ohlapna, vendar pa tudi avtorji sami ne določajo podrobneje, kaj natančno je koristnostno podatkovno rudarjenje. Zgoraj omenjeni končni cilj je morda nedosegljiv, že zaradi optimizacijske narave problema, vendar lahko uvedbo novega pojma smatramo za smiselno, če bo vodila v razvoj novih, kompleksnejših načinov vrednotenja pridobljenega znanja.

1.3 Zgradba diplomskega dela

V drugem poglavju je podan pregled najpomembnejših oblik koristnostnega podatkovnega rudarjenja. Opisano je področje *cenovno občutljivega učenja* ter področje *iskanja koristnih n-teric*. Pojasnjena je tesna vez s področjem iskanja *povezovalnih pravil* in *pogostih n-teric* ter razlika med koristnimi in pogostimi n-tericami. Tretje poglavje opisuje najhitrejše algoritme za iskanje koristnih n-teric ter podatkovne strukture, ki so potrebne za učinkovito realizacijo. Četrto poglavje vsebuje predstavitev implementacije algoritmov za iskanje koristnih n-teric, povzetke in analizo izvedenih testov, komentar ter grafične in tabelarične predstavitve rezultatov. Kratko sta predstavljena tudi oba uporabljena generatorja umetnih podatkovnih baz ter njune prednosti in slabosti. V zaključku povzemamo opravljeno delo ter nakažemo možne razširitve in dopolnitve.

2. Pregled področja koristnostnega podatkovnega rudarjenja

Kot smo omenili v uvodu, koristnostno podatkovno rudarjenje vključuje metode napovednega in opisnega podatkovnega rudarjenja. V tem poglavju predstavljeni pregled področja koristnostnega podatkovnega rudarjenja je zato razdeljen v dve podpoglavji. V prvem je opisano področje cenovno občutljivega učenja, ki je najpomembnejša in najobsežnejša oblika napovednega koristnostnega podatkovnega rudarjenja. V drugem podpoglavju pa je predstavljeno področje iskanje koristnih n-teric, ki temelji na metodah iskanja pogostih n-teric ter povezovalnih pravil. Podane so vse potrebne definicije in dokazi, opisana pa je tudi splošna oblika učinkovitega algoritma za iskanje koristnih n-teric. Omenjeno je tudi podpodročje iskanja koristnih in pogostih n-teric in splošna ocena pomembnosti in uporabnosti področja iskanja koristnih n-teric.

2.1 Cenovno občutljivo učenje

O cenovno občutljivem učenje govorimo, kadar v procesu učenja upoštevamo takšno ali drugačno vrsto cen. V strokovni literaturi izraz cenovno občutljivo učenje pogosto označuje upoštevanje le ene vrste cene: ceno napačne klasifikacije. Naša obravnava je splošnejša in v skladu z najpopolnejšo opredelitvijo cen [56], ki jih srečamo v procesu učenja iz primerov oz. nadzorovanega učenja (angl. learning from examples, supervised learning).

V tem podpoglavju je predstavljena klasifikacija najpomembnejših vrst cen, ki jih srečamo v procesu učenja iz primerov. Opisane so lastnosti cenovne matrike in njene posplošitve, dobičkovne matrike. Predstavljeni so načini, kako lahko v enotnem ogrodju upoštevamo različne vrste cen in tako posplošimo obravnavo. Takšne metode so še posebej pomembne v smislu koristnostnega podatkovnega rudarjenja, saj predstavljajo razvoj v smeri analize koristi (cene) celotnega procesa pridobivanja znanja.

Vhod v nek splošni algoritem učenja iz primerov je množica označenih primerov (učna množica): $S = \{\langle \boldsymbol{x}_i, y_i \rangle\}$, $i = 1, 2, 3, \ldots, n$, kjer je $\boldsymbol{x}_i$ vektor diskretnih ali zveznih

vrednosti, imenovanih atributi (značilke), y_i pa oznaka primera $\boldsymbol{x}_i$. V naši obravnavi cenovno občutljivega učenja privzemamo, da so vrednosti spremenljivke y_i diskretne (razredi), torej govorimo o klasifikacijskih problemih. Ta predpostavka sicer ni nujna pri vseh vrstah cen, vendar pa poenostavi obravnavo.

2.1.1 Uporabnost cenovno občutljivega učenja

Klasični klasifikacijski algoritmi in sistemi kot merilo uspešnosti upoštevajo samo klasifikacijsko točnost. Takšni modeli se v praksi obnesejo dobro le tedaj, če so cene (posledice) napačne klasifikacije vedno enake in neodvisne od napovedanega razreda, kar pomeni, da minimiziranje števila napačnih napovedi hkrati zadosti tudi kriteriju minimalne cene napake.

V realnih domenah se cene napačnih napovedi razlikujejo. Najenostavneši primer, ki odkrije pomankljivost kriterija klasifikacijske točnosti, je primer redkih razredov (angl. class imbalance problem). Tu je eden (ali tudi več) razredov redko zastopan [29, 55], kar pomeni, da lahko z večinskim klasifikatorjem, ki vse nove primere označi z oznako prevladujočega razreda, dosežemo veliko klasifikacijsko točnost, pri čemer pa smo ignorirali vse manjšinske razrede. Odveč je poudarjati, da je takšen model neuporaben, še posebej tedaj, kadar je redko zastopanim razredom namenjena posebna pozornost (kar je v praksi običajno). Kot primer [55] lahko navedemo nadzor proizvodnje linije, kjer brezhibni proizvodi številčno prevladujejo nad defektnimi. Soroden primer so sleparske transakcije z bančnimi karticami, kjer prav tako močno prevladujejo legitimne transakcije.

Nadaljnji primeri, v katerih sicer ni problema redkih razredov, cena napačne odločitve pa je kritičen faktor (pogosto je tudi napovedna točnost nizka – v absolutnem smislu), so problemi medicinske diagnostike [24, 31] in odobritve kreditov [15]. V problemu medicinske diagnostike je ekspert (ali pa tudi ekspertni sistem) postavljen pred odločitev, kako ravnati v primeru, ko diagnoza ni povsem očitna in obstaja določena mera dvoma. Če bolnega pacienta razglasimo za zdravega, so v primeru nevarne bolezni posledice usodne. Nasprotno pa dodatni testi in zdravljenje pacienta, ki je zdrav, niso tako zelo kritični (čeprav vodijo v nepotrebno porabo sredstev, lahko pa tudi s stranskimi učinki škodujejo pacientu). Naiven model, ki bi minimiziral kritične posledice, bi vse paciente proglasil za bolne. Seveda pa bi v tem primeru

potrebovali nerazumno veliko število ekspertov (zdravnikov) in sredstev, s stranskimi učinki zdravljenja pa bi lahko škodovali (številnim) popolnoma zdravim pacientom

S podobnim težavami se srečamo v problemu odobritve kreditov. Tu kot kritično odločitev prepoznamo odobritev kredita komitentom, ki ga iz določenega razloga ne bodo odplačali. Preveliko število odobrenih kreditov nezanesljivim osebam lahko vodi v stečaj posojilodajalca, veliko število zavrnjenih prošenj zanesljivim osebam pa zmanjšuje njegov ugled in obseg poslovanja.

2.1.2 Cena

Besedo cena si moramo razlagati v najbolj abstraktnem smislu [56]. Lahko je merjena v denarnih enotah, časovnih enotah, ali pa v najbolj splošnem primeru v abstraktnih enotah koristnosti (angl. utils). V primerih medicinske diagnostike pridejo v poštev tudi enote kot je npr. kvaliteta življenja pacienta, v kolikor lahko takšno enoto sploh izmerimo. Pri računalniški prepoznavi slik pa bi ceno lahko opredelili kot procesorski čas, potreben za izvedbo določenih izračunov.

Najpopolnejšo taksonomijo cen, ki lahko nastopajo v induktivnem učenju (učenju iz primerov), podaja P. Turney [56]. Razlikuje nič manj kot 9 glavnih tipov cen, pri tem pa še privzema, da so cene z gotovostjo določene (v nasprotnem primeru bi namreč morali negotovost modelirati z verjetnostno porazdelitvijo po vseh možnih cenah, kar pa bi analizo močno zapletlo).

Glavni tipi cen so:

1. **cena napačne klasifikacije:**

   Ta vrsta cene je najpogosteje upoštevana in pogosto tudi najpomembnejša. Upoštevanje te cene v literaturi običajno imenujejo cenovno občutljivo učenje [41] (kar je glede na obravnavano taksonomijo zavajajoče). Ceno napačne klasifikacije definiramo kot ceno klasifikacije primera v razred *i*, če je njegov pravi razred *j*. Cena pravilne klasifikacije je običajno 0, ni pa nujno. Ena od možnih različic je tudi obstoj dodatnega razreda "*neznan"*, ki je seveda primerno utežen. V najpogostejši obliki so cene posameznih razredov konstantne [4, 20], lahko pa so odvisne od posameznega

primera, časa klasifikacije, klasifikacije ostalih primerov in vrednosti atributov.

2. **cena testov (meritev):**

   Vsaka meritev oz. test lahko ima ceno. V primeru medicinske diagnostike je to npr. cena analize krvi ali cena rentgenske fotografije. Odločitev, ali je smiselno plačati ceno nekega testa, je odvisna od tega, če poznamo cene napačnih klasifikacij [57]. Kadar je cena napačne klasifikacije veliko večja kot cene testov, je smiselno opraviti vse teste, ki so relevantni. V nasprotnem primeru, ko so cene testov veliko večje kot cene napačne klasifikacije, pa sploh ni racionalno izvesti kateregakoli testa. Tudi tu je lahko cena konstantna ali odvisna od okoliščin. Konstantna cena je enaka za vse učne primere (cene različnih testov pa se seveda razlikujejo). Okoliščine [56], ki lahko vplivajo na ceno testov, so: predhodni testi, rezultati predhodnih testov, pravi razred primera, stranski učinki, posebnosti posameznega primera in čas testa.

3. **cena učitelja:**

   Lahko si predstavljamo primer, ko imamo na voljo dovolj (ali neomejeno) učnih podatkov, ki pa niso označeni. Oznake lahko pridobimo, vendar moramo zanje plačati določeno ceno. Učni algoritem (ali učenec) izbira primere, ki naj jih klasificira ekspert (učitelj), lažje primere pa lahko poskuša klasificirati sam. Kadar pa učenec nima možnosti izbire, temelji odločitev na primerjavi cene napačne klasifikacije in cene odgovora eksperta. Izbiranju primerov, ki naj jih klasificira učitelj, pravimo *aktivno učenje*. Podobno kot pri ceni napačne klasifikacije in ceni testov tudi tu ločimo konstantno in pogojno ceno učitelja. Konstantna cena označitve poljubnega primera je poenostavljena in nerealna predpostavka, saj je v praksi cena običajno odvisna od težavnosti primera.

4. **cena izračunov:**

   Ker so računski viri skoraj vedno omejeni, je smiselno upoštevati tudi cene izračunov. Delimo jih na statične in dinamične ter cene izračunov med učenjem in cene izračunov med uporabo. Med statične cene izračunov štejemo velikost programa (bajti, vrstice) in strukturno kompleksnost (število zank, število rekurzivnih klicev), med dinamične pa časovno in prostorsko kompleksnost [22], ki sta povezani. Cena izračunov med učenjem je praviloma različna od cene izračunov med uporabo. Kot primer lahko

navedemo učenje z nevronskimi mrežami, ki ima visoko dinamično časovno zahtevnost med učenjem ter nizko med uporabo. Nasprotno pa ima algoritem k najbližjih sosedov med učenjem visoko dinamično prostorsko zahtevnost ter nizko dinamično časovno zahtevnost, med uporabo pa tipično visoko dinamično časovno zahtevnost.

5. **cena učnih primerov:**

   Tu za razliko od cene testov obravnavamo ceno celotnega učnega primera. V tipičnem primeru imamo na voljo omejeno število primerov, pridobitev nadaljnjih pa je draga (včasih je lahko seveda tudi nemogoča: neponovljivost, zgodovinski podatki). V primeru gradnje modela, ki bo po učenju vgrajen v nek sistem in se ne bo več spreminjal, lahko minimiziramo združeno ceno učenja in izvajanja kot funkcijo velikosti učne množice, če poznamo (ocenimo) [49]: (1) razmerje med stopnjo napačne klasifikacije in velikostjo učne množice (t.i. krivulja učenja, angl. learning curve), (2) pričakovano število klasifikacij, ki jih bo sistem opravil v času delovanja, (3) ceno napačnih klasifikacij in (4) ceno pridobivanja učnih podatkov. Podobno lahko s prilagodljivim (adaptivnim) modelom optimiziramo združeno ceno učenja in izvajanja s prilagajanjem krivulje učenja in velikosti učne množice, če poznamo (2), (3) in (4). Enak zaključek velja tudi v primeru inkrementalnega modela (ta se spreminja v času delovanja sistema), le da tu pričakujemo, da se bo stopnja napačnih klasifikacij s časom zmanjševala.

Ostale, v praksi manj pomembne vrste cen, ki jih ne bomo podrobneje opisovali, so: *cena vpliva človek-stroj* (npr. iskanje primernih parametrov, priprava podatkov, analiza rezultatov), *cena nestabilnosti* (cena, ki je povezana z nestabilnostjo zgrajenega modela), *cena posega* in *cena stranskih učinkov*. Zadnji dve sta povezani z vplivom človeka na proces, katerega lastnosti so atributi učnega problema (npr. destilacija nafte, kjer npr. ugotovimo pravilo: če ima senzor A večjo vrednost kot senzor B, bo narasla količina proizvoda C. Če želimo povečati proizvodnjo C, moramo za to plačati ceno, potrebno za spremembo A. Če to pravilo velja npr. samo v 90% primerov, imamo v 10% še dodatno ceno stranskega učinka).

2.1.3 Cena napačne klasifikacije

Kadar cene klasifikacij v napačne razrede niso enake, moramo uporabiti tehniko, ki bo omogočila vpeljavo različnih cen v proces učenja. Najsplošnejša delitev loči dve strategiji za obravnavo cen napačne klasifikacije: *spreminjanje učnih podatkov* in *spreminjanje učnega algoritma*. Med tehnike spreminjanja učnih podatkov štejemo stratifikacijske metode [41, 4], boosting metode [17] in meta učne algoritme [13]. Tehnike spreminjanja učnega algoritma so specializirane in pogosto prilagojene posameznim domenam. Omenimo le, da imajo vsi pomembnejši učni algoritmi tudi cenovno občutljive različice, npr. nevronske mreže [28], odločitvena drevesa [57], učenje z genetskim programiranjem [33], naivni in delno naivni Bayes [31], k-najbližjih sosedov [31], ipd.

Ker obe opisani vrsti strategij nista neposredno povezani s koristnostnim podatkovnim rudarjenjem, ju ne bomo obravnavali. Podrobneje bomo opisali le matriko cen, ki jo imenujemo tudi *cenovna matrika* (angl. cost matrix) in njeno posplošitev, *dobičkovno* ali *koristnostno matriko* (angl. benefit matrix).

Privzemamo tudi, da je cenovna matrika (dobičkovna matrika) konstantna (z izjemo enega primera), torej imajo vse celice matrike enake vrednosti ne glede na učne primere, čas in vrednosti atributov.

3.A Cenovna matrika

Cenovna matrika podaja cene, ki so povezane z napačno in pravilno klasifikacijo učnih primerov.

Definicija 2.1: C je cenovna matrika velikosti $n \times m$ za domeno D, če je n enak številu mogočih napovedi klasifikatorja, m pa je enak številu razredov v domeni D. Velja:

$$C = \begin{bmatrix} c_{0,0} & \cdots & c_{0,m} \\ \vdots & \ddots & \vdots \\ c_{n,0} & \cdots & c_{n,m} \end{bmatrix}, \quad n \geqslant m, \quad \forall i, j : c_{i,j} \geqslant 0$$

Vrednost elementa $c_{i,j}$ pomeni ceno klasifikacije primera v razred i, če je njegov pravi razred j. Kadar matrika ni kvadratna (vrstic oz. možnih napovedi je več), gre za že omenjeni primer dodatnega razreda *"neznan"*.

Če imamo podano cenovno matriko in verjetnosti $P(j|\boldsymbol{x})$ za dani primer **x**, lahko definiramo cenovno optimalno napoved:

$$r_{OPT} = \underset{i}{argmin} \sum_{j=1}^{m} P(j|\boldsymbol{x}) \cdot c_{i,j}$$

Optimalna napoved je torej tisti razred, ki minimizira zgornjo vsoto (pravimo ji p*ogojno tveganje* ali tudi *pričakovana cena napovedi*). Naloga učnega algoritma, ki ga pri tem uporabimo, je oceniti verjetnosti $P(j|\boldsymbol{x})$ po vseh razredih j za vse učne primere.

Smiselna predpostavka [15, 24] je, da velja: $c_{j,j} < c_{i,j} \forall \; i \neq j$, kar pomeni, da je cena pravilne klasifikacije vedno nižja kot cena napačne klasifikacije [24, 15]. Če to ne drži, se lahko zgodi, da katerega od razredov klasifikator sploh nikoli ne napove. Preprost test, kdaj pride do takšne situacije, je dominanca vrstic v cenovni matriki. Kadar vrstica *m* dominira nad vrstico *n* za vse vrednosti *j*: $C(m,j) \geqslant C(n,j)$, je napoved razreda *n* cenejša od napovedi razreda *m* ne glede na dejanski razred primera.

V tabeli 2.1 je podan primer cenovne matrike, kjer vrstici 2 in 3 dominirata nad vrstico 1. V tem primeru cenovno občutljiva klasifikacija ni potrebna, ker je optimalna napoved v smislu enačbe o cenovno optimalni napovedi vedno razred C1.

Tabela 2.1: Primer cenovne matrike, kjer je cenovno optimalna napoved vedno razred C1.

	pravi razred		
napovedan razred	**C1**	**C2**	**C3**
C1	1	2	4
C2	2	3	5
C3	3	10	7

Transformacije cenovne matrike

Na cenovni matriki so dovoljene naslednje operacije, ki ne spremenijo optimalne napovedi [41, 15]:

1. **skaliranje:**

   Če vsako celico v matriki pomnožimo s pozitivno konstanto *k*, se napoved optimalnega razreda ne spremeni. Da to drži, se lahko prepričamo tako, da v enačbi cenovno optimalne napovedi izpostavimo konstanto *k*. Operacijo skaliranja lahko

interpretiramo kot spremembo cenovne enote.

2. **premik:**

   Podobno kot pri skaliranju se pri prištevanju pozitivne konstante vsem celicam cenovne matrike ne spremeni napoved optimalnega razreda. Dokaz izpeljemo podobno kot pri skaliranju: iz vsote v enačbi cenovno optimalne napovedi ločimo člen

$$k \cdot \sum_{j=1}^{m} P(j|\boldsymbol{x}),$$

   za katerega lahko ugotovimo, da je enak *k*, ker je vsota vseh verjetnosti v porazdelitvi enaka 1. Operacija premika je koristna, ker pomeni spremembo osnove, iz katere merimo cene napačnih napovedi.

3.B Dobičkovna matrika

Posplošitev cenovnih matrik so t.i. *dobičkovne* ali *koristnostne matrike* [41, 15] (angl. benefit matrices), ki so definirane takole:

Definicija 2.2: B je dobičkovna matrika velikosti $n \times m$ za domeno D, če je *n* enak številu mogočih napovedi klasifikatorja, *m* pa je enak številu razredov v domeni D. Velja:

$$B = \begin{bmatrix} b_{0,0} & \cdots & b_{0,m} \\ \vdots & \ddots & \vdots \\ b_{n,0} & \cdots & b_{n,m} \end{bmatrix}, \quad n \geqslant m, \quad b_{i,i} \geqslant 0 \quad \text{in} \quad b_{i,j} < b_{i,i} \quad \text{za} \quad i \neq j$$

Tu se lažje izognemo napakam pri določanju vrednosti v matriki, ker imamo naravno mejo, iz katere merimo dobiček oz. korist. Ta meja je stanje agenta pred odločitvijo glede obravnavanega primera. Če je odločitev koristna, ima agent dobiček (pozitivna vrednost), v nasprotnem primeru pa izgubo (negativna vrednost).

Primer spremenljive dobičkovne matrike za klasifikacijo bančnih transakcij, katere vrednosti so odvisne od posameznega učnega primera, je podan v tabeli 2.2. Tu je cena odobrene sleparske transakcije odvisna od njene velikosti, ker banka nosi odgovornost za zlorabo. Odobrena legitimna transakcija vodi v dobiček, ki je odvisen od višine zneska transakcije, npr. 2%. Zavrnitev legitimne transakcije je netrivialna, ker banka ne more vnaprej predvideti, kako bo na zavrnitev reagiral kupec. Če ta npr. po več neuspešnih transakcijah zapre račun,

ima banka dolgoročno izgubo. Zato vrednost -20€ v tabeli ne predstavlja dejanske izgube, pač pa nekakšno povprečje izgube na posamezno zavrnjeno legitimno transakcijo. Prav tako dobiček 20€ pri zavrnitvi sleparske transakcije ni dejanski ampak dolgoročni, saj pomeni preprečitev nadaljnjih sleparij in morebitno aretacijo.

Tabela 2.2: Primer spremenljive dobičkovne matrike bančnih transakcij.

	pravi razred	
napovedan razred	**sleparska**	**legitimna**
sleparska	20€	-20€
legitimna	**-x**	0.02**x**

Dobičkovne matrike imajo vse lastnosti cenovnih matrik in izvajanje operacij skaliranja in premika ne spremeni na njej temelječe optimalne odločitve, ki je definirana takole:

$$r_{OPT} = \underset{i}{argmax} \sum_{j=1}^{m} P(j|\boldsymbol{x}) \cdot b_{i,j}$$

Vsoto v zgornji enačbi imenujemo tudi pričakovani dobiček. Celotni pričakovani dobiček (korist) pa je definiran kot:

$$TB = \sum_{x \in S} \sum_{j=1}^{m} P(j|\boldsymbol{x}) \cdot b_{i,j}$$

2.1.4 Cena testov

V klasifikacijskih problemih se pogosto srečamo z manjkajočimi vrednostmi atributov. Kadar je manjkajočih vrednosti le malo v primerjavi z vsemi učnimi podatki, imamo na voljo več tehnik [55], ki se v praksi obnesejo zadovoljivo:

- odstranitev nepopolnih primerov, odstranitev atributov. Pri slednji je potrebna pazljivost, ker je odstranjeni atribut lahko ključnega pomena.
- ocenjevanje manjkajočih vrednosti. Tu lahko uporabimo tehnike interpolacije, iskanja najbližje vrednosti in najpogostejše vrednosti.
- ignoriranje manjkajočih vrednosti med učenjem.

Opisani pristopi so primerni le, kadar je manjkajočih vrednosti malo, popolnih podatkov pa dovolj. Kadar pa manjka veliko število vrednosti in ni mogoče zgraditi dovolj točnega modela, potrebujemo inteligenten pristop k izbiranju testov (pridobivanju manjkajočih vrednosti), ker je izvajanje vseh testov največkrat predrago in nepotrebno, naključno izbiranje podmnožice testov pa cenovno neučinkovito, ker ne upošteva podatkov, ki jih že imamo. Uveljavljeno ime takšnega pristopa je *aktivna izbira testov* (angl. active feature-value acquisition).

Definicija 2.3: Imamo *m* učnih primerov $x_1, \ldots, x_m$, vsak od njih ima *n* atributov $a_1, \ldots, a_n$. Pri vsakem primeru poznamo vrednosti nekaterih atributov in razred *r*. Vrednosti ostalih atributov so neznane, lahko pa jih pridobimo za določeno ceno. Problem aktivne izbire testov se glasi: katere primere in katere teste naj izberemo, da bo zgrajeni model (klasifikator) imel zahtevane lastnosti (npr. velika točnost, nizka cena).

F. Provost in sod. so razvili splošno metodo za aktivno izbiro testov [51], ki izbira tiste teste, za katere je ocenjeno, da bodo cenovno najbolj učinkoviti (glede na točnost grajenega modela). Metoda je popolnoma splošna, ker jo lahko uporabimo s poljubnih učnim algoritmom, merilom kakovosti modela in cenami posameznih testov. Edina (trenutna) omejitev je pogoj diskretnosti atributov.

Postopek je sestavljen iz več iteracij, v vsaki izberemo *b* testov, ki smo jih najbolje ocenili. Ocenjevanje testov temelji na *pričakovani koristi* (angl. expected utility) testa *j* za primer i ($q_{i,j}$), ki jo izračunamo takole:

$$E(q_{i,j}) = \sum_{k=1}^{K} P(F_{i,j}=V_k) \cdot U(F_{i,j}=V_k)$$

$P(F_{i,j} = V_k)$ je verjetnost, da ima *j*-ti atribut primera *i* vrednost V_k (*F* je tabela učnih podatkov), $U(F_{i,j} = V_k)$ pa korist podatka o vrednosti V_k *j*-tega atributa, ki je definirana kot:

$$U(F_{i,j} = V_k) = \frac{A(F, F_{i,j} = V_k) - A(F)}{C_{i,j}}$$

kjer je *A(F)* točnost trenutnega klasifikatorja, $A(F, F_{i,j} = V_k)$ točnost klasifikatorja, ki upošteva, da je $F_{ij} = V_k$, $C_{i,j}$ pa cena testa q_{ij}. Ker točnih vrednosti $P(F_{i,j} = V_k)$ in $A(\cdot)$ ne poznamo, jih je potrebno na primeren način oceniti [51]. Izčrpno iskanje najkoristnejših testov

je časovno zelo potratno (zgraditi moramo nov klasifikator za vsako možno vrednost vsakega atributa), zato je smiselno uporabiti tehnike jemanja vzorcev (tu govorimo o vzorčni pričakovani koristi). Testi kažejo, da lahko s primernim izbiranjem vzorcev, npr. z vzorčno pričakovano koristjo, izvajanje občutno pohitrimo brez pomebnejše izgube točnosti [51].

S tehniko vzorčne pričakovane koristi izberemo vzorec testov velikosti $\alpha \cdot b$ in zanje izračunamo pričakovano korist po zgornjih enačbah. Uporabljeni parameter α določa obseg preiskovanja, vrednosti zanj pa izbiramo iz območja $[1, \ldots, \frac{m \cdot n}{b}]$. Obravnavani vzorec testov lahko izberemo naključno ali pa uporabimo strategijo *error sampling*, pri kateri teste izberemo med tistimi primeri, ki jih je trenutni model klasificiral napačno oz. je njegova sicer pravilna klasifikacija najbolj nedoločena (neizrazita porazdelitev verjetnosti po možnih razredih).

Opisana korist rezultata testa $q_{i,j}$ upošteva izboljšavo točnosti modela glede na ceno testa, seveda pa jo lahko priredimo za poljuben cilj učenja (čim večja točnost ali pa npr. najcenejši klasifikator). Predstavljena metoda aktivne izbire testov je učinkovita in daje občutno boljše rezultate kot naključno izbiranje testov. Še posebej dobro pa se obnese v primerih velikih razlik med posameznimi testi glede njihovega prispevka k izboljšanju modela na enoto cene.

Med sorodne, čeprav nekoliko manj splošne metode, lahko štejemo še:

- učenje z zalogo (angl. budgeted learning), kjer je največja cena testov omejena z vnaprej določeno količino denarja [34, 26]. Kot primer lahko navedemo gradnjo sistema za diagnosticiranje neke bolezni. Na voljo je določena vsota denarja, vzorci z ugotovljeno boleznijo (npr. vzorci tkiva) ter množica testov z znanimi cenami in neznano pomembnostjo za diagnosticiranje. Sedaj smo postavljeni pred vprašanje, kako najbolje porabiti dana sredstva za gradnjo sistema. Ali na vseh vzorcih izvajamo vse teste, dokler ne izčrpamo finančnih sredstev, ali pa izbiramo, katere teste izvesti na posameznem vzorcu. Preprostejša oblika [35], ki ji pravimo *aktivna izbira modela* (angl. active model selection), poskuša ugotoviti, katere teste izvesti na vseh primerih. Množici testov v tem primeru pravimo model, v splošnem pa je to lahko tudi množica klasifikatorjev z znanimi cenami in neznano točnostjo ali pa npr. množica vhodnih parametrov programa, pri katerem želimo dobiti optimalen izhod v omejenem številu poskusov.

  Uporabimo lahko tehnike spodbujevanega učenja (angl. reinforcement learning), vendar

testiranja kažejo [27], da se obnesejo slabše kot preprostejše strategije (npr. round robin, biased robin). Običajna teoretična formulacija aktivne izbire modela temelji na problemu *n* kovancev z neznanimi verjetnostmi cifre in grba, kjer je cilj v končnem številu korakov (metov izbranega kovanca z znano ceno) določiti najboljši kovanec (kvaliteta je npr. verjetnost grba). Z omenjeno strategijo round robin mete pravično porazdelimo med vse kovance, pri strategiji biased robin pa met trenutnega kovanca ponavljamo, dokler ne dobimo negativnega izida (cifra). Problem izbire kovancev prevedemo v problem izbire testov tako, da kovanec predstavlja model (množico testov), verjetnost pozitivnega izida pa kriterij izbire modela (npr. napovedna točnost).

- izbiranje primerov, na katerih izvedemo vse možne teste [50]. Ta pristop je poenostavljena oblika opisanega postopka, izbira primera pa temelji na napačnih odločitvah trenutnega klasifikatorja (error sampling) ali pa na nedoločenosti odločitve klasifikatorja (t.i. uncertainty sampling), kjer je kriterij izbire razlika $P_{c_1}(x) - P_{c_2}(x)$ med najvišje ocenjenima verjetnostma v porazdelitvi, ki jo vrne trenutni klasifikator.

Zanimiv pristop, ki nekoliko spominja na tehnike inverzne transdukcije, je uporabljen v algoritmu GODA [66] (angl. **G**oal-**O**riented **D**ata **A**cquisition). Primere z manjkajočimi vrednostmi obravnavamo po vrsti, tako da jih dodajamo v učno množico (ki že vsebuje nekaj popolnih primerov), pri čemer manjkajoče vrednosti ocenimo z modelom, zgrajenim na popolnih primerih. Za tiste primere, ki pri tem najbolj povečajo kvaliteto klasifikatorja, računano na vseh podatkih, predvidimo izvedbo testov za pridobitev manjkajočih vrednosti.

2.1.5 Cena učitelja

Upoštevanje težavnosti (cene) označevanja primerov se imenuje *aktivno učenje* (angl. active learning) in ga po predstavljeni klasifikaciji obravnavamo kot obliko cenovno občutljivega učenja. Izraz “aktivno” ni najbolj primeren, ker zajema vse načine učenja, kjer ima učenec določen vpliv na učne podatke. Tako bi lahko med aktivno učenje prištevali tudi aktivno izbiro testov, vendar je ta v literaturi strogo ločena [50, 51]. Kot praktične primere uporabe tehnik aktivnega učenja lahko navedemo navigacijo mobilnih robotov, prepoznavo govora in pisave, analizo naravnih jezikov in kategorizacijo tekstov (npr. tistih, ki so dostopni na spletu).

Cilj aktivnega učenja je zgraditi primerno natančen model, pri čemer pa označeni podatki niso zastonj, zato upamo, da bomo z inteligentno in prilagodljivo strategijo potrebovali manj podatkov kot bi jih sicer pri običajnem, "pasivnem" učenju. Intuitivno lahko sklepamo [47], da peščica informativnih primerov koristi bolj kot naključna množica.

Uporabnost aktivnega učenja je v literaturi običajno utemeljena s preprostim primerom [12]:

Primer 2.1: Imamo podatke, ki ležijo na premici v ravnini, klasifikatorji pa so preproste pragovne funkcije $H = \{h_w : w \in \mathbb{R}\}$, $h_w(\boldsymbol{x}) = 1$ za $\boldsymbol{x} \geqslant w$ in $h_w(\boldsymbol{x}) = 0$ za $\boldsymbol{x} < w$. Če je porazdelitev podatkov takšna, da obstaja popoln klasifikator (meja med pozitivnimi in negativnimi primeri), je za mejo točnosti *e* dovolj izbrati

$$n = O\left(\frac{1}{e}\right)$$

naključnih primerov in zgraditi klasifikator, ki je z njimi konsistenten. Če pa izberemo *n* neoznačenih primerov, pa za zagotovitev meje točnosti *e* potrebujemo le

$$n = O\left(\log(\frac{1}{e})\right)$$

oznak primerov, ker si lahko pomagamo z binarnim iskanjem.

V opisanem primeru nam aktivno učenje zmanjša potrebno število označitev primerov za eksponentni faktor, v splošnem pa stvari niso tako enostavne. Izkaže se [12], da že v dvodimenzionalni različici primera 2.1 obstajajo hipoteze, kjer je potrebnih

$$n = \Omega\left(\frac{1}{e}\right)$$

primerov ne glede na način učenja.

Zahtevnost nadzorovanega učenja običajno izrazimo kot funkcijo točnosti *e* in verjetnostne porazdelitve *P*. Pri aktivnem učenju pa je potrebno upoštevati tudi ciljno hipotezo in število neoznačenih primerov [12]. S. Dasgupta je dokazal, da pri določenih lastnosti prostora hipotez velja, da ta nujno vsebuje hipoteze, ki so za aktivno učenje neugodne (težke). Nadaljnjih teoretičnih osnov aktivnega učenja ne bomo podrobneje obravnavali, ker z vidika koristnostnega podatkovnega rudarjenja niso zanimive in tudi daleč presegajo okvir tega dela [12, 11, 2].

Ločimo dva glavna pristopa k aktivnemu učenju:

- konstrukcija vprašanj in
- selektivno jemanje vzorcev (angl. selective sampling).

V primeru konstrukcije vprašanj učenec (model) sestavi umeten primer tako, da bo odgovor o njegovem razredu kar najbolj informativen. V praksi se pri takšnem pristopu srečamo s težavo, da je konstruiran primer izjemno težko ali tudi nemogoče klasificirati [8]. Algoritem za prepoznavo pisave Bauma in Langa, temelječ na nevronskih mrežah, je tako za označitev sestavil primere, ki za eksperta niso imeli pomena in jih je bilo nemogoče klasificirati. Razširitve tako dopuščajo tudi nepopolna vprašanja, na katera je dovoljeno odgovoriti z "ne vem" ter zavajajoča vprašanja, pri katerih je odgovor lahko napačen [45].

Pri selektivnem jemanju vzorcev, ki je popularnejši pristop [45], proces učenja začnemo z (majhno) množico označenih in veliko množico neoznačenih primerov. Osnovna shema postopka je prikazana na sliki 2.1.

Vhod *:*
$-$*učni algoritem* $\boldsymbol{A}$
$-$*množica označenih primerov* $\boldsymbol{L}$
$-$*množica neoznačenih primerov* $\boldsymbol{U}$
$-$*ustavitveni kriterij* $\boldsymbol{k}$
$-$*koristnostna funkcija* $\boldsymbol{Utility} : X \times H \to \mathbb{R}$
Izhod *:*
$-$*oznake izbranih primerov*

while $\neg \boldsymbol{k}$:
  $h = \boldsymbol{A}(\boldsymbol{L})$
  $q = \underset{u \in U}{argmax}\ \boldsymbol{Utility}(u, h)$
  remove q from $\boldsymbol{U}$
  ask for $label(q)$
  add q to $\boldsymbol{L}$
end

Slika 2.1: Osnovna oblika postopka selektivnega jemanja vzorcev.

Kot rezultat dobimo povečano množico *L*, ki vsebuje poleg začetnih označenih primerov še tiste, katere je funkcija $\boldsymbol{Utility}(\cdot, \cdot)$ ocenila kot najboljše. Glede na konstrukcijo hipoteze *h* (trenutni model), ločimo dve vrsti učenja: *ena sama hipoteza* in *več hipotez* (t.i. komite, angl. comitee-based). Kadar upoštevamo eno samo hipotezo, večinoma uporabljamo eno od

naslednjih dveh funkcij koristnosti:

- *zmanjšanje nedoločenosti*. Za označitev predlagamo tisti primer, pri katerem je odločitev trenutnega klasifikatorja najbolj nedoločena. Spomnimo se, da enako tehniko uporabljamo tudi pri aktivni izbiri testov.
- *minimiziranje pričakovane napake*. Tu izbira primera temelji na maksimizaciji pričakovanega zmanjšanja napake pri klasifikaciji. Izbrati hočemo tisti primer, katerega uporaba po označitvi bo minimizirala napako bodočega klasifikatorja. Za nekatere trivialne klasifikatorje je sicer to mogoče, v splošnem pa si moramo pomagati s hevristično oceno zmanjšanja napake.

Če pa upoštevamo več hipotez (množica trenutnih klasifikatorjev oz. komite), predlagamo označitev tistega primera, pri katerem so napovedi najbolj raznolike (kar je pravzaprav enakovredno zmanjšanju nedoločenosti, ker ima takšen primer najmanj zanesljivo označitev).

Pristop selektivnega jemanja vzorcev je soroden delno nadzorovanemu učenju (angl. semi-supervised learning), vendar le-to ni interaktiven proces [47]. Neoznačeni primeri so tu le dodatek, s katerim lahko izboljšamo točnost in splošnost modela (ali pa tudi poslabšamo). Uporabimo jih npr. za zmanjšanje pristranskosti, če označeni učni primeri niso bili izbrani naključno.

Omenimo še pristop s t. i. *filtriranjem* (angl. querry-filtering), ki je soroden selektivnemu jemanju vzorcev, le da imamo namesto množice na voljo tok neoznačenih primerov, sestavljen in naključno izbranih primerov vhodne populacije, pri čemer se o morebitni označitvi odločamo sproti. Velja, da je tak način v splošnem inferioren glede na selektivno jemanje vzorcev, ker lahko pri slednjem uporabimo večjo količino informacij (v vsakem trenutku imamo na voljo množico neoznačenih primerov).

Posebna oblika aktivnega učenja in delno nadzorovanega učenja uporablja t.i. *metodo različnih pogledov* (angl. multiple views), kjer pogled na primer predstavlja podmnožico njegovih atributov. Algoritem Co-training Bluma in Mitchella [9, 44] uporablja dva pogleda in v množico označenih primerov dodaja tiste primere, katerih oznake trenutna klasifikatorja iz obeh pogledov določita z največjo gotovostjo. Poleg praktične uporabnosti je zanimiv tudi dokaz [9], da se ciljnega koncepta za problem z dvema pogledoma (dva atributa in razred) lahko naučimo na nekaj označenih primerih in množici neoznačenih, če sta pogleda združljiva

in nekorelirana [45, 9] (posplošeni dokaz velja tudi za delno združljiva pogleda). Pri tem združljivost pomeni, da ciljni model vse primere v obeh pogledih označi enako, nekoreliranost pa je predpostavka o neodvisnosti obeh atributov pri danem razredu. Algoritem Co-training v procesu aktivnega učenja uporabimo tako, da za označitev predlagamo tisti primer, pri katerem se napovedi klasifikatorjev iz različnih pogledov najbolj razlikujejo. Avtor I. Muslea je takšen pristop poimenoval Co-testing [45].

V nekaterih primerih pa ne želimo najtočnejšega klasifikatorja, pač pa čim boljšo oceno porazdelitve verjetnosti po razredih. Tipičen primer so učni problemi, kjer imamo neenake cene napačnih klasifikacij. Tudi v splošnem velja, da je dobra ocena porazdelitve verjetnosti po razredih bolj zaželena kot pa napoved le enega razreda, četudi je slednja nekoliko točnejša. Obstajajo metode aktivnega učenja, s katerimi lahko pridobimo kvalitetne oceno porazdelitve po razredih, pri tem pa je potrebnih manj označenih učnih primerov. Kot primere takšnih pristopov lahko navedemo algoritem Bootstrap-LV [53] in uporabo Jensen-Shannonove divergence [46] (posplošitev Kullback-Leiblerjeve divergence na več porazdelitev) pri izbiranju primerov za označitev.

2.1.6 Združevanje različnih vrst cen

Kot smo omenili v uvodnem poglavju, je cilj koristnostnega podatkovnega rudarjenja upoštevanje vseh koristnostnih faktorjev (običajno cen), ki nastopajo v procesu podatkovnega rudarjenja. Takšna analiza je pogosto zapletena, zato ne preseneča, da je večina raziskav omejenih izključno na eno samo vrsto cene. Predstavljena sta dva pristopa, ki v enotnem ogrodju združujeta različne vrste cen.

6.A Skupna koristnost klasifikatorja

G. M. Weiss in Y. Tian pri *skupni koristnosti klasifikatorja* [59] upoštevata tako ceno uporabe klasifikatorja (cena klasifikacijskih napak) kot tudi ceno učnih primerov. Če je npr. klasifikator A nekoliko slabši kot klasifikator B, toda občutno cenejši, je po predstavljenem merilu boljši. Izhajamo iz enačbe:

$$skupna\ cena = cena\ podatkov + cena\ napake$$

Predpostavljamo obstoj ciljne množice *S*, na kateri bomo uporabili zgrajeni model. Cena napake na ciljni množici temelji na številu napak na ciljni množici, ki jih lahko ocenimo s točnostjo na učni množici, pomnoženo z velikostjo ciljne množice (predpostavljamo, da je učna množica reprezentativen primerek). Velikost ciljne množice *S* je v praksi sicer običajno neznana, vendar jo ekspert lahko dovolj natančno oceni. Poznamo še *n* (velikost učne množice) in *e* (stopnjo napačne klasifikacije). Privzemamo tudi konstantno ceno pridobitve posameznega učnega primera (C_{tr}) in konstantno ceno napake na ciljni množici (C_{err}). Z navedenimi predpostavkami lahko skupno ceno definiramo kot:

$$skupna\ cena = n \cdot C_{tr} + e \cdot |S| \cdot C_{err}$$

kjer so vrednosti C_{tr}, C_{err} in |S| odvisne od problemske domene. V praksi bi se lahko pri tako definirani meri koristnosti pojavila prevlada drugega člena zaradi faktorja |S|, vendar je po mnenju avtorjev to netipično, ker je cena učnih podatkov pogosto zelo pomembna, poleg tega pa množica *S* ni vedno velika.

Predstavljena mera skupne koristnosti je zanimiva, vendar ima še več pomankljivosti oz. omejitev. Ceni pridobitve učnega primera in napake sta konstantni, kot cenovni faktor ni upoštevan čas računanja, poleg tega pa je za uspešno uporabo v praksi potrebno specializirano znanje o domeni. Med najzanimivejše rezultate raziskave lahko štejemo ugotavljanje optimalne velikosti učne množice glede na množico vnaprej določenih razmerij med C_{tr} in C_{err} (npr. 1:1, 1:20, … , 1:50000) in empirični dokaz, da lahko celotno koristnost klasifikatorja izboljšamo, ne da bi uporabili vse dostopne učne podatke.

6.B Aktivno cenovno občutljivo učenje

Aktivno cenovno občutljivo učenje (angl. active cost-sensitive learning) [40] predstavlja združitev cene učitelja in cene napačne klasifikacije. Cilj je minimizirati celotno ceno, ki jo izračunamo kot vsoto cene označevanja učnih primerov (cena učitelja) ter cene izgube, povezane z odločitvami. Predstavljeni algoritem ACTIVE-CSL na vhodu zahteva množico označenih primerov S_l, množico neoznačenih primerov S_u, cenovno matriko *C*, cenovno funkcijo označevanja L, učni algoritem in ustavljalni kriterij. Koraki algoritma so sledeči:

1. v prvem koraku ocenimo verjetnosti razredov $P(j|\boldsymbol{x})$ za vse neoznačene primere s pomočjo množice S_l in učnega algoritma;

2. izberemo primer, ki naj ga klasificira ekspert in sicer tako, da informacija o njegovem pravem razredu največ prispeva v smislu skupne cene (cena označevanja + cena napačne klasifikacije). Pri tem je pomembno, da sta matrika C in funkcija L oblikovani tako, da dražji primeri označitve s strani eksperta bolj vplivajo na zmanjšanje cene napačne klasifikacije kot pa cenejši primeri (kar je povsem realističen predpostavka);
3. označeni primer dodamo v množico S_l;
4. če ustavljalnemu kriteriju še ni zadoščeno, se vrnemo na korak 1, sicer pa končamo. Rezultat je optimalna klasifikacija izbranih primerov glede na matriko C in verjetnosti P.

Pri uporabi v praksi velja še posebno pozornost posvetiti izbiri učnega algoritma, s katerim ocenimo verjetnosti razredov.

Kot primer uporabe aktivnega cenovno občutljivega učenja lahko navedemo klasifikatorje za ugotavljanje vdorov v računalniška omrežja. Pri gradnji takšnih modelov označene učne primere pridobimo z analizo dogodkov v omrežju s strani eksperta. Naključno izbiranje primerov za označitev tipično vodi v visoke stroške (potrebnih je veliko primerov), upoštevati pa moramo tudi neenako ceno napačnih klasifikacij. Če poskusa vdora ne zaznamo, lahko to vodi v različne kritične situacije. Očitno je, da potrebujemo inteligenten pristop h gradnji modelov, ki upošteva tako cene označitev kot tudi cene napačnih klasifikacij.

2.2 Iskanje koristnih n-teric

Področje iskanja koristnih n-teric se je razvilo iz področja iskanja povezovalnih (asociativnih) pravil. Slednja se uporabljajo pri analizi velikih podatkovih baz (iskanje vzorcev, relacij), izvorno tistih, ki nastajajo pri transakcijah v trgovinskih podjetjih (t.i. market-basket analysis), s primernimi posplošitvami pa tudi poljubnih diskretnih [25] in v določenih primerih zveznih podatkih [55]. S povezovalnimi pravili sicer lahko odkrijemo potencialno zanimive zakonitosti, vendar imajo v smislu koristnostnega podatkovnega rudarjenja določene pomanjkljivosti (npr. v primeru transakcijskih baz trgovinskih podjetij), saj ne upoštevajo najpomembnejšega faktorja: cene. Pri iskanju koristnih n-teric zato v vsaki transakciji upoštevamo tudi različne cene in količine prisotnih predmetov (artiklov, atributov). Rezultat procesa iskanja koristnih n-teric je sicer manj splošen, ker namesto pravil pridobimo le

množice predmetov, vendar v koristnostnem smislu primernejši, saj ni le statistično pomemben, ampak tudi neposredno uporaben v poslovnem procesu.

V tem podpoglavju najprej podajamo kratek opis povezovalnih pravil, idejno podlago za tehnike iskanja koristnih n-teric. Podrobno so predstavljene teoretične osnove iskanja najkoristnejših n-teric in shema učinkovitega iskalnega algoritma. Opisana je tudi posebna oblika koristnih n-teric, koristne pogoste n-terice.

Omeniti velja še resno težavo glede izrazoslovja. Raziskovalci področja iskanja koristnih n-teric kot enakovredna uporabljajo dva izraza: rudarjenje koristnih n-teric [32, 62] (angl. high utility itemset mining) in rudarjenje koristi [65, 64] (angl. utility mining). Ker je slednji močno podoben izrazu koristnostno podatkovno rudarjenje in pomensko nepotrebno preobsežen, se mu bomo v naši obravnavi izognili.

2.2.1 Povezovalna pravila

Iskanje povezovalnih pravil je pomebno področje opisnega podatkovnega rudarjenja. Prvi učinkovit iskalni algoritem, imenovan APriori, so predstavili Agrawal in sod. [1]. Ta je bil sprva omejen na pravila s sklepim delom dolžine 1 (klasifikacijska oblika pravila), kasneje pa posplošen na sklepne dele poljubne dolžine [3].

Najpogostejše področje uporabe povezovalnih pravil so že omenjene podatkovne baze velikih trgovin, kjer lahko pridobljeno znanje uporabimo npr. za načrtovanje prodaje, urejanje zalog, analizo navad kupcev ipd. Tudi v naši obravnavi se bomo omejili le na takšno obliko podatkov, ker je iskanje koristnih n-teric prilagojeno za obravnavo kvantitativnih vrednosti (s privzetkom “več je bolje”). Še enkrat poudarimo, da za povezovalna pravila te omejitve ne veljajo, saj lahko poljubne podatke (diskretne in zvezne) prevedemo v binarno obliko, primerno za uporabo algoritma APriori [25]. Področja uporabe tako vključujejo medicinsko diagnostiko, bioinformatiko, rudarjenje na spletu in raziskavo obnašanja obiskovalcev internetnih strani ter analizo različnih znanstvenih podatkov.

Povezovalno pravilo formalno definiramo takole:

Definicija 2.4: Povezovalno pravilo je izraz oblike $X \rightarrow Y$, kjer sta X in Y neprazni in tuji množici predmetov (oz. v splošnem parov oblike $atribut = vrednost$).

Zanimivost povezovalnega pravila običajno merimo s *podporo* in *zaupanjem* (angl. support, confidence). Podpora je delež transakcij v katerih se pojavijo vsi predmeti, nastopajoči v X in Y, zaupanje pa delež transakcij med transakcijami z X, ki vsebujejo tudi Y.

$$podpora(X \rightarrow Y) = \frac{|\{T \mid X \cup Y \subseteq T,\ T \in DB\}|}{|DB|}$$

$$zaupanje(X \rightarrow Y) = \frac{|\{T \mid X \cup Y \subseteq T,\ T \in DB\}|}{|\{T \mid X \subseteq T,\ T \in DB\}|}$$

Obe meri sta objektivni, saj temeljita na statističnih zakonitostih v podatkih. Mera podpore ima tudi lastnost antimonotonosti, ki je temelj algoritma APriori, saj omogoča učinkovito omejevanje iskalnega prostora.

Definicija 2.5: Naj bo A množica vseh predmetov in J potenčna množica množice I. Funkcija f je antimonotona, če velja: $\forall X, Y \in J: \ X \subseteq Y \Rightarrow f(Y) \leqslant f(X)$.

Omenimo še, da obstajajo številne druge objektivne mere [55], ki prav tako temeljijo na statistiki, vendar jih ne bomo posebej opisovali, ker za našo obravnavo niso zanimive. V smislu koristnostnega podatkovnega rudarjenja so pomembnejše subjektivne mere (presenetljivost, neobičajnost) in koristnostne mere, ki jih Geng in Hamilton [18] klasificirata kot semantične.

Razlike med merami lahko ponazorimo s preprostim primerom:

Primer 2.2: V tabeli 2.3 je prikazana majhna podatkovna baza s petimi transakcijami, ki podaja fiktivne nakupe petih kupcev v neki trgovini. Tabela je binarna (količina 0 ali 1) in v t.i. redki (angl. sparse) obliki, pri kateri so v vsaki transakciji navedeni samo kupljeni predmeti in ne vsi obstoječi (s količinami 0, ker niso bili kupljeni).

Tabela 2.3: Primer majhne binarne podatkovne baze.

ID transakcije	predmeti
1	kruh, mleko, maslo
2	kruh, jajca, pivo, plenice
3	kokakola, mleko, pivo, plenice
4	kruh, mleko, pivo, plenice
5	kruh, mleko, plenice, salama

Z analizo podatkovne baze lahko ugotovimo npr. sledeči pravili: $kruh \rightarrow mleko$ in $plenice \rightarrow pivo$. V obeh primerih velja: podpora = 0.6, zaupanje = 0.75. Prvo pravilo je po subjektivnih merilih nezanimivo, ker se ujema z nakupovalnimi navadami večine kupcev in ga lahko ugotovimo tudi brez analize asociacij. Mnogo zanimivejše je drugo pravilo, saj odkriva nepričakovan vzorec, ki mu le s težavo določimo vzrok. Na podlagi objektivnih mer sta torej zanimivi obe pravili (ustrezno velika podpora in zaupanje), na podlagi subjektivnih mer, ki vključujejo tudi uporabnikovo znanje, pa le drugo. Kot zanimivost omenimo, da naj bi bilo drugo pravilo ugotovljeno pri analizi realnih podatkov trgovske verige Wal-mart. Ukrep, ki naj bi ga po tej ugotovitvi sprejeli v podjetju, je bilo prestrukturiranje postavitve obeh artiklov in posledično povečanje dobička (oz. vsaj zadovoljstva nekaterih kupcev, ki jim je bilo prihranjeno iskanje). Pristnost tega primera je sicer nekoliko sporna [42], vendar pa nazorno prikazuje, kako je mogoče s tehnikami odkrivanja zakonitosti v podatkih povečati dobiček.

2.2.2 Koristne n-terice

Postopek iskanja povezovalnih pravil je sestavljen iz dveh faz:

1. iskanje pogostih n-teric
2. generiranje povezovalnih pravil na podlagi pogostih n-teric

Prva faza je idejno sorodna procesu iskanja koristnih n-teric, saj v obeh primerih iščemo množice predmetov z želenimi lastnostmi. Rezultat te faze je množica n-teric (množica množic predmetov), ki imajo podporo večjo ali enako dani spodnji meji. Proces iskanja pogostih n-teric poteka v več stopnjah, pri čemer uporabimo lastnost antimonotonosti, zaradi katere lahko generiramo kandidate naraščajočih dolžin, ki so sestavljeni iz kandidatov prejšnje stopnje. Kot bomo videli, pa pogoste n-terice niso nujno tudi koristne (in obratno), funkcija, s katero računamo korist n-teric, pa ni antimonotona. To pomeni, da ne moremo uporabiti postopka iskanja pogostih n-teric, kjer bi mero podpore nadomestili z mero koristi. Izkaže pa se, da je mogoče sestaviti funkcijo za omejevanje množice kandidatov, ki ima lastnost antimonotonosti, čeprav je v primerjavi z mero podpore precej ohlapna in zato mnoge kandidate precenjuje.

Razliko med pogostimi in koristnimi n-tericami lahko ponazorimo na primeru 3.1, če vpeljemo še podatke o dobičku za vsak predmet:

Tabela 2.4: Dobički pri prodaji predmetov

predmet	**kruh**	**mleko**	**maslo**	**jajca**	**plenice**	**kokakola**	**salama**	**pivo**
dobiček	2	2	3	4	4	6	6	6

Na podlagi tabele 2.4 je n-terica {pivo} ob enaki podpori v tabeli 2.3 koristnejša od n-terice {kruh, mleko}, saj prinaša večji dobiček ($3*6 > 3*(2+2)$).

Za potrditev, da objektivne mere, temelječe na statistiki, niso vedno zadovoljiv pokazatelj koristnosti, navajamo tri definicije:

Definicija 2.6: Vzorec je koristen za uporabnika, če njegova uporaba prispeva k dosegu zastavljenega cilja [65].

Definicija 2.7: Vzorec je zanimiv le tedaj, ča ga lahko uporabimo v odločitvenem procesu tako, da povečuje korist [30].

Definicija 2.8: Zanimivost vzorca = verjetnost vzorca + korist [54].

Zapisane definicije so splošne in veljajo za vsakršne oblike vzorcev, v našem primeru pa jih bomo uporabili pri opredelitvi koristnih n-teric. Mera za ocenjevanje vzorcev, ki temelji na koristi, torej poleg statističnih lastnosti vzorca upošteva tudi subjektivno definirano korist. Očitno je, da za izračun koristi neke n-terice potrebujemo koristnostno funkcijo in podatke o koristi posameznih elementov n-terice. Formalno to lahko zapišemo takole:

Definicija 2.9: Zunanja korist predmeta i_p je numerična vrednost y_p, ki jo določi uporabnik. Je neodvisna od transakcije in običajno odraža dobiček tega elementa. Zunanje koristi hranimo v *tabeli koristi* (ang. utility table).

Definicija 2.10: Notranja korist elementa i_p je numerična vrednost x_p in je odvisna od transakcije. Običajno jo opredelimo kot količino elementa v transakciji.

Definicija 2.11: Funkcija koristnosti f je funkcija dveh spremenljivk:

$$f(x, y): (\mathbb{R}^+, \mathbb{R}^+) \rightarrow \mathbb{R}^+ .$$

Na podlagi teh definicij lahko formalno opredelimo korist nekega predmeta v transakciji,

korist n-terice v transakciji, korist elementa v n-terici, skupno korist n-terice v podatkovni bazi, korist transakcije, korist celotne podatkovne baze in delež skupne koristnosti, ki jo prispeva neka n-terica.

Definicija 2.12: Korist elementa i_p v transakciji T je: $u(i_p, T) = f(x_p, y_p),\ \ i_p \in T$.

Definicija 2.13: Korist n-terice S v transakciji T je: $u(S, T) = \sum_{i_p \in S} u(i_p, T),\ \ S \subseteq T$.

Definicija 2.14: Korist elementa i_p v n-terici S je: $u(i_p, S) = \sum_{T \in PB,\ S \subseteq T} u(i_p, T)$.

Definicija 2.15: Skupna korist (oz. okrajšano korist) n-terice S v podatkovni bazi PB je:

$$u(S) = \sum_{T \in PB,\ S \subseteq T} u(S, T) = \sum_{T \in PB,\ S \subseteq T} \sum_{i_p \in S} f(x_p, y_p).$$

Definicija 2.16: Korist transakcije T je: $u(T) = \sum_{i_p \in T} u(i_p, T)$.

Definicija 2.17: Korist podatkovne baze PB je: $u(PB) = \sum_{T \in PB} u(T)$.

Definicija 2.18: Procentualni delež koristnosti n-terice S v podatkovni bazi PB je:

$$U(S) = \frac{u(S)}{u(PB)}.$$

Potrebno je omeniti, da obstaja veliko mer zanimivosti, temelječih na koristnosti, vendar za vse veljajo enačbe v zgornjih definicijah, če primerno definiramo [65] funkcijo koristnosti $f(\cdot, \cdot)$. V naši obravnavi bomo uporabljali obliko $f(x_p, y_p) = x_p * y_p$, kar pomeni, da korist definiramo nad notranjo in zunajo koristjo v obliki produkta [63]. Zavedati se moramo, da je izbira mere koristnosti odvisna od obravanega primera – izberemo tisto mero, ki nam za dani primer najbolj ustreza.

Na podlagi definicij 2.9 – 2.18 lahko formalno definiramo (visoko) koristno n-terico in problem iskanja koristnih n-teric.

Definicija 2.19: N-terica S je (visoko) koristna, če velja: $U(S) \geq minUtil$, kjer je *minUtil* prag koristnosti v procentih glede na korist celotne podatkovne baze.

Definicija 2.20: Problem iskanja vseh koristnih n-teric je problem iskanja množice *H*, ki je

definirana takole: $H = \{S \mid S \subseteq I,\ u(S) \geq minUtil\}$, I je množica predmetov (atributov).

Za lažje razumevanje navajamo nekaj izračunov koristnosti na nekoliko večji podatkovni bazi, ki vključuje tudi različne količine kupljenih artiklov:

Primer 2.3: Dana je podatkovna baza PB z desetimi transakcijami in petimi artikli [62]. Zunanje koristi so navedene v tabeli 2.5 in predstavljajo čisti dobiček v denarnih enotah pri prodaji posameznih artiklov.

Tabela 2.5: Podatkovna baza z desetimi transakcijami in različnimi količinami kupljenih artiklov.

TID	A	B	C	D	E
1	0	0	18	0	1
2	0	6	0	1	1
3	2	0	1	0	1
4	1	0	0	1	1
5	0	0	4	0	2
6	1	1	0	0	0
7	0	10	0	1	1
8	3	0	25	3	1
9	1	1	0	0	0
10	0	6	2	0	2

Tabela 2.6: Zunanje koristi artiklov iz tabele 3.3.

artikel	dobiček v €
A	3
B	10
C	1
D	6
E	5

Primeri izračunov:

1. Korist n-terice {B,E} v transakciji 2:
   $u(\{B, E\},\ T_2) = u(\{B\},\ T_2) + u(\{E\},\ T_2) = 6*10 + 1*5 = 65$

2. Korist transakcije 10:
$$u(T_{10}) = u(\{B\},\ T_{10}) + u(\{C\},\ T_{10}) + u(\{E\},\ T_{10}) = 6*10 + 2*1 + 2*5 = 72$$

3. Korist celotne podatkovne baze:
$$u(PB) = u(T_1) + \ldots + u(T_{10}) = 23 + \ldots + 72 = 400$$

4. Skupna korist n-terice {A, E}:
$$u(\{A, E\}) = u(\{A, E\},\ T_3) + u(\{A, E\},\ T_4) + u(\{A, E\},\ T_8) = 11 + 8 + 14 = 33$$

5. Delež koristnosti n-terice {A, E}:
$$U(\{A, E\}) = \frac{33}{400} = 0.0825 = 8.25\%$$

6. Skupna korist n-terice {A, D, E}:
$$u(\{A, D, E\}) = u(\{A, D, E\},\ T_4) + u(\{A, E\},\ T_8) = 14 + 32 = 46$$

7. Delež koristnosti n-terice {A, D, E}:
$$U(\{A, D, E\}) = \frac{46}{400} = 0.115 = 11.5\%$$

8. Množica koristnih n-teric pri omejitvi $minUtil \geq 20\%$:
$$\{\{B\}, \{B, D\}, \{B, E\}, \{C, E\}, \{B, D, E\}\}$$

S protiprimerom lahko dokažemo, da ne velja lastnost antimonotonosti:
$$\{A, E\} \subseteq \{A, D, E\},\ u(\{A, E\}) \leq u(\{A, D, E\})\ .$$

Prav tako ne velja lastnost monotonosti:
$$\{B\} \subseteq \{B, C\},\ u(\{B\}) \geq u(\{B, C\})$$

Teoretične osnove učinkovitega iskanja koristnih n-teric

Pri iskanju koristnih n-teric imamo na voljo naslednje možnosti (po kronološkem vrstnem redu):

1. izčrpno preiskovanje vseh možnih n-teric;
2. uporaba hevrističnih ocen;

3. omejevanje števila kandidatov s pomočjo kritične funkcije.

Z izčrpnim preiskovanjem množice kandidatov sicer odkrijemo vse koristne n-terice, vendar je takšen postopek časovno in prostorsko zelo potraten. Pri *n* artiklih moram torej preveriti vseh $2^n - 1$ n-teric, kar že za majhne vrednosti *n* predstavlja neobvadljiv problem ($n \approx 30$). Primera takšnih algoritmov sta ZP in ZSP [5, 7].

Pri iskanju koristnih n-teric si lahko pomagamo tudi s hevurističnimi ocenami kandidatov, seveda pa takšne ocene ne zagotavljajo, da bomo našli vse koristne n-terice. Kot primere uporabe hevurističnih ocen lahko navedemo algoritme SIP, CAC, IAB, MEU [5, 6, 7, 32, 64]. Po navedbah avtorjev se le-ti v praksi sicer dobro obnesejo, vendar pa je njihova teoretična utemeljitev pogosto nezadostna. Hamilton in Barber navajata [6], da CAC in IAB odkrijeta "večino" koristnih n-teric. Pomembno izboljšavo omenjenih algoritmov predstavlja vpeljava *pragovnega faktorja*, s katerim lahko povečamo verjetnost odkritja najkoristnejših n-teric z obravnavo večjega števila kandidatov (in obratno).

Najučinkovitejši način iskanja vseh koristnih n-teric je uporaba t.i. *kritične funkcije* (angl. critical function), s pomočjo katere lahko učinkovito omejujemo število kandidatov v postopku iskanja. Izraz kritična funkcija prvič zasledimo v članku, Lija, Yeha in Changa [37], v katerem opisujejo algoritem FSM za iskanje vseh koristnih n-teric. Enakovredna, toda še nekoliko bolj omejujoča (in zato boljša) funkcija je uporabljena v dvofaznem algoritmu Liuja, Liaoja in Choudaryja. Slednja je uporabljena tudi v dveh najhitrejših algoritmih, ShFSM in DCG, ki sta podrobneje obravnavana v tretjem poglavju.

Potrebno je omeniti, da se koristi v algoritmih DCG in FSM (ter vseh izboljšavah: EFSM, SuSFM, ShFSM) računajo z nekoliko drugače definirano koristnostno funkcijo, kot je uporabljena v naši obravnavi. Definirana je samo na podlagi notranjih koristi, ker pa je po lastnostih enaka (pravzaprav je poseben primer naše funkcije, kjer so vse zunanje koristi enake 1), lahko algoritme uporabljamo brez dodatnih sprememb.

Da bi lahko formalno zapisali kritično funkcijo, potrebujemo naslednjo definicijo:

Definicija 2.21: Za dano n-terico S je podatkovna baza $PB_S \subseteq PB$ definirana takole:

$$PB_S = \{T \mid \{S \subseteq T \wedge T \in PB\}\} .$$

PB_S je torej podatkovna baza s tistimi transakcijami, ki vsebujejo n-terico *S*. Sedaj lahko

definiramo kritično funkcijo:

Definicija 2.22: Za dano n-terico *S* je kritična funkcija *CF(S)* enaka vrednosti $u(PB_S)$.

Trditev 2.1: Imamo podatkovno bazo PB in omejitev $minU = minUtil * u(PB)$. Velja: $u(PB_S) < minU \Rightarrow \forall X \supseteq S: u(X) < minU$.

Z besedami: če korist podatkovne baze PB_S ne zadošča pogoju minimalne koristi, tudi n-terica S in vse njene nadmnožice ne zadoščajo pogoju minimalne koristi. Dokaz je enostaven:

Dokaz trditve 2.1: Imamo podatkovno bazo *PB*, n-terico *S* in omejitev *minU*. Naj velja $u(PB_S) < minU$. Iz definicij 2.15, 2.17 in 2.21 sledi: $u(S) \leq u(PB_S)$. Torej za vsak $X \supseteq S$ velja $u(X) \leq u(PB_X)$ in po definicijah 2.17 ter 2.21 $u(PB_X) \leq u(PB_S)$. Torej velja: $u(X) \leq u(PB_S)$ in od tod z upoštevanjem predpostavke: $u(X) < minU$. □

Dokaz trditve 2.1 je tudi dokaz antimonotonosti kritične funkcije. Velja namreč: $\forall X, Y \in J: \; X \subseteq Y \Rightarrow CF(Y) \leqslant CF(X)$, kjer je *J* potenčna množica množice predmetov v podatkovni bazi.

Lastnost antimonotonosti nam zagotavlja obstoj enostavnega in učinkovitega postopka eliminiranja neobetavnih kandidatov. Zavedati pa se moramo, da je kritična funkcija iz definicije 2.22 le groba ocena koristnosti nadmnožic, kar pomeni, da je število kandidatov med postopkom iskanja še vedno veliko. Ta ugotovitev velja ne glede na dejansko število koristnih n-teric pri dani meji koristnosti in je posledica oblike funkcije koristnosti.

Osnovna shema vsakega algoritma za iskanje koristnih n-teric, ki temelji na antimonotoni kritični funkciji, je takšna:

Vhod *:*
− podatkovna baza PB
− omejitev ***minUtil***
Izhod *:*
− vse koristne n − terice

1. *generiraj začetno množico kandidatov*
2. *v množici kandidatov poišči morebitne koristne n − terice*
3. *generiraj novo množico kandidatov za ena večje dolžine iz stare množice s pomočjo kritične funkcije in generatorskega postopka*
4. *če je nova množica neprazna, se vrni na korak 2, sicer končaj*

Slika 2.2: Osnovna shema postopka iskanja koristnih n-teric.

Zaradi lastnosti antimonotonosti kritične funkcije je postopek podoben iskanju pogostih n-teric (prva faza iskanja povezovalnih pravil). Izvajanje se ustavi, ko je množica novih kandidatov prazna (po uporabi kritične funkcije) oz. ko iz množice kandidatov s pomočjo generatorskega postopka ne moremo pridobiti nobenega novega kandidata več.

2.2.3 Koristne pogoste n-terice

Z omenjenima algoritmoma ShFSM in DCG lahko v dani podatkovni bazi poiščemo vse koristne n-terice. Ker pa pri iskanju upoštevamo le omejitev koristnosti, ne pridobimo informacije o pogostosti (podpori) najdenih n-teric. Posebna oblika iskanja n-teric tako poleg subjektivno definirane koristi upošteva tudi statistične zakonitosti v podatkih, kar je v skladu z definicijo 2.8.

Podpodročje iskanja koristnih pogostih n-teric (angl. utility-frequent itemset mining) je definiral J. S. Yeh [62], ki tudi navaja prvi iskalni algoritem TP (angl. **T**wo **P**hase). Ta je nekoliko podrobneje opisan v tretjem poglavju.

Če želimo definirati koristno pogosto n-terico, potrebujemo formalno definicijo razširjene oblike podpore, ki vključuje tudi korist:

Definicija 2.23: $podpora(S,\ \mu) = \frac{|\tau_{S,\mu}|}{|DB|} = \frac{|\{T \mid S \subseteq T \wedge u(S,T) \geq \mu \wedge T \in DB\}|}{|DB|}$.

Podpora iz zgornje definicije predstavlja delež transakcij, ki vsebujejo n-terico S in v katerih je korist S večja od dane spodnje meje μ. Pravimo, da je n-terica S koristna in pogosta, če velja: $podpora(S,\ \mu) \geq s$, kjer je s minimalna zahtevana podpora.

Tako definirana podpora nima lastnosti monotonosti, saj vsebuje ne-monotono funkcijo koristnosti $u(\cdot,\cdot)$. To lahko dokažemo s protiprimerom:

Primer 2.4: Naj bo PB podatkovna baza iz primera 2.3. Pri omejitvah
$\mu = 5\% = 20$, $podpora = 20\%$ je n-terica {C, E} koristna in pogosta, n-terici {C} in {C, D, E} pa ne. Velja: $\tau_{\{C\},\mu} = \{T_8\}$, $\tau_{\{C,\ E\},\mu} = \{T_1,\ T_8\}$, $\tau_{\{C,\ D,\ E\},\mu} = \{T_8\}$
$podpora(\{C\},\ \mu) = 10\%$, $podpora(\{C,\ D,\ E\},\ \mu) = 10\%$ in
$podpora(\{C,\ E\},\ \mu) = 20\%$.

2.2.4 Uporabnost tehnik iskanja koristnih n-teric

Področje iskanja koristnih n-teric predstavlja nadgradnjo področja iskanja pogostih n-teric. Slednje ima solidno teoretično podlago in je v kombinaciji s iskanjem povezovalnih pravil uporabno na različnih področjih. Glede vrste in oblike podatkov pri tem ni posebnih zahtev, saj lahko povezovalna pravila in podskupine iščemo v poljubnih diskretnih podatkih. Metode iskanja koristnih n-teric pa postavljajo precejšnje omejitve. V naši obravnavi definicija koristi in koristne n-terice predpostavlja numerične vrednosti atributov in zunanjih koristi s predpostavko "več je bolje". Uporaba implementiranih tehnik iskanja koristnih n-teric je zato vezana izključno na podatkovne baze, kjer atributi predstavljajo artikle, vrednosti atributov pa prodane količine (gre za t.i. posplošene binarne transakcijske podatkovne baze). Glede razvoja teorije iskanja koristnih n-teric lahko zapišemo, da je že dosegla vrh in pomembnejših izboljšav ne moremo več pričakovati. Edina smer, v kateri so še mogoče izboljšave, je optimiziranje kritične funkcije in posledična pohitritev opisanih algoritmov. Končni cilj področja iskanja koristnih n-teric je seveda uporaba v praksi, npr. v poslovnem procesu trgovinskih podjetij. Za podjetja v splošnem velja, da je pravilna postavitev prodajne cene najhitrejši in najuspešnejši način doseganja kar največjega dobička [43], saj lahko primerno postavljena cena poveča dobiček hitreje, kot pa bi to dosegli s povečanjem obsega poslovanja. Ker so prav koristne n-terice tiste množice produktov iz ponudbe, ki prinašajo največji dobiček (večji od dane spodnje meje), je potrebno nameniti posebno pozornost določanju njihove največje, za kupce še sprejemljive cene.

Kljub tej potencialni uporabnosti še ni mogoče zaslediti konkretnih primerov uporabe. Vzrok lahko pripišemo dejstvu, da je področje iskanja koristnih n-teric dokaj novo, uspešen prenos tehnik podatkovnega rudarjenja iz teorije v prakso pa je v splošnem dolgotrajen proces.

3. Algoritmi za učinkovito iskanje vseh koristnih n-teric

V drugem poglavju smo omenili, da učinkoviti algoritmi za iskanje vseh koristnih n-teric temeljijo na uporabi kritične funkcije. Osnovna oblika vseh je enaka tisti na sliki 2.2, se pa razlikujejo v načinu generiranja kandidatov. Kot bomo videli, je slednje v določenih primerih ključnega pomena za hitrost implementacije.

V tem poglavju je najprej opisana posebna podatkovna struktura, ki je v vseh opisanih algoritmih uporabljena za učinkovito računanje koristi in pogostosti n-teric. Predstavljena sta najhitrejša algoritma za iskanje koristnih n-teric, ShFSM in DCG ter dva načina generiranja kandidatov in primerjava med njima. Kratko je opisan dvofazni algoritem TP za iskanje koristnih in pogostih n-teric ter nov, izboljšan algoritem (FUFM), ki ima številne prednosti.

3.1 Učinkovito računanje koristi n-teric

Iz definicije 2.15 je razvidno, da moramo za izračun koristi n-terice S poznati vse transakcije, ki vsebujejo n-terico S. V splošnem to pomeni, da je potrebno pregledati vse transakcije in uporabiti primerno obliko funkcije $subset(\cdot,\cdot)$, da ugotovimo, ali je n-terica vsebovana v transakciji. Izkaže pa se, da lahko v primeru zaporednega računanja koristi več enako dolgim n-tericam (npr. v točki 2 v osnovnem algoritmu na sliki 2.2) uporabimo posebno podatkovno strukturo – razpršeno drevo (angl. hash tree ali tudi trie), s katerim lahko računanje občutno pohitrimo. Pri računanju podpore n-tericam (prva faza iskanja povezovalnih pravil) so ga uporabili že Agrawal in sod. [3]. Uporabljena oblika razpršenega drevesa ima naslednje lastnosti:

1. vsi elementi so shranjeni v listih in so enake dolžine. Predpostavljamo, da je element takšne oblike, da njegove komponente lahko naslavljamo (npr. seznam, niz).
2. komponente elementov so urejene, npr. po abecednem redu. Vsi elementi se razlikujejo vsaj v eni komponenti (v drevesu ni duplikatov).

3. notranja vozlišča so razpršene tabele s kazalci na potomce;
4. največje število elementov v listih je *k*. Ko postane število elementov preveliko, list spremenimo v notranje vozlišče.

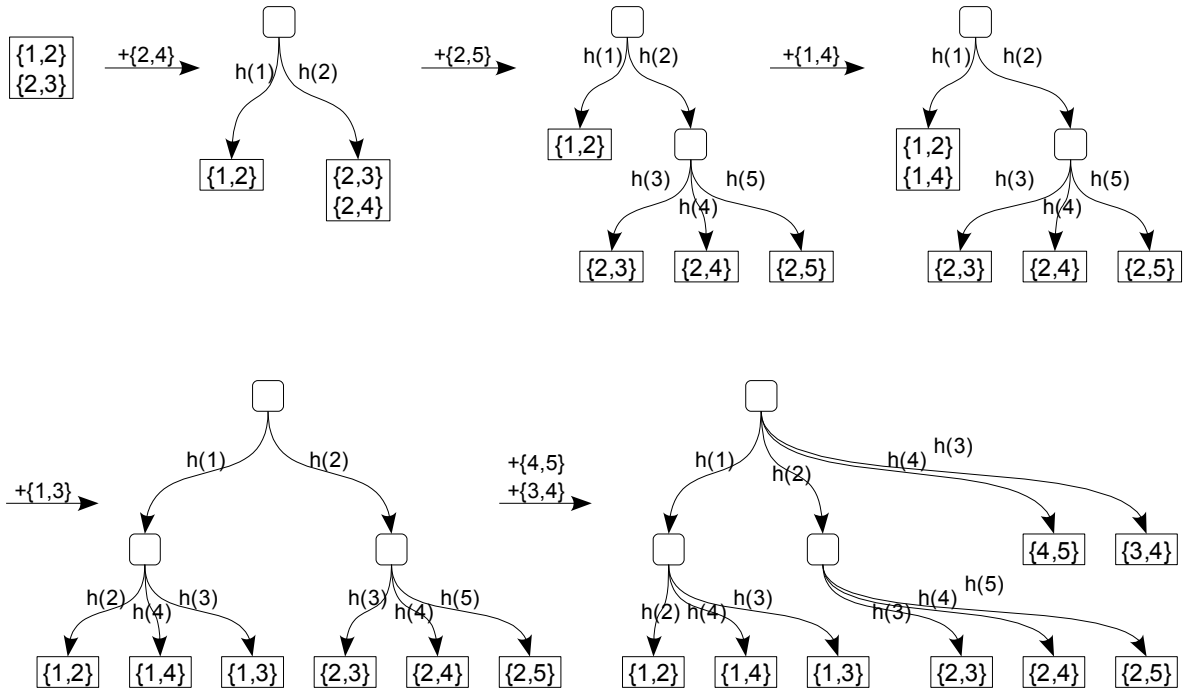
Pri gradnji razpršenega drevesa je ključnega pomena razpršilna funkcija, ki jo uporabljamo tako pri gradnji kot pri iskanju. V naši implementaciji, ki je v celoti napisana v programskem jeziku Python, je za ta namen uporabljena vgrajena funkcija $hash(\cdot)$, notranja vozlišča pa so podatkovne strukture tipa slovar, ki so interno (v interpreterju jezika Python) realizirane z razpršenimi tabelami. Omenjena funkcija $hash(\cdot)$ je hitra in kvalitetna (dobro razpršuje), morebitna moteča lastnost pa je, da za enaka števila različnih podatkovnih tipov (integer, long float) vrača isto vrednost (npr. $hash(1) = hash(1.0)$). Ker pa so v našem primeru oznake elementov vedno nizi, ne prihaja do težav. Funkcija ni injektivna, vendar to za gradnjo in ohranjanje strukture razpršenega drevesa tudi ni potrebno. Velja pa, da dva različna elementa ne smeta imeti vseh komponent enakih glede na funkcijo $hash(\cdot)$, ker sta sicer s stališča funkcije enaka, to pa je v nasprotju z lastnostjo 1, navedeno zgoraj.

Gradnja razpršenega drevesa poteka po naslednjih pravilih:

1. začnemo s praznim drevesom, kjer je koren drevesa hkrati list in se nahaja na globini 1;
2. ko dodajamo nov element v drevo na globini *d*, gremo v poddrevo, v katerega nas vodi rezultat funkcije $hash(\cdot)$, izračunan na d-ti komponenti novega elementa;
3. nov element dodamo v list, če velikost lista še ni presegla meje (če takega lista še ni, ga ustvarimo);
4. če velikost lista preseže mejo, ga spremenimo v notranje vozlišče (razpršeno tabelo), v katero kot ključe vstavimo rezultate funkcije $hash(\cdot)$, izračunane na d-tih komponentah elementov v bivšem listu (*d* je globina bivšega lista).

Postopek gradnje razpršenega drevesa prikažimo na majhnem primeru:

Primer 3.1: Imamo dane n-terice {{1,2}, {2,3}, {2,4}, {2,5}, {1,4}, {1,3}, {4,5}, {3,4}}, ki jih želimo urediti v razpršeno drevo, pri čemer je število elementov v listih omejeno na 2. Izračunane vrednosti funkcije $hash(\cdot)$ so na sliki prikazane okrajšano kot $h(\cdot)$. Postopek gradnje je prikazan na sliki 3.1.



Slika 3.1: Gradnja razpršenega drevesa iz primera 3.1.

Razpršena drevesa v postopku iskanja koristnih n-teric uporabimo v vsaki iteraciji osnovnega algoritma na sliki 2.2 pri računanju koristi. Vse trenutne kandidate (n-terice) uredimo v razpršeno drevo, potem pa na drevesu "uporabimo" vse transakcije. Z "uporabo" je mišljeno določanje kandidatov, ki so vsebovani v transakciji. Postopek poteka takole:

1. če se nahajamo v listu, s funkcijo $subset(\cdot,\cdot)$ preverimo, kateri elementi v listu so vsebovani v transakciji;
2. če se nahajamo v notranjem vozlišču, izvedemo rekurzivni klic za vsako komponento transakcije z indeksom, večjim od indeksa trenutno obravnavane komponente;
3. če se nahajamo v korenu drevesa, izvedemo rekurzivni klic za vse komponente transakcije;
4. z rekurzivnim klicem obiščemo poddrevo, v katerega kaže vrednost $hash(\cdot)$, izračunana na obravnavani komponenti transakcije.

Ker v našem primeru za vsakega kandidata potrebujemo seznam indeksov transakcij, v katerih je vsebovan, vsakemu elementu v listu dodamo še seznam indeksov lastnih transakcij. S tem

seznamom postane računanje koristi n-terice učinkovito.

"Uporabo" transakcije na razpršenem drevesu prikažimo na drevesu iz primera 3.1:

Primer 3.2: Naj bo dana transakcija $T = \{1,2,3,5\}$. Rekurzivni klici pri uporabi na drevesu iz primera 3.1 se kopičijo takole:

use(T, root):
find(1, *root*)
find(2, *root*[1])
found {1,2}
find(3, *root*[1])
found {1,3}
find(5, *root*[1])
return
find(2, *root*)
find(3, *root*[2])
found {2,3}
find(5, *root*[2])
found {2,5}
find(3, *root*)
return
find(5, *root*)
return

Če pri računanju koristi ne uporabljamo razpršenega drevesa, v vsaki iteraciji pri n transakcijah s povprečno dolžino t in k kandidatih z dolžino d potrebujemo v povprečju $n*k*d*\log_2 t$ enostavnih operacij (predpostavljamo urejenost elementov transakcije). Z razpršenim drevesom pa potrebujemo približno $k*d + n*t*d$ enostavnih operacij (gradnja + "uporaba"). Prihranek je očiten, ker sta največja faktorja v produktu, n in k, sedaj ločena. Če velja npr. $n=10000$, $t=15$, $k=500$, $d=4$, imamo brez drevesa 78 137 812 operacij, z drevesom pa le 602 000. Navedene formule so samo grob približek, vendar pa zadoščajo za občutek o dejanski pohitritvi.

3.2 Algoritmi za učinkovito iskanje vseh koristnih n-teric

Vsi najučinkovitejši algoritmi za iskanje koristnih n-teric v osnovi temeljijo na opisani shemi splošnega postopka (algoritem 2.2). Podrobneje bomo obravnavali dva: ShFSM [38] in DCG [39], dvofaznega algoritma [32] pa zaradi podobnosti ShFSM ne bomo opisali. Poudarimo, da

ta dvofazni algoritem ni enak dvofaznemu algoritmu za iskanje koristnih in pogostih n-teric, opisanem v tem poglavju.

3.2.1 Algoritem ShFSM

Algoritem ShFSM je najučinkovitejša izpeljanka algoritma FSM [37] (angl. **F**ast **S**hare **M**easure). Uporablja kritično funkcijo iz definicije 2.22, ki je še dodatno izboljšana tako, da jo za poljubno n-terico S dolžine k definiramo kot $u(PB_{S^{k+1}})$, kjer indeks $k+1$ pomeni, da upoštevamo le tiste transakcije, ki vsebujejo vsaj eno nadmnožico n-terice S, dolgo k+1. Povedano preprosteje, pri računanju izboljšane kritične funkcije iz PB_S izpustimo tiste transakcije, ki so enako dolge kot S. Brez težav se lahko prepričamo, da še vedno velja lastnost antimonotonosti. Potrebno pa je upoštevati, da za tako definirano kritično funkcijo velja: $u(PB_{S^{k+1}}) < minU \Rightarrow \forall X \supset S: u(X) < minU$, kar pomeni, da lahko pride do primera (zaradi stroge vsebovanosti S v X), kjer $u(S) \geq minUtil \wedge CF(S) < minUtil$. Takšne n-terice S dodamo med najdene koristne n-terice, ne dodamo pa jih v množico kandidatov (po kritični funkciji iz definicije 2.22 bi jih ocenili višje in dodali tudi v množico kandidatov). Opis algoritma ShFSM je podan na sliki 3.2.

```
Algoritem ShFSM ()
Vhod:
 − podatkovna baza DB z množico predmetov I
 − meja koristnosti minUtil
Izhod:
 − vse n-terice s koristjo ≥ minUtil

      /* priprava začetnih množic */
[1]   k = 1; C_1 = I;
[2]   RC_1 = {i | i ∈ C_1 ∧ CF(i) ≥ minUtil}
[3]   H_1 = {i | i ∈ C_1 ∧ u(i) ≥ minUtil}
[4]   while RC_k ≠ Ø:
[5]       k = k + 1; H_k = Ø;
[6]       C_k = generateCandidates(RC_{k-1})
          /* obhod podatkovne baze (računanje koristi kandidatov) */
[7]       foreach T ∈ DB:
[8]           compute u(X) ∀ X ∈ C_k
          /* med kandidati poiščemo koristne n − terice
             in zavržemo neobetavne kandidate */
[9]       foreach X ∈ C_k:
[10]          if u(X) ≥ minUtil
[11]              H_k.append(X)
[12]          if CF(X) ≥ minUtil:
[13]              RC_k.append(X)
[14]  return H_1 ∪ H_2 ∪ ... ∪ H_k
```

Slika 3.2: Opis algoritma ShFSM. C_k – množica kandidatov dolžine k, RC_k – množica preostalih kandidatov dolžine k, H_k – množica (visoko) koristnih n-teric dolžine k.

Zaradi razumljivosti je v opisu izpuščena uporaba razpršenih dreves. Le-to bi v vsaki iteraciji while zanke zgradili v vrsticah 7-8, kjer je v zgornjem opisu nakazan obhod podatkovne baze, med katerim računamo koristi kandidatov. Imena spremenljivk v opisu so okrajšani angleški izrazi.

Posebej velja omeniti funkcijo generiranja kandidatov $generateCandidates(\cdot)$. Ta je lahko standardna funkcija $APrioriGen(\cdot)$, kot so jo definirali Agrawal in sod. [3], lahko pa uporabimo nekoliko drugačno obliko, uporabljeno v algoritmu EFSM [38] (izboljšan FSM). Obe sta sestavljeni iz dveh korakov:

1. generiranje nove množice kandidatov z združevanjem trenutnih kandidatov (angl. join

step);

2. odstranjevanje nepravih novih kandidatov (angl. prune step).

Drugi korak je pri obeh funkcijah enak, prvi pa se razlikuje. V funkciji $APrioriGen(\cdot)$ generiramo nove kandidate z združevanjem tistih trenutnih kandidatov, katerih komponente se ujemajo do predzadnjega mesta. Pri generiranju novih kandidatov v algoritmu EFSM pa trenutne kandidate dopolnjujemo s tistimi elementi množice posameznih artiklov v podatkovni bazi, ki so večji od največje komponente trenutnega kandidata. Funkciji sta prikazani na slikah 3.3 in 3.4. Obe na vhodu zahtevata urejen seznam urejenih kandidatov enake dolžine.

```
APrioriGen(Candidates):
    newCandidates = ∅
    foreach X ∈ Candidates:
        foreach Y ∈ Candidates ∧ Y > X:
            L = len(X)
            if ∀ i ∈ {1, ..., L−1}: X[i] == Y[i] ∧ X[L] < Y[L]:
                newCandidate = {X[1], ..., X[L−1], Y[L]}
                newCandidates.append(newCandidate)

    remainingNewCandidates = prune(Candidates, newCandidates)
    return remainingNewCandidates
```

Slika 3.3: Standardni (APriori) način generiranja kandidatov.

```
EFSMGen(Candidates, PB.items):
    newCandidates = ∅
    foreach X ∈ Candidates:
        L = len(X)
        foreach item ∈ PB.items ∧ item > X[L]:
            newCandidate = {X[1], ..., X[L−1], item}
            newCandidates.append(newCandidate)

    remainingNewCandidates = prune(Candidates, newCandidates)
    return remainingNewCandidates
```

Slika 3.4: EFSM način generiranja kandidatov.

Odstranjevanje nepravih kandidatov v obeh primerih poteka s preverjanjem vseh podmnožic dolžine *k-1* množice novih kandidatov (*k* je dolžina novih kandidatov) in je prikazano na sliki 3.5.

```
prune(Candidates, newCandidates):
    remainingNewCandidates = Ø
    for newCandidate ∈ newCandidates:
        if ∀ S ∈ allSubsets_{k-1}(newCandidate) ∈ Candidates:
            remainingNewCandidates.append(newCandidate)

    return remainingNewCandidates
```

Slika 3.5: Postopek odstranjevanja kandidatov.

Primerjava zahtevnosti obeh načinov generiranja je dokaj zapletena, saj število novih kandidatov, dobljenih v prvi fazi, določa zahtevnost druge faze. Zahtevnosti posameznih faz v grobem ocenimo takole:

1. **faza 1**: *m* kandidatov
    1. APrioriGen: v $\frac{(m^2-m)}{2} * \bar{k}$ korakih dobimo največ $n \leq \frac{(m^2-m)}{2}$ novih kandidatov ($\bar{k}$ je povprečna dolžina kandidatov glede na vse klice funkcije)
    2. EFSM način: v $m * |PB.items|$ korakih dobimo $n = m * |PB.items|$ novih kandidatov
2. **faza 2**: *n* novih kandidatov
    1. APrioriGen: največ $\frac{(m^2-m)}{2} * (k-1) * \log_2(m)$ korakov
    2. EFSM način: $m * |PB.items| * (k-1) * \log_2(m)$ korakov

V praksi se izkaže, da sta hitrosti obeh funkcij dokaj podobni, saj večjo časovno zahtevnost prve faze funkcije $APrioriGen(\cdot)$ odtehta manjše število kandidatov, ki jih obravnavamo v drugi fazi. Podobno pri EFSM načinu generiranja manjšo zahtevnost prve faze poslabša večje število kandidatov za drugo fazo. V naši implementaciji uporabljamo EFSM način generiranja kandidatov, ker je bila s poskusi ugotovljena nekoliko večja hitrost.

3.2.2 Algoritem DCG

Algoritem DCG uporablja kritično funkcijo iz definicije 2.22, ki je v primerjavi s kritično funkcije algoritma ShFSM nekoliko slabša. Razlikuje se tudi v načinu generiranja kandidatov, saj ne uporabljamo več faze združevanja in faze odstranjevanja, ampak kandidate generiramo sproti: za vsakega trenutnega kandidata S dolžine k vzdržujemo polje števcev koristnosti novih, za ena daljših kandidatov S_{k+1}, ki jih dobimo z dodajanjem vseh elementov v transakcijah T_S, večjih od največjega elementa v S (predpostavljamo urejenost kandidatov in transakcij). Na ta način sicer dobimo večje število potencialnih kandidatov, vendar pa se izognemu zamudnima fazama združevanja in odstranjevanja, pa tudi kandidati obstajajo le kot indeksi ne pa kot dejanske n-terice. Vrednosti kritične funkcije enako kot pri algoritmu ShFSM učinkovito izračunamo z razpršenim drevesom.

Na tem mestu je potrebno opozoriti, da avtorji algoritma DCG napačno navajajo [39], da je število kandidatov manjše kot pri algoritmu ShFSM. Velja, da je začetno število kandidatov (števcev) večje, po uporabi kritične funkcije pa enako kot v algoritmu ShFSM, če v le-tem uporabimo isto kritično funkcijo. Kot smo omenili, v algoritmu DCG pravzaprav kandidatov ne generiramo eksplicitno, klub temu pa moramo zanje pripraviti prostor v velikosti števca koristnosti, torej lahko rečemo, da jih v nekem smislu generiramo (oz. obiščemo).

Testiranja so pokazala [39], da je algoritem DCG nekoliko hitrejši kot algoritem ShFSM (tipično nekaj sekund) v primeru implementacije v enem od nižjenivojskih jezikov. V našem primeru pa se zaradi posebnosti jezika Python razlika poveča za faktor 3 in več. Zaključimo lahko, da je uporaba algoritma DCG smiselna takrat, ko število števcev za potencialne kandidate ne presega velikosti glavnega pomnilnika in v primeru posebnosti uporabljenega programskega jezika. Podrobnejši opis algoritma DCG je podan na sliki 3.6.

Tudi tukaj smo v opisu izpustili uporabo razpršenih dreves. Koristi kandidatov bi z razpršenim drevesom računali v vrstici 9, zgradili pa bi ga v vrstici 3. Vrstice 4-6 nakazujejo sprehod skozi podatkovno bazo, v implementaciji pa so učinkoviteje realizirane z uporabo informacije o lastnih transakcijah vsakega kandidata, ki se nahaja v razpršenem drevesu. Prav tako kot v opisu algoritma ShFSM so imena spremenljivk okrajšani angleški izrazi. Opozoriti velja, da v algoritmu DCG kritične funkcije ne kličemo eksplicitno, ker se njena vrednost akumulira v vrstici 7. Manjša izboljšava, ki ni prisotna v opisu [39] algoritma DCG in na sliki 3.6, je odstranitev kandidatov S dolžine 1 iz začetne množice C_1, za katere velja

$CF(S) < minUtil$. Z njo algoritem nekoliko upočasnimo zaradi dodatnih operacij (tipično nekaj sekund), vendar pa služi za lažjo primerjavo z algoritmom ShFSM, ker z njo dosežemo enako število začetnih kandidatov (razen v primeru obstoja transakcij z enim samim artiklom zaradi razlike v kritičnih funkcijah). Zato je bila uporabljena tudi v naši implementaciji.

Algoritem $DCG()$
***Vhod**:*
- *podatkovna baza* DB *z množico predmetov* I
- *meja koristnosti* ***minUtil***

***Izhod**:*
- *vse* n*-terice s koristjo* $\geq minUtil$

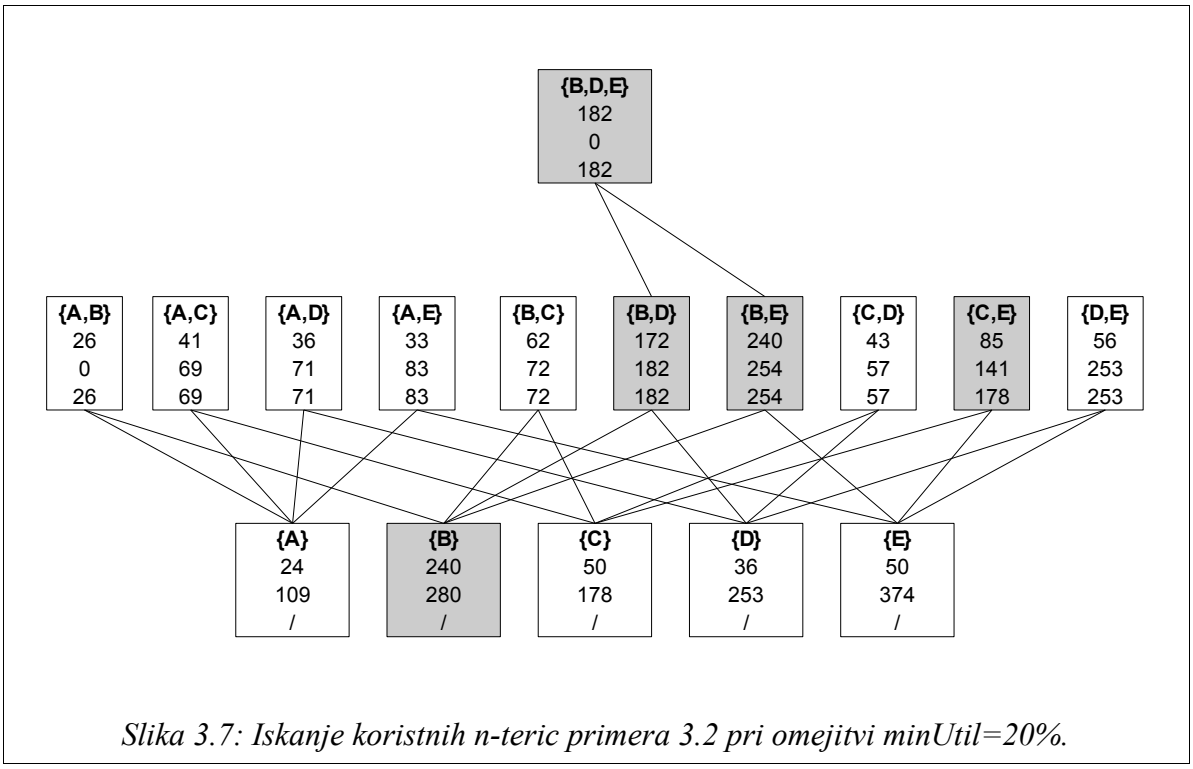
```
[1]   k=1; C_1=I; ∀T∈DB compute u[T]
[2]   while C_k ≠ Ø:
          /* priprava množic ter števcev koristi novih kandidatov */
[3]       H_k=Ø; C_{k+1}=Ø; ∀X∈C_k: u[DB_{X^{i_q}_{k+1}}]=0
          /* obhod podatkovne baze (računanje kritične funkcije) */
[4]       foreach T∈DB:
[5]           foreach X∈C_k:
[6]               foreach i_q > i_k ∧ i_q∈T ∧ X⊂T:
[7]                   u[DB_{X^{i_q}_{k+1}}] += u[T]
          /* med kandidati poiščemo koristne n−terice,
             med novimi kandidati pa zavržemo neobetavne */
[8]       foreach X∈C_k:
[9]           if u(X) ≥ minUtil:
[10]              H_k.append(X)
[11]          foreach i_q > i_k ∧ i_q∈T ∧ X⊂T:
                  /* vrednost kritične fukncije za n−terico
                  X^{i_q}_{k+1} je enaka u[DB_{X^{i_q}_{k+1}}] */
[12]              if u[DB_{X^{i_q}_{k+1}}] ≥ minUtil:
[13]                  C_{k+1}.append(X^{i_q}_{k+1})
[14]      k=k+1
[15]  return H_1∪H_2∪...∪H_k
```

Slika 3.6: Opis algoritma DCG. C_k – množica kandidatov dolžine k, H_k – množica (visoko) koristnih n-teric dolžine k.

Algoritem DCG je mogoče prilagoditi na izboljšano kritično funkcijo, vendar v tem primeru naletimo na težavo zaradi možnosti obstoja n-terice S z lastnostjo:

$u(S) \geq minUtil \wedge CF(S) < minUtil$. Takšne n-terice bi v primeru uporabe izboljšane kritične funkcije v vrstici 12 izpadle iz množice kandidatov za naslednjo iteracijo. To pa pomeni, da jih ne bi dodali v množico najdenih koristnih n-teric, saj jih v množici, obravnavani v vrstici 8 v naslednji iteraciji, ne bi bilo. Poleg kritične funkcije bi torej morali izračunati še koristi vseh kandidatov ter tiste z opisano lastnostjo prenesti v naslednjo iteracijo, kjer bi jih v vrstici 9 brez preverjanja pogoja dodali množici najdenih koristnih n-teric te iteracije. Tako spremenjen algoritem je nekoliko počasnejši kot osnovni, saj izboljšava kritične funkcije ne odtehta dodatnih računskih operacij.

Delovanje obeh algoritmov prikažimo na podatkovni bazi iz primera 2.3, pri čemer mejo koristnosti postavimo na 20%, kar pomeni, da mora biti korist n-terice večja ali enaka $20\% * u(DB) = 20 * 400 = 80$, da jo smatramo kot (visoko) koristno. Na sliki 3.7 so v okvirjih prikazane n-terice, ki jih obravnavamo med procesom iskanja z algoritmoma ShFSM in DCG. Obravnavane n-terice so v našem primeru enake pri obeh algoritmih (kar v splošnem ne drži), se pa razlikujejo vrednosti kritičnih funkcij.



Slika 3.7: Iskanje koristnih n-teric primera 3.2 pri omejitvi minUtil=20%.

Okvirji, ki so temno obarvani, vsebujejo n-terice, ki izpolnjujejo pogoj koristnosti. Pri tem je

zgornja vrednost v vsakem okvirju korist n-terice, vrednost v sredini rezultat kritične funkcije algoritma ShFSM, spodnja vrednost pa je kritična funkcija algoritma DCG. Slednji se v obravnavanem primeru razlikujeta pri n-tericah {C, E} in {B, D, E}. Algoritem ShFSM se po izračunu kritične funkcije za n-terico {B, D, E} konča, DCG pa izvede še poslednjo iteracijo, v kateri pa iz edinega kandidata {B, D, E} ne more pridobiti nobenega novega kandidata več in se zato ustavi. Iz tabele 2.5 in obeh definicij kritične funkcije je razvidno, zakaj se ShFSM ustavi eno iteracijo prej. Transakcije 2, 7 in 10, ki vsebujejo n-terico {B, D, E}, ne vsebujejo nobenega drugega elementa: izboljšana kritična funkcija koristi nadmnožic pravilno oceni z 0 (nadmnožic sploh ni). Kritična funkcija algoritma DCG pa te tri transakcije upošteva v izračunu kritične funkcije in kratkovidno predpostavlja obstoj nadmnožic.

3.2.3 Dvofazni algoritem za iskanje koristnih pogostih n-teric

V drugem poglavju smo omenili algoritem TP za iskanje koristnih pogostih n-teric. Sestavljen je iz dveh faz in dokazano poišče vse koristne pogoste n-terice, ki zadoščajo danim omejitvam. Temelj algoritma je monotona funkcija kvazi podpore, ki omogoča strukturirano preiskovanje prostora:

Definicija 3.1:

$$kvazi\ podpora(S,\ \mu) = \frac{|\tau'_{S,\mu}|}{|DB|} = \frac{|\{T \mid \exists\, X \subseteq S:\ X \subseteq T \wedge u(X,T) \geq \mu \wedge T \in DB\}|}{|DB|}.$$

Tako definirana kvazi podpora se razlikuje od razširjene podpore iz definicije 2.23 v tem, da vsebovanost n-terice *S* v transakciji ni več obvezna – zadostuje že delna vsebovanost. Dokaz monotonosti je enostaven.

Monotonost kvazi podpore: Naj bo *PB* podatkovna baza, *J* potenčna množica množice njenih artiklov, *X* in *Y* pa n-terici, za kateri velja: $X \subseteq Y;\ X, Y \subseteq J$. Iz definicije 3.21 je očitno, da je $kvazi\ podpora(Y,\ \mu)$ vedno večja ali enaka $kvazi\ podpora(X,\ \mu)$, saj je $|\tau'_{Y,\mu}| \geq |\tau'_{X,\mu}|$ (ker *Y* vsebuje vse podmnožice, ki jih vsebuje *X*, lahko pa tudi takšne, ki jih v *X* ni). Ker sklep velja za poljubni n-terici $X,\ Y;\ X \subseteq Y$, je kvazi podpora monotona. □

Očitno je, da je vsaka koristna pogosta n-terica S tudi kvazi pogosta in koristna, saj vedno velja $kvazi\ podpora(S,\ \mu) \geq podpora(S,\ \mu)$.

Na navedenih ugotovitvah temelji omenjeni dvofazni algoritem. V prvi fazi poiščemo vse kvazi pogoste in koristne n-terice, v drugi fazi pa odstranimo tiste, ki niso koristne in pogoste (ta faza predstavlja le sprehod skozi podatkovno bazo). Algoritem je dokaj preprost, vendar pa ima določene lastnosti, zaradi katerih je neprimeren za implementacijo v programskem jeziku Python. Zaradi lastnosti monotonosti kandidate generiramo po padajočih dolžinah, torej n-terice v vsaki iteraciji razgradimo na za ena krajše podmnožice, med katerimi izločimo tiste, ki ne zadoščajo pogoju $kvazi\ podpora(\cdot, \mu) \geq s$ oz. imajo kakšno kvazi nepogosto nadmnožico (lastnost monotonosti). Preostale gredo kot kandidati v naslednjo iteracijo. Generiranje in izločanje podmnožic je zamudno in občutno upočasni iskanje.

3.2.4 Algoritem FUFM za iskanje koristnih pogostih n-teric

Učinkovitejši algoritem temelji na preprosti ugotovitvi, da je razširjena podpora vedno manjša ali enaka "navadni" podpori. Očitno je, da to drži, saj pri navadni podpori štejemo vse transakcije, ki vsebujejo dano n-terico, pri razširjeni pa le tiste med njimi, v katerih ima n-terica dovolj veliko korist. Na podlagi te ugotovitve lahko sestavimo enostaven in hiter iskalni algoritem, katerega osnovo predstavlja iskanje pogostih n-teric. Ta lastnost je glavna privlačnost našega novega algoritma, saj so algoritmi za iskanje pogostih n-teric dobro raziskani in tudi zelo učinkoviti, npr. APriori [1], FP drevesa [21], ECLAT [67], Relim [10], Partition [52]. V naši implementaciji je bil zaradi preprostosti uporabljen APriori način iskanja pogostih n-teric, ki je bil učinkovito realiziran z razpršenim drevesom.

Shema novega iskalnega algoritma, ki ga bomo imenovali FUFM (**F**ast **U**tility-**F**requent **M**ining), je podana na sliki 3.8.

Algoritem FUFM ( )
Vhod *:*
− *podatkovna baza PB*
− *omejitvi* ***minUtil*** *in* ***minSup***
Izhod *:*
− *množica vseh pogostih in koristnih n − teric*

1. *generiraj začetno množico pogostih kandidatov s podporo* ≥ *minSup*
2. *med njimi poišči morebitne koristne in pogoste n − terice*
3. *generiraj novo množico kandidatov za ena večje dolžine iz stare množice pogostih kandidatov s pomočjo generatorskega postopka*
4. *če je nova množica neprazna, se vrni na korak 2, sicer končaj*

Slika 3.8: Opis algoritma novega algoritma FUFM za iskanje vseh koristnih in pogostih n-teric.

3.2.5 Primerjava algoritmov TP in FUFM

Algoritma TP in FUFM se razlikujeta tudi v številu obravnavanih kandidatov. V splošnem ni mogoče enostavno določiti, kateri bo generiral več kandidatov, intuitivno pa lahko sklepamo, da bo v večini primerov to algoritem TP, saj kandidate obravnava po padajoči dolžini, zato bo v začetnih iteracijah prve faze večina kandidatov imela visoko vrednost kvazi podpore in se bo zato prenesla v naslednjo iteracijo. Posledično je potrebno tudi več časa za preverjanje podmnožic. Nasprotno pa v je v algoritmu FUFM število kandidatov odvisno le od minimalne podpore. Kadar bo le-ta zelo nizka, lahko sicer pričakujemo veliko kandidatov, vendar se problem prenese na (zunanji) algoritem iskanja pogostih n-teric. Iskanje koristnih pogostih n-teric med kandidati pa ostane učinkovito, saj lahko uporabimo razpršena drevesa (pri algoritmu TP jih uporabimo šele v drugi fazi).

Omenimo še, da lahko pri iskanju koristnih pogostih n-teric uporabimo tudi algoritme za iskanje koristnih n-teric, npr. ShFSM in DCG, saj je njihov rezultat pri enaki omejitvi *minUtil* vedno nadmnožica rezultata algoritmov TP in FUFM. Med koristnimi n-tericami, ki jih najdeta ShFSM in DCG, moramo poiskati tiste, ki zadoščaju danemu pogoju minimalne razširjene podpore. Težava, na katero naletimo, je ta, da je dobljeno število koristnih n-teric zelo veliko, saj je prag minimalne koristi pri iskanju pogostih koristnih n-teric definiran glede na transakcijo, pri iskanju koristnih n-teric pa glede na celotno bazo. Če torej minimalno

zahtevano korist n-terice v transakciji izrazimo kot delež celotne koristi podatkove baze, dobimo zelo majhno število (še posebej pri velikih podatkovnih bazah). Posledično je v vsaki iteraciji algoritmov ShFSM in DCG veliko kandidatov in tudi veliko koristnih n-teric (med slednjimi pa je seveda le malo pogostih in koristnih).

4. Implementacija algoritmov in rezultati testiranj

V tem poglavju so predstavljeni rezultati, dobljeni z izvajanjem algoritmov za iskanje koristnih n-teric na sintetičnih podatkovnih bazah. Te so pridobljene z dvema generatorjema podatkov: IBM Quest in LUCS-KDD ARM. Kratko so opisane nekatere posebnosti implementacije algoritmov iskanja koristnih n-teric ter parametri obeh generatorjev. Opisan je tudi pretvornik podatkov, ki podatkovne baze, pridobljene s navedenima generatorjema, pretvori v obliko, primerno za uporabo v okolju Orange. Rezultati testiranj so predstavljeni grafično in tabelarično, pri čemer je pozornost posvečena času izvajanja ter spreminjanju množic kandidatov in koristnih n-teric.

4.1 Generatorji podatkov

V uvodnem poglavju smo omenili, da posebno težavo pri izvajanju algoritmov iskanja koristnih n-teric predstavlja pomanjkanje prosto dostopnih podatkov, ki bi vključevali tudi zunanje koristi (dobiček). To dejstvo ni presenetljivo, saj je predvsem tabela zunanjih koristi tista, v kateri so zbrane kritične informacije o poslovanju podjetja in kot takšna predstavlja poslovno skrivnost. Raziskovalci so si zato prisiljeni pomagati z generatorji podatkov in koristi, ki bolj ali manj uspešno poskušajo oponašati obnašanje kupcev. V nadaljevanju podajamo kratek opis kompleksnejšega generatorja IBM Quest in preprostejšega LUCS-KDD ARM.

4.1.1 Generator podatkov IBM Quest

Generator IBM Quest je nastal v okviru projekta Quest v raziskovalnem centru IBM Almaden. Predstavitev in prva uporaba je opisana v članku [3] R. Agrawala in R. Srikanta, kjer so generirani podatki uporabljeni za iskanje povezovalnih pravil. Pri implementiranju generatorja so bile upoštevane zakonitosti, ki jih lahko zasledimo v realnih podatkovnih

bazah. Najpomembnejša predpostavka je, da kupci artikle kupujejo v množicah, vsaka takšna (velika) množica pa je lahko tudi pogosta, torej je v množici kupljenih artiklov pri več kupcih. Seveda pa ni nujno, da je v množici kupljenih artiklov teh kupcev vsebovana celotna opazovana množica. Poleg tega se lahko zgodi, da transakcija vsebuje več kot le eno (celotno ali delno) pogosto množico. Velikosti transakcij so zgoščene okoli povprečne dolžine, nekaj transakcij pa odstopa v pozitivni in negativni smeri.

Zunanji parametri generatorja IBM Quest so:

- D – število transakcij;
- T – povprečna dolžina transakcije;
- I – povprečna velikost (velike) pogoste množice;
- L – število pogostih (velikih) množic;
- N – število artiklov;
- parameter stopnje korelacije med (velikimi) množicami kupljenih artiklov, ki modelira opisan pojav, da imajo le-te pogosto skupne artikle.

Uveljavljen zapis generirane podatkovne baze ima obliko: Ta.Ib.Dc.Nd, kjer so a, b, c in d vrednosti parametrov, npr. T10.I4.D10000.N200 je podatkovna baza z 10000 transakcijami povprečne dolžine 10 in 200 artikli, povprečna dolžina pogostih n-teric pa je 4.

Avtorji generatorja (in algoritma APriori) navajajo, da stopnja korelacije (0.25 … 0.75) nima opaznega vpliva na rezultate, zato smo v naših eksperimentih upoštevali privzeto vrednost 0.25.

4.1.2 Generator podatkov LUCS-KDD ARM

Avtor generatorja podatkov LUCS-KDD ARM je F. Coenen, nastal pa je na oddelku za računalništvo univerze v Liverpoolu [36]. Ta generator je mnogo preprostejši, saj upošteva le ugotovitev, da so dolžine večine transakcij zbrane okoli povprečja, dolžine nekaterih transakcij pa odstopajo. Ker so artikli, prisotni v transakcijah, izbrani popolnoma naključno, ni upoštevano dejstvo, da so v realnih podatkih nekatere n-terice pogostejše kot druge. Ta slabost je razvidna tudi iz rezultatov testiranj. Nastavljivi parametri so:

- D – število vrstic (transakcij);
- N – število artiklov;
- g – "gostota" podatkovne baze.

Parameter gostote je uporabljen tako, da se z njim interno določi verjetnost vsakega posameznega artikla, da bo nastopal v obravnavani transakciji, dobljena podatkovna baza pa ima tudi skupno gostoto enako *g* (gostota podatkovne baze je razmerje med vsoto kupljenih artiklov v vseh transakcijah in produktom števila transakcij ter števila artiklov).

4.1.3 Generiranje koristi in količin artiklov

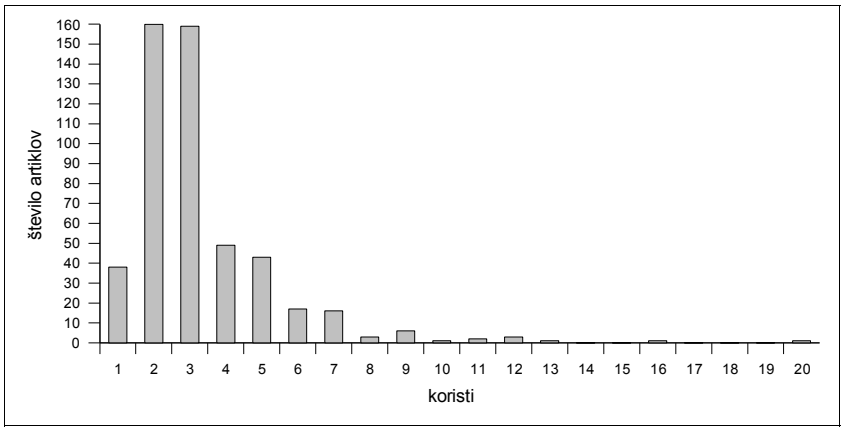
Oba opisana generatorja podatkov sta namenjena generiranju podatkovnih baz za iskanje povezovalnih pravil, zato je oblika dobljenih baz binarna. Če želimo na takšni podatkovni bazi uporabiti algoritme iskanja koristnih n-teric, potrebujemo tudi tabelo zunanjih koristi. Seveda pa je smiselno generirati še različne notranje koristi, s katerimi nadomestimo binarne vrednosti, saj se v nasprotnem primeru koristnostna funkcija izrodi v uteževanje atributov (artiklov).

Za potrebe pretvorbe podatkovnih baz in generiranja koristi ter količin je bil v okviru tega dela implementiran manjši samostojen program v jeziku Python, z grafičnim vmesnikom, ki izhodne datoteke generatorjev IBM Quest in LUCS-KDD ARM pretvori v obliko, uporabno v okolju Orange. Rezultat pretvorbe sta dve datoteki. Prva vsebuje generirano podatkovno bazo v t.i. *basket* obliki, druga pa hrani koristi za vse artikle, ki nastopajo v podatkovni bazi. Omenjena basket oblika datoteke je za naše potrebe zelo primerna, saj so obravnavane baze "redke", kar pomeni, da je povprečna dolžina transakcije mnogo manjša kot število vseh artiklov. Ta lastnost je interno upoštevana tudi v Orangeu, saj je domena takšnih podatkov prazna, vse vrednosti atributov (količine artiklov) pa so podane z meta atributi. Datoteka tipa basket je tekstovna datoteka, v kateri vsaka vrstica opisuje eno transakcijo. Artikli, ki nastopajo v posamezni transakciji, so navedeni v poljubnem vrstnem redu. Če je količina različna od 1, je to predstavljeno s parom $artikel = vrednost$. Posamezni artikli oz. pari so med seboj ločeni z vejicami.

Pri generiranju koristi in količin lahko v programu izbiramo med normalno in log-normalno

porazdelitvijo, pri čemer je privzeto upoštevana slednja, saj bolje opisuje realne podatke. Naključna spremenljivka, porazdeljena log-normalno, bo v večini primerov imela vrednosti blizu 0, v nekaj primerih pa bo vrednost veliko število [14]. Za pridobivanje tako porazdeljenih naključnih vrednosti sta v transformacijskem programu uporabljeni funkciji $normalvariate(\cdot,\cdot)$ in $lognormvariate(\cdot,\cdot)$ modula *random*, ki je vključen v standardno distribucijo interpreterja jezika Python. Dobljene vrednosti se končno še preslikajo v interval $[minVrednost, maxVrednost]$, ki ga določi uporabnik.

Na sliki 4.1 je podana porazdelitev koristi v primeru 500 artiklov po log-normalni porazdelitvi, na sliki 4.2 na naslednji strani pa primer uporabe vmesnika za pretvorbo datotek.



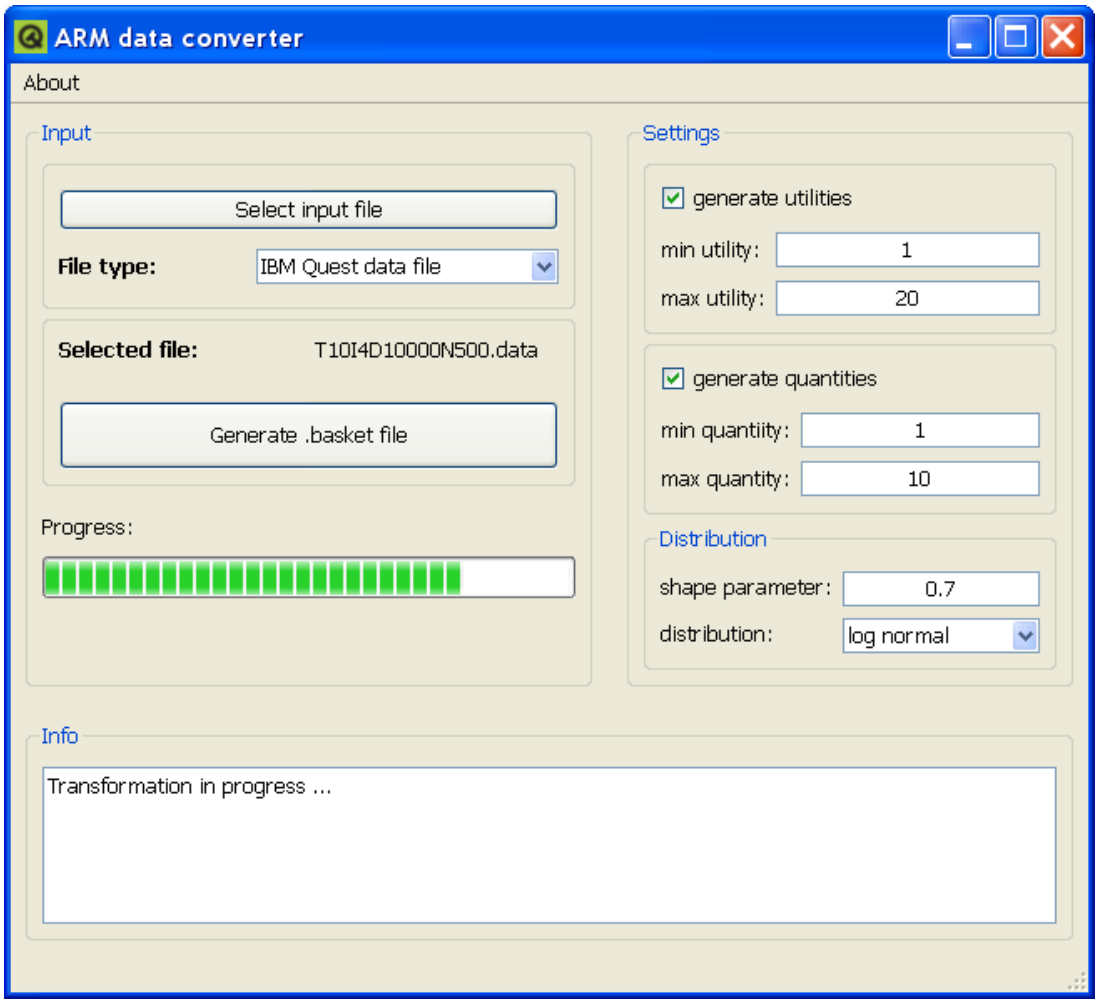
Slika 4.1: Porazdelitev (zaokroženih) koristi 500 artiklov po log-normalni porazdelitvi s parametrom 0.7 (standardna deviacija).

4.2 Implementacija algoritmov za iskanje koristnih n-teric

Celotni praktični del naše naloge, iskalni algoritmi, podatkovna baza in operacije na njej ter program za pretvorbo podatkov, je napisan v programskem jeziku Python. Ta odločitev sicer postavlja določene omejitve, ima pa tudi številne prednosti.

Največjo (in edino pomebno) omejitev naše implementacije predstavlja hitrost izvajanja. Tipične velikosti podatkovnih baz, ki se uporabljajo pri iskanju povezovalnih pravil, pogostih ter koristnih n-teric, so reda 10^6 in več. Tako velikih baz v naših eksperimentih ne moremo uporabiti, saj postane čas izvajanja algoritmov predolg, število kandidatov pa preveliko za glavni pomnilnik. Največja podatkovna baza, uporabljena v naših eksperimentih, je

sestavljena iz 100 000 transakcij in 1000 artiklov. Seveda pa je potrebno upoštevati, da je glavni faktor zahtevnosti omejitev minimalne koristi. Če je le-ta nizka, bo število kandidatov v zaporednih iteracijah skokovito naraslo, prav tako pa tudi število najdenih koristnih n-teric. V praksi je zato proces iskanja koristnih n-teric smiselno organizirati tako, da vnaprej določimo, kolikšno število koristnih n-teric je še obvladljivo in uporabno ter mejo koristnosti postopoma znižujemo, dokler nismo zadovoljni z dobljeno množico koristnih n-teric.



Slika 4.2: Primer uporabe programa za pretvorbo datotek.

Med prednostmi, ki jih ima implementacija v jeziku Python, lahko na prvem mestu navedemo preprostost, prilagodljivost ter integracijo s sistemom Orange. Nove module in funkcije zanj lahko sicer pišemo tudi v nižjenivojskem jeziku (C++), vendar pa težimo k uporabi jezika Python, kjer je le mogoče. Tako napisane module je enostavno vzdrževati, spreminjati in

testirati, ker ni potrebna faza prevajanja. Implementacija algoritmov za iskanje koristnih n-teric v jedru Orangea bi bila smiselna v primeru, če bi se pojavila potreba po učinkoviti analizi velike količine podatkov, npr. pri obdelavi več realnih transakcijskih podatkovnih baz z 10^6 in več transakcijami.

Temelj podatkovne baze, ki je uporabljena v iskalnih algoritmih, je struktura *orange.ExampleTable*, ki se zgradi pri branju podatkovne datoteke tipa basket. Ta struktura predstavlja osnovo novega razreda *Database*, ki služi kot vmesnik med iskalnimi algoritmi in podatkovno strukturo ExampleTable. Definira funkcije, ki so potrebne pri izvajanju algoritmov, npr. $utility(\cdot,\cdot)$, $support(\cdot,\cdot)$, ipd. Najpomembnejši atribut tega razreda je tabela zunanjih koristi $utilityTable$, ki je definirana kot slovar, v katerem so ključi vsi artikli v podatkovni bazi. Tabela zunanjih koristi se napolni z branjem datoteke s koristmi. Le-ta je tekstovna datoteka, v kateri vsaka vrstica vsebuje dve vrednosti: ime artikla in korist. Lahko jo sestavimo sami, v običajnem primeru pa je stranski produkt pretvorbe datoteke enega od opisanih generatorjev v basket obliko.

Dodaten podporni modul *supFuncs* definira nekatere funkcije in razrede, ki so potrebni pri izvajanju algoritmov za iskanje koristnih n-teric: funkcije za delo s seznami (odstranjevanje dvojnikov, razlika seznamov, APrioriGen in EFSM generiranje kandidatov) ter razreda *HashTree* in *Report*. Slednji služi za izpis rezultatov med izvajanjem algoritmov, *HashTree* pa definira podatkovno strukturo razpršeno drevo in operacije na njej. Implementacija razpršenega drevesa temelji na podatkovnih strukturah tipa slovar (notranja vozlišča) in seznam (listi) ter uporablja vgrajeno funkcijo $hash(\cdot)$. Kot smo opisali v tretjem poglavju, razpršeno drevo uporabljamo pri računanju koristi in podpore n-teric.

Najpočasnejši del implementiranih algoritmov ShFSM, DCG in FUFM predstavljajo večkratne vgnezdene zanke (procedure za generiranje kandidatov), ki so tudi v splošnem glede hitrosti šibka točka jezika Python. Prav zaradi tega je algoritem DCG hitrejši od algoritma ShFSM kljub večjemu številu kandidatov (števcev koristi za kandidate) v iteracijah, saj le-te pridobi na preprostejši način. Kot smo omenili, algoritem DCG ne uporablja standardne oblike generiranja (proizvedi in odstrani), pač pa kandidate pridobi s preprostim stikom seznamov (stik kandidata z vsakim elementom trenutne obravnavane transakcije, ki je večji od največje komponente kandidata).

4.3 Rezultati testiranj

Pri testiranjih algoritmov so bile uporabljene naslednje podatkovne baze:

1. generator IBM Quest: T10.I4.D1000.N200.L100, T10.I4.D2000.N300.L100, T10.I4.-D5000.N400.L200, T10.I4.D10000.N500.L300, T10.I4.D100000.N1000.L2000,
2. generator LUCS-KDD ARM: D2000.N300, $g = 3\%$.

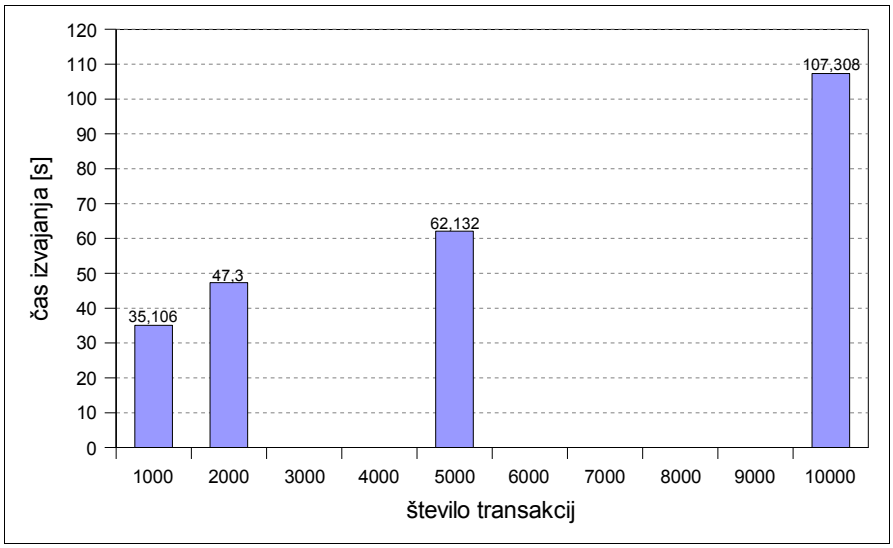
Gostota podatkovne baze v drugem primeru je bila izbrana tako, da je približno ustrezala gostoti baze T10.I4.D2000.N300.L100: $\frac{10 * 2000}{300 * 2000} = 3.33\%$.

V vseh primerih so bile koristi in količine generirane po log-normalni porazdelitvi s parametrom 0.7 v intervalih $[1, \ldots, 20]$ (koristi) ter $[1, \ldots, 10]$ (količine).

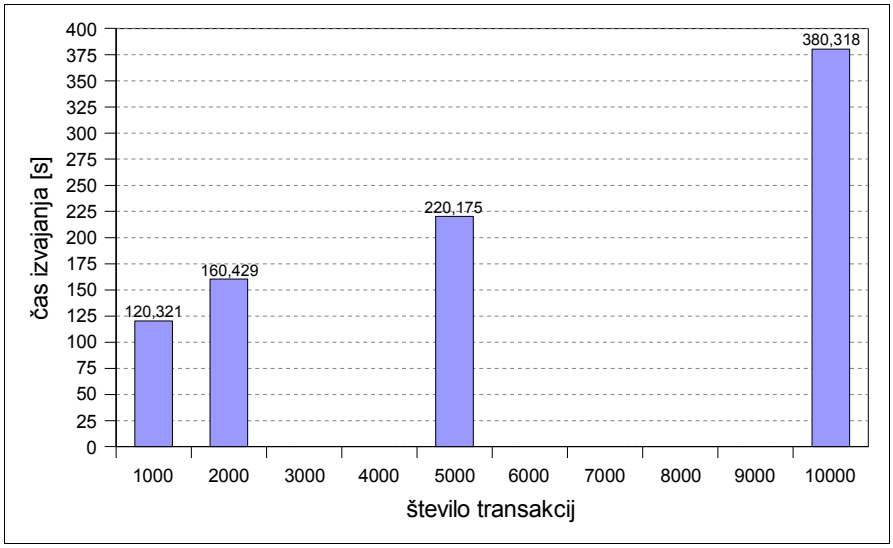
Vsi testi so bili izvedeni na računalniku s procesorjem AMD 3000+ z 1GB glavnega pomnilnika. Uporabljena je bila verzija 2.5 interpreterja jezika Python in Orange 0.99b (17. maj 2007).

Pri obravnavi rezultatov je pozornost namenjena predvsem zahtevnosti iskanja n-teric pri različnih velikostih baz in omejitvah koristi ter številu obravnavanih kandidatov. Konkretne najdene n-terice pa niso zanimive, saj so rezultat naključnega procesa (umetne podatkovne baze). Seveda pa bi v primeru realne podatkovne baze bila situacija ravno obratna.

Grafi 4.1 – 4.4 predstavljajo časovno zahtevnost algoritmov DCG in ShFSM. Razvidno je, da je zahtevnost iskanja linearno odvisna od števila transakcij in nelinearno (eksponentno) odvisna od omejitve koristi. Pri tem moramo poudariti, da so pri grafih 4.1 in 4.2 vrednosti na osi x, ki nimajo podane vrednosti časovne zahtevnosti, prisotne zato, da je merilo osi linearno. Vzrok linearne odvisnosti v prvem primeru je očiten: z večanjem podatkovne baze upada število koristnih n-teric pri fiksni omejitvi. Pri zelo velikih bazah se iskanje pri (premalo strogi) fiksni omejitvi izrodi v eden ali dva obhoda podatkovne baze, saj kritična funkcija za vse kratke n-terice (npr. dolžine 1, 2) ugotovi, da nimajo koristnih nadmnožic. Tudi eksponentna odvisnost zahtevnosti od omejitve minimalne koristi je očitna. Če le-to postavimo blizu 0, bodo koristne (skoraj) vse n-terice (podmnožice množice artiklov), postopek iskanja pa bo imel eksponentno zahtevnost, ker bo moral proučiti eksponentno število kandidatov.



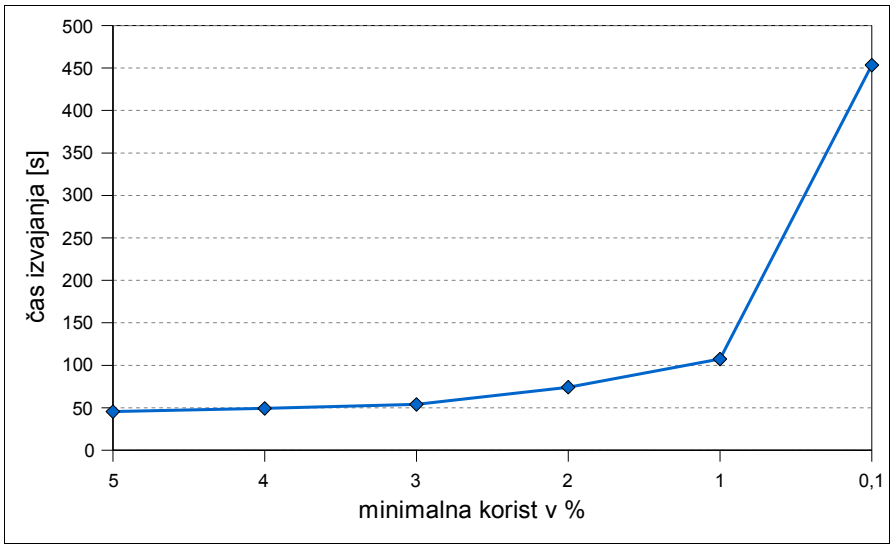
Graf 4.1: Povzetek izvajanja algoritma DCG na različnih podatkovnih bazah pri omejitvi koristi 1%.



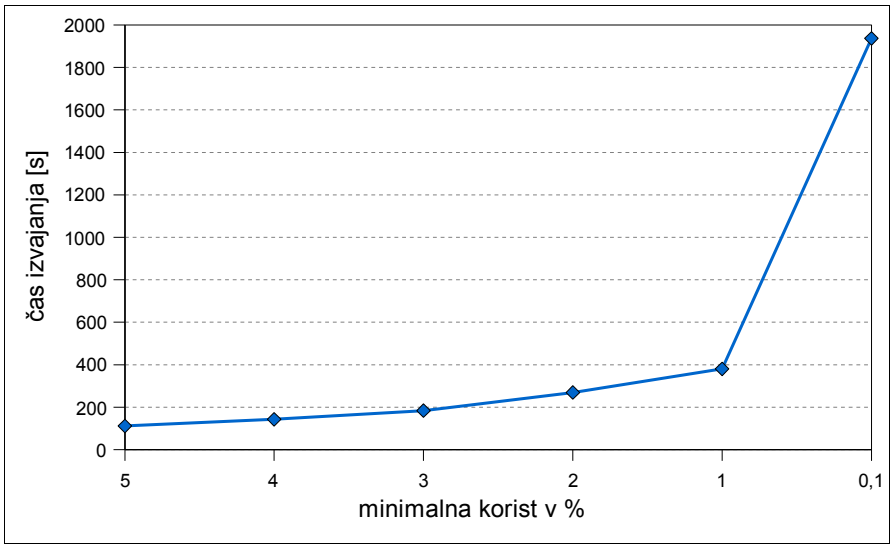
Graf 4.2: Povzetek izvajanja algoritma ShFSM na različnih podatkovnih bazah pri omejitvi koristi 1%.

Na grafu 4.5 je predstavljeno spreminjanje števila vseh kandidatov v zaporednih iteracijah algoritmov DCG in ShFSM, preden neobetavne odstranimo z uporabo kritične funkcije (pri DCG štejemo število števcev koristi, ne pa dejanskih kandidatov) . Rezultat je še posebej zanimiv pri upoštevanju podatkov iz grafov 4.1 in 4.2. Kljub občutno večjemu številu

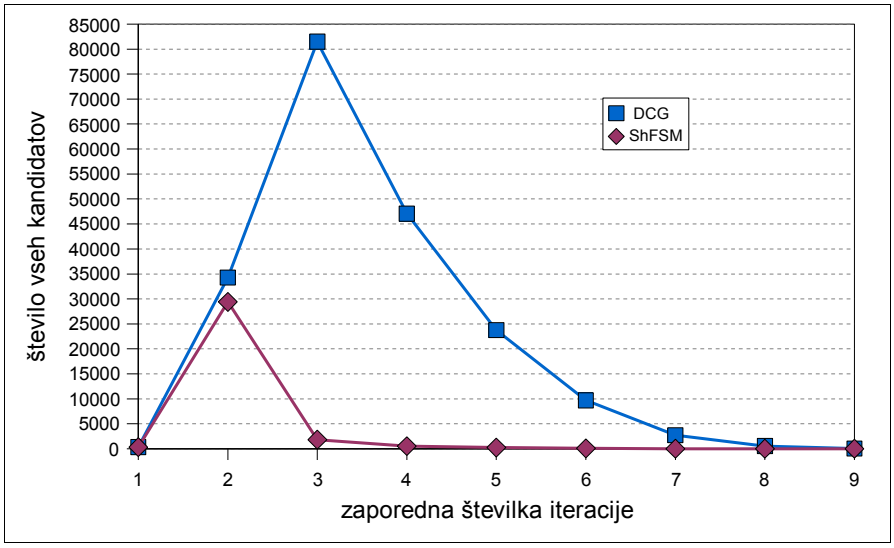
potencialnih kandidatov v vsaki iteraciji je DCG skoraj 4x hitrejši (DCG: 107 sekund, ShFSM: 380 sekund). Vzrok smo že pojasnili v tretjem poglavju, tukaj pa še enkrat poudarimo, da bi bila v primeru implementacije v enem od nižje nivojskih jezikov časovna razlika pri izvajanju obeh algoritmov zanemarljiva [39].



Graf 4.3: Čas izvajanja algoritma DCG na podatkovni bazi T10.I4.D10000.N500.L300 za dane omejitve koristi.



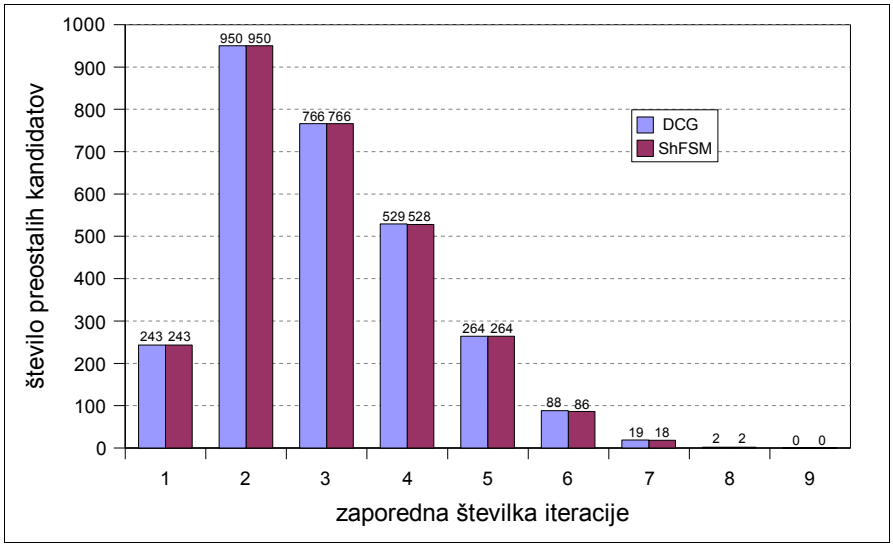
Graf 4.4: Čas izvajanja algoritma ShFSM na podatkovni bazi T10.I4.D10000.N500.L300 za dane omejitve koristi.



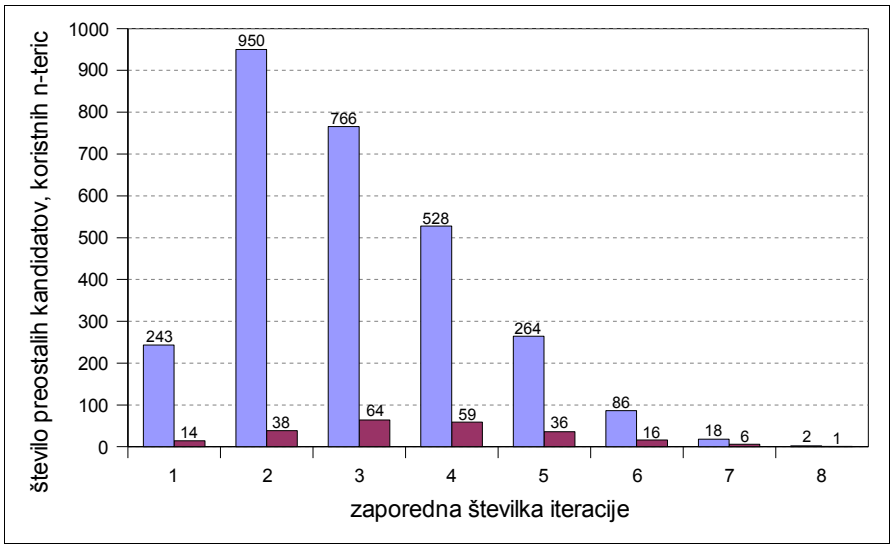
Graf 4.5: Število vseh kandidatov (oz. števcev v primeru DCG) v zaporednih iteracijah (pred uporabo kritične funkcije) med izvajanjem algoritmov DCG in ShFSM na podatkovni bazi T10.I4.D10000.N500.L300 pri omejitvi koristi 1%.

Graf 4.6 prikazuje število preostalih kandidatov in dopolnjuje graf 4.5. Očitno je, da je razlika v kvaliteti kritičnih funkcij algoritmov ShFSM in DCG zanemarljiva, saj algoritem ShFSM, ki uporablja optimalnejšo kritično funkcijo, skupno število kandidatov zmanjša le za 4 (iteracije 4, 6 in 7) v primerjavi z neoptimalno verzijo v algoritmu DCG. Vidimo tudi, da je uporaba kritične funkcije (optimalne in neoptimalne) uspešna, saj število kandidatov močno omeji, npr. od 29304 kandidatov jih ostane le 950.

Tipično spreminjanje velikosti množice preostalih kandidatov in koristnih n-teric v zaporednih iteracijah pri obeh algoritmih prikazuje graf 4.7. Ker v začetnih iteracijah število potencialnih kandidatov narašča, narašča tudi število koristnih n-teric. Število koristnih n-teric začne upadati nekaj iteracij kasneje (2 na grafu 4.7) kot število potencialnih kandidatov. Podatkovne baze, pridobljene z generatorjem IBM Quest, so dober približek realnih, torej je upravičeno pričakovati podobno obnašanje tudi pri analizi realnih podatkov.



Graf 4.6: Število preostalih kandidatov (po uporabi kritične funkcije) pri izvajanju algoritmov ShFSM in DCG na podatkovni bazi T10.I4.D10000.N500.L300 pri omejitvi koristi 1%. Število vseh kandidatov je podano na grafu 4.5.

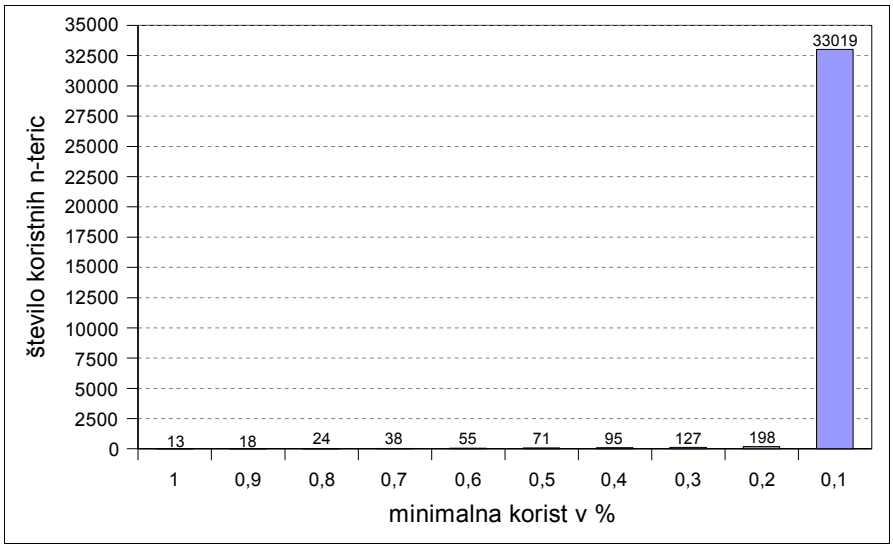


Graf 4.7: Spreminjanje velikosti množic preostalih kandidatov in koristnih n-teric pri izvajanju algoritma ShFSM (DCG) na podatkovni bazi T10.I4.D10000.N500.L300 pri omejitvi koristi 1%.

Slabost generatorja LUCS-KDD ARM prikažimo na podatkovni bazi D2000.N300. Pri analizi z algoritmom DCG lahko ugotovimo, da so pri omejitvi $minUtil = 1\% \ldots 0.4\%$ vse koristne

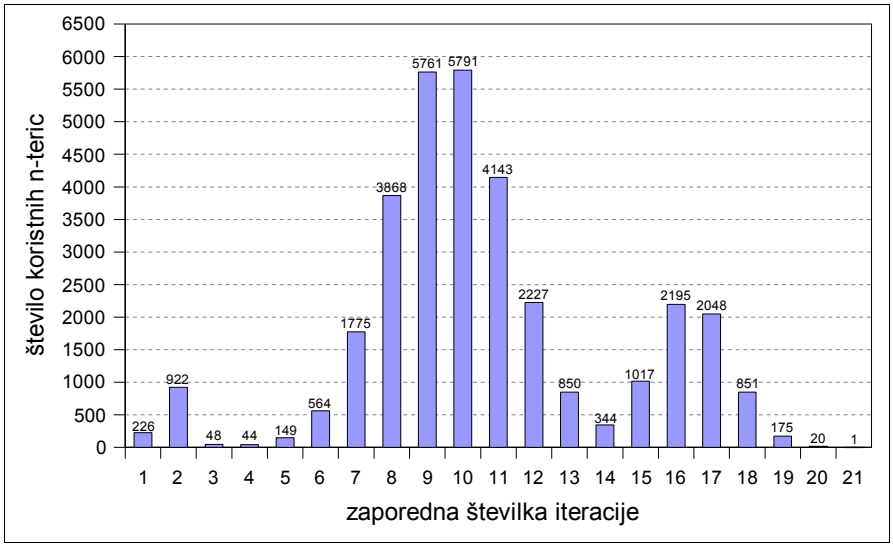
n-terice dolžine 1 (posamezni artikli). Za vrednosti 0.3% in 0.2% je najdenih tudi nekaj koristnih n-teric dolžine 2 (1 in 129), pri čemer je omejitev 0.2% "prelomna", saj dobimo pri omejitvi 0.1% izjemno veliko število koristnih n-teric dolžine vse do 21. To je prikazano na grafih 4.8 in 4.9.

Spreminjanje števila najdenih koristnih n-teric v ekstremnem primeru, ko je omejitev koristi 0.1%, je prikazano na grafu 4.9. Opazimo lahko, da se trend spreminjanja velikosti množice koristnih n-teric ne ujema s tistim, ki je razviden iz grafa 4.7. Vzrok, da so na grafu 4.9 trije izrazitejši "vrhovi" v velikosti najdenih n-teric, ni povsem jasen, najverjetneje pa je posledica naključja, ker v generatorju LUCS-KDD ARM niso upoštevane nobene zakonitosti pri generiranju transakcij (razen povprečne velikosti).



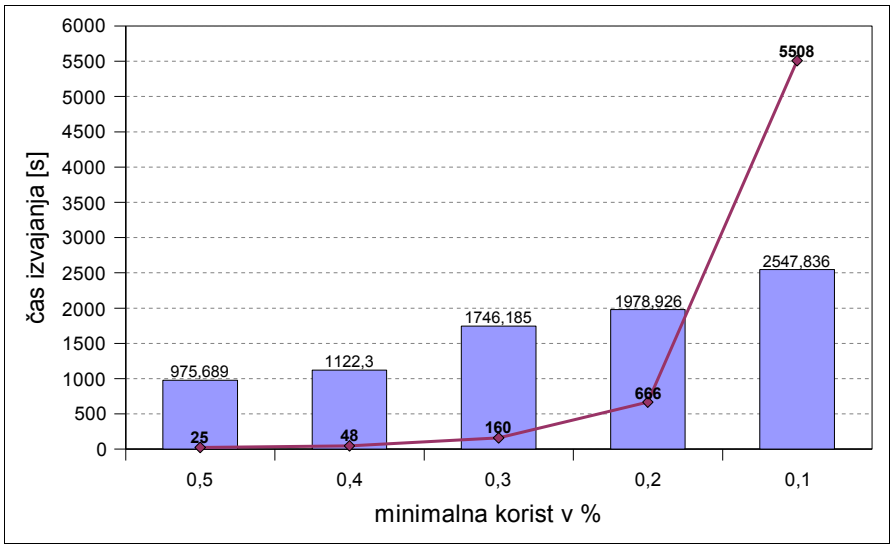
Graf 4.8: Število najdenih koristnih n-teric pri izvajanju algoritma DCG na podatkovni bazi D2000.N300 z gostoto 3%.

Kot zanimivost podajamo še povzetek analize podatkovne baze T10.I4.D100000.N1000.L2000. Le-ta je, vsaj glede velikosti, že precej dober približek realnim podatkovnim bazam. Vsebuje 100 000 transakcij, datoteka z opisom (basket oblika) pa ima velikost 6,5MB.



Graf 4.9: Spreminjanje velikosti množice koristnih n-teric pri izvajanju algoritma DCG na podatkovni bazi D2000.N300 z gostoto 3% pri omejitvi koristi 1%.

Graf 4.10 prikazuje rezultate izvajanja algoritma DCG na podatkovni bazi T10.I4.D100000.N1000.L2000. Višina stolpcev predstavlja čas izvajanja v sekundah, vrednosti na črti, ki povezuje stolpce, pa število najdenih koristnih n-teric pri dani omejitvi minimalne koristi. Pri omejitvi *minUtil=0.3%* v 29 minutah najdemo 160 koristnih n-teric. V primeru analize realne podatkovne baze bi bil takšen rezultat sprejemljiv, če seveda ni strožjih zahtev glede časovne omejitve.



Graf 4.10: Povzetek izvajanja algoritma DCG na podatkovni bazi T10.I4.D100000.N1000.L2000.

Na koncu prikažimo še učinkovitost novega algoritma FUFM za iskanje koristnih pogostih n-teric. Ker je dvofazni algoritem prepočasen za izvajanje na uporabljenih podatkovnih bazah, ga v primerjavi ne moremo uporabiti. Zato podajamo le povzetek izvajanja algoritma FUFM na podatkovni bazi T10.I4.D100000.N1000.L2000, na podlagi katerega lahko realistično ocenimo praktično uporabnost.

Tabela 4.1: Rezultat izvajanja algoritma FUFM na podatkovni bazi T10.I4.D100000.N1000.L2000.

	podpora = 0.5%		**podpora = 0.1%**	
omejitev koristi	**število n-teric**	**čas [s]**	**število n-teric**	**čas [s]**
$1 \cdot 10^{-5}$ %	0	241,3	121	1727,3
$5 \cdot 10^{-6}$ %	9	240,6	3644	1749,9
$1 \cdot 10^{-6}$ %	537	248,8	24587	1701.1

Ker so omejitve koristi podane procentualno glede na korist transakcije (zato so vrednosti nizke), navajamo še številčne vrednosti zaradi lažjega razumevanja:

- korist celotne podatkovne baze: 14 794 526,786 €
- številčno izražene koristi: $1 \cdot 10^{-5}$ % ≈ 148 €, $5 \cdot 10^{-6}$ % ≈ 74 €, $1 \cdot 10^{-6}$ % ≈ 15 €
- podpora 0.5% = 500 transakcij, podpora 0.1% = 100 transakcij

Očitno je, da je algoritem FUFM učinkovit, saj pri podpori 0.5% v podatkovni bazi s 100 000 transakcijami poišče koristne pogoste n-terice v približno 4 minutah. Pri omejitvi podpore 0.1% čas izvajanja sicer precej naraste, kar je posledica velikega števila n-teric, ki zadoščajo pogoju minimalne podpore, vendar pa je tudi dobljeno število koristnih in pogostih n-teric že zelo veliko. Tudi tu je v primeru uporabe v praksi smiselno vnaprej določiti, kolikšno število dobljenih n-teric je še obvladljivo. Pri željeni omejitvi podpore znižujemo omejitev koristi tako dolgo, dokler nismo zadovoljni s številom najdenih koristnih in pogostih n-teric.

Iz tabele 4.1 je razvidna še ena lepa lastnost algoritma FUFM. Čas izvajanja pri dani omejitvi podpore je namreč neodvisen od omejitve koristi, saj ugotavljanje koristnih n-teric med kandidati (pogostimi n-tericami) zahteva le sprehod skozi elemente v razpršenem drevesu in izračun njihove koristi. Ker je poleg tega razpršeno drevo že zgrajeno (APriori način računanja podpore), je ta korak trivialen in hiter.

5. Zaključek

V tem diplomskem delu smo proučili novonastalo področje koristnostnega podatkovnega rudarjenja. Kot smo omenili v uvodnem poglavju, ne gre za novo področje, pač pa za krovni pojem, ki združuje nekatera obstoječa podpodročja podatkovnega rudarjenja in strojnega učenja. Končni cilj uvedbe tega novega pojma je posplošena obravnava koristi celotnega procesa rudarjenja (vseh faz) pri napovednem in opisnem podatkovnem rudarjenju. Ker takšnih posplošenih obravnav še ni, se moramo pri opisu koristnostnega podatkovnega rudarjenja zadovoljiti s pregledom obstoječih metod napovednega in opisnega rudarjenja, ki vključujejo faktor koristnosti.

Predstavljena je opredelitev cen, ki nastopajo v procesu učenja iz primerov. Takšno vrsto učenja imenujemo cenovno občutljivo učenje in jo štejemo za najpomembnejšo in najobsežnejšo obliko koristnostnega podatkovnega rudarjenja. Na osnovi opredelitve cen je podan zgoščen pregled tega področja, vključno z navedbami specializirane literature. Opisana sta dva poskusa združitve različnih vrst cen v enotno ogrodje, ki predstavljata prvi korak k uresničitvi cilja uvedbe področja koristnostnega podatkovnega rudarjenja.

Med metodami opisnega podatkovnega rudarjenja smo podrobno proučili področje iskanja koristnih n-teric, ki predstavlja teoretični temelj naše implementacije v okolju Orange. To področje temelji na povezovalnih pravilih in pogostih n-tericah, vendar ima bolj omejen obseg uporabe, saj je v trenutno obliki primerno le za analizo transakcijskih podatkovnih baz. Prizadevanja raziskovalcev področja iskanja koristnih n-teric so tako usmerjena v izboljšave obstoječih algoritmov ne pa v možnosti splošnejše uporabe.

Naša obravnava področja iskanja koristnih n-teric vsebuje tudi opis posebne oblike n-teric, koristne pogoste n-terice. Kratko je opisan prvi obstoječ iskalni algoritem TP (angl. **T**wo-**P**hase), predstavljen pa je tudi nov algoritem FUFM (**F**ast **U**tility-**F**requent **M**ining). Ta ima številne prednosti pred algoritmom TP: hitrost, enostavnost in možnost uporabe obstoječih, učinkovitih algoritmov za iskanje pogostih n-teric. Razvoj algoritma FUFM je tudi najpomembnejši avtorski prispevek tega diplomskega dela.

Kratko sta opisana dva generatorja umetnih podatkov, uporabljena pri testiranju implementiranih algoritmov, pri čemer je za enega od njiju empirično dokazana slaba

kakovost generiranih baz. Za potrebe testiranja algoritmov je bil napisan tudi program z grafičnim vmesnikom, ki podatkovne baze, pridobljene z obema generatorjema, pretvori v obliko, primerno za uporabo v okolju Orange.

Pomanjkljivost praktičnega dela naše naloge predstavlja odsotnost realnih podatkovnih baz z realnimi koristmi (cenami), vendar je ta problem splošen in prisoten na vseh področjih koristnostnega podatkovnega rudarjenja. Delno omejitev uporabnosti implementiranih algoritmov postavlja tudi izbira programskega jezika, saj se v primeru podatkovnih baz realnih velikosti čas izvajanja močno podaljša. Kljub temu pa so poskusi pokazali, da so v primerih, ko čas ni kritičen faktor, implementirane rešitve uporabne tudi na podatkovnih bazah realnih velikosti.

Implementirana algoritma za iskanje koristnih n-teric smo preizkusili na več umetnih podatkovnih bazah, rezultati pa kažejo, da je hitrost sicer dokaj enakovrednih algoritmov močno odvisna od izbire programskega jezika.

Možnih smeri nadaljnjega dela je več. Prva, teoretična smer, je sistematizacija področja koristnostnega podatkovnega rudarjenja, ki je sedaj še dokaj neurejeno in nenatančno definirano. Druga, praktična smer, pa je implementacija algoritmov za iskanje koristnih n-teric v nižjenivojskem jeziku v jedro sistema Orange, ki bi omogočila hitrostno optimalno analizo danih podatkovnih baz. Pri tem bi lahko upoštevali še nekatere dodatne izboljšave, npr. iskanje in odstranjevanje izoliranih elementov, ki izboljša funkcijo ocenjevanja n-teric. Možna nadaljnja praktična smer dela bi lahko bila tudi implementacija in izčrpno testiranje postopkov, ki v enotnem ogrodju združujejo različne vrste cen. Le-ti so namreč še v povojih, objektivnih analiz različnih raziskovalcev pa še ni.

Viri in literatura

1. Agrawal R., Imielinski T., Swami A., “Mining association rules between sets of items in large databases,” v zborniku *ACM SIGMOD Intl. Conf. on Management of Data*, Washington, D.C., maj 1993, str. 207-216.
2. Angluin D., “Queries and concept learning”, *Machine Learning*, 2, str. 319–342, 1988.
3. Agrawal R., Srikant R., “Fast algorithms for mining association rules”, v zborniku *20th Intl. Conf. on Very Large Data Bases*, Santiago, Chile, september 1994, str. 487-499.
4. Breiman, L., Friedman, J., Olshen, R., Stone, C., *Classification and regression trees*, California, Wadsworth, 1984.
5. Barber B., Hamilton H. J., “Algorithms for mining share frequent itemsets containing infrequent subsets”, *4th European Conf. on Principles of Data Mining and Knowledge Discovery*, Lecture Notes in Computer Sciences, vol. 1910, Springer-Verlag, str. 316-324, 2000.
6. Barber B., Hamilton H. J., “Parametric algorithms for mining share-frequent itemsets”, *Journal of Intelligent Information Systems*, 16, str. 277-293, 2001.
7. Barber B., Hamilton H. J., “Extracting share frequent itemsets with infrequent subsets”, *Data Mining and Knowledge Discovery*, 7, str. 153-185, 2003.
8. Baum E., Lang K., “Query learning can work poorly when a human oracle is used”, International Joint Conference in Neural Networks, Beijing, China, 1992.
9. Blum, A., Mitchell T., “Combining labeled and unlabeled data with co-training”, v zborniku *1998 Conference on Computational Learning Theory*, str. 92–100, 1998.
10. Borgelt C., “Keeping Things Simple: Finding Frequent Item Sets by Recursive Elimination”, *Workshop Open Source Data Mining Software*, ACM Press, New York, str. 66-70, 2005.
11. Cohn D., Atlas L., Ladner R., “Improving generalization with active learning”, *Machine Learning*, 15, str. 201–221, 1994.
12. Dasgupta S., “Coarse sample complexity bounds for active learning”, *Neural Information Processing Systems*, 2005.

13. Domingos P., “Metacost: A general method for making classifiers cost-sensitive”, v zborniku *International Conference on Knowledge Discovery and Data Mining*, pages 155-64, San Diego, CA, str. 155-164, 1999.

14. Evans M., Hastings N., Peacock B., *Statistical distributions*, tretja izdaja, John Wiley & Sons, 2000.

15. Elkan C., “The Foundations of Cost-Sensitive Learning“ v zborniku *17th International Joint Conference on Articial Intelligence*, avgust 2001.

16. Fayyad U., Piatetsky-Shapiro G., Smyth P., “From data mining to knowledge discovery in databases“, *AI magazine*, 17(3), str. 37-54, 1996.

17. Freund, Y., Schapire R., E., “A decision theoretic generalization of on-line learning and an application to boosting”, *Journal of Computer and System Sciences*, 55(1), str. 119-139, 1997.

18. Geng L., Hamilton H. J., “Interestingness measures for data mining: a survey”, A*CM Computing Surveys*. To appear.

19. Hilderman R. J., Hamilton H. J., “Knowledge discovery and interestingness measures: A survey“, *Technical Report*, CS 99-04, Department of Computer Science, University of Regina, 1999.

20. Hermans, J., Habbema, J.D.F., Van der Burght, A.T., “Cases of doubt in allocation problems, k populations”, *Bulletin of the International Statistics Institute*, 45, str. 523-529, 1974.

21. Han J., Pei J., Yin Y., “Mining frequent patterns without candidate generation”, v zborniku *Int. Conf. on Management of Data*, str. 1-12, 2000.

22. Hopcroft J. E., Ullman J. D., *Introduction to automata theory, languages, and computation*, Addison-Wesley, 1979.

23. IBM Almaden research center, “Synthetic data generation code for associations and sequential patterns“. Dostopno na: http://www.almaden.ibm.com/cs/projects/iis/hdb/Projects/data_mining/datasets/syndata.html#assocSynData

24. Ikizler N., “Benefit maximizing classification using feature intervals”, *M.Sc. thesis*, Bilkent University, 2002.

25. Jovanovski V., Lavrač N., “Classication Rule Learning with APRIORI-C”, v zborniku *10th Portuguese Conference on Articial Intelligence*, str. 44-51, 2001.

26. Kapoor A., Learning and Classifying under Hard Budgets, *M.Sc. thesis*, University of Alberta, 2005.

27. Kapoor A., Greiner R., “Reinforcement Learning for Active Model Selection”, *First International Workshop on Utility-Based Data Mining*, Chicago, Illinois, 2005.

28. Kukar M., Kononenko I., “Cost-sensitive learning with neural networks“, *13th European Conference on Artificial Intelligence*, John Wiley & Sons, str. 445-449, 1998.

29. Kononenko I., *Strojno učenje*, založba FE in FRI, 2005.

30. Kleinberg J., Papadimitriou C., Raghvan P., “A Microeconomic view of Data Mining”, *J. Data Mining and Knowledge Discovery*, Kluwer Academic Publishers, 1999.

31. Kukar M., “Cenovno občutljivo učenje v medicinski diagnostiki”, *magistrsko delo*, Univerza v Ljubljani, Fakulteta za računalništvo in informatiko, 1996.

32. Liu Y., Liao W. K., Choudhary A., “A two-phase algorithm for fast discovery of high utility itemsets”, *First International Workshop on Utility-Based Data Mining*, Chicago, Illinois, 2005.

33. Li J., Li X., Yao X., “Cost-Sensitive Classification with Genetic Programming”, v zborniku *Congress on Evolutionary Computation*, Vol. 3, IEEE Press, New York, str. 2114-2121, 2005.

34. Lizotte D., Madani O., Greiner R., “Budgeted learning of naive-Bayes classifiers”, v zborniku *19th Conference on Uncertainty in Artificial Intelligence*, Acapulco, Mexico, 2003.

35. Lizzote D., Madani O., J., Greiner R., “Active model selection”, v zborniku *Uncertainty In Artificial Intelligence*, 2003.

36. LUCS-KDD ARM data generator, LUCS-KDD Software Library, Liverpool University of Computer Science. Dostopno na: http://www.csc.liv.ac.uk/~frans/KDD/Software/LUCS-KDD-DataGen/generator.html

37. Li Y. C., Yeh J. S., Chang, C. C., “A fast algorithm for mining share-frequent itemsets”, *7th Asia Pacific Web Conf. Lecture Notes in Computer Science*, vol. 3399,

Springer-Verlag, str. 417-428, 2005.

38. Li Y. C., Yeh J. S., Chang, C. C., "Efficient algorithms for mining share-frequent itemsets", v zborniku 1*1th World Congress of Intl. Fuzzy Systems Association*, str. 543-539, 2005.

39. Li Y. C., Yeh J. S., Chang, C. C., "Direct candidates generation: a novel algorithm for discovering complete share-frequent itemsets", v zborniku *2nd Intl. Conf. on Fuzzy Systems and Knowledge Discovery*, str. 551-560, 2005.

40. Margineantu D. D., "Active Cost-Sensitive Learning", *International Joint Conference on Artificial Intelligence*, str. 1622, 2005.

41. Margineantu D. D., "Methods for cost-sensitive learning", *Ph.D. dissertation*, Oregon State University, 2002.

42. McGarry K., "A Survey of Interestingness Measures for Knowledge Discovery", *The Knowledge Engineering Review*, Cambridge University Press, str. 1-24, 2005.

43. Mihelčič M., *Ekonomika poslovanja za inženirje*, založba FE in FRI, 8. dopolnjena izdaja, 2003.

44. Muslea I. A., Minton S., Knoblock C., "Adaptive view validation: A first step towards automatic view detection", *19th International Conference on Machine Learning*, str. 443–450, 2002.

45. Muslea, I. A., "Active learning with multiple views", *Ph.D. dissertation*, University of Southern California, 2002.

46. Melville P., Yang S. M., Saar-Tsechansky M., Mooney R., "Active Learning for Probability Estimation using Jensen-Shannon Divergence", v zborniku *16th European Conference on Machine Learning*, Porto, Portugal, str. 268-279, 2005.

47. Novak B., "Use of unlabeled data in supervised machine learning", *SIKDD multiconference*, Ljubljana, Slovenija, 2004.

48. Park J. S., Chen M. S., Yu P. S., "An effective hash-based algorithm for mining association rules", *SIGMOD Record*, 25(2), str. 175-186, 1995.

49. Provost, F. J., Jensen, D., and Oates, T., "Efficient progressive sampling", v zborniku *5th International Conference on Knowledge Discovery and Data Mining*, KDD-99, 1999.

50. F. Provost, Melville P., Saar-Tsechansky M., Mooney R., “Active Feature-Value Acquisition for Classifier Induction”, v zborniku *4th IEEE International Conference on Data Mining*, Brighton, UK, november, 2004.

51. F. Provost, Melville P., Saar-Tsechansky M., Mooney R., "Economical Active Feature value Acquisition through Expected Utility Estimation", *First International Workshop on Utility-Based Data Mining, Chicago*, Illinois, 2005.

52. Savasere A., Omiecinski E, Navathe S., “An efficient algorithm for mining association rules in large databases”, v zborniku *21th Intl. Conf. on Very Large Data Bases*, Zurich, Switzerland, str. 432-444, 1995.

53. Saar-Tsechansky, M., Provost F., "Active Sampling for Class Probability Estimation and Ranking", *Machine Learning* 54:2, str. 153-178, 2004.

54. Shen Y. D., Zhang Z., Yang Q., “Objective-Oriented utility-based association mining”, v zborniku *IEEE International Conference on Data Mining*, Maebashi, Japan, str 426-433, 2002.

55. Tan, P., Steinbach M., Kumar V., *Introduction to data mining*, Addison-Wesley, 2006.

56. Turney P. D., “Types of cost in inductive concept learning“, *Workshop on Cost-Sensitive Learning at the Seventeenth International Conference on Machine Learning*, Stanford University, California, str. 15-21, 2000.

57. Turney, P. D., “Cost-sensitive classification: Empirical evaluation of a hybrid genetic decision tree induction algorithm”, *Journal of Artificial Intelligence Research*, 2, str. 369-409, 1995.

58. ACM SIGKDD workshop on utility-based data mining, 2005. Dostopno na: http://storm.cis.fordham.edu/~gweiss/ubdm-kdd05.html

59. ACM SIGKDD workshop on utility-based data mining, 2006. Dostopno na: http://www.ic.uff.br/~bianca/ubdm-kdd06.html

60. Weiss G., Zadrozny B., Saar-Tsechansky M., “Utility-based data mining 2006 workshop report“, *SIGKDD Explorations*, volume 8, issue 2.

61. Witten, I., Frank E., *Data mining*, second edition, Morgan Kaufmann, 2005.

62. Yeh J. S., Li, Y. C., Chang C. C., “A Two-Phase Algorithm for Utility-Frequent Mining”, to appear in *Lecture Notes in Computer Science, International Workshop on*

High Performance Data Mining and Applications, 2007.

63. Yao H., Hamilton H. J., “Mining itemset utilities from transaction databases”, *Data & Knowledge Engineering*. To appear.
64. Yao H., Hamilton H. J., Butz C. J., “A Foundational Approach to Mining Itemset Utilities from Databases”, *SDM*, 2004.
65. Yao H., Hamilton H. J., Geng L., "A Unified framework for Utility based Measures for Mining Itemsets”, *Second International Workshop on Utility-Based Data Mining*, Philadelphia, Pennsylvania, 2006.
66. Zheng Z., Padmanabhan B., “On active learning for data acquisition”, v zborniku *IEEE Intl. Conf. on Data Mining*, 2002.
67. Zaki M. J., Parthasarathy S., Ogihara M., Li W., “New Algorithms for Fast Discovery of Association rules”, v zborniku *3rd Intl. Conf. on Knowledge discovery and Data Mining*, Newport Beach, California, str. 283-286, 1997.

Izjava o avtorstvu

Podpisani Vid Podpečan, absolvent Fakultete za računalništvo in informatiko Univerze v Ljubljani, izjavljam, da sem diplomsko nalogo "Koristnostno podatkovno rudarjenje" izdelal samostojno pod mentorstvom prof. dr. Igorja Kononenka in somentorstvom prof. dr. Nade Lavrač. Sodelavci, ki so mi pri tem pomagali, so navedeni v zahvali.

Vid Podpečan