Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Roman Orač

Strojno učenje v porazdeljenem okolju z uporabo paradigme MapReduce

magistrsko delo

magistrski program druge stopnje
računalništvo in informatika

Mentor: izr. prof. dr. Marko Robnik-Šikonja
Somentorica: prof. dr. Nada Lavrač

Ljubljana 2014

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Roman Orač, z vpisno številko **63080351**, sem avtor diplomskega dela z naslovom:

Strojno učenje v porazdeljenem okolju z uporabo paradigme MapReduce

S svojim podpisom zagotavljam, da:

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

V Ljubljani, dne 10. oktobra 2014 Podpis avtorja:

Zahvaljujem se mentorju izr. prof. dr. Marku Robniku-Šikonji za usmeritev in dobre napotke, somentorici prof. dr. Nadi Lavrač za začetno idejo in podporo pri delu in dr. Vidu Podpečanu za pomoč pri delu na magistrski nalogi.

Kazalo

Povzetek

Naraščajoči trend zbiranja podatkov je zahteval razvoj novih orodij za obdelavo in hranjenje velikih podatkov. To omogočajo nova orodja s porazdeljenim računanjem na računalniški gruči. Implementacija algoritmov strojnega učenja v porazdeljenem okolju prinaša več prednosti, kot sta zmožnost obdelave velikih množic podatkov in linearna pospešitev izvajanja z dodatnimi računskimi enotami. V magistrski nalogi opišemo paradigmo MapReduce, ki omogoča porazdeljeno računanje na računalniški gruči, in ogrodje Disco, ki ga implementira. Predstavimo sumarno obliko, ki je pogoj za učinkovito implementacijo algoritmov strojnega učenja s paradigmo MapReduce in opišemo implementacije izbranih algoritmov. Poleg tega predstavimo nove različice porazdeljenih naključnih gozdov, ki gradijo model na podmnožicah podatkov. Implementirane algoritme ovrednotimo s primerjavo z uveljavljenimi programi strojnega učenja. Pri tem izmerimo pohitritev implementiranih algoritmov z dodajanjem vozlišč v računalniško gručo. Magistrsko delo zaključimo z opisom vključitve implementiranih algoritmov v platformo ClowdFlows, ki omogoča sestavljanje, izvajanje in deljenje interaktivnih delotokov podatkovnega rudarjenja. S tem omogočimo obdelavo velikih paketnih podatkov z vizualnim programiranjem.

Ključne besede: MapReduce, porazdeljeno računanje, Disco, strojno učenje, sumarna oblika, DiscoMLL, porazdeljeni naključni gozdovi, ClowdFlows.

Abstract

The upward trend of data collection requested the development of new tools for processing and storing big data. New distributed computing models enable computation on computer clusters. Implementation of machine learning algorithms in a distributed environment ensures us multiple advantages, like processing of very large datasets and linear speedup with additional processing units. In this master thesis, we describe the MapReduce paradigm, which enables distributed computing, and the Disco framework, which implements it. We present the summation form, which is a condition for efficient implementation of machine learning algorithms with the MapReduce paradigm, and describe the implementations of the selected algorithms. We propose novel distributed random forest algorithms that build models on subsets of the dataset. We compare time and accuracy of the implemented algorithms with the well recognized data analytics tools. We measure speedups of implemented algorithms as a function of adding nodes to the cluster. We end our master thesis by describing the integration of the implemented algorithms into the ClowdFlows platform, which is a web platform for construction, execution and sharing of interactive workflows for data mining. With this integration, we enabled processing of big batch data with visual programming.

Keywords: MapReduce, distributed computing, Disco, machine learning, summation form, DiscoMLL, distributed random forest, ClowdFlows.

Poglavje 1

Uvod

Naraščajoči trend zbiranja podatkov je zahteval razvoj novih orodij za obdelavo in hranjenje velikih podatkov. V zadnjem času pogosto zasledimo besedno zvezo "veliki podatki" [1], ki se nanaša na množico podatkov, ki je prezahtevna za obdelavo s klasičnimi metodami. Nova orodja omogočajo obdelavo velikih množic podatkov s porazdeljenim računanjem na računalniški gruči. Lastnost takšnih sistemov je linearno povečanje zmogljivosti z dodajanjem novih vozlišč v gručo, izbrane algoritme pa je potrebno na novo implementirati za delovanje v porazdeljenem okolju.

Strojno učenje [2] je proces odkrivanja koristnih informacij iz podatkov. Algoritmi strojnega učenja se vsakodnevno uporabljajo v poslovnem in znanstvenem okolju. Zaradi širokega spektra uporabe je področje strojnega učenja vedno podvrženo novim izzivom. Eden izmed teh je obdelava velikih množic podatkov. Večina uveljavljenih programov z algoritmi strojnega učenja predpostavlja, da so množice podatkov manjše od velikosti glavnega pomnilnika, za obdelavo večjih množic podatkov pa potrebujemo zmogljivejši sistem. Obstajajo pristopi, ki zmanjšajo uporabo glavnega pomnilnika s shranjevanjem vmesnih izračunov na disk, na račun daljšega časa izvajanja. Čeprav lahko s takšnim pristopom obdelamo večjo množico podatkov, smo še vedno omejeni z zmogljivostjo posameznega sistema. Implementacija algoritmov strojnega učenja v porazdeljenem okolju prinaša več prednosti, kot sta zmožnost ob-

delave velikih množic podatkov in linearna pospešitev izvajanja z dodatnimi računskimi enotami. Pri tem pa obstajajo omejitve, saj lahko učinkovito implementiramo le algoritme strojnega učenja, ki pri porazdeljenih izračunih statistik nimajo računskih odvisnosti.

Cilj magistrskega dela je implementacija in validacija izbranih algoritmov strojnega učenja v porazdeljenem okolju. Algoritme izvedemo na različnem številu vozlišč in pri tem izmerimo pohitritve. K ciljem dodamo implementacije novih različic porazdeljenih naključnih gozdov [3, 4, 5], ki so primerne za izvajanje v porazdeljenem okolju. V poglavju 2 opišemo paradigmo MapReduce [1], ki predstavlja standard na področju obdelave velikih podatkov s porazdeljenimi sistemi. V poglavju 3 predstavimo implementacijo paradigme MapReduce v ogrodju Disco [6], ki omogoča porazdeljeno izvajanje algoritmov na računalniški gruči. Opišemo porazdeljeni datotečni sistem, ki je vključen v ogrodje Disco, in ogrodje primerjamo z ogrodjem Hadoop [7]. Poglavje 4 predstavlja osrednji del magistrskega dela, v katerem opišemo implementirane algoritme strojnega učenja. Na začetku predstavimo sumarno obliko, ki je pogoj za učinkovito implementacijo algoritmov strojnega učenja s paradigmo MapReduce. V nadaljevanju predstavimo izbrane algoritme in podrobno opišemo način implementacije s paradigmo MapReduce. Poglavje zaključimo z opisom novih različic porazdeljenih naključnih gozdov, ki se od obstoječih razlikujejo pri gradnji odločitvenih dreves na podmnožicah podatkov. V poglavju 5 opišemo ovrednotenje implementiranih algoritmov, ki jih primerjamo z uveljavljenimi programi za strojno učenje. Primerjamo pravilnost algoritmov in čase izvajanja. Pri tem izmerimo pohitritve implementiranih algoritmov z dodajanjem vozlišč v računalniško gručo. Magistrsko delo zaključimo z opisom vključitve implementiranih algoritmov v platformo ClowdFlows [8], kar opišemo v poglavju 6. Platforma ClowdFlows je odprtokodna oblačna platforma za sestavljanje, izvajanje in deljenje interaktivnih delotokov (angl. workflows) podatkovnega rudarjenja. Z vključitvijo implementiranih algoritmov v platformo ClowdFlows omogočimo obdelavo velikih množic podatkov z uporabo vizualnega programiranja.

Poglavje 2

Paradigma MapReduce

Paradigma MapReduce [1] je programski model in izvajalno ogrodje za obdelavo velikih podatkovnih množic s porazdeljenimi algoritmi na gruči računalnikov. Paradigmo MapReduce so razvili leta 2004 v podjetju Google.

Paradigma MapReduce predstavlja abstrakten nivo za porazdeljeno računanje, kjer ogrodje samodejno poskrbi za porazdelitev podatkov, vzporedno izvajanje algoritmov, razvrščanje opravil, izenačevanje obremenitev in odpornost na napake. Paradigma MapReduce predstavlja višji nivo abstrakcije v primerjavi z nizko nivojskimi abstrakcijami za vzporedno računanje, kjer mora programer poskrbeti za sinhronizacijo podatkov, zaznavo tekmovanja za vire in preprečevanje smrtnih objemov. Programerji brez izkušenj z vzporednimi in porazdeljenimi sistemi se lahko posvetijo logiki algoritmov in se ne ukvarjajo z izvedbenimi podrobnostmi. Ostale značilnosti ogrodja MapReduce so:

- možnost zaporednega in vzporednega računanja [9],
- računanje se prenese na vozlišča, ki imajo lokalen dostop do podatkov,
- za povečanje zmogljivosti sistema lahko dodamo dodatna vozlišča (horizontalna skalabilnost),
- poleg izvajalnega okolja ponavadi vsebuje tudi porazdeljen datotečni sistem,

- vozlišča vsebujejo izvajalno in shranjevalno okolje, ki sta medsebojno neodvisna,
- učinkovitost pri reševanju problemov, ki jih lahko brez težav razdelimo na manjše podprobleme (angl. embarrassingly parallel problems).

Najbolj znani odprtokodni implementaciji paradigme MapReduce sta ogrodji Hadoop [7] in Disco [6]. Čeprav je ogrodje Hadoop bolj razširjena implementacija paradigme MapReduce, se v magistrski nalogi osredotočamo na opis in uporabo ogrodja Disco, saj je primernejše za delovanje z aplikacijami, ki so napisane v programskem jeziku python.

V nadaljevanju poglavja podrobnejše predstavimo osnovni koncept paradigme MapReduce z opisom programskega modela in izvajalnega ogrodja ter poglavje zaključimo s primerom štetja besed v knjigi s paradigmo MapReduce.

2.1 Programski model

Paradigma MapReduce s funkcijo ***map*** razdeli problem na podprobleme in s funkcijo ***reduce*** rešitve podproblemov združi v končno rešitev. Podoben pristop uporabljajo funkcijski programski jeziki, kjer pa istoimenske funkcije niso ekvivalentne funkcijama map in reduce v paradigmi MapReduce [10]. Uporabnik določi funkciji map in reduce glede na problem ter kodo za pričetek izvajanja, za izvedbene podrobnosti pa poskrbi ogrodje MapReduce. Postopek MapReduce sprejme množico podatkov, iz katere ustvari nabor vhodnih parov $\langle ključ, vrednost \rangle$ in po zaključeni obdelavi vrne nabor izhodnih parov $\langle ključ, vrednost \rangle$, ki so z druge domene kot vhodni pari.

Programski model paradigme MapReduce temelji na naslednjem zaporedju korakov [10]:

1. Iteracija čez vhodne podatke.
2. Tvorjenje vmesnih parov $\langle ključ, vrednost \rangle$ iz vhodnih podatkov.

3. Združevanje vmesnih vrednosti z enakim ključem v skupine.

4. Iteracija čez vmesne vrednosti vsake skupine.

5. Redukcija vsake skupine z uporabo določene operacije.

Funkcija map sprejme vhodni par $\langle ključ, vrednost \rangle$, tvori poljubno število vmesnih parov in jih vrne. Funkcija reduce sprejme vmesni par s ključem in združenim seznamom vrednosti istega ključa, kar predstavlja par $\langle ključ, seznam(vrednost) \rangle$. Funkcija reduce za vsak vhodni ključ vrne obdelan nabor vrednosti, ki je ponavadi manjši od vhodnega. Zgornji opis formalno določita spodnji enačbi.

$$map \quad \langle ključ_1, vrednost_1 \rangle \longrightarrow seznam(\langle ključ_2, vrednost_2 \rangle) \quad (2.1)$$

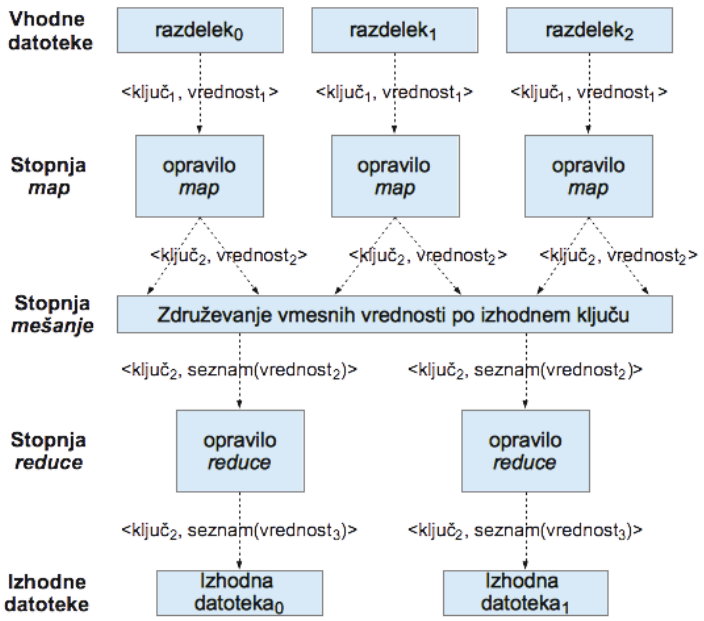
$$reduce \quad \langle ključ_2, seznam(vrednost_2) \rangle \longrightarrow seznam(vrednost_3) \quad (2.2)$$

Iz enačb (2.1) in (2.2) je razvidno, da je vhodni par iz druge domene kot izhodni par, saj funkcija map sprejme par iz ene domene in tvori seznam parov iz druge domene. Vmesni pari so iz iste domene kot izhodni pari. To pomeni, da lahko funkcija map vhodne ključe poljubno spreminja, pri funkciji reduce pa to ni mogoče.

2.2 Izvajalno ogrodje

Postopek MapReduce se izvede v treh stopnjah ***map***, ***mešanje*** in ***reduce*** (slika 2.1). Pri tem se mora predhodna stopnja zaključiti pred začetkom naslednje. Ob začetku izvajanja ogrodje razdeli vhodne podatke na več manjših razdelkov (angl. chunks) in jih porazdeli po gruči.

Izvajanje se začne s stopnjo map, kjer glavno vozlišče dodeli opravila delovnim vozliščem. Opravilo je dodeljeno vozlišču, ki je fizično najbližje vhodnemu razdelku. S tem se zmanjša omrežni prenos podatkov, saj se računanje približa podatkom. Posamezno opravilo map obdela posamezen



Slika 2.1: Prikaz izvedbe postopka MapReduce.

razdelek z vhodnimi podatki. Število razdelkov je enako številu opravil map, ki se izvede v postopku MapReduce. Podatki v razdelku se pretvorijo v vhodne pare $\langle ključ, vrednost \rangle$ in se posredujejo funkciji map, ki jih obdela in vrne vmesne pare. Delitvena funkcija razdeli vmesne pare po ključu v več skupin.

Stopnja mešanje posreduje vse pare z enako skupino določenemu opravilu reduce. Ta postopek poteka samodejno, brez posega programerja. Za pohitritev stopnje mešanje se lahko uporabi lokalni združevalnik (angl. combiner), ki je v ogrodju Disco opisan v poglavju 3.

V stopnji reduce, opravila reduce sprejmejo skupino parov, ki jo določi delitvena funkcija. Število skupin je enako številu opravil reduce, ki se izvede v postopku MapReduce. Delitvena funkcija je privzeto nastavljena tako, da razvrsti vse pare v isto skupino. V tem primeru se vsi vmesni pari posredujejo enemu opravilu reduce, ki jih obdela in vrne. Ta pristop se uporablja za probleme, ki za izračun potrebujejo dostop do celotnega nabora vrednosti.

V primeru več skupin, posamezno opravilo reduce sprejme pare posamezne skupine. Ta pristop omogoča vzporedno izvajanje več opravil reduce.

Opravila map ne ohranjajo stanja (angl. stateless), kar pomeni, da se posamezen vhodni par obdela neodvisno od ostalih. Ta lastnost omogoča vzporedno izvajanje, saj se lahko razdelki obdelajo na različnih strojih neodvisno od ostalih. Dostop do vseh parov v funkciji reduce predstavlja glavni razlog za zaporedno izvajanje postopka MapReduce, saj se mora pred začetkom stopnje reduce, zaključiti stopnja map. Vzporednost izvajanja se zagotavlja z izvajanjem vzporednih opravil reduce nad različnimi skupinami.

Za kompleksnejše izračune lahko verižimo več postopkov MapReduce, kjer lahko vsakem postopku določimo različni funkciji map in reduce. Pri tem izhod trenutnega postopka MapReduce predstavlja vhod naslednjemu postopku.

Primer štetja besed

Štetje besed v knjigi je klasičen primer, ki prikaže izvajanje postopka MapReduce. Denimo, da želimo prešteti besede v knjigi, ki je objavljena na spletu. Ogrodju Disco podamo vir knjige, določimo funkciji map in reduce ter sprožimo postopek.

Uporabnik določi funkciji map in reduce kot prikazuje algoritem 2.1. Opravila map se izvajajo istočasno na več delovnih vozliščih in jih določa enaka funkcija map. Posamezna funkcija map sprejme posamezno vrstico iz knjige na vhod. Na primer, vrstica *"Knjiga opisuje avtorjevo življenje. Knjiga"*. Funkcija vrstico obdela, besedo za besedo, in vrača vmesne pare. V tem primeru se tvorijo naslednji pari: $\langle Knjiga, 1\rangle$, $\langle opisuje, 1\rangle$, $\langle avtorjevo, 1\rangle$, $\langle življenje., 1\rangle$ in $\langle Knjiga, 1\rangle$. Vsak par označuje eno pojavitev besede. Ker uporabljamo privzeto delitveno funkcijo, so vsi vmesni pari v isti skupini. Stopnja mešanje poda vse vmesne pare enakemu opravilu reduce.

Opravilo reduce združuje vmesne vrednosti in se izvaja na enem delovnem vozlišču. Funkcija reduce sprejme parameter iterator, ki omogoča prehod po vseh vmesnih parih, ki jih vrnejo opravila map. Funkcija kvgroup

```
1  def map(vrstica, params):
2      for beseda in vrstica.split():
3          yield beseda, 1
4
5  def reduce(iterator, params):
6      #funkcija kvgroup za združevanje vrednosti
7      from disco.util import kvgroup
8
9      #frekvence združimo po besedi
10     for beseda, frekvence in kvgroup(iterator):
11         yield beseda, sum(frekvence) #seštevek frekvenc
```

Algoritem 2.1: Algoritem za štetje besed z ogrodjem Disco.

vrednosti združi po ključu. Beseda Knjiga se pojavi dvakrat v zgornji vrstici, kar pomeni, da je funkcija map vrnila enaka para $\langle Knjiga, 1\rangle$. Zato po združevanju vrednosti dobimo par $\langle Knjiga, [1, 1]\rangle$, ki označuje dve pojavitvi besede Knjiga. Funkcija reduce sešteje vrednosti za vsak ključ in vrne pare. V tem primeru, par $\langle Knjiga, 2\rangle$ predstavlja enega izmed izhodnih parov. Za izvedbo algoritma 2.1 z ogrodjem Disco je še potrebno dodati kodo, ki določi postopek MapReduce.

Poglavje 3

Ogrodje Disco

V poglavju opišemo implementacijo paradigme MapReduce v ogrodju Disco [6, 11]. To ogrodje smo izbrali zaradi dobre integracije s platformo ClowdFlows [8] opisano v poglavju 6, saj sta oba sistema napisana v programskem jeziku python.

Ogrodje Disco je odprtokodna implementacija paradigme MapReduce za porazdeljeno računanje na gruči računalnikov. Pri analizi in obdelovanju velikih podatkovnih množic ogrodje samodejno poskrbi za porazdelitev podatkov, vzporedno izvajanje opravil, razvrščanje opravil, izenačevanje obremenitev ter zagotovi odpornost na napake. Poleg izvajalnega okolja ogrodje Disco vsebuje tudi porazdeljen datotečni sistem, ki poskrbi za porazdeljeno in podvojeno shranjevalno plast. Jedro ogrodja Disco je napisano v programskem jeziku erlang, ki je namenjen razvoju robustnih, porazdeljenih ter na napake odpornih aplikacij. Uporabniška interakcija z ogrodjem Disco poteka z orodji ukazne vrstice in z uporabo knjižnice Disco, ki je implementirana v programskem jeziku python. Knjižnica Disco zagotovi vmesnik za komunikacijo z glavnim vozliščem in vmesnik za dostop do porazdeljenega datotečnega sistema. Ogrodje Disco so razvili leta 2008 v raziskovalnem središču podjetja Nokia in ga ažurno razvijajo naprej.

Glavne prednosti ogrodja Disco so:

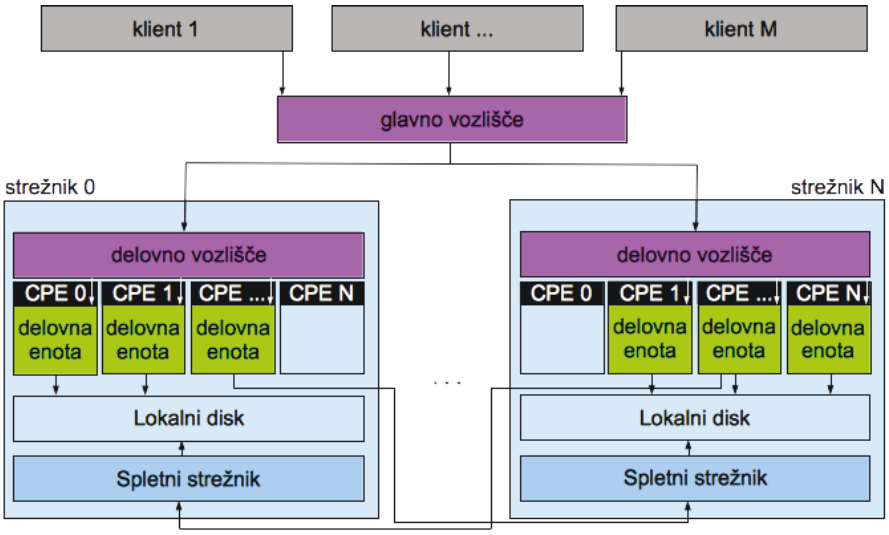
- nadzor in konfiguracija se vršita preko spletnega vmesnika,

- za določitev postopka MapReduce je potrebno napisati le funkciji map in reduce v programskem jeziku python, čeprav ogrodje podpira katerikoli programski jezik,
- poleg funkcij map in reduce lahko določimo razčlenitveno funkcijo, združevalnik in delitveno funkcijo, ki jih podrobnejše opisujemo spodaj,
- vhodni podatki so lahko v kateremkoli formatu. Podatki so lahko shranjeni na lokalnem disku ali pa dostopni preko spletne povezave,
- ogrodje je odporno na napake, saj sesutje posameznega delovnega vozlišča ne prekine izvajanja. Sproti lahko v gručo dodajamo nova delovna vozlišča,
- preprosta integracija v kompleksnejše aplikacije, kjer lahko računsko zahtevna opravila ločimo od same aplikacije,
- vgrajen porazdeljen datotečni sistem,
- večletna uporaba v podjetju Nokia, kjer dnevno analizirajo več terabajtov podatkov.

V nadaljevanju poglavja opišemo arhitekturo ogrodja Disco, porazdeljen datotečni sistem in osnovne gradnike, ki omogočajo izvajanje paradigme MapReduce. S primerom prikažemo izvajanje paradigme MapReduce v ogrodju Disco in poglavje zaključimo s primerjavo ogrodij Disco in Hadoop.

3.1 Arhitektura in osnovni gradniki

Ogrodje Disco je zasnovano za izvajanje na veliki gruči računalnikov, ki so povezani v omrežje. Ogrodje temelji na arhitekturi z glavnim vozliščem in več delovnimi vozlišči (slika 3.1). Aktivnosti koordinira glavno vozlišče, ki upravlja nadzor, dodeljuje računalniške vire, razvršča posle in opravila, obravnava dnevniške datoteke in interakcije med klienti. Klienti s proženjem programov



Slika 3.1: Pregled arhitekture ogrodja Disco.

predložijo posle v izvajanje na glavno vozlišče. Glavno vozlišče posle sprejme, doda v vrsto za izvajanje in jih izvede na prostih delovnih vozliščih.

Glavno vozlišče zažene delovna vozlišča na gruči, ki so sestavljena iz delovnih enot, lokalnega diska in spletnega strežnika. Delovna vozlišča so odgovorna za zagon in nadzor delovnih enot, ki izvajajo opravila map in reduce predloženega posla. Delovna enota izvede opravila posameznega posla in rezultat shrani v izhodne datoteke na lokalni disk, glavnemu vozlišču pa posreduje lokacije. Ogrodje Disco poskuša ohranjati lokalnost z razporejanjem opravil na vozlišča, ki gostijo podatke. V primeru, da je delovno vozlišče nedosegljivo, se sproži prerazporeditev opravil na ostala delovna vozlišča. S tem se zagotovi robustnost.

3.1.1 Predložitev poslov

Klient določi posel in s tem posreduje paket posla (angl. Job-packet) glavnemu vozlišču z uporabo spletnega vmesnika. Paketi posla so jezikovno

agnostični[1] in lahko vsebujejo binarne izvršljive datoteke poljubnega programskega jezika. Paketi se po predložitvi shranijo na glavno vozlišče. Ko je opravilo map ali reduce razporejeno na delovno vozlišče, vozlišče pridobi paket posla od glavnega vozlišča in zažene izvršljivo datoteko. Vsaka zagnana delovna enota ustvari opravilu lastno mapo, v kateri ustvari začasne datoteke in datoteke z rezultati. Čiščenje pomnilnika izbriše zastarele mape delovnih enot, pri tem pa ohrani datoteke z rezultati. Nedosegljivost delovnega vozlišča onemogoči dostop do rezultatov, zato je dobro rezultate porazdeliti po gruči ob koncu izvajanja. Knjižnica Disco to omogoča z zastavico save pri določitvi posla.

3.1.2 Protokol delovnih enot Disco

Protokol delovnih enot Disco je jezikovno agnostičen in se uporabi za komunikacijo med delovnimi vozlišči in zagnano delovno enoto. Pri komunikaciji se uporabi sočasen protokol s potrjevanjem, ki temelji na sporočilih JSON (angl. JavaScript Object Notation). Delovna enota uporabi ta protokol za pridobitev lokacij vhodnih podatkov za opravila, določitev izhodnih lokacij opravil in pošiljanje dnevniških zapisov ali sporočil o napakah. Z uporabo protokola se pošiljajo samo nadzorna sporočila. Vhodni paketni podatki se pridobijo neposredno iz lokalne ali oddaljene lokacije, ki je določena v sporočilih protokola.

3.1.3 Razporejevalnik poslov

Vsak porazdeljen posel MapReduce vključuje izvedbo več opravil, kjer opravilo predstavlja eno enoto za izvedbo in razporejanje. Vsako opravilo v izvajanju zavzame delovno enoto na vozlišču, na katerem se izvaja. Razporejevalnik se sproži, ko se pojavi nov posel za izvajanje, ko se zaključi opravilo v izvajanju ali ko pride do spremembe v topologiji gruče (na primer, zaradi ne-

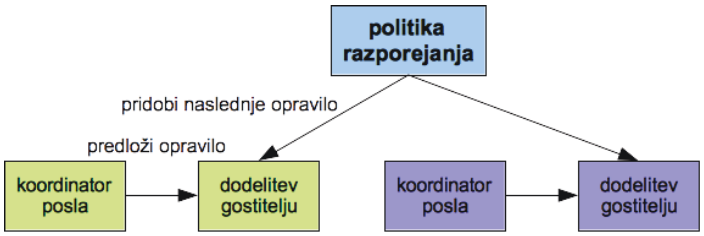
[1] Jezikovna agnostičnost pomeni, da se lahko struktura uporabi v navezi s poljubnim programskim jezikom.

dosegljivih ali ponovno dosegljivih gostiteljev). Razporejevalnik poslov skrbi za:

- pravično dodelitev računskih virov v gruči vsem poslom v izvajanju,
- zmanjšanje podatkovnega prenosa preko omrežja z lociranjem računanja blizu vhodnim podatkov,
- izenačevanje obremenitev,
- obravnava sprememb na gruči, ko delovna vozlišča postanejo nedosegljiva ali se pridružijo gruči.

Razporejevalnik je sestavljen iz več procesov (slika 3.2), ki jih opišemo spodaj. Za vsak aktiven posel se izvaja proces koordinator posla (angl. job coordinator), ki upravlja cevovod opravil (angl. task pipeline) in podatkovni cevovod (angl. data pipeline). Zažene se ob predložitvi posla na ogrodje Disco. Koordinator posla izračuna število opravil map in reduce na podlagi vhodov v posel in izhodov predhodnih opravil v cevovodu. Ko so vhodni podatki na voljo, opravilo postane pripravljeno za izvedbo in ga koordinator posla pošlje v izvajanje na proces dodelitev gostitelju (angl. Host allocator). Ta se izvaja za vsak aktiven posel in dodeljuje opravila delovnim vozliščem. Pri tem poskuša opravilo dodeliti delovnemu vozlišču, ki je lokalno vhodnim podatkom. Koordinator posla za vsako opravilo vzdržuje seznam delovnih vozlišč, na katerih je opravilo spodletelo in ob takšni napaki ponovno razporedi opravila. S tem se zagotovi odpornost na napake na ravni opravil.

Proces globalne politike razporejanja izbere naslednji posel za izvedbo. Politika razporejanja poskuša dodeliti vse razpoložljive procesorske vire poslom v izvajanju, tako da vsak posel dobi enak delež. Proces periodično posodablja prioritetno vrsto, v kateri so posli urejeni v skladu s primanjkljajem virov. Razporejevaljnik izbere posel z najmanjšim deležem zagotovljenih virov in pokliče proces dodelitev gostitelju, da izvede opravilo na prostih delovnih vozliščih.



Slika 3.2: Razporejanje poslov v ogrodju Disco je sestavljeno iz politike razporejanja ter koordinatorja posla in procesa dodelitve gostitelju za vsak posel posebej.

3.2 Porazdeljen datotečni sistem Disco

Porazdeljen datotečni sistem Disco (DDFS) zagotovi porazdeljeno shranjevalno plast. DDFS dopolnjuje tradicionalne relacijske podatkovne baze in porazdeljene strukture $\langle ključ, vrednost \rangle$, ki imajo pogosto težave z dinamičnim razširjanjem ob veliki količini paketnih podatkov. Ogrodje Disco shranjuje rezultate na DDFS, kar obdelanim podatkom omogoča trajnost in toleranco pri okvarah. Kot ogrodje Disco tudi DDFS uporablja arhitekturo z glavnim vozliščem in delovnimi vozlišči, kjer je glavno vozlišče DDFS sočasno tudi glavno vozlišče ogrodja Disco. Arhitektura DDFS z enim glavnim vozliščem je pogojena z večjo pomembnostjo konsistentnosti kot dostopnosti[2]. Visoka zakasnitev zaradi nedostopnega okolja MapReduce je bolj sprejemljiva kot pa nekonsistentnost. Arhitektura DDFS je primerljiva z ostalimi podobnimi sistemi.

3.2.1 Zasnova

DDFS je zasnovan za primere uporabe, ki so značilni za ogrodje Disco in paradigmo MapReduce, torej za hranjenje in obdelavo velikih količin nespre-

[2]Teorem CAP (angl. Consistency, Availability, Partition tolerance) govori o tem, da je v enem sistemu mogoče implementirati le dve od treh lastnosti: konsistenca, dostopnost in večparticijsko delovanje.

menljivih podatkov (angl. immutable data), kot so dnevniški zapisi, veliki binarni objekti (slike, posnetki), ali s spletnimi pajki zbrani neobdelani podatki. DDFS je datotečni sistem, ki temelji na oznakah namesto na datotečni hierarhiji. Podatkovne objekte lahko označimo s poljubno oznako in jih kasneje pridobimo na podlagi te oznake.

Oznake in zbirke binarnih podatkov

DDFS hrani paketne podatke v obliki zbirk binarnih podatkov (angl. Binary Large Object) in jih poimenuje v skladu z imenom izvirnih podatkov. Dostop do podatkov je omogočen s povezavo (URL) gostitelja[3] v gruči. Zaradi podvajanja ima zbirka binarnih podatkov več lokacij, ki so zajete v seznamu kopij zbirke binarnih podatkov. Oznake omogočajo gradnjo usmerjenih asociativnih grafov (slika 3.3) in zagotovijo prilagodljiv način za upravljanje obsežnih podatkovnih objektov.

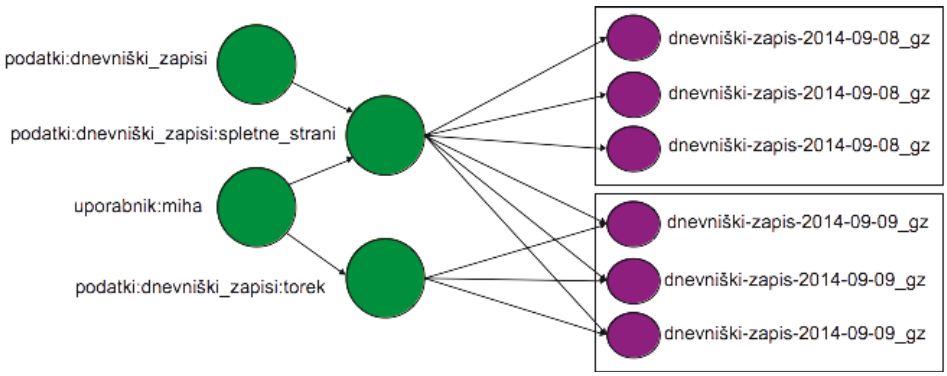
Vse operacije z oznakami obravnava glavno vozlišče DDFS, kar zagotovi konsistentnost v vsakem trenutku, paketni podatki pa se preberejo iz ali zapišejo na gostitelje neposredno z uporabo protokola za prenos hiperteksta. Stopnja podvajanja podatkov in oznak je določena s faktorjem 3 za zagotovitev odpornosti na napake in visoke dostopnosti.

Branje podatkov iz DDFS

Klient dostopa do vseh zbirk binarnih podatkov z uporabo oznak. Ustrezna oznaka zagotovi seznam kopij, ki vsebuje lokacije zbirk binarnih podatkov. Hierarhija oznak se obdela rekurzivno, da pridobimo vse zbirke binarnih podatkov povezanih s korensko oznako. V primeru napake pri branju iz določene lokacije lahko klient prebere podatke iz druge lokacije.

Ko gostitelj predstavlja tudi delovno vozlišče in to pridobiva zbirke vhodnih binarnih podatkov za opravilo, ga protokol delovnih enot Disco informira o lokalnih zbirkah binarnih podatkov. Tako lahko delovno vozlišče dostopa

[3]Gostitelj je delovno vozlišče DDFS, na katerega se shranjujejo podatki.



Slika 3.3: DDFS temelji na oznakah, ki omogočajo gradnjo usmerjenih asociativnih grafov.

neposredno do podatkov z uporabo lokalnega datotečnega sistema, brez potrebe po zaklepanju (angl. lock), kar omogoči visoko učinkovitost branja podatkov. Učinkovitost je posledica zasnove DDFS za hranjenje nespremenljivih podatkov in hranjenja podatkov brez spreminjanja formata.

Nalaganje paketnih podatkov na DDFS

DDFS pri nalaganju podatke razdeli in podvojeno shrani kot zbirke binarnih podatkov. Pri tem ustvari novo oznako ali pa spremeni že obstoječo. Klient sprva zahteva lokacije za nalaganje od glavnega vozlišča, ki mu vrne množico povezav URL na gostiteljih. Glavno vozlišče izbere gostitelja na podlagi razpoložljivega diskovnega prostora in sproži nalaganje podatkov. Dana množica povezav URL tvori seznam kopij zbirke binarnih podatkov.

Spletni programski vmesnik (angl. WebAPI) omogoča nalaganje podatkov na DDFS ter je osnovni gradnik orodij ukazne vrstice in modula za interakcijo z DDFS. Knjižnica Disco vsebuje orodji chunk in push za nalaganje podatkov na DDFS. Orodje chunk razdeli vhodne podatke na več razdelkov velikosti 64 megabajtov in razdelke zgosti s programom gzip. Pri razdelitvi podatkov ohranja celotne zapise in jih ne razbije na več delov, saj bi to predstavljalo težave pri obdelavi. Orodje push naloži podatke brez razdelitve in

zgoščevanja na DDFS.

Brisanje podatkov iz DDFS

Zbirke binarnih podatkov na DDFS naslavljamo z uporabo oznak in jih odstranimo tako, da izbrišemo vse oznake, ki se sklicujejo nanje. Čiščenje pomnilnika izvede dejansko brisanje nenaslovljenih paketnih podatkov. V času brisanja so lahko nekatere kopije na nedosegljivem gostitelju. Če se takšen gostitelj pridruži gruči kasneje, se vse kopije izbrisane oznake smatrajo za zastarele.

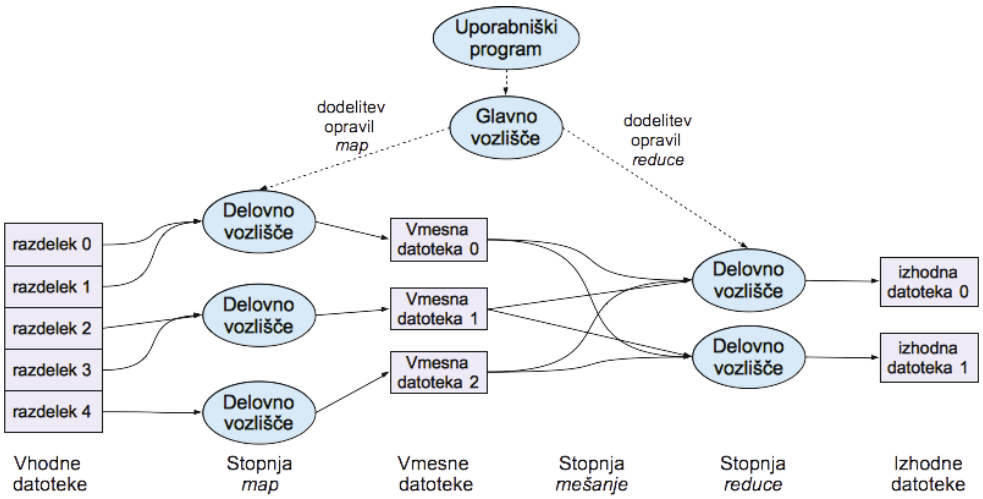
Čiščenja pomnilnika in podvajanje

Čiščenje pomnilnika izbriše zbirke binarnih podatkov, na katere se ne sklicuje nobena oznaka, zastarele oznake in oznake, ki jih odstrani uporabnik. Popoln pregled datotečnega sistema, ki ga izvede čiščenje pomnilnika, odkrije, katere oznake in zbirke binarnih podatkov imajo manj kopij in jih podvoji. Podvajanje se izvede v času čiščenja pomnilnika.

3.3 Paradigma MapReduce v ogrodju Disco

Na primeru štetja besed v knjigi prikažemo izvajanje paradigme MapReduce v ogrodju Disco (slika 3.4). Podrobnejše opišemo izvedbo postopka MapReduce, za razliko od primera v poglavju 2, ki se osredotoča na opis operacij nad pari. Poleg funkcij map in reduce, opišemo razčlenitveno funkcijo za razčlenitev vhodnih podatkov (angl. map reader), delitveno funkcijo za porazdelitev parov v skupine (angl. partition function) in združevalnik za lokalno združevanje parov (angl. combiner).

Za izvedbo posla na ogrodju Disco je potrebno napisati algoritem, ki uvozi knjižnico Disco, določi funkciji map in reduce ter izvede ukaz za začetek posla. Dodatno lahko določimo razčlenitveno funkcijo, delitveno funkcijo, združevalnik ter parametre, ki so dostopni vsem opravilom posla. Ogrodje Disco porazdeli računanje sorazmerno z razdelitvijo podatkov na DDFS, zato se



Slika 3.4: Izvedba postopka MapReduce v ogrodju Disco.

podatki pri nalaganju razdelijo na manjše razdelke. Število opravil map posameznega posla je enako številu vhodnih razdelkov. Manjša velikost razdelka omogoča hranjenje celotnega razdelka v glavnem pomnilniku računalnika, kar zmanjša bralno-pisalne operacije z diskom. V stopnji map se nad posameznim razdelkom izvede opravilo map, ki je sestavljeno iz razčlenitvene funkcije, funkcije map in združevalnika.

Razčlenitvena funkcija se izvede pred funkcijo map, razčleni pare in jih vrne. Uporaba razčlenitvene funkcije omogoča ločitev razčlenitve od obdelave podatkov, saj lahko za vsak vhodni format podatkov določimo drugo razčlenitveno funkcijo. Algoritem 3.1 prikazuje razčlenitev formata CSV (angl. Comma-Separated Values), kjer parametre fd, size in url nastavi ogrodje Disco, parameter params pa vsebuje parametre, ki jih pred začetkom izvedbe določi uporabnik. Parameter params je dostopen v vseh funkcijah posla. Razčlenitvena funkcija razčleni vse vrstice posameznega razdelka in za vsako vrstico vrne seznam njenih vrednosti. Seznam vrednosti se nato poda funkciji map.

Funkcija map se izvede za vsak par, ga obdela in vrne poljubno število pa-

```
1 def map_reader(fd, size, url, params):
2     #fd - objekt, ki vzpostavi povezavo
3     #size - velikost razdelka
4     #url - povezava URL do razdelka
5
6     razdelek = csv.reader(fd)
7     for vrstica in razdelek:
8         yield vrstica.split(",")
```

Algoritem 3.1: Razčlenitvena funkcija v ogrodju Disco.

rov. Sprejme parametra vrstica in params, kjer vrstica predstavlja razčlenjeno vrstico, ki jo vrne razčlenitvena funkcija. Algoritem 3.2 za vsako pojavitev besede v vrstici tvori in vrne vmesni par $\langle beseda, 1\rangle$.

```
1 def map(vrstica, params):
2     for beseda in vrstica:
3         yield (beseda, 1)
```

Algoritem 3.2: Funkcija map v ogrodju Disco.

Delitvena funkcija razdeli izhodne pare opravila map v več skupin kot določa enačba (3.1), kjer parameter razdelki (angl. partitions) poda uporabnik pri določitvi posla, privzeto pa je nastavljen na 1. Parameter razdelki s privzeto delitveno funkcijo je enak številu opravil reduce, ki se izvede v enem poslu MapReduce. Delitveno funkcijo lahko prilagodimo problemu, potrebno pa je zagotoviti enakomerno porazdelitev parov v skupine.

$$skupina = razpršitvena\_funkcija(ključ) \bmod razdelki \tag{3.1}$$

V stopnji mešanje ogrodje Disco posreduje vmesne pare opravilom reduce, kar pri velikem številu parov predstavlja obremenitev za omrežje. Združevalnik omogoča lokalno združevanje vrednosti izhodnih parov opravila map in se izvede, preden se pari shranijo v vmesne datoteke. Delitvena funkcija s številom skupin določi število združevalnikov, ki se izvede v opravilu

map. Združevalnik sprejme pare za posamezno skupino in združuje vrednosti z enakim ključem. Pari se nato shranijo v vmesne datoteke in se posredujejo stopnji mešanje. Združevalnik pare $\langle beseda, 1\rangle$ združi v pare $\langle beseda, frekvenca\rangle$. Ogrodje Disco samodejno nastavi parametre združevalnika (algoritem 3.3):

- parametra *kljuc* in *vrednost* predstavljata posamezen par, ki ju vrne funkcija map,
- parameter izravnalnik (angl. buffer) je slovar, ki ga inicializira ogrodje Disco. Izravnalnik ohranja stanje združevalnika s shranjevanjem ključev in združevanjem vrednosti,
- parameter *zakljucek* označi zadnji klic združevalnika, ko funkcija map vrne zadnji par. Takrat združevalnik vrne združene pare, ki se shranijo v vmesno datoteko.

```
1 def zdruzevalnik(kljuc, vrednost, izravnalnik,zakljucek,params):
2     if zakljucek:
3         #vrne združene vmesne pare
4         return izravnalnik
5     #seštevanjanje vrednosti z enakim ključem
6     izravnalnik[kljuc] = izravnalnik.get(kljuc, 0) + vrednost
```

Algoritem 3.3: Združevalnik v ogrodju Disco.

Stopnja mešanje poteka samodejno brez posredovanja programerja. V tej stopnji se pari iz vmesnih datotek posredujejo opravilom reduce glede na skupino, ki jo določi delitvena funkcija.

V stopnji reduce, opravilo reduce sprejme vmesne pare za določeno skupino. Pri določitvi posla lahko določimo parameter za sortiranje parov po ključu. To je potrebno za pravilno združitev vrednosti po ključu, saj funkcija kvgroup združuje le enake zaporedne vrednosti. Funkcija pretvori vmesne pare $\langle ključ, vrednost\rangle$ v pare $\langle ključ, seznam(vrednost)\rangle$. Brez te funkcije

vmesni pari ostanejo v enaki obliki, kot jih vrne funkcija map ali združevalnik. Funkcija reduce sprejme parametra iterator in params (algoritem 3.4), kjer iterator omogoča prehod čez vmesne pare. Funkcija reduce naredi prehod čez vse vmesne pare, uporabi določeno operacijo in vrne pare, ki se zapišejo v izhodno datoteko.

```
1 def reduce(iterator, params):
2     #funkcija za združevanje vrednosti
3     from disco.util import kvgroup
4     #frekvence združimo po besedi
5     for beseda, frekvence in kvgroup(iterator):
6         #seštevek frekvenc posamezne besede
7         yield  beseda, sum(frekvence)
```

Algoritem 3.4: Funkcija reduce v ogrodju Disco.

Pri določitvi posla v ogrodju Disco lahko poleg funkcij nastavimo tudi izbirne parametre (algoritem 3.5). Za izvedbo posla algoritem uvozi naslednje gradnike iz knjižnice Disco:

- razred Job vsebuje objekte za določitev poslov in interakcijo z njimi,
- razred Params omogoča določitev parametrov, ki so dostopni v vseh funkcijah posla,
- funkcija result_iterator omogoča izpis rezultatov, ki so shranjeni na DDFS.

Vhodni podatki se lahko nahajajo na DDFS ali pa so dostopni preko spletne povezave. Lokacije vhodnih podatkov podamo v seznamu, saj lahko določimo več lokacij. Funkcije map_reader, map, combiner in reduce določimo z lastnimi funkcijami, ki jih lahko poljubno poimenujemo. Za sortiranje parov v stopnji reduce označimo parameter sort. Rezultat posla se shrani v izhodne datoteke na DDFS. Število izhodnih datotek je enako številu opravil reduce določenega posla. Za izpis rezultatov uporabimo funkcijo result_iterator, ki vrne iterator za prehod po vseh parih izhodnih datotek.

```
1  from disco.core import Job, Params  #uvoz razredov
2  from disco.core import result_iterator #funkcija za izpis
3
4  #lokacije vhodnih podatkov
5  vhod = ["http://discoproject.org/media/text/chekhov.txt"]
6  params = Params(primer = 1) # primer določitve parametra
7
8  posel = Job().run(input = vhod, #določitev vhoda
9                    map_reader = map_reader, #razčlenitvena funkcija
10                   map = map,
11                   combiner = combiner, #določitev združevalnika
12                   reduce = reduce,
13                   partitions = 1, #določitev razdelkov
14                   sort = True, #sortiranje vrednosti
15                   params = params) #določitev dodatnih parametrov
16 rezultati = posel.wait() #začetek izvajanja posla
17
18 #izpis rezultatov
19 for beseda, frekvence in result_iterator(rezultati):
20     print beseda, frekvence
```

Algoritem 3.5: Določitev posla MapReduce v ogrodju Disco.

3.4 Primerjava ogrodij Disco in Hadoop

Čeprav imata ogrodji Disco in Hadoop veliko skupnih lastnosti gre za različna sistema. Ogrodji sta odprtokodna projetka, ki omogočata porazdeljeno obdelavo velikih podatkovnih množic na računalniški gruči s programskim modelom MapReduce. Skupna jima je zmožnost izvajanja poslov MapReduce, uporabljata podobno arhitekturo in imata vgrajen porazdeljen datotečni sistem.

Naštete podobnosti pa se razlikujejo v načinu implementacije. Opazna razlika je uporabljen programski jezik, saj je ogrodje Disco implementirano v programskih jezik python in erlang, ogrodje Hadoop pa v programskem jeziku java. Prvoten namen ogrodij je bil izvajanje paradigme MapReduce, s

tokom razvoja pa se je njuna funkcionalnost razširila. Ogrodje Disco podpira cevovod opravil, s katerim se ne omejuje na zaporedje stopenj map, mešanje in reduce. S tem je določitev zaporedja in števila stopenj prepuščena uporabniku. Ogrodje Hadoop je večnamenska platforma, ki podpira več načinov za obdelavo podatkov. Pod to šteje proženje interaktivnih poizvedb, obdelava tokov podatkov in paketna obdelava podatkov z MapReduce. Za ogrodje Hadoop obstaja več odprtokodnih razširitev, ki dodajo funkcionalnost ogrodju.

Porazdeljena datotečna sistema pri nalaganju podatke podvojita in s tem zagotovita odpornost na napake. Oba sta zasnovana za hranjenje nespremenljivih paketnih podatkov. Kot izvajalni ogrodji se razlikujeta tudi porazdeljena datotečna sistema. DDFS je implementiran s programskim jezikom erlang, HDFS (angl. Hadoop Distributed File System) pa s programskim jezikom java. DDFS naslavlja datoteke na podlagi oznak, ki omogočajo gradnjo usmerjenih asociativnih grafov in jih lahko obdelamo rekurzivno. HDFS uporablja drevesno datotečno strukturo kot klasični datotečni sistemi. DDFS podvoji oznake in podatke za razliko od HDFS, ki podvoji le podatke. HDFS omogoča nastavitev za vzdrževanje več kopij podatkovnih struktur, ki vsebujejo strukturo z datotekami.

Ogrodje Hadoop je največkrat uporabljena platforma za upravljanje in obdelavo velikih podatkov. Predstavlja zmogljivejše ogrodje v primerjavi z ogrodjem Disco, saj omogoča več načinov za obdelavo podatkov. Kritika ogrodja Hadoop je zapletenost uporabe, ki je posledica večnamenske zasnove. Ogrodje Disco je enostavnejše za uporabo, pri tem pa nudi manj razširitev. Izbira ogrodja je odvisna od problema, ki ga želimo rešiti. Ogrodje Disco je primernejše za vključitev v aplikacije, kjer je paketna obdelava velikih podatkov le ena izmed funkcionalnosti, ogrodje Hadoop pa za celovitejšo rešitev pri upravljanju in obdelavi velikih podatkov.

Poglavje 4

Strojno učenje s paradigmo MapReduce

Število jeder v procesorjih se povečuje iz leta v leto. Z vzporednim izvajanjem algoritmov izkoristimo večjedrno arhitekturo, za to pa je potrebno na novo razviti in implementirati algoritme. Obstaja več programskih jezikov, ki omogočajo vzporedno izvajanje algoritmov, vendar nihče od njih ne ponuja očitne implementacije vzporednih algoritmov strojnega učenja. Literatura opisuje več pristopov za strojno učenje in podatkovno rudarjenje v porazdeljenem okolju, vendar so to večinoma prilagojene implementacije posameznih algoritmov in ne splošen pristop. Splošni pristopi obstajajo na računalnikih s skupnim pomnilnikom, vendar niso primerni za računalnike z lokalnim predpomnilnikom.

V poglavju se osredotočimo na opis splošnega in eksaktnega pristopa za velik razred algoritmov strojnega učenja, ki se izvajajo vzporedno na procesorjih z več jedri. Opišemo sumarno obliko, ki predstavlja pogoj za učinkovito implementacijo algoritma s paradigmo MapReduce. V [12] avtorji nakažejo implementacije več algoritmov strojnega učenja s paradigmo MapReduce. Nekatere smo implementirali in združili v knjižnico algoritmov za strojno učenje s paradigmo MapReduce, kar podrobneje opisujemo v nadaljevanju poglavja. Poglavje zaključimo z opisom lastnih različic porazdeljenih na-

ključnih gozdov, ki odločitvena drevesa zgradijo na podmnožicah podatkov.

4.1 Sumarna oblika

Izkoristek sistemov z več jedri je največji pri sočasnih aplikacijah, ki ne zahtevajo veliko komunikacije. Pomembno je, da so podatki razdeljeni in da so podmnožice podatkov jedrom lokalno dostopne. Učni model, ki to formalizira, je model statističnih poizvedb (angl. Statistical Query model) [13].

Pri modelu statističnih poizvedb učni algoritem, namesto posameznih učnih primerov, združi več učnih primerov. Statistične poizvedbe potekajo v dveh stopnjah, kjer prva obdela posamezno podmnožico podatkov, druga pa združi rezultate in zgradi napovedni model. Lastnosti modela statističnih poizvedb omogočajo učinkovito vzporedno izvajanje algoritmov na sistemih z več jedri. Algoritme, ki spadajo v model statističnih poizvedb, lahko zapišemo v sumarni obliki (angl. summation form). To so algoritmi, ki računajo določeno globalno statistiko, ki jo lahko izračunamo z vsoto lokalno izračunanih statistik iz podmnožic podatkov. Pri paradigmi MapReduce lokalne statistike izračunajo opravila map, opravilo reduce pa jih združi v globalne statistike. Algoritem s takšnimi lastnostmi lahko zapišemo v sumarni obliki. Zapis v sumarni obliki ne obstaja za algoritme, ki uporabljajo stohastični najstrmejši sestop (angl. stochastic gradient descent). Za vsak korak najstrmejšega sestopa (en učni primer) je potrebno posodobiti skupno množico parametrov, kar pa predstavlja ozko grlo pri vzporednem izvajanju algoritma.

4.2 Knjižnica DiscoMLL

Po našem vedenju ne obstaja paket algoritmov strojnega učenja v programskem jeziku python, ki temelji na paradigmi MapReduce in deluje z ogrodjem Disco. To nas je motiviralo za razvoj knjižnice DiscoMLL (angl. Disco Machine Learning Library) [14].

Knjižnica DiscoMLL je odprtokodna knjižnica, ki temelji na knjižnici NumPy [15] in deluje z ogrodjem Disco. Knjižnica vsebuje algoritme strojnega učenja s paradigmo MapReduce za področja klasifikacije, regresije, združevanja in ansambelskih metod. Knjižnica omogoča več nastavitev: določitev več vhodnih virov podatkov, izbiro atributov in obravnavo manjkajočih podatkov. Podpira več podatkovnih formatov: tekstovne datoteke, podatke v razdelkih na DDFS in format gzip. Podatkovni vir se lahko nahaja na DDFS ali pa je dosegljiv preko spletne povezave.

Knjižnico DiscoMLL smo vključili v platformo ClowdFlows [8], ki je opisana v poglavju 6. Uporabnikom platforme ClowdFlows omogoča obdelavo velikih paketnih podatkov z uporabo vizualnega programiranja.

4.3 Algoritmi s paradigmo MapReduce

Opišemo implementacijo algoritmov strojnega učenja, ki za pospešitev izvajanja uporabijo dodatna jedra v računalniškem sistemu. Glavne lastnosti pristopa so naslednje:

- vsak algoritem, ki spada v model statističnih poizvedb lahko zapišemo v sumarni obliki. To spremeni način implementacije in ne algoritma samega, zato rezultat predstavlja eksaktno rešitev in ne približek,
- sumarna oblika se lahko enostavno izrazi s paradigmo MapReduce,
- opisan pristop omogoča linearno pohitritev z dodajanjem procesnih enot,
- opisane implementacije algoritmov niso nujno hitrejše od specializiranih algoritmov. V [12] dosežejo linearno pohitritev, ki je pogosto hitrejša od specifičnih prilagojenih implementacij.

Algoritme smo implementirali s paradigmo MapReduce in se izvajajo na ogrodju Disco. Algoritmi v prvi stopnji zgradijo napovedni model iz učne množice, v drugi pa ga uporabijo za napovedovanje primerov iz testne

množice. Za vsak algoritem podrobneje opišemo gradnjo napovednega modela. Pri tem se v stopnji map izvajanje opravil porazdeli, stopnja reduce pa združi vse vmesne pare ter vrne povezavo URL do napovednega modela na DDFS. Pri algoritmih, ki omogočajo izvajanje več vzporednih opravil reduce za gradnjo napovednega modela, je to posebej poudarjeno. Napovedni model se razlikuje za vsak algoritem, saj vsebuje model in parametre za napovedovanje. Napovedovanje sestavljajo le opravila map, ki se porazdelijo po gruči. Napovedni model preberemo iz DDFS in ga podamo kot parameter opravilom map. Ta ga uporabijo na testnih primerih in tvorijo napovedi. Napovedovanje vrne povezavo URL, s katero dostopamo do rezultatov na DDFS.

V psevdokodi algoritmov uporabljamo naslednje funkcije. Funkcija map sprejme parametra primer in params, kjer parameter primer vsebuje učni primer, parameter params pa vsebuje nastavitve, ki jih določimo pred izvajanjem. Parameter params sprejmejo funkcije map, združevalnik in reduce. Združevalnik sprejme parametre kljuc, vrednost, izravnalnik in zakljucek, ki jih nastavi ogrodje Disco glede na predhodno stanje funkcije map. Pomen parametrov združevalnika je podrobneje opisan v poglavju 3 o ogrodju Disco. Funkcija reduce sprejme parameter iterator, ki omogoča prehod po parih, ki jih tvori stopnja map. V funkciji reduce lahko uporabimo funkcijo kvgroup za združitev vrednosti parov po ključu. Pri algoritmih, ki funkcijo uporabijo, to posebej poudarimo.

Uporabljamo naslednjo notacijo: $x^{(i)}$ ali x označuje učni primer, $y^{(i)}$ ali y pa razred primera i. Naj x_j označuje atribut j. Množica podatkov vsebuje n atributov in m primerov, kjer velja $1 \leq j \leq n$ in $1 \leq i \leq m$.

Implementirali smo naslednje algoritme (podrobnosti implementacije so opisane v [12]): naivni Bayes, linearna regresija, lokalno utežena linearna regresija, logistična regresija, metoda podpornih vektorjev in razvrščanje z voditelji. Razvili in implementirali smo več različic porazdeljenih naključnih gozdov, ki se od ostalih razlikujejo po gradnji odločitvenih dreves na podmnožicah podatkov. Uporabili smo podoben pristop kot je opisan v [3, 4].

4.3.1 Naivni Bayes

Naivni Bayes [2] oceni apriorno verjetnost in pogojno verjetnost razreda ob predpostavki, da so atributi pri danem razredu pogojno neodvisni. Apriorno verjetnost razreda ocenimo z izračunom deleža učnih primerov, ki pripadajo razredu. Predpostavko o pogojni neodvisnosti izrazimo z enačbo (4.1).

$$P(x|y = c) = \prod_{j=1}^{n} P(x_j|y = c). \tag{4.1}$$

Namesto računanja pogojnih verjetnosti za vsako kombinacijo vrednosti v x, lahko zaradi predpostavke o pogojni neodvisnosti, izračunamo le pogojne verjetnosti za vsako vrednost x_j pri danem y. Ta pristop za dobro oceno verjetnosti ne zahteva velike učne množice.

Ocenjevanje pogojnih verjetnosti

Pri ocenjevanju pogojnih verjetnosti ločimo izračun s kategoričnimi vrednostmi atributov in izračun s številskimi vrednostmi atributov.

Pri ocenjevanju pogojnih verjetnosti s kategoričnimi atributi, za kategorični atribut x_j izračunamo pogojno verjetnost $P(x_j = v|y = c)$ z deležem učnih primerov razreda c, ki imajo pri atributu x_j vrednost v. Formalni zapis zgornjega opisa določa enačba (4.2), kjer n označuje število učnih primerov z razredom c, n_c označuje število učnih primerov razreda c pri vrednosti v, m je parameter ocene m in p označuje apriorno verjetnost razreda c.

$$P(x_j = v|y = c) = \frac{n_c + mp}{n + m}. \tag{4.2}$$

Pri ocenjevanju pogojnih verjetnosti s številskimi atributi lahko uporabimo dva pristopa. Prvi pristop pretvori vse številske atribute v kategorične s pretvorbo vrednosti v intervale in oceni pogojne verjetnosti z zgoraj opisanim pristopom za kategorične atribute.

Naša implementacija uporabi pristop, ki predpostavi določeno verjetnostno porazdelitev za številske atribute in oceni parametre porazdelitve z

uporabo učne množice. Ponavadi se izbere Gaussova porazdelitev za pogojne verjetnosti številskih atributov. Porazdelitev je parametrizirana z matematičnim upanjem (angl. mean), označenim z μ, in varianco (angl. variance), označeno z σ^2. Za vsak razred c_k izračunamo pogojno verjetnost za številski atribut x_j z enačbo (4.3). Parameter μ_{jk} ocenimo z izračunom matematičnega upanja atributa x_j pri razredu c_k. Parameter σ^2_{jk} ocenimo z enačbo (4.4), kjer l označuje število vrednosti x_j pri razredu c_k.

$$P(x_j = v_j|y = c_k) = \frac{1}{\sqrt{2\pi}\sigma_{jk}} \exp^{-\frac{(v_j-\mu_{jk})^2}{2\sigma^2_{jk}}}. \tag{4.3}$$

$$\sigma^2_{jk} = \frac{1}{l}\sum_{s=1}^{l}(x_{js} - \mu_{jk})^2. \tag{4.4}$$

Klasifikacija

Pri klasifikaciji testnega primera izračunamo aposteriorno verjetnost za vsak razred y (enačba 4.5). Ker je $P(x)$ enak za vsak y, je zadostno izbrati razred, ki maksimira števec enačbe $P(y)\prod_{j=1}^{n} P(x_j|y)$. Testnemu primeru določimo razred y, ki doseže največjo oceno verjetnosti.

$$P(y|x) = \frac{P(y)\prod_{j=1}^{n} P(x_j|y)}{P(x)}. \tag{4.5}$$

Naivni Bayes s paradigmo MapReduce

Implementirali smo algoritem naivni Bayes s paradigmo MapReduce, v katerem ločeno izračunamo pogojne verjetnosti za kategorične in številske atribute. Za lažje razumevanje algoritma smo opis razdelili na dva dela, kjer prvi opisuje izračun s kategoričnimi, drugi pa s številskimi atributi.

Naivni Bayes s kategoričnimi atributi

Funkcija map (algoritem 4.1) sprejme učni primer in za vsak atribut učnega primera tvori pare $\langle ključ, vrednost\rangle$. Vsak par vsebuje razred y, indeks

atributa j in vrednost atributa v, ki predstavljajo ključ in 1, ki predstavlja vrednost. Vrnjen par označi pojavitev vrednosti atributa pri danem razredu. Na koncu funkcija map vrne par $\langle y, 1 \rangle$ s katerim označi pojavitev razreda, kar potrebujemo za izračun apriornih verjetnosti.

```
1 def map(primer, params):
2     x, y = primer
3     for j, v in enumerate(x):
4         yield ((y, j, v), 1) #((razred, indeks, vrednost), 1)
5     yield (y, 1) #označitev pojavitve razreda
```

Algoritem 4.1: Funkcija map z algoritmom naivni Bayes za gradnjo napovednega modela s kategoričnimi vrednostmi atributov.

Združevalnik (algoritem 4.2) se izvede za vsak vrnjen par funkcije map in lokalno združuje pare z enakim ključem. Ko funkcija map vrne zadnji par, združevalnik vrne združene pare, ki jih vsebuje izravnalnik.

```
1 def zdruzevalnik(kljuc,vrednost,izravnalnik,zakljucek,params):
2     if zakljucek: #zadnja iteracija
3         return izravnalnik #vrne združene pare
4     #sešteje vrednosti z enakim ključem
5     izravnalnik[kljuc] = izravnalnik.get(kljuc, 0) + vrednost
```

Algoritem 4.2: Združevalnik z algoritmom naivni Bayes za gradnjo napovednega modela s kategoričnimi vrednostmi atributov.

Funkcija reduce (algoritem 4.3) sortira pare po ključu in s funkcijo kvgroup združi vrednosti po ključu. Funkcija sprejme parameter iterator za prehod po vmesnih parih, ki jih loči glede na število elementov ključa. Če ključ vsebuje en element, predstavlja razred in frekvence, ki jih funkcija sešteje in shrani za izračun apriornih verjetnosti. Pari, pri katerih je ključ sestavljen iz treh elementov predstavljajo frekvence $x_j = v \ \wedge \ y = c$ za izračun pogojnih verjetnosti $P(x|y)$, zato funkcija frekvence sešteje in vrne. Na koncu funkcija reduce izračuna apriorne verjetnosti za vsak razred in jih vrne. Funkcijo

reduce izvaja eno opravilo, saj izračun apriornih verjetnosti zahteva dostop do vseh vrednosti. Funkcija reduce zgradi napovedni model, ki se uporabi v stopnji napovedovanja.

```
1  def reduce(iterator, params):
2      model = {}
3      #združi vrednosti po ključu
4      for kljuc, vrednost in kvgroup(iterator):
5          if len(kljuc) == 1: #frekvence razreda
6              model[kljuc] = sestej(vrednost)
7          elif len(kljuc) == 3:
8              #frekvence x_j = v in y = c
9              yield (kljuc, sestej(vrednost))
10     apriori = izracunaj_apriori(model)
11     yield ("apriori", apriori) #P(y)
```

Algoritem 4.3: Funkcija reduce z algoritmom naivni Bayes za gradnjo napovednega modela s kategoričnimi vrednostmi atributov.

Primer

S primerom opišemo gradnjo napovednega modela z algoritmom naivni Bayes s kategoričnimi atributi na manjši množici podatkov (tabela 4.1). V stopnji map, funkcija map prebere prvi učni primer in nastavi vrednosti spremenljivk $x = [Kratka,\ Visoka]$ in $y = M$. Zanka for obdela x in vrne para $\langle(M,\ 0,\ Kratka),\ 1\rangle$ in $\langle(M,\ 1,\ Visoka),\ 1\rangle$, kjer je ključ določen z $(razred, indeks, vrednost\ atributa)$, vrednost 1 pa označuje pojavitev vrednosti atributa pri danem razredu. Funkcija map vrne dva para, saj ima učni primer dva atributa. Na koncu funkcija map vrne par $\langle M,\ 1\rangle$, ki označi pojavitev razreda v učnem primeru. Ta postopek se ponovi za vsak učni primer. V tabeli 4.1 sta prvi in drugi učni primer enaka in tvorita enake pare.

V primer nismo vključili združevalnika, saj izvrši podobne operacije kot funkcija reduce. Funkcija reduce sortira vmesne pare po ključu, da lahko

Dolžina las	Telesna višina	Spol
Kratka	Visoka	M
Kratka	Visoka	M
Dolga	Srednja	Ž

Tabela 4.1: Primer množice podatkov s fizičnimi opisi ljudi.

funkcija kvgroup pravilno združi vrednosti. Pri tem nastanejo sledeči pari: $\langle(M,\ 0,\ Kratka),\ [1,1]\rangle$, $\langle(M,\ 1, Visoka),\ [1,1]\rangle$, $\langle(\check{Z},\ 0,\ Dolga),\ [1]\rangle$, $\langle(\check{Z},\ 1,\ Srednja),\ [1]\rangle$. Pari označijo pojavitve vrednosti atributov pri danem razredu. Vrednosti prvega in drugega učnega primera so se združile po ključu v seznam $[1,1]$. Prav tako nastaneta sledeča para $\langle M, [1,1]\rangle$ in $\langle\check{Z}, [1]\rangle$, ki označita pojavitev razreda. Funkcija reduce sprejme parameter iterator za prehod po prej navedenih parih. Pri prehodu skozi pare za vsak ključ seštejemo vrednosti. Ključe, ki označujejo frekvence razredov, uporabimo za izračun apriornih verjetnosti, ostale pare pa vrnemo v obliki $\langle(M,\ 0,\ Kratka),\ 2\rangle$. Izračunane apriorne verjetnosti vrnemo, saj skupaj z ostalimi pari tvorijo napovedni model.

Naivni Bayes s številskimi atributi

Funkcija map (algoritem 4.4) je podobna funkciji za izračun pogojnih verjetnosti s kategoričnimi atributi. Pri tvorjenju parov za izračun pogojnih verjetnosti je ključ določen z $(razred, indeks)$, vrednost pa s številsko vrednostjo atributa.

```
1 def map(primer, params):
2     x, y = primer
3     for j, v in enumerate(x):
4         yield ((y, j), v) #((razred, indeks), vrednost)
5     yield (y, 1) #označitev pojavitve razreda
```

Algoritem 4.4: Funkcija map z algoritmom naivni Bayes za gradnjo napovednega modela s številskimi vrednostmi atributov.

Združevalnik (algoritem 4.5) lokalno shranjuje vrednosti za vsak ključ v seznam. Združevalnik vrednosti sešteje, če ključ sestavlja en element. Seštevek predstavlja delne frekvence razredov (delne, saj se lahko istočasno izvaja več združevalnikov). Če ključ sestavljata dva elementa (razred in indeks), združevalnik vrne par s številom elementov, matematičnim upanjem in varianco. Z izračunom teh statistik zmanjšamo omrežen promet, saj funkciji reduce pošljemo delne izračune namesto vseh vrednosti.

```
1  def zdruzevalnik(kljuc, vrednost, izravnalnik, zakljucek, params):
2      if zakljucek: #zadnja iteracija
3          for kljuc, vrednost in izravnalnik:
4              if len(kljuc) == 1:
5                  #frekvence razredov
6                  izravnalnik[kljuc] = sestej(vrednost)
7              elif len(kljuc) == 2:
8                  #delni izračuni matematičnega upanja in variance
9                  izravnalnik[kljuc] = (len(vrednost), mat_up(
                       vrednost), varianca(vrednost))
10         yield izravnalnik
11     #združevanje vrednosti po ključu
12     elif kljuc in izravnalnik:
13         izravnalnik[kljuc].append(vrednost)
14     else:
15         izravnalnik[kljuc] = [vrednost]
```

Algoritem 4.5: Združevalnik z algoritmom naivni Bayes za gradnjo napovednega modela s številskimi vrednostmi atributov.

Funkcija reduce (algoritem 4.6) združi delne vrednosti statistik v končni izračun. Sprva sortira pare po ključu in s funkcijo kvgroup združi vrednosti po ključu. S sortiranjem pare razvrsti po razredih in indeksih atributov. Tako lahko obdela vse vrednosti atributov za posamezen razred, izračuna statistike in jih shrani v vektor. Matematično upanje (μ) iz dveh delnih izračunov izračunamo z enačbo (4.6), kjer n označuje število vrednosti, s katerim smo izračunali delni μ. V [16] predstavijo enačbo (4.7) za natančen

izračun variance (σ) iz dveh delnih izračunov.

```
1  def reduce(iterator, params):
2      model = {}, mat_up = [], varianca = []
3      prejsnji_kljuc = ""
4      #zdruzi vrednosti po kljucu
5      for kljuc, vrednost in kvgroup(iterator):
6          if len(kljuc) == 1: #frekvence razredov
7              model[kljuc] = sestej(vrednost)
8          elif len(kljuc) == 2: #razred in indeks
9              if kljuc != prejsnji_kljuc: #ali je drug razred
10                 yield (kljuc, mat_up)
11                 yield (kljuc, varianca)
12                 mat_up = [], varianca = [] #poenostavitev
13             else:
14                 mat_up[kljuc] = izracunaj_mat_up(vrednost)
15                 varianca[kljuc] = izracunaj_varianco(vrednost)
16                 prejsnji_kljuc = kljuc
17     apriori = izracunaj_apriori(model)
18     yield ("apriori", apriori) #P(y)
```

Algoritem 4.6: Funkcija reduce z algoritmom naivni Bayes za gradnjo napovednega modela s številskimi vrednostmi atributov.

$$\mu_{ab} = \frac{n_a\mu_a + n_b\mu_b}{n_a + n_b}. \tag{4.6}$$

$$\sigma_{ab} = \frac{n_a\sigma_a + n_b\sigma_b}{n_a + n_b} + n_a n_b \left(\frac{\mu_b - \mu_a}{n_a + n_b}\right)^2. \tag{4.7}$$

Funkcija reduce vrne vektor matematičnega upanja in vektor variance za vsak atribut. Vektorske podatkovne strukture omogočajo matrično množenje in v stopnji napovedovanja pohitrijo napovedovanje testnih primerov. Na koncu funkcija reduce izračuna apriorne verjetnosti za vsak razred in jih vrne. Te vrednosti tvorijo napovedni model naivnega Bayesa s številskimi atributi.

4.3.2 Linearna regresija

Linearna regresija predpostavlja, da so vsi atributi številski ter da obstaja linearna relacija med regresijsko spremenljivko y in atributi x. Linearna funkcija nad x se imenuje hipoteza in aproksimira y. Določa jo enačba $h(x) = \theta_1 x_1 + \theta_0$, kjer je x vektor z 1 dimenzijo. Za poenostavitev notacije dodamo $x_0 = 1$, saj to omogoča vektorski zapis učnega primera x in parametrov θ. Enačba (4.8) določa hipotezo, kjer parametri θ predstavljajo uteži.

$$h(x) = \sum_{j=0}^{n} \theta_j x_j = \theta^T x. \tag{4.8}$$

Linearna regresija določi parametre θ s prileganjem $h(x)$ odvisni spremenljivki y. Cenilna funkcija $J(\theta)$ to formalizira in za vrednosti θ izmeri, kako blizu je $h(x)$ vrednostim y. Cenilka je določena z enačbo (4.9).

$$J(\theta) = \frac{1}{2} \sum_{i=1}^{m} (h(x^{(i)}) - y^{(i)})^2. \tag{4.9}$$

Poznamo več načinov za minimizacijo cenilne funkcije, ki se razlikujejo po hitrosti konvergence. Na primer, paketni najstrmejši spust in stohastični najstrmejši spust. Oba pristopa potrebujeta več prehodov skozi učno množico podatkov, da aproksimacija konvergira. Analitična rešitev je metoda, s katero izračunamo eksaktne vrednosti parametrov θ brez minimizacije cenilne funkcije. Ta metoda je najprimernejša za izračun parametrov linearne regresije s paradigmo MapReduce in jo uporabimo v naši implementaciji.

Linearna regresija s paradigmo MapReduce

Linearna regresija prilega parametre θ učnim primerom z uporabo enačbe $\theta^* = A^{-1}b$, kjer izračunamo $A = \sum_{i=1}^{m} (x^{(i)}(x^{(i)})^T)$ in $b = \sum_{i=1}^{m} (x^{(i)} y^{(i)})$ z m učnimi primeri, kjer $x^{(i)}$ označuje učni primer, $y^{(i)}$ pa razred primera i. Za izračun v sumarni obliki sprva izračunamo A in b s funkcijo map, operacije na matrikah pa izračunamo v funkciji reduce. Za vektorski zapis

vsak učni primer razširimo s členom 1 (angl. intercept term), kar zapišemo z $x = \begin{bmatrix}1\\x\end{bmatrix}$. Funkcija map (algoritem 4.7) izračuna $\sum_{podmnožica}(x^{(i)}(x^{(i)})^T)$ in $\sum_{podmnožica}(x^{(i)}y^{(i)})$ ter vrne delne izračune matrik A in b za vsak učni primer.

```
1 def map(primer, params):
2     x, y = primer
3     A = zunanji_produkt(x, x)
4     b = vektorski_produkt(x, y)
5     yield ("Ab", (A,b))
```

Algoritem 4.7: Funkcija map z linearno regresijo za izračun parametrov θ.

Združevalnik (algoritem 4.8) sešteje delne izračune matrik A in b in jih vrne.

```
1 def zdruzevalnik(kljuc,vrednost,izravnalnik,zakljucek,params):
2     if zakljucek: #zadnja iteracija
3         return izravnalnik
4     A, b = vrednost
5     izravnalnik["A"] = izravnalnik.get("A", 0) + A
6     izravnalnik["b"] = izravnalnik.get("b", 0) + b
```

Algoritem 4.8: Združevalnik z linearno regresijo za izračun parametrov θ.

Ker združevalnik vrača le ključa A in b ni potrebno sortirati in združevati parov po ključu. Funkcija reduce (algoritem 4.9) sešteje matriki A in b in izračuna enačbo $\theta^* = A^{-1}b$. Funkcija reduce vrne parametre θ, ki se prilegajo učnim primerom. V stopnji napovedovanja vsak testni primer razširimo s členom 1, kar zapišemo z $x = \begin{bmatrix}1\\x\end{bmatrix}$. S tem omogočimo vektorsko množenje s parametri θ, izračunana vrednost pa predstavlja napoved testnega primera.

4.3.3 Lokalno utežena linearna regresija

Izbira kvalitetnih atributov je pomembna za zagotovitev uspešnosti učnega algoritma. Lokalno utežena linearna regresija (LOESS, angl. local regres-

```
1 def reduce(iterator, params):
2     for kljuc, vrednost in iterator:
3         if kljuc == "A"
4             A = sum(vrednost)
5         else:
6             b = sum(vrednost)
7     thetas = vektorski_produkt(inverz(A), b)
8     output("thetas", thetas)
```

Algoritem 4.9: Funkcija reduce z linearno regresijo za izračun parametrov θ.

sion) spada med neparametrične algoritme, pri katerih je izbira atributov manj pomembna pri zadostni učni množici. Neparametrični algoritmi nimajo fiksnega števila parametrov, saj je število parametrov odvisno od učne množice. Za razliko od linearne regresije, LOESS pri napovedovanju poleg parametrov θ potrebuje še učno množico. LOESS v oceno napovedi hipoteze vključi uteži (enačba 4.10), kjer $w^{(i)}$ predstavlja utež z nenegativno vrednostjo. Intuitivno, če je $w^{(i)}$ velika utež na mestu i, je težko zmanjšati napako $(y^{(i)} - \theta^T x^{(i)})^2$. Če je $w^{(i)}$ majhen, se napaka skoraj ignorira.

$$\sum_i w^{(i)}(y^{(i)} - \theta^T x^{(i)})^2, \tag{4.10}$$

kjer je

$$w^{(i)} = \exp\left(-\frac{(x^{(i)} - x)^2}{2\tau^2}\right). \tag{4.11}$$

Enačba (4.11) predstavlja Gaussovo utežno funkcijo, ki se uporablja z LOESS. Vrednost uteži je odvisna predvsem od točke $x^{(i)}$, pri kateri poskušamo oceniti x. Če je $|x^{(i)} - x|$ manjša vrednost, je $w^{(i)}$ blizu 1. Zato izbran parameter θ daje večjo utež učnim primerom, ki so bliže točki x, ki jo želimo napovedati. Parameter τ določa, kako hitro se utež učnega primera zmanjšuje z razdaljo med $x^{(i)}$ in x.

Lokalno utežena linearna regresija s paradigmo MapReduce

Lokalno utežena linearna regresija (LOESS) s paradigmo MapReduce poišče rešitev enačbe $A\theta = b$, kjer sta

$$A = \sum_{i=1}^{m} w^{(i)}(x^{(i)}(x^{(i)})^T),$$

$$b = \sum_{i=1}^{m} w^{(i)}(x^{(i)}y^{(i)}),$$

kjer $w^{(i)}$ označuje utež, $x^{(i)}$ označuje učni primer, $y^{(i)}$ pa razred primera i. Za izračun v sumarni obliki s funkcijo map izračunamo podmnožici za A in b, funkcija reduce pa sešteje podmnožici in izračuna enačbo $\theta = A^{-1}b$.

LOESS naredi prehod skozi učno množico za napoved testnega primera. To pri velikem številu testnih primerov predstavlja preveč režije. Naša implementacija LOESS zahteva manj prehodov skozi učno množico, saj pri enem prehodu izračunamo parametre θ za več testnih primerov (algoritem 4.10). Pri branju testne množice, testne primere shranjujemo v slovar. Določimo maksimalno število testnih primerov, ki jih lahko shranimo, saj je slovar omejen z glavnim pomnilnikom računalnika, velikosti testnih primerov pa se razlikujejo med podatkovnimi množicami. Slovar s podmnožico testnih primerov podamo poslu s parametrom params, ki je dostopen vsem opravilom v poslu. To je pomembno, saj je potrebno za izračun parametrov θ testnega primera uporabiti vse učne primere, ki jih preberejo različna opravila map. Posel izračuna parametre θ in regresijsko spremenljivko za testne primere. Algoritem vrne seznam povezav URL do rezultatov.

Funkcija map (algoritem 4.11) s parametrom params dostopa do slovarja testnih primerov. Posamezen učni primer uporabimo pri izračunu utežne funkcije in izračunu podmnožic matrik A in b za vse testne primere. Funkcija map vrne par, kjer identifikator testnega primera predstavlja ključ, podmnožici matrik A in b pa vrednost.

Funkcija map vrne število parov, ki je enako številu tesnih primerov v slovarju. Za zmanjšanje obremenitve omrežja, združevalnik (algoritem 4.12) združuje podmnožici matrik A in b za testne primere z enakim identifikatorjem.

```
def izracunaj_thetas(ucna_mnozica, testna_mnozica, tau = 1):
    rezultati_url = [], testni_primeri = {}
    preberi_testne_primere = posel(testna_mnozica)

    for id, x in preberi_testne_primere:
        testni_primeri[id] = x
        #preverimo število primerov v slovarju
        if max_kapaciteta(testni_primeri):
            params = Params(testni_primeri, tau)
            #izračunamo thetas za podmnožico testnih primerov
            thetas = posel(ucna_mnozica, params)
            rezultati_url.append(thetas)
            testni_primeri = {}
    return rezultati_url
```

Algoritem 4.10: Izračun parametrov θ z lokalno uteženo linearno regresijo.

```
def map(primer, params):
    xi, y = primer
    for id, x in params.testni_primeri:
        w = utezna_funkcija(xi, x, params.tau)
        sub_A = w * zunanji_produkt(xi, xi)
        sub_b = w * xi * y
        yield (id, (sub_A, sub_b))
```

Algoritem 4.11: Funkcija map z lokalno uteženo linearno regresijo za izračun parametrov θ.

Funkcija reduce (algoritem 4.13) izvaja podobno operacijo kot združevalnik, le da sešteva vse podmnožice matrik za testne primere z enakim identifikatorjem v končni matriki A in b. Na začetku pare sortiramo po ključu in s funkcijo kvgroup združimo vrednosti po ključu (identifikatorju tesnih primerov). Za vsak testni primer seštejemo podmnožici matrik A in b in izračunamo parametre θ z enačbo $\theta = A^{-1}b$. Funkcija reduce vrne par za vsak testni primer, kjer identifikator predstavlja ključ, parametri θ in napovedana regresijska spremenljivka pa vrednost. V funkciji reduce po-

```
1 def zdruzevalnik(kljuc,vrednost,izravnalnik,zakljucek,params):
2     if zakljucek:
3         return izravnalnik
4     sub_A, sub_b = vrednost
5     izravnalnik[kljuc] = izravnalnik.get(kljuc, (0,0)) + (sub_A,
          sub_b)
```

Algoritem 4.12: Združevalnik z lokalno uteženo linearno regresijo za izračun parametrov θ.

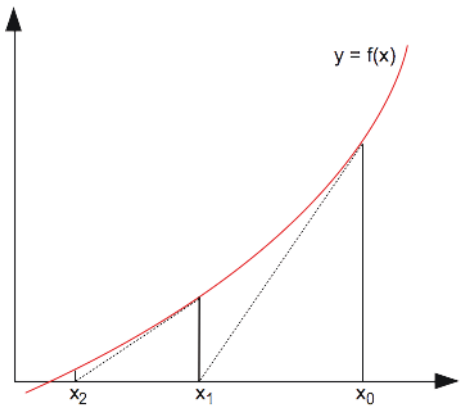
leg parametrov θ izračunamo tudi napoved zato napovedovanje ni potrebno. Opravila reduce se lahko izvajajo vzporedno, saj stopnja mešanje posreduje vse pare s skupnim identifikatorjem enakem opravilu reduce.

```
1 def reduce(iterator, params):
2     for id, vrednost in kvgroup(iterator):
3         A, b = 0, 0
4         for sub_A, sub_b in vrednost:
5             A += sub_A
6             b += sub_b
7         thetas = vektorski_produkt(inverz(A), b)
8         napoved = vektorski_produkt(thetas, params.
              testni_primeri[id])
9         yield (id, (thetas, napoved))
```

Algoritem 4.13: Funkcija reduce z lokalno uteženo linearno regresijo za izračun parametrov θ in napoved.

4.3.4 Logistična regresija

Logistična regresija uporabi linearno regresijo za klasifikacijo. Lahko bi ignorirali dejstvo, da y zavzema kategorične vrednosti in ga poskušali napovedati z danim x, vendar ta pristop ni smiseln, saj bi $h(x)$ zavzemala vrednosti večje od 1 in manjše od 0. Hipoteza logistične regresije se imenuje logistična funkcija in vedno vrača vrednosti med 0 in 1. Določa jo enačba (4.12).



Slika 4.1: Prvi trije približki izračunane ničle funkcije f z Newtonovo metodo.

$$h(x) = g(\theta^T x) = \frac{1}{1 + e^{-\theta^T x}}, \tag{4.12}$$

Za logistično regresijo ne obstaja analitična rešitev kot pri linearni regresiji, zato je potrebno uporabiti metodo, ki minimizira kriterijsko funkcijo in s tem določi parametre θ. Uporabili smo Newtonovo metodo za iskanje ničel funkcije, saj ta tipično konvergira hitreje kot paketni najstrmejši spust. Iteracija Newtonove metode je računsko zahtevnejša, saj zahteva iskanje inverza matrike, konvergira pa veliko hitreje, dokler atributni prostor ni prevelik. Osnovno idejo Newtonove metode prikaže slika 4.1, podrobnejši opis metode pa je pri opisu implementacije s paradigmo MapReduce.

1. Izberemo začetni približek x_0 za ničlo funkcije f.
2. V točko x_i postavimo tangento in poiščemo ničlo tangente.
3. Ničlo tangente izberemo za naslednji približek x_{i+1}.
4. Postopek nadaljujemo s ponavljanjem korakov 2. in 3. dokler funkcija ne konvergira.

Logistična regresija s paradigmo MapReduce

Logistična regresija s paradigmo MapReduce je binarni klasifikator, ki prilagodi parametre θ. Za posodobitev parametrov θ uporabimo Newtonovo metodo $\theta = \theta - H^{-1}\nabla_\theta \ell(\theta)$. Za sumarno obliko izračunamo podskupine gradienta $\nabla_\theta \ell(\theta)$ z opravili map, ki izračunajo $\sum_{podmnožica}(y - h(x))x_j$ in Hessovo matriko $H(j,k) = H(j,k) + h(x)(h(x) - 1)x_j x_k$. Podskupine gradienta in Hesseva matrika se lahko vzporedno izračunata z opravili map. Logistična regresija parametre θ posodobi vsako iteracijo, kjer ena iteracija predstavlja en posel MapReduce. Parametri θ se shranijo v objekt params in se podajo kot parameter poslu. Funkcija map (algoritem 4.14) izračuna hipotezo z učnim primerom in parametri θ iz prejšnje iteracije ter izračuna in vrne podskupine gradienta in Hessove matrike.

```
1 def map(primer, params):
2     x, y = primer
3     h = izracunaj_hipotezo(x, params.thetas)
4     yield ("grad", izracunaj_gradient(x, y, h))
5     yield ("H", izracunaj_hessovo_mat(x, h))
```

Algoritem 4.14: Funkcija map z logistično regresijo za izračun parametrov θ.

Združevalnik (algoritem 4.15) združi podskupine gradienta in Hessove matrike in ju vrne.

```
1 def zdruzevalnik(kljuc,vrednost,izravnalnik,zakljucek,params):
2     if zakljucek:
3         return izravnalnik #vrne združene pare
4     #sešteje vrednosti z enakim ključem
5     izravnalnik[kljuc] = izravnalnik.get(kljuc, 0) + vrednost
```

Algoritem 4.15: Združevalnik z logistično regresijo za izračun parametrov θ.

Funkcija reduce (algoritem 4.16) sprejme iterator za prehod po parih, ki predstavljajo podskupine gradienta in Hessove matrike. S funkcijo kvgroup združimo vrednosti z enakim ključem, nato pa seštejemo vrednosti matrike in

gradienta, posodobimo parametre θ in jih vrnemo. Ta postopek se ponavlja do konvergence ali pa do števila iteracij, ki jih določi uporabnik. Napoved razreda izračunamo enako kot pri linearni regresiji, le da so napovedi 0 ali 1 zaradi uporabe logistične funkcije.

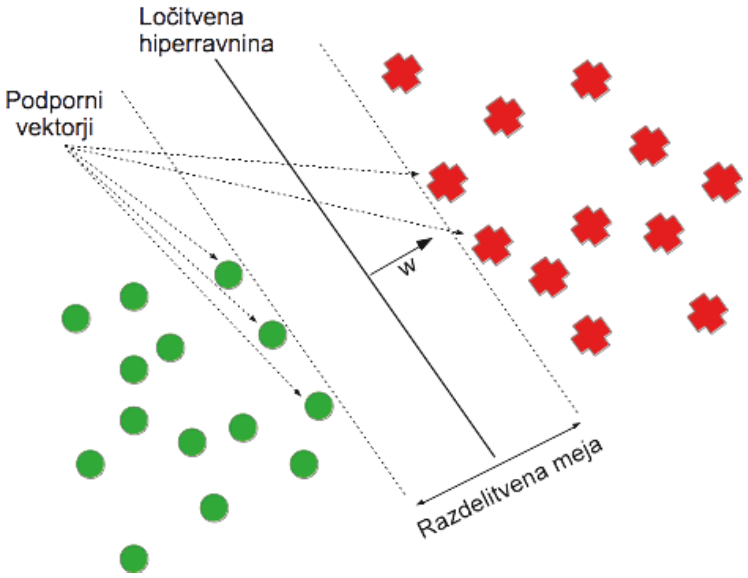
```
1 def reduce(iterator, params):
2     for kljuc, vrednost in kvgroup(iterator):
3         if kljuc == "H":
4             H = sum(vrednost)
5         else:
6             grad = sum(vrednost)
7     thetas = params.thetas - vektorski_produkt(inverz(H), grad)
8     yield ("thetas", thetas)
```

Algoritem 4.16: Funkcija reduce z logistično regresijo za izračun parametrov θ.

4.3.5 Metoda podpornih vektorjev

Metoda podpornih vektorjev (SVM) [2] spada med algoritme nadzorovanega učenja z različicami algoritma za klasifikacijo in regresijo. V nadaljevanju opisujemo linearni SVM, ki se uporablja za klasifikacijo linearno ločljivih binarnih razredov.

Linearni SVM išče hiperravnino z največjo razdelitveno mejo, ki ločijo različna razreda (slika 4.2). SVM razdelitveno mejo predstavi s podmnožico učnih primerov, ki jih imenujemo podporni vektorji. Podporni vektorji predstavljajo točke v prostoru, ki so blizu ločevalne hiperravnine. Pri linearno ločljivih učnih primerih obstaja neskončno mnogo hiperravnin, ki loči različna razreda. Čeprav vsaka hiperravnina enakovredno loči razreda učnih primerov, to ni zagotovilo, da bodo hiperravnine enako dobro ločevale testne primere. Klasifikator zato izbere hiperravnino, za katero pričakuje, da bo najbolje ločila razrede testnih primerov. Hiperravnine z veliko razdelitveno mejo minimizirajo posplošitveno napako (angl. generalization error). Če je razde-



Slika 4.2: Hiperravnina binarnega SVM z razdelitveno mejo.

litvena meja ozka, ima že manjši šum velik učinek na klasifikacijo, prav tako pa je takšen model bolj prilagojen učni množici.

Metoda podpornih vektorjev s paradigmo MapReduce

Implementirali smo linearne SVM, ki je opisan v [17]. Linearni SVM dodeli učne primere najbližji izmed dveh vzporednih ravnin, ki so čim bolj razmaknjene, ter izračuna rešitev sistema linearnih enačb. Klasičen SVM dodeli primere enemu izmed dveh podprostorov in izračuna linearno ali kvadratno funkcijo, kar zahteva več časa. Linearni SVM vzdržuje pozitivno definitno matriko [1] dimenzije $(n+1) \times (n+1)$, ki vsebuje sistem linearnih enačb, kjer n označuje število atributov učne množice.

Linearni SVM ima naslednje lastnosti.

- Za gradnjo napovednega modela zahteva en prehod čez učno množico

[1]Realna simetrična matrika M, razsežnosti $n \times n$, je pozitivno definitna, če za vse neničelne vektorje z z realnimi elementi velja $z^T Mz > 0$. Uporabljamo jih v optimizacijskih algoritmih in pri gradnji različnih regresijskih modelov.

podatkov.

- Pomnilniška zahtevnost je reda $(n+1)^2$.
- Časovna zahtevnost je reda $(n+1)^3$.
- Stara učna množica se po učenju opusti, medtem ko lahko nova posodobi napovedni model.
- Velike množice podatkov lahko klasificiramo inkrementalno.
- Veliko učno množico velikosti $m \times n$ lahko zgostimo na velikost $(n+1)^2$ za uporabo z linearnim SVM.

Klasifikator predpostavlja, da so atributi številski s pozitivnimi vrednostmi, pri katerih je binarni razred označen z -1 ali 1. Učenje modela linearnega SVM poteka po naslednjem postopku. Dana učna množica z n atributi in m primeri se shrani v matriko A in ustvari se diagonalna matrika D, ki na diagonali vsebuje oznake razredov (-1 in 1) vsakega primera v A. Določi se matrika E z enačbo (4.13), kjer je e vektor števil 1 dimenzije $m \times 1$. Izračuna se $\left[\begin{smallmatrix} w \\ \gamma \end{smallmatrix}\right]$, kot določa enačba (4.14), za pozitiven ν, kjer ν predstavlja parameter za prilagoditev klasifikatorja.

$$E = [A \quad -e]. \tag{4.13}$$

$$\begin{bmatrix} w \\ \gamma \end{bmatrix} = \left(\frac{I}{\nu} + E^T E\right)^{-1} E^T D e. \tag{4.14}$$

Z opravili map lahko porazdeljeno izračunamo $E^T E$ in $E^T De$. Funkcija map (algoritem 4.17) z vhodnim primerom določi matriko A. Diagonalne matrike D ni potrebno ustvariti, saj vsebuje le razred primera, ki se trenutno obdeluje in je dimenzije 1×1. Funkcija map izračuna $E^T E$, ki je dimenzije $(n+1) \times (n+1)$, in $E^T De$, ki je dimenzije $(n+1) \times 1$, za posamezen primer in ju vrne.

```
1 def map(primer, params):
2     A, D = primer
3     e = vektor_enic(len(A), 1)
4     E = zdruzi_po_stolpcu(A, -e)
5     ETE = notranji_produkt(trasponiraj(E), E)
6     ETDe = notranji_produkt(trasponiraj(E), D, e)
7     yield ("kljuc", (ETE, ETDe))
```

Algoritem 4.17: Funkcija map z linearnim SVM za učenje modela.

Združevalnik (algoritem 4.18) zmanjša prenos podatkov preko omrežja, saj funkcija map za vsak primer vrne dve matriki. Združevalnik sešteje matriki ETE in $ETDe$ za vse učne primere posameznega razdelka in ju vrne.

```
1 def zdruzevalnik(kljuc,vrednost,izravnalnik,zakljucek,params):
2     if zakljucek:
3         return izravnalnik
4     ETE, ETDe = vrednost
5     izravnalnik["ETE"] = izravnalnik.get("ETE", 0) + ETE
6     izravnalnik["ETDe"] = izravnalnik.get("ETDe", 0) + ETDe
```

Algoritem 4.18: Združevalnik z linearnim SVM za učenje modela.

Pri linearnem SVM ni potrebno sortirati in združiti vrednosti po ključu, saj opravila map vračajo le pare s ključi ETE in $ETDe$. Za izračun parametrov seštejemo matriki ETE in $ETDe$ in ustvarimo enotsko matriko I dimenzije $(n+1) \times (n+1)$, ki jo delimo s parametrom ν. Nato izračunamo enačbo (4.14) in vrnemo parametre modela linearnega SVM dimenzije $(n+1) \times 1$.

Pri klasifikaciji z modelom linearnega SVM vsak testni primer x razširimo $z = \left[\begin{smallmatrix} x \\ -1 \end{smallmatrix}\right]$, saj to zahteva algoritem. Napoved tesnega primera določimo z enačbo (4.15).

$$sign(z^T\big(\frac{I}{\nu} + E^T E\big)^{-1} E^T De)\begin{cases} = 1, \; razred\, 1 \\ = -1, \; razred\; -1 \end{cases}. \tag{4.15}$$

```
1  def reduce(iterator, params):
2      sum_ETE, sum_ETDe = 0,0
3      for kljuc, vrednost in iterator:
4          if kljuc == "ETE":
5              sum_ETE += vrednost
6          else:
7              sum_ETDe += vrednost
8      I = enotska_matrika(sum_ETE.dimenzija_x)
9      sum_ETE += I/params.nu
10     yield ("kljuc", vektorski_produkt(inverz(sum_ETE), sum_ETDe)
           )
```

Algoritem 4.19: Funkcija reduce z linearnim SVM za učenje modela.

4.3.6 Razvrščanje z voditelji

Razvrščanje z voditelji (angl. k-means) je metoda nenadzorovanega učenja, ki učne primere s številskimi vrednostmi atributov razvrsti v k skupin. Središče posamezne skupine se imenuje centroid. Na začetku algoritem izbere k začetnih centroida, kjer parameter k poda uporabnik. Vsak učni primer dodeli najbližjemu centroidu z izračunom evklidske razdalje. Nabor primerov, ki pripada centroidu, predstavlja skupino. Razvrstitev primerov po skupinah lahko ocenimo z vsoto kvadratov napake. Centroid vsake skupine se posodobi na podlagi primerov v skupini. Postopek dodelitve primerov in posodobitve centroidov se ponavlja dokler noben primer ne zamenja skupine, ali ekvivalentno, dokler centroidi ne ostanejo nespremenjeni.

Razvrščanje z voditelji s paradigmo MapReduce

Pri razvrščanju z voditelji s paradigmo MapReduce lahko vzporedno računamo evklidsko razdaljo med učnimi primeri in centroidi. V začetni iteraciji funkcija map naključno razdeli učne primere v k skupin. Povprečje vrednosti primerov v skupini uporabimo za določitev centroidov. Posel sprejme skupine s centroidi in jih uporabi pri izračunu novih skupin. Opisani postopek se ponavlja dokler ne dosežemo zastavljenega števila iteracij, ki ga določi

uporabnik. Ena iteracija predstavlja en posel MapReduce.

Funkcija map (algoritem 4.20) sprejme parameter primer in izračuna evklidsko razdaljo med primerom in vsakim centroidom. Primer dodeli najbližji skupini. Funkcija map vrne identifikator skupine kot ključ in element kot vrednost.

```
1 def map(primer, params):
2     razdalje = izracunaj_razdalje(primer, params.skupine)
3     skupina_id = min(razdalje)
4     yield (skupina_id, primer)
```

Algoritem 4.20: Funkcija map pri razvrščanju z voditelji.

Združevalnik (algoritem 4.21) sešteje vrednosti in števec primerov za posamezno skupino. Za vsako skupino vrne identifikator, seštevek vrednosti primerov in števec primerov.

```
1 def zdruzevalnik(kljuc,vrednost,izravnalnik,zakljucek,params):
2     if zakljucek:
3         return izravnalnik
4     elif kljuc in izravnalnik:
5         izravnalnik[kljuc] = vrednost
6     else:
7         izravnalnik[kljuc] = posodobi(izravnalnik[kljuc],
              vrednost)
```

Algoritem 4.21: Združevalnik pri razvrščanju z voditelji.

Razvrščanje z voditelji ne sortira parov in združuje vrednosti po ključu, saj podobno operacijo opravlja funkcija reduce. Funkcija reduce (algoritem 4.22) vsaki skupini sešteje vrednosti in števec primerov. Za vsako skupino izračuna povprečje vrednosti primerov (centroid) in vrne identifikator skupine z njenim središčem. Vzporedno lahko izvajamo več opravil reduce, saj se skupine z enakim identifikatorjem posredujejo enakemu opravilu reduce. Parameter k je enak številu opravil reduce, ki se izvede v posameznem poslu.

```
1 def reduce(iterator, params):
2     skupine = {}
3     for skupina_id, primeri in iterator:
4         skupine[skupina_id] = posodobi(skupine[skupina_id],
              primeri)
5     for skupina_id, primeri in skupine:
6         yield (skupina_id, povprecje(primeri))
```

Algoritem 4.22: Funkcija reduce pri razvrščanju z voditelji.

Testnemu primeru določimo skupino z minimalno razdaljo med centroidom in testnim primerom. Implementacija algoritma razvrščanja z voditelji prihaja v sklopu ogrodja Disco, kateri pa smo dodali združevalnik in jo prilagodili za delovanje s knjižnico DiscoMLL. Implementacija algoritma ne pregleduje konvergence centroida in ne obravnava praznih skupin.

4.4 Drevesne metode

Naključni gozdovi [2] so ansambelski algoritmi, ki za klasifikacijo uporabljajo odločitvena drevesa. Ansambelski algoritmi združujejo več različnih modelov za dosego večje točnosti napovedovanja. V magistrski nalogi predstavimo tri različice porazdeljenih naključnih gozdov, ki odločitvena drevesa zgradijo na lokalnih podatkovnih množicah. Opis pričnemo s predstavitvijo odločitvenih dreves in mer za ocenjevanje atributov. V pregledu področja opišemo že obstoječe različice naključnih gozdov, ki delujejo na velikih podatkih, nato pa opišemo razvite različice porazdeljenih naključnih gozdov.

4.4.1 Odločitveno drevo

Odločitveno drevo [2] je klasifikator, ki se pogosto uporablja pri analizi podatkov. Z odločitvenim drevesom rešimo klasifikacijski problem z zaporedjem testov na atributih testnega primera. Zaporedje testov tvori obliko odločitvenega drevesa, ki je hierarhična struktura sestavljena iz vozlišč in

usmerjenih povezav. Drevo ima tri tipe vozlišč:

- koren nima vhodnih povezav in ima nič ali več izhodnih povezav,
- notranje vozlišče ima natanko eno vhodno povezavo in dve ali več izhodnih povezav,
- list ali končno vozlišče ima natanko eno vhodno povezavo in nič izhodnih povezav.

Vsak list v odločitvenemu drevesu označuje določen razred, povezave predstavljajo vrednosti atributov, koren in ostala notranja vozlišča pa vsebujejo pravila, ki razločijo primere z različnimi lastnostmi. Odločitveno drevo klasificira testni primer z uporabo pravil od korena do lista odločitvenega drevesa. Izbrani list predstavlja napoved za testni primer.

Iz atributov dane učne množice lahko zgradimo ogromno odločitvenih dreves, kjer so nekatera drevesa točnejša od drugih. Iskanje optimalnega odločitvenega drevesa zaradi eksponentne velikosti iskalnega prostora ni mogoče. Algoritmi za gradnjo odločitvenih dreves ponavadi uporabljajo požrešno metodo, ki zgradi odločitveno drevo s serijo lokalno optimalnih odločitev glede izbranega atributa za razdelitev podatkov. Osnovni algoritem učenja so predstavili (Hunt in sodelavci, 1962), ki gradi odločitveno drevo na rekurziven način z razdeljevanjem učnih primerov v čistejše podmnožice. V vsakem rekurzivnem koraku gradnje odločitvenega drevesa se izbere atribut, ki razdeli učno množico na manjše podmnožice.

Implementirali smo algoritem za gradnjo odločitvenih dreves, ki vozlišča razvija v prioritetnem vrstnem redu glede na kvaliteto atributa. To nam omogoča, da razvijemo določeno število najperspektivnejših vozlišč v drevesu. Vozlišča razcepljamo binarno, saj se s tem izognemo precenjevanju večvrednostih atributov. Algoritem lahko izmeri kvaliteto vseh atributov ali pa le naključno izbrane podmnožice atributov v vsakem vozlišču, kar omogoča gradnjo raznolikih odločitvenih dreves.

Naša implementacija (algoritem 4.23) hrani model odločitvenega drevesa v slovarju, kjer je ključ določen z identifikatorjem starša, vrednost pa s se-

znamom, ki vsebuje levo in desno vozlišče razcepa. Vozlišče določa peterka, ki vsebuje identifikator sina, indeks najboljšega atributa, vrednost za razcep, porazdelitev razredov v vozlišču in globino vozlišča v drevesu. Na začetku izvajanja izmerimo kvaliteto atributov in določimo razcep. Atribut z višjo kvaliteto ima prednost v prioritetni vrsti. Iz učne množice ustvarimo dve podmnožici glede na razcep. Za vsako podmnožico izmerimo porazdelitev razredov in jo shranimo v model odločitvenega drevesa. V primeru, da je podmnožica primerna za nadaljnji razcep, ponovno ocenimo kvaliteto atributov in podmnožico z razcepom podamo v prioritetno vrsto. Postopek gradnje se zaključi z izpraznjeno prioritetno vrsto ali pa z razvitjem zadostnega števila vozlišč.

Napovedovanje (algoritem 4.24) je sprehod od korena do lista. Napoved izračunamo s porazdelitvijo razredov v listu in izberemo najverjetnejši razred. Pri kategoričnih vrednostih se lahko zgodi, da določene vrednosti ni ne v levem, ne v desnem razcepu. V takšnem primeru uporabimo porazdelitev razredov starša za napoved. Z metodo napovedovanja lahko izračunamo tudi zaupanje v napoved [18], ki izmeri razliko med verjetnostjo pravega razreda in verjetnostjo njemu neenakega razreda z največjo verjetnostjo. Na primer, da so verjetnosti napovedanih razredov A, B in C, $P(A) = 0.6$, $P(B) = 0.3$, $P(C) = 0.1$. Če je pravilni razred A, izračunamo zaupanje v napoved, $zaupanje = 0.6 - 0.3 = 0.3$. Če je pravilni razred C izračunamo zaupanje v napoved, $zaupanje = 0.1 - 0.6 = -0.5$.

Binarni razcep atributov

Pri iskanju najboljšega binarnega razcepa številskih in kategoričnih atributov uporabimo različni metodi. Za številske atribute izvedemo diskretizacijo, saj uporabljene mere za ocenjevanje kvalitete atributov delujejo le s kategoričnimi atributi. Implementirali smo dve metodi za diskretizacijo, kjer prva številske vrednosti atributov naključno razbije na določeno število kategoričnih intervalov. Druga metoda primere razdeli v kategorične intervale z enakim številom primerov. Metoda sprejme parameter za število intervalov

```
def gradnja_modela(ucna_mnozica, st_vozlisc, mera):
  #model_drevesa = {id_starša:
  #[(id_levi_sin, naj_atr, razcep, porazdelitev, globina),
  #(id_desni_sin, naj_atr, razcep, porazdelitev, globina)]}

  vrsta = prioritetna_vrsta()
  drevo = {}
  id_sin, globina = 0, 0
  #določi indeks najboljšega atributa in razcep
  naj_atr, razcep = mera(ucna_mnozica)
  vrsta.put(naj_atr, (ucna_mnozica, razcep, id_sin, globina))

  while not vrsta.empty() and len(drevo) < st_vozlisc:
    naj_atr, (ucna_mnozica, razcep, id_stars, globina) = vrsta.
        get()

    for i in range(2): #za levi in desni razcep
      id_sin += 1 #ustvari nov indentifikator
      #razcepi učno množico glede na razcep
      podmnozica = razcepi(ucna_mnozica, naj_atr, razcep)
      #izračunaj porazdelitev razredov v vozlišču
      porazdelitev = porazdelitev_razredov(podmnozica)
      #dodaj vozlišče v model drevesa
      drevo[id_stars].append((id_sin, naj_atr, razcep,
          porazdelitev, globina))

      #oceni, če lahko vozlišče še razcepimo
      if pogoji_za_razcep(podmnozica):
          #določi indeks najboljšega atributa in razcep
          novi_max_atribut, novi_razcep = mera(podmnozica)
          vrsta.put(novi_max_atribut, (podmnozica, novi_razcep,
              id_vozlisca, globina+1))

  return drevo
```

Algoritem 4.23: Algoritem za gradnjo odločitvenega drevesa.

```
1  def napovedovanje(drevo, testni_primer, razred = ""):
2    id_vozlisca = 0 #koren
3
4    while = 1:
5      sinovi = drevo[id_vozlisca]
6      for i in range(2): #levi in desni sin
7        vozlisce = sinovi[i]
8        razcep = vozlisce["razcep"]
9        naj_atribut = vozlisce["naj_atribut"]
10
11       #pri številskih atributih je operator < ali >
12       #pri kategoričnih atributih operator oceni,
13       #če je vrednost testnega primera v razcepu
14       if operator(testni_primer[naj_atribut], razcep):
15         id_vozlisca = vozlisce["id_vozlisca"]
16       if id_vozlisca in drevo:
17         continue
18
19       #izračunaj napoved
20       elif razred = "":
21         napoved = max(vozlisce["porazdelitev"])
22         return napoved
23
24       #izračunaj zaupanje
25       else:
26         verjetnosti = izracunaj_v(vozlisce["porazdelitev"])
27         napoved = max(verjetnosti)
28         if napoved == razred: #napovemo pravilno
29           zaupanje = verjetnosti[0] - verjetnosti[1]
30         else: #napovemo napačno
31           zaupanje = verjetnosti[razred] - verjetnosti[0]
32         return id_vozlisca, zaupanje
```

Algoritem 4.24: Algoritem napovedovanja z odločitvenim drevesom.

(privzeto nastavljen 100). Če je število primerov manjše kot je število intervalov, metoda izračuna srednje vrednosti med sosednjimi primeri z različnimi

razredi. Izračunane vrednosti predstavljajo kandidate za razcep. Za vsak interval izmerimo porazdelitev razredov in nato z mero za ocenjevanje kvalitete atributov izberemo vrednost za najboljši razcep. Če je parameter za število razcepov druge metode večji od števila primerov, je metoda primerna za uporabo z odločitvenim drevesom, saj najde dobre kandidate za razcep.

Pri iskanju najboljšega binarnega razcepa za kategorične atribute uporabimo hevristiko, ki jo opišejo v [19]. Hevristika je definirana za binarne razrede, mi pa jo posplošimo na večrazredne probleme z iskanjem razcepa za razred z najmanj pojavitvami. Na začetku izberemo razred z najmanj pojavitvami in izračunamo apriorno verjetnost. Za vsako kategorično vrednost atributa izračunamo pogojno verjetnost z izbranim razredom $P(razred|\ kategoricna\_vrednost)$. S sortiranjem pogojnih verjetnosti sortiramo tudi kategorične vrednosti. Najboljši razcep določimo z mero za ocenjevanje kvalitete atributov po vrstnem redu kategoričnih vrednosti.

Mere za izbiro atributov

Za določitev kvalitete atributa v podatkovni množici uporabimo informacijski prispevek in princip najkrajšega opisa.

Informacijski prispevek je mera za izbiro najboljšega atributa, ki temelji na stopnji nečistoče naslednikov. Manjša kot je stopnja nečistoče, večja je nesimetričnost porazdelitve razredov. Na primer, vozlišče s porazdelitvijo razredov $(0, 1)$ ima stopnjo nečistoče 0, medtem ko ima vozlišče z enakomerno porazdelitvijo $(0.5, 0.5)$ najvišjo stopnjo nečistoče. Pogosto se za mero nečistoče uporabi entropija, ki jo določa enačba (4.16), kjer $p(y|t)$ označuje delež primerov, ki pripada razredu y v vozlišču t, c označuje število razredov.

$$H(t) = -\sum_{y=1}^{c} p(y|t) \log_2 p(y|t). \tag{4.16}$$

Kvaliteta pravila se oceni s primerjavo stopnje nečistoče prednika (pred delitvijo) s stopnjo nečistoče naslednikov (po delitvi). Večja kot je razlika,

kvalitetnejše je pravilo. Informacijski prispevek (Δ_{info}) oceni kvaliteto razcepa in za mero nečistoče uporabi entropijo. Določa ga enačba (4.17), kjer N označuje število primerov prednika, k označuje število vrednosti atributa in $N(v_j)$ določa število primerov povezanih z naslednikom v_j. Informacijski prispevek precenjuje atribute z veliko različnimi vrednostmi (na primer, identifikator primerov). Temu se izognemo z normalizacijo informacijskega prispevka z entropijo atributa ali z gradnjo binarnih odločitvenih dreves.

$$\Delta_{info} = H(prednik) - \sum_{j=1}^{k} \frac{N(v_j)}{N} H(v_j). \tag{4.17}$$

Princip najkrajšega opisa (angl. Minimum description length) izbere atribut, ki najbolj zgosti podatke. Denimo, da želi pošiljatelj prejemniku poslati informacijo o razredih. Oba imata podatkovno množico z znanimi vrednostmi atributov, pri tem pa le pošiljatelj pozna vrednosti razredov za vsak primer. Pošiljatelj lahko prejemniku pošlje razred za vsak primer, kar zahteva prenos $H(c) \cdot n$ bitov, kjer $H(c)$ označuje entropijo razredov c, n pa število primerov. Za zmanjšanje prenosa podatkov pošiljatelj zgradi klasifikacijski model, ga zakodira in pošlje prejemniku. Če je model točen za vse primere, je potrebno prenesti le kodiran napovedni model. Sicer mora pošiljatelj poslati še informacijo o primerih, ki jih model nepravilno klasificira. Skupna cena prenosa je:

$$cena(model, podatki) = cena(model) + cena(podatki|model),$$

kjer prvi člen na desni strani enačbe označuje ceno kodiranega modela, drugi člen pa ceno kodiranih nepravilno klasificiranih primerov. Princip najkrajšega opisa minimizira skupno ceno prenosa podatkov med prejemnikom in pošiljateljem. Princip najkrajšega opisa, ki ga uporabimo za oceno atributov je podrobnejše opisan v [20].

4.4.2 Pregled pristopov z naključnimi gozdovi

Naključni gozdovi so ansambelski algoritmi, ki zgradijo več odločitvenih dreves na podatkovni množici in združijo njihove napovedi v končno napoved. Upoštevajo princip večkratne razlage, kar pomeni, da moramo za optimalno rešitev problema upoštevati več hipotez, ki sledijo iz vhodnih podatkov. Da bo ansambel dobro deloval, moramo zagotoviti različnost in točnost članov ansambla. Višja korelacija med odločitvenimi drevesi niža točnost naključnih gozdov. Za nižanje korelacije pri gradnji odločitvenih dreves vpeljemo več naključnih faktorjev.

V naključnem gozdu zgradimo odločitvena drevesa iz različnih podmnožic podatkov, kar zagotovi raznolikost klasifikatorjev. Podmnožico podatkov ustvarimo z naključno izbiro primerov z vračanjem iz začetne množice podatkov in na podlagi te zgradimo odločitveno drevo. Metoda, ki to omogoča, je strmensko vzorčenje (angl. bootstrap). Metoda učne primere vzorči z zamenjavo, kar pomeni, da so že izbrani primeri lahko ponovno izbrani. Dokazano je, da v povprečju ustvarjena podmnožica vsebuje 63.2% različnih primerov iz začetne množice podatkov. Neizbrane primere (angl. out of bag) lahko uporabimo za preizkus klasifikatorja. Pri gradnji odločitvenega drevesa naključno izberemo F atributov za razcep v vsakem vozlišču. Manjši F omogoča manjšo korelacijo med drevesi, večji F pa zviša točnost klasifikatorja. Za kompromis med korelacijo in točnostjo določimo $F = \lceil \log_2 n + 1 \rceil$, kjer n označuje število atributov. Naključna izbira F atributov zmanjša pristranskost drevesa in pohitri izvajanje algoritma.

Pri pregledu literature smo našli tri različne pristope, ki gradijo naključne gozdove na velikih podatkih s paradigmo MapReduce. Opisani pristopi niso odprtokodni, zato jih nismo mogli primerjati z razvitimi različicami porazdeljenih naključnih gozdov.

Pristop MReC4.5 [3] zgradi odločitveno drevo na lokalni podmnožici podatkov in drevesa združi v ansambel. Pri gradnji drevesa pri vsakem vozlišču oceni vse atribute in pri napovedovanju uporabi vsa drevesa. Za gradnjo modela potrebuje en prehod skozi podatkovno množico, kar zmanjša bralne ope-

racije z diskom in režijo pri proženju poslov MapReduce. Značilnost pristopa je minimalna komunikacija in koordinacija med vozlišči. Pristop predpostavlja, da so podatki enakomerno porazdeljeni po gruči.

Pristop COMET [4] zgradi več naključnih gozdov iz lokalnih podatkovnih množic in jih združi v velik ansambel. Pri pristopu COMET posamezno opravilo map zgradi naključni gozd iz lokalne podatkovne množice, opravilo reduce pa združi zgrajene modele v velik ansambel. Ta vsebuje veliko klasifikatorjev, ki lahko upočasnijo napovedovanje. Za hitrejše napovedovanje pristop uporabi hevristiko, ki izbere toliko članov ansambla, kot je potrebno za zanesljivo napoved. V opravilih map lahko namesto dreves uporabimo drug klasifikator. Pristop COMET, tako kot pristop MReC4.5, potrebuje en prehod čez podatkovno množico za izgradnjo modela.

Pristop PLANET [5] omogoča gradnjo odločitvenega drevesa na velikih podatkih. Uporablja prilagojeno ogrodje MapReduce, ki posle razvršča z upoštevanjem odvisnosti med vozlišči pri gradnji drevesa. Pristop izvede več poslov MapReduce za izgradnjo odločitvenega drevesa. Pristop PLANET uporablja komponento za nadzor, ki nadzoruje gradnjo drevesa, razvršča vozlišča za razcep in vzdržuje datoteko s trenutno zgrajenim modelom drevesa. Po izvedenem poslu se ocene atributov posredujejo komponenti za nadzor, ki izbere najboljši razcep in ga shrani v datoteko z modelom. Na podlagi te datoteke se določi naslednje vozlišče za razcep. Posel MapReduce sprejme datoteko z modelom za določitev vhodnih podatkov pri razcepu. Komponenta za nadzor vzdržuje dve vrsti, ki hranita vozlišča za naslednji razcep. Prva vrsta vsebuje vozlišča, ki imajo manj primerov in jih lahko hranimo v glavnem pomnilniku. Ta vozlišča obdela posel, ki je optimiziran za iskanje razcepa vozlišča s podatki v glavnem pomnilniku. Druga vrsta vsebuje vozlišča z več primeri, ki jih ni mogoče hraniti v glavnem pomnilniku. Pristop PLANET gradi odločitveno drevo v širino, za razliko od rekurzivnih različic, ki gradijo odločitveno drevo v globino. Zaradi gradnje v širino lahko ogrodje z enim poslom izračuna razcepe za več vozlišč. Ogrodje PLANET so razvili v podjetju Google za analizo podatkov o klikih na oglase.

4.4.3 Porazdeljeni naključni gozdovi

Razvili smo tri različice porazdeljenih naključnih gozdov za obdelavo velikih podatkov s paradigmo MapReduce. Različice gradijo odločitvena drevesa z algoritmom 4.23 in pri napovedovanju uporabijo algoritem 4.24. Gradnja naključnega gozda na lokalnih podmnožicah omogoča učinkovito implementacijo algoritma s paradigmo MapReduce, saj algoritem potrebuje le en prehod čez učno množico za izgradnjo modela.

Prvo različico porazdeljenih naključnih gozdov smo poimenovali gozd porazdeljenih odločitvenih dreves (v nadaljevanju imenujemo to različico FDDT, angl. Forest of Distributed Decision Trees). FDDT uporabi enak pristop kot pristop MReC4.5, ki zgradi odločitveno drevo na lokalni podmnožici podatkov, v vsakem vozlišču odločitvenega drevesa oceni vse atribute in drevesa združi v ansambel. Pri napovedovanju uporabi vsa drevesa in vrne napoved, za katero glasuje največ dreves. Drugo različico smo poimenovali porazdeljeni naključni gozdovi (v nadaljevanju imenujemo to različico DRF, angl. Distributed Random Forest). DRF uporabi podoben pristop kot pristop COMET, ki z opravili map zgradi naključni gozd z uporabo lokalnega strmenskega vzorčenja (angl. bootstrap) na lokalni podatkovni množici, v vsakem vozlišču odločitvenega drevesa oceni naključno izbrano podmnožico atributov in z opravilom reduce združi naključne gozdove v velik ansambel. Pri napovedovanju uporabi hevristiko, ki napoved določi z glasovanjem podmnožice dreves. Hevristika izbere naključno brez vračanja določeno število dreves (privzeto 15) in shrani napovedi. Uporabnik poda parameter *razlika*, ki določi razliko med prvo in drugo najverjetnejšo napovedjo. Ob zadostni razliki v verjetnosti, hevristika vrne napoved, drugače pa ponovno izbere določeno število dreves. Postopek se ponavlja dokler ne glasujejo vsa drevesa ali pa razlika med prvo in drugo najverjetnejšo napovedjo postane dovolj velika. Hevristika omogoča, da se za manj zahtevne napovedi uporabi manj dreves in s tem pohitri izvajanje. Tretja različica naključnih gozdov predstavlja razširitev prve in druge različice, zato v nadaljevanju podrobnejše opišemo le tretjo različico.

Tretjo različico smo poimenovali porazdeljeni uteženi naključni gozdovi (v nadaljevanju imenujemo to različico DWRF, angl. Distributed Weighted Random Forest). DWRF pri gradnji modela za vsako odločitveno drevo izračuna statistike in jih uporabi kot uteži pri napovedovanju. S tem izboljša točnost napovedovanja, saj napoved določijo le drevesa, ki dobro napovedujejo primere z določenimi karakteristikami. DWRF zgradi odločitveno drevo na vzorčeni učni množici (algoritem 4.25) ter drevo uporabi za napoved neizbranih primerov (angl. out of bag). Algoritem za vsak napovedan primer shrani zaupanje v napoved in identifikator lista, ki ga napove. Primerom, ki jih napove isti list, poveča podobnost v matriki podobnosti, velikosti ($št.\ primerov \times št.\ primerov$). Število kombinacij, ki jih dodamo v matriko podobnosti določa enačba $n * (n - 1)$, kjer n označuje število primerov, ki jih napove isti list. Zaradi velikega števila primerov, večine od teh nikoli ne napove isti list in zato je matrika podobnosti redka. Namesto simetrične matrike zato uporabimo slovar, ki je primernejši za hranjenje parov podobnosti. Po zaključeni gradnji modela, algoritem normalizira matriko podobnosti s številom zgrajenih odločitvenih dreves. Tako je najvišja možna podobnost med primeroma 1, kar pomeni, da je primera vsako drevo razvrstilo v isti list. Algoritem matriko podobnosti poda prilagojenemu algoritmu razvrščanja z voditelji (sekcija 4.3.6), ki namesto centroida vrača medoide. Medoidi so primeri, katerih povprečna razdalja do vseh ostalih primerov v gruči je minimalna. Medoidi so dejanski primeri iz podatkovne množice za razliko od centroida, ki so centri gruč. Algoritmu razvrščanja z voditelji določimo parameter k z $\lceil\sqrt{št.atr} + 1\rceil$. Višji parameter k pomeni višjo točnost pri napovedovanju, vendar podaljša čas gradnje modela in napovedovanja. Algoritem razvrščanja vrne oznako gruče za vsak primer in medoide. Algoritem izračuna povprečje zaupanja v napovedi za vsak medoid za vsako odločitveno drevo. Statistiko shrani v model drevesa in predstavlja utež pri napovedovanju. Algoritem vrne model naključnega gozda, medoide in mediane medoidov. Pri gradnji modela algoritem zgradi gozd in izračuna medoide na lokalni podmnožici podatkov. Z združitvijo ansamblov v velik

ansambel, algoritem vrednosti ne združuje, saj so izračunani medoidi in ostali parametri veljavni le za pripadajoči model naključnega gozda.

Pri napovedovanju izračunamo podobnost med testnim primerom in vsakim medoidom. Za izračun podobnosti uporabimo Gowerjev koeficient [21] S_{ij}, ki je definiran za numerične in kategorične vrednosti atributov. Za primera x_i in x_j izračunamo podobnost z enačbo (4.18), kjer S_{ijk} označuje prispevek spremenljivke k, z $w_{ijk} = 1$ upoštevamo prispevek, z $w_{ijk} = 0$ pa ne upoštevamo prispevka spremenljivke k. Za kategorične vrednosti atributov določimo S_{ijk} z 1, če je $x_{ik} = x_{jk}$, drugače pa z 0. Numerične vrednosti atributov izračunamo z enačbo (4.19), kjer r_k označuje interval spremenljivke k.

$$S_{ij} = \frac{\sum_k^n w_{ijk} S_{ijk}}{\sum_k^n w_{ijk}} \tag{4.18}$$

$$S_{ijk} = 1 - \frac{|x_{ik} - x_{jk}|}{r_k} \tag{4.19}$$

Pri napovedovanju (algoritem 4.26) izberemo medoide, ki jim Gowerjev koeficient dodeli najvišjo podobnost s testnim primerom (več medoidov ima lahko enako podobnost). Pri določanju napovedi uporabimo drevesa iz gozda, ki vsebuje izbrani medoid, kar pomeni, da v primeru več izbranih medoidov uporabimo več gozdov. Odločitveno drevo mora imeti zaupanje v napovedi večje od mediane, da ga izberemo za glasovanje. Preizkusili smo zaupanje v napovedi večje od 0, povprečje in mediano, od katerih se je zadnja izkazala za najboljšo. Z izbranimi odločitvenimi drevesi napovemo testni primer in za vsako napoved hranimo vrednost zaupanja. Za vsako napoved izračunamo $povprečje(zaupanje) * število\_napovedi$ in končno napoved določimo glede na najvišjo vrednost. Z opisanim postopkom želimo izboljšati točnost napovedovanja, saj pri izračunu napovedi sodelujejo le drevesa, ki dosegajo visoko točnost pri napovedovanju podobnih primerov.

Pri implementaciji različice DWRF s paradigmo MapReduce smo uporabili algoritem 4.25 za gradnjo modela. Algoritem potrebuje več glavnega po-

mnilnika v primerjavi s prej opisanimi algoritmi. Pri obdelavi velikega števila primerov postane matrika podobnosti prevelika za glavni pomnilnik, zato smo število neizbranih primerov, ki jih napovemo z zgrajenim odločitvenim drevesom, omejili na 500. Pri implementaciji algoritma s paradigmo MapReduce funkcija map (algoritem 4.27) vhodni primer razčleni, določi nov par, kjer identifikator primera predstavlja ključ, atributi in razred pa vrednost, in primer vrne.

Združevalnik (algoritem 4.28) združuje primere z izravnalnikom. Ko funkcija map vrne zadnji par, uvozimo algoritem 4.25 za modela in mu podamo podatkovno množico. Celotna lokalna podmnožica podatkov je v glavnem pomnilniku in njena velikost predstavlja omejitev. Za zmanjšanje porabe glavnega pomnilnika kategorične vrednosti atributov tipa niz kodiramo v števila. Če odločitveno drevo zgradimo na kodiranih vrednostih atributov, je potrebno številske vrednosti atributov pretvoriti v kategorične preden odločitveno drevo shranimo v končen model. Algoritmu za gradnjo naključnih gozdov podamo podatkovno množico in parameter za število dreves. Po izvedbi algoritma združevalnik vrne model naključnega gozda.

Funkcija reduce (algoritem 4.29) vsako drevo označi z identifikatorjem in ga vrne. Združevanje vrednosti po ključu v tem primeru ni potrebno.

Pri napovedovanju v funkciji map (algoritem 4.30) uvozimo algoritem za napovedovanje z različico DWRF (algoritem 4.26) in mu podamo testni primer, model naključnega gozda, medoide in mediane medoidov. Funkcija map vrne napoved za vsak primer.

```
1  def gradnja_modela(ucna_mnozica, st_dreves):
2      gozd, zaupanje_napovedi = [], [] #zaupanje drevesa v napoved
           za vsak neizbrani primer
3      matrika_podobnosti = {}
4
5      for i in range(st_dreves):
6          ucna, neizbrani = bootstrap(ucna_mnozica)
7          drevo = odlocitveno_drevo.gradnja_modela(ucna)
8          zaupanje, listi =  drevo.napovedovanje(neizbrani)
9          zaupanje_napovedi.append(zaupanje)
10         gozd.append(drevo)
11         #idje primerov združimo po listu
12         skupine = grupiraj_id_primerov(listi)
13
14         for skupina in skupine:
15             #za vsak par idjev neizbranih primerov v skupini
16             for id_a, id_b in kombinacije(skupina):
17                 #matrika je simetrična in povečamo 2x
18                 matrika_podobnosti[id_a, id_b] += 1
19                 matrika_podobnosti[id_b, id_a] += 1
20
21     #matriko podobnosti normaliziramo s številom dreves
22     matrika_podobnosti = matrika_podobnosti / st_dreves
23     k = koren(stevilo_atributov) + 1 #število medoidov
24     #vsak primer označimo z gručo in izračunamo medoide
25     gruce, medoidi=k_medoid.gradnja_modela(matrika_podobnosti,k)
26
27     for i, drevo in enumerate(gozd):
28         #gruča vsebuje idje primerov, ki so najbližji medoidu
29         for j, gruca in enumerate(gruce):
30             #izračunamo povprečje zaupanja za vsak medoid, ki ga
                   je napovedalo drevo
31             drevo["zaupanje" + j] = izracunaj_avg(gruca)
32     return gozd, medoidi
```

Algoritem 4.25: Gradnja modela z različico porazdeljenih uteženih naključnih gozdov.

```
1
2  def napovej(testni_primer, gozdovi, medoidi, mediana):
3      #izračunamo podobnosti
4      podobnosti = []
5      for medoid in medoidi:
6          podobnosti.append(gower(testni_primer, medoid))
7      podobnosti = sort(podobnosti)
8      #več medoid ima lahko enako podobnost
9      identifikatorji = izberi_najvisje_ocenjene(podobnosti)
10
11     napovedi = {}
12     for id_medoida, id_gozda in identifikatorji:
13         for drevo in gozdovi[id_gozda]:
14             zaupanje = "zaupanje"+id_medoida
15             #drevesa z zadostnim zaupanjem uporabimo za napoved
16             if zaupanje in drevo and drevo[zaupanje] > mediana[
                   id]:
17                 napoved = drevo.napovedovanje(testni_primer)
18                 napovedi[napoved].append(drevo[zaupanje])
19
20     napovedi = [len(napoved) * avg(napoved) for napoved in
           napovedi]
21     napoved = izberi_najvisje_ocenjeno(napovedi)
22     return napoved
```

Algoritem 4.26: Napovedovanje z različico porazdeljenih uteženih naključnih gozdov.

```
1  def map(primer, params):
2      x_id, x, y = primer
3      yield (x_id, (x, y))
```

Algoritem 4.27: Funkcija map za gradnjo modela z različico porazdeljenih uteženih naključnih gozdov.

```
1  def zdruzevalnik(kljuc,vrednost,izravnalnik,zakljucek,params):
2      if zakljucek: #zadnja iteracija
3          import nakljucni_gozd
4          #zgradimo podatkovno množico
5          X, Y = [], []
6          for k, v in izravnalnik.iteritems()
7            X.append(v[0])
8            Y.append(v[1])
9          X = kodiraj(X) #nize kodiramo v stevila
10
11         gozd = nakljucni_gozd.gradnja_modela(X, Y, params.
               stevilo_dreves)
12         gozd = preslikaj(gozd)
13         return ("gozd", gozd)
14     izravnalnik[kljuc] = vrednost
```

Algoritem 4.28: Združevalnik za gradnjo modela zz različico porazdeljenih uteženih naključnih gozdov.

```
1  def reduce(iterator, params):
2      id = 0
3      for _, gozd in iterator:
4        for drevo in gozd:
5          yield (id, drevo)
6          id += 1
```

Algoritem 4.29: Funkcija reduce za gradnjo modela z različico porazdeljenih uteženih naključnih gozdov.

```
1  def map(primer, params):
2      import nakljucni_gozd
3      x_id, x = primer
4      y = nakljucni_gozd.napovej(x, params.gozd, params.medoidi,
           params.mediane)
5      yield (x_id, y)
```

Algoritem 4.30: Funkcija map za napovedovanje z različico porazdeljenih uteženih naključnih gozdov.

Poglavje 5

Ovrednotenje algoritmov

V poglavju opišemo ovrednotenje implementiranih algoritmov in predstavimo rezultate. Ovrednotenje algoritmov smo izvedli na več računalniških sistemih, ki jih podrobnejše opišemo spodaj. Začeli smo z ustvarjanjem podatkovnih množic, saj smo želeli obdelati enako količino podatkov z vsakim algoritmom. Podatkovne množice smo pripravili tako, da so bile primernejše za obdelavo z algoritmi v porazdeljenem okolju. Pri ovrednotenju smo primerjali točnost napovedi implementiranih algoritmov z algoritmi v paketih Knime in scikit. Vsak algoritem smo testirali s podatkovno množico z veliko atributi in poiskali mejno število atributov, ki jih lahko algoritem obdela na delovnem vozlišču z dano specifikacijo. Za vsak algoritem smo izmerili čas izvajanja in primerjali pohitritve z dodajanjem več delovnih vozlišč.

5.1 Testno okolje

Algoritme smo ovrednotili na računalniških sistemih s specifikacijami v tabeli 5.1. Strežnik Vihar uporabljajo na raziskovalnem Odseku za tehnologije znanja na Institutu Jožef Štefan. Ogrodje Disco smo namestili na strežnik Vihar in mu dodelili vire, kot opišemo v tabeli 5.2. Ogrodje uporablja 10 vozlišč (glavno vozlišče in 9 delovnih vozlišč), kjer glavno vozlišče skrbi le za nadzor poslov. Delovnim vozliščem v gruči smo določili uporabo celotnega

glavnega pomnilnika (privzeto je 80%).

ime	Vihar	MacBook
tip	strežnik	osebni računalnik
CPE	AMD Opteron 8431	Intel Core i5
frekvenca CPE	2,4 Ghz	2,3 GHz
število CPE	4	1
število jeder v CPE	6	2
glavni pomnilnik	128 GB	8 GB
operacijski sistem	Slackware 13.37, 64 bit	OS X 10.9.4, 64 bit
virtualizacija	Virtualbox 4.3.8	/

Tabela 5.1: Specifikacija računalniških sistemov.

gruča Disco	
gostitelj	Vihar
različica	Disco 0.5.1-104-gaaa2665
število vozlišč	10
delovno vozliče Disco	
Število CPE	2
glavni pomnilnik	1 GB
prosti prostor na disku	6 GB
operacijski sistem (OS)	Ubuntu 12.04.4 LTS, 32 bit
OS - poraba pomnilnika	110 MB
Disco - poraba pomnilnika	14 MB
Prost pomnilnik vozlišča	876 MB

Tabela 5.2: Specifikacije ogrodja Disco.

Sprva smo želeli primerjave izvesti s paketom Knime s specifikacijo *Knime 1* (tabela 5.3), ki je enaka specifikaciji delovnega vozlišča Disco. Paket Knime nudi možnost shranjevanja podatkov iz glavnega pomnilnika na disk, kar omogoča obdelavo večje količine podatkov na račun daljšega časa izvajanja. Izkazalo se je, da je bila količina podatkov prevelika, glavnega pomnilnika pa premalo. S specifikacijo Knime 1 smo izvedli le algoritem naivni Bayes. Zato smo dodali specifikacijo *Knime 2* z več glavnega pomnilnika, na kateri smo izvedli linearno in logistično regresijo. Algoritme naključni gozdovi, linearni

SVM in razvrščanje z voditelji smo izvedli s paketom scikit, različica 0.14.1, na računalniku MacBook, saj jih paket Knime ni bil zmožen izvesti. V času izvajanja testov na računalniku nismo izvajali drugih procesov.

	Knime 1	Knime 2
računalnik	Vihar	Vihar
operacijski sistem	Ubuntu 12.04, 32 bit	Ubuntu 14.04, 64 bit
glavni pomnilnik	1 GB	4 GB
različica	v2.10.0, 32 bit	v2.10.0, 64 bit

Tabela 5.3: Specifikacija računalnikov z nameščenim paketom Knime.

5.2 Ustvarjanje podatkovnih množic

Potrebovali smo dve vrsti podatkovnih množic, saj implementirani algoritmi delujejo na področjih klasifikacije in regresije. Podatke za klasifikacijo smo ustvarili z razširitvijo obstoječih podatkovnih množic. Izbrali smo podatkovne množice v tabeli 5.4 in jih razmnožili s paketom *semiArtificial* [22] v programu *R*. Paket semiArtificial ustvari nove primere glede na porazdelitev podatkov. Na izbranih podatkovnih množicah paket semiArtificial dobro deluje, kot je opisano v [22]. Velikosti podatkovnih množic smo določili na 3 GB za učno in 3 GB za testno množico. Časi ustvarjanja podatkov so nenatančni, saj smo podatke ustvarili na računalniku Vihar in MacBook, čase izvajanja pa združili (tabela 5.5).

	lymphography	segmentation	sonar	ionosphere
število primerov	148	2310	208	351
numerični atributi	3	19	60	34
kategorični atributi	15	0	0	0
število razredov	4	7	2	2
večinski razred	54,7	14,2	53,4	64,1

Tabela 5.4: Karakteristike izbranih podatkovnih množic za klasifikacijo.

	lymphography	segmentation	sonar	ionosphere
ustvarjanje generatorja	00:00:03	00:00:35	00:00:07	00:00:06
ustvarjanje podatkov (6 GB)	19:48:17	141:12:22	61:45:53	36:35:49
skupni čas	19:48:20	141:12:57	61:46:00	36:35:55

Tabela 5.5: Časi ustvarjanja podatkovnih množic s paketom semiArtificial.

Ustvarili smo tudi podatkovni množici linear in fraction z regresijsko napovedno spremenljivko. Atributi teh podatkovnih množic imajo numerične vrednosti in se ločijo na pomembne in naključne atribute. Podatkovni množici linear smo napovedno spremenljivko določili z enačbo (5.1), kjer n predstavlja število pomembnih atributov, A_i pa naključno število na intervalu $[0, 1]$. Regresijsko spremenljivko v podatkovni množici fraction smo določili z enačbo (5.2), kjer regresijska spremenljivka predstavlja neceli del vsote vrednosti atributov. Podatkovni množici smo ustvarili z računalnikom Vihar. Tabela 5.6 prikaže skupni čas ustvarjanja učne in testne množice, število primerov pa opisuje število primerov v eni množici podatkov (v učni ali v testni množici).

$$linear\_y = \sum_{i=1}^{n} i * A_i \tag{5.1}$$

$$fraction\_y = \sum_{i=1}^{n} A_i \mod 1 \tag{5.2}$$

	linear	fraction
število primerov	14.500.000	23.166.666
pomembni atributi	10	3
naključni atributi	10	10
skupni čas ustvarjanja	00:05:56	00:05:59

Tabela 5.6: Podatkovne množice z regresijsko spremenljivko.

Vsakemu primeru v podatkovni množici smo dodali identifikator, podatkovno množico razdelili na več manjših kosov, kose stisnili in naložili na datotečni strežnik (tabela 5.7). Razdelitev podatkov na manjše kose je potrebna

za boljši izkoristek ogrodja Disco. Manjše kose lahko delovna vozlišča obdelajo istočasno, celotno množico podatkov pa lahko obdeluje le eno vozlišče hkrati. Podatkovne množice smo razdelili na 36 kosov, saj smo izračunali, da lahko tako izkoristimo vsa delovna vozlišča. Na primer, imamo 9 delovnih vozlišč in vsako vozlišče ima 2 delovni enoti, torej lahko naenkrat obdelamo 18 kosov podatkov. Za zmanjšanje prenosa preko spleta do delovnih vozlišč, smo podatke stisnili. Podatkovne množice za klasifikacijo smo obdelali na računalniku MacBook, podatkovne množice z regresijsko spremenljivko pa na računalniku Vihar.

	lymphography	segmentation	sonar	ionosphere	linear	fraction
velikost podatkov	3 GB	3 GB	3 GB	3 GB	3 GB	3 GB
število primerov	20.903.225	10.781.249	2.980.000	5.587.500	14.500.000	23.166.666
število primerov v kosu	580.646	299.480	82.777	155.207	404.000	645.000
število kosov	36	36	36	36	36	36
velikost kosa	85 MB	85 MB	86 MB	86 MB	84 MB	83 MB
velikost stisnjenega kosa	19 MB	40 MB	39 MB	39 MB	32 MB	33 MB
čas obdelave	0:20:21	0:29:09	0:27:52	0:27:20	0:18:23	0:20:41

Tabela 5.7: Obdelava podatkovnih množic.

5.3 Ovrednotenje algoritmov

Ocenimo pravilnost implementiranih algoritmov in količino podatkov, ki jo lahko obdelamo v določenem času. Pravilnost algoritmov smo ocenili s primerjavo klasifikacijkih točnosti s paketom Knime ali scikit na dveh podatkovnih množicah. Implementirani algoritmi nimajo omejitve v številu primerov (omejitev je diskovni prostor delovnih vozlišč), ampak so omejeni s številom atributov, ki jih lahko obdelajo. Pri gradnji modelov statistike hranimo v podatkovnih strukturah seznam ali slovar, ki so omejene z glavnim pomnilnikom delovnega vozlišča. Ustvarili smo več podatkovnih množic z velikim številom atributov in za vsak algoritem poiskali mejno število atributov, ki jo lahko obdelamo na delovnem vozlišču z dano specifikacijo. Implementirane različice porazdeljenih naključnih gozdov so omejene z velikostjo lokalne pod-

množice podatkov, saj so v času gradnje drevesa vsi podatki v glavnem pomnilniku. Različic ne preizkusimo s podatkovno množico z velikim številom atributov, saj v primeru, da algoritem naloži podatke v glavni pomnilnik, število atributov ne predstavlja omejitve.

Točnost različic porazdeljenih naključnih gozdov je odvisna od porazdelitve razredov v učnih podmnožicah podatkov, na katerih algoritem zgradi napovedni model. Modele smo zgradili na enakomerno in nesimetrično porazdeljenih razredih v učnih podmnožicah podatkov in primerjali klasifikacijske točnosti.

Za vsak algoritem smo izmerili čas izvajanja in čase primerjali. Za implementirane algoritme smo izmerili čas izvajanja z 1 delovnim vozliščem in primerjali pohitritev z 3, 6 in 9 delovnimi vozlišči. Za osnovo vzamemo čas z enim delovnim vozliščem in pohitritve primerjamo z idealnimi linearnimi pohitritvami (tabela 5.8).

Št. delovnih vozlišč	1	3	6	9
pohitritev	0,00%	66,67%	83,33%	88,89%

Tabela 5.8: Idealna linearna pohitritev, če za osnovo vzamemo čas izvedbe z enim delovnim vozliščem.

Pri ovrednotenju algoritmov smo upoštevali naslednje. Algoritmi v knjižnici DiscoMLL preberejo podatke v stopnji map, v kateri že gradijo model ali vračajo napovedi, zato prikažemo skupni čas branja učne množice in gradnje modela ali branja testne množice in napovedovanja. Paket Knime prebere učno in testno množico vzporedno, zato smo vključili le čas branja učne množice. Paket scikit prebere podatkovne množice zaporedno, zato smo združili časa branja podatkovnih množic. Paket scikit zahteva podatkovne množice s številskimi vrednostmi atributov, pri čimer smo čase kodiranja vrednosti dodali k branju podatkov. Pri testu s podatkovnimi množicami z veliko atributi podamo velikost modela, kar smo izmerili s shranitvijo modela v datoteko.

5.3.1 Naivni Bayes

Primerjamo klasifikacijsko točnost in čase izvajanja algoritma naivni Bayes iz paketov DiscoMLL, Knime in scikit. V paketu scikit smo izbrali različico, ki predpostavlja Gaussovo porazdelitev vrednosti atributov, saj je ta dosegla najvišjo točnost na izbranih podatkovnih množicah.

Podatkovna množica lymphography

Podatkovna množica lymphography ima kategorične in številske vrednosti atributov, zato smo kategorične vrednosti kodirali v številske, saj to zahteva paket scikit. Algoritmom smo nastavili parameter m ocene [2] na 1, paket Knime pa ni ponudil te možnosti.

Algoritem v paketu DiscoMLL je dosegel najvišjo klasifikacijsko točnost na podatkovni množici lymphography (tabela 5.9), ki pa je le malo višja od klasifikacijske točnosti algoritma v paketu Knime. Algoritem v paketu scikit je dosegel nižjo klasifikacijsko točnost, ker smo kategorične atribute kodirali v številske, pri tem pa je algoritem vrednosti kodiranih atributov upošteval kot razdalje. Algoritem v paketu DiscoMLL se je na enem delovnem vozlišču izvedel hitreje v primerjavi z algoritmoma v paketih Knime in scikit (tabela 5.10). Pričakovali smo, da se bo najhitreje izvedel algoritem v paketu scikit, ki pa je večino časa porabil za branje podatkov. Domnevamo, da branje podatkov ni optimizirano za branje kategoričnih vrednosti v programskem jeziku python. Opazimo 50% pohitritev izvajanja algoritma s specifikacijo Knime 2 v primerjavi s specifikacijo Knime 1, ki ima 3 GB manj glavnega pomnilnika od specifikacije Knime 2 in uporablja 32 bitno različico paketa Knime. Na grafu (slika 5.1) opazimo, da z dodajanjem procesnih enot algoritem počasi odstopa od idealne linearne pohitritve.

podatkovna množica	segmentation			lymphography		
paket	Knime	scikit	DiscoMLL	Knime	scikit	DiscoMLL
klasifikacijska točnost	0,7574	0,8046	0,8046	0,9511	0,8276	0,9516

Tabela 5.9: Primerjava klasifikacijskih točnosti algoritma naivni Bayes.

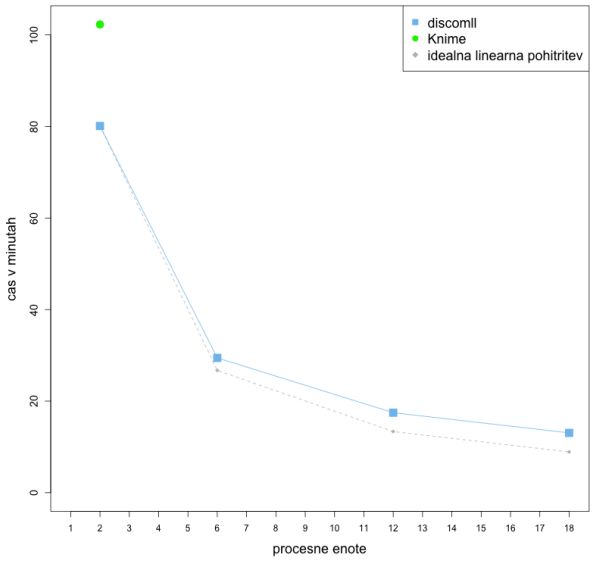
	Knime 1	Knime 2	scikit	DiscoMLL			
računalnik	Vihar	Vihar	MacBook	Vihar	Vihar	Vihar	Vihar
virtualni stroji	1	1	1	1	3	6	9
CPE	2	2	1	2	6	12	18
branje	02:30:25	01:16:06	07:15:15	00:20:56	00:07:24	00:04:05	00:02:49
učenje	00:16:36	00:06:36	00:00:53				
napovedovanje	00:38:58	00:14:58	0:05:25	00:56:54	00:19:59	00:11:21	00:07:55
zapisovanje	00:09:56	00:04:49	00:03:02	00:02:21	00:02:21	00:02:21	00:02:21
skupni čas	3:35:55	1:42:29	7:24:35	1:20:11	0:29:44	0:17:47	0:13:05

pohitritev	0%	53%	-51%	63%	86%	92%	94%
		0%	-77%	22%	71%	83%	87%
			0%	82%	93%	96%	97%
				0%	63%	78%	84%
					0%	40%	56%
						0%	26%

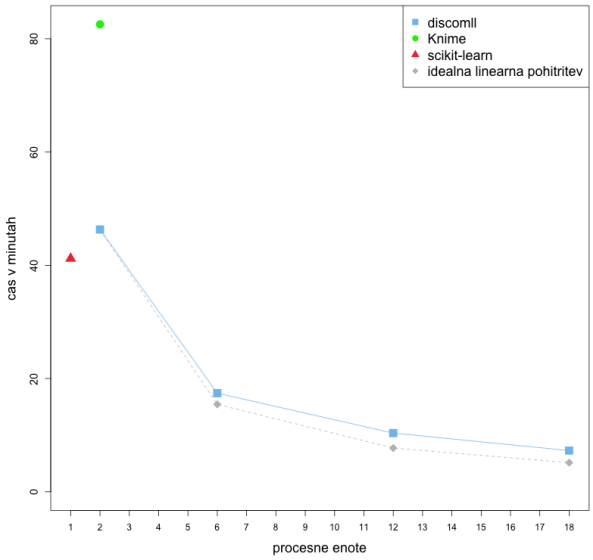
Tabela 5.10: Primerjava časov izvajanja in pohitritev z algoritmom naivni Bayes na podatkovni množici lymphography.

Podatkovna množica segmentation

Algoritma v paketih DiscoMLL in scikit sta dosegla enaki klasifikacijski točnosti na podatkovni množici segmentation (tabela 5.9). Algoritem v paketu Knime je dosegel nižjo klasifikacijsko točnost, čeprav je algoritem prepoznal vse atribute kot številske in vsebuje implementacijo algoritma naivni Bayes, ki predpostavi Gaussovo porazdelitev vrednosti atributov. Najhitreje se je izvedel algoritem v paketu scikit (tabela 5.11). To potrjuje našo domnevo, da je branje kategoričnih vrednosti počasnejše v programskem jeziku python, saj smo pri branju podatkov uporabili enako kodo. Algoritem v paketu DiscoMLL se je izvedel 11% počasneje na manj zmogljivem delovnem vozlišču, kar potrjuje učinkovito implementacijo algoritma. Na grafu (slika 5.2) opazimo linearno pohitritev algoritma z dodajanjem procesnih enot, saj se čas izvajanja zmanjšuje v skladu z idealno linearno pohitritvijo.



Slika 5.1: Primerjava časov izvajanja algoritma naivni Bayes na podatkovni množici lymphography.



Slika 5.2: Primerjava časov izvajanja algoritma naivni Bayes na podatkovni množici segmentation.

	Knime 1	Knime 2	scikit	DiscoMLL			
računalnik	Vihar	Vihar	MacBook	Vihar	Vihar	Vihar	Vihar
virtualni stroji	1	1	1	1	3	6	9
CPE	2	2	1	2	6	12	18
branje	01:58:18	00:52:30	0:17:08	00:19:51	00:07:01	00:03:53	00:02:42
učenje	00:07:40	00:03:32	00:00:53				
napovedovanje	00:36:26	00:19:49	0:17:31	00:25:15	00:09:14	00:05:17	00:03:19
zapisovanje	00:05:37	00:07:04	00:05:47	00:01:26	00:01:26	00:01:26	00:01:26
skupni čas	02:48:01	01:22:55	00:41:19	00:46:32	00:17:41	00:10:36	00:07:27

pohitritve	0%	51%	75%	72%	89%	94%	96%
		0%	50%	44%	79%	87%	91%
			0%	-11%	57%	74%	82%
				0%	62%	77%	84%
					0%	40%	58%
						0%	30%

Tabela 5.11: Primerjava časov izvajanja in pohitritev z algoritmom naivni Bayes na podatkovni množici segmentation.

Podatkovna množica z velikim številom številskih atributov

Algoritem naivni Bayes lahko obdela številske in kategorične vrednosti atributov zato smo test razdelili na dva dela. Z ustvarjeno podatkovno množico z velikim številom številskih atributov (tabela 5.12), smo našli zgornjo mejo pri 120.000 atributih na delovnem vozlišču z dano specifikacijo. Pri tem je velikost modela narasla na 67 MB. Velikost modela je določena s $št.atributov * št.razredov * 3$, kjer število tri predstavlja vektor matematičnega upanja, variance in logaritma variance. Logaritem variance smo dodali za pohitritev napovedovanja, ker algoritem vrednosti izračuna enkrat namesto večkrat v vsakem opravilu map.

Podatkovna množica z velikim številom kategoričnih atributov

Z ustvarjeno podatkovno množico z velikim številom kategoričnih atributov smo našli zgornjo mejo pri 60.000 atributih na delovnem vozlišču z dano speci-

podatkovna množica	
št. atributov	500.000
primeri	100
razredi	11
min. vrednost	0
max vrednost	0,99
št. decimalnih mest	2
velikost podatkov	234 MB

zmogljivost	
št. atributov	120.000
učenje	00:14:07
napovedovanje	00:01:05
velikost modela	67 MB

Tabela 5.12: Podatkovna množica z velikim številom številskih atributov in mejna zmogljivost algoritma naivni Bayes.

podatkovna množica	
št. atributov	70.000
primeri	100
razredi	11
v/a	9
velikost podatkov	20 MB

zmogljivost	
št. atributov	60.000
učenje	00:03:34
obdelava	00:01:25
napovedovanje	00:01:46
velikost modela	90 MB

Tabela 5.13: Podatkovna množica z velikim številom kategoričnih atributov in mejna zmogljivost algoritma naivni Bayes. v/a označuje povprečno število vrednosti na atribut.

fikacijo (tabela 5.13). Algoritem pred napovedovanjem kategorične vrednosti atributov shrani v vektorsko obliko in izračuna logaritme vrednosti za pohitritev napovedovanja. Velikost modela naraste na 90 MB, kar je več od velikosti podatkovne množice. To je posledica zasnove modela, kjer smo sprejeli kompromis med velikostjo modela in hitrostjo napovedovanja. V primeru večje podatkovne množice z enakim naborom vrednosti atributov, velikost modela ne bi več naraščala. Velikost modela s kategoričnimi vrednostmi je določena s $\check{s}t.atributov * povprečno\ število\ vrednosti\ na\ atribut * \check{s}t.razredov$.

5.3.2 Gozd porazdeljenih odločitvenih dreves

Različica gozd porazdeljenih odločitvenih dreves (v nadaljevanju imenujemo to različico FDDT, angl. Forest of Distributed Decision Trees) zgradi število

podatkovna množica	segmentation		lymphography	
paket	scikit	DiscoMLL	scikit	DiscoMLL
klasifikacijska točnost	0,9939	0,8952	0,9567	0,9521

Tabela 5.14: Primerjava klasifikacijskih točnosti različice gozd porazdeljenih odločitvenih dreves z naključnimi gozdovi v paketu scikit.

odločitvenih dreves, ki je enako številu lokalnih podmnožic podatkov (v našem primeru 36) in v vsakem vozlišču oceni vse atribute. Algoritem primerjamo z naključnimi gozdovi v paketu scikit. Algoritmoma smo nastavili globino drevesa na 50, minimalno število primerov v listu pa na 5. Algoritmu v paketu scikit smo nastavili število odločitvenih dreves na 50 in entropijo za mero za ocenjevanje atributov. Različici FDDT smo nastavili informacijski prispevek za mero ocenjevanja atributov. Različici naključnih gozdov smo ovrednotili na podatkovnih množicah lymphography in segmentation, saj vsebujeta kategorične in številske atribute ter več razredov.

Podatkovna množica lymphography

Algoritem v paketu scikit je dosegel malo višjo klasifikacijsko točnost kot algoritem v paketu DiscoMLL na podatkovni množici lymphography (tabela 5.14). Pri tem je potrebno poudariti, da algoritem v paketu scikit gradi odločitvena drevesa iz celotne podatkovne množice za razliko od algoritma v paketu DiscoMLL, ki gradi odločitvena drevesa na lokalnih podmnožicah podatkov. Različica FDDT z enim delovnim vozliščem se je izvedla 78% hitreje od algoritma v paketu scikit na podatkovni množici lymphography (tabela 5.15). Na grafu (slika 5.3) opazimo, da se čas izvedbe algoritma približa idealni linearni pohitritvi (časa izvedbe algoritma v paketu scikit nismo narisali zaradi boljše preglednosti grafa).

Podatkovna množica segmentation

Algoritem v paketu scikit je dosegel občutno višjo klasifikacijsko točnost kot algoritem v paketu DiscoMLL na podatkovni množici segmentation (tabela

	scikit	DiscoMLL			
računalnik	MacBook	Vihar	Vihar	Vihar	Vihar
virtualni stroji	1	1	3	6	9
CPE	2	2	6	12	18
branje	6:40:53	01:04:35	00:22:49	00:11:50	00:05:47
učenje	02:18:43				
napovedovanje	00:04:03	00:52:09	00:18:53	00:09:36	00:08:03
zapisovanje	00:00:18	00:01:29	00:01:29	00:01:29	00:01:29
skupni čas	09:03:57	01:58:13	00:43:11	00:22:55	00:15:19

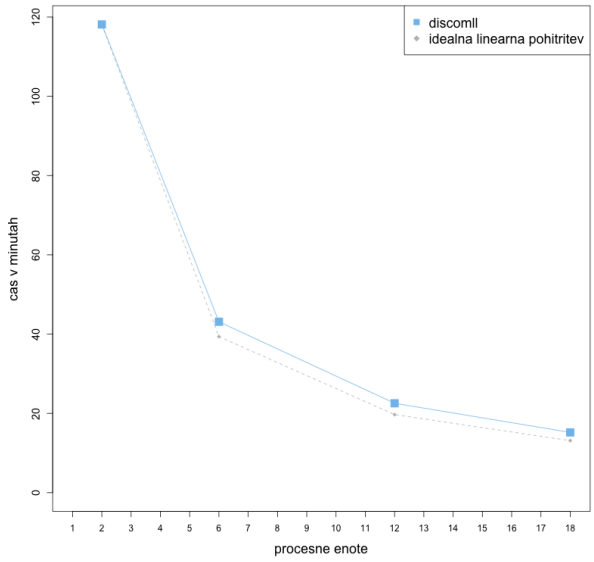
pohitritve	0%	78%	92%	96%	97%
		0%	63%	81%	87%
			0%	47%	65%
				0%	33%

Tabela 5.15: Primerjava časov izvajanja in pohitritev gozda porazdeljenih odločitvenih dreves z naključni gozdovi na podatkovni množici lymphography.

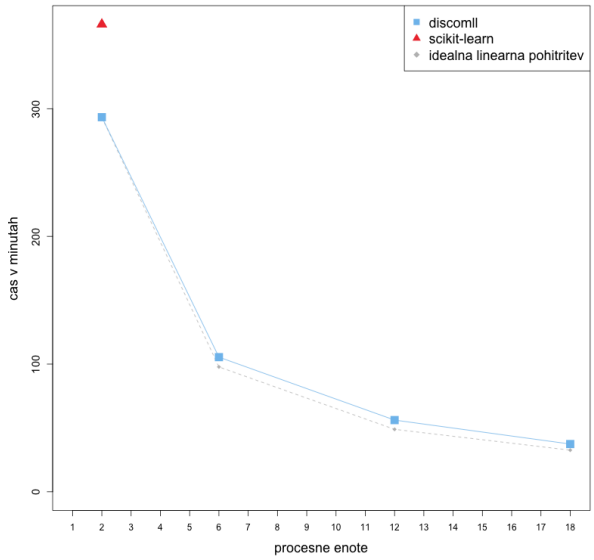
5.14). Razlog je v manj zmogljivi funkciji za iskanje najboljšega razcepa s številskimi atributi algoritma v paketu DiscoMLL, kjer smo morali sprejeti kompromis med točnostjo in hitrostjo izvedbe. Različica FDDT z enim delovnim vozliščem se je izvedla 20% hitreje od algoritma v paketu scikit na podatkovni množici segmentation (tabela 5.16). Različica FDDT se izvede počasneje na podatkovni množici segmentation v primerjavi s podatkovno množico lymphography, saj vsak atribut razdeli na 100 intervalov z enakim številom primerov in za vsakega izračuna informacijski prispevek. Na grafu (slika 5.4) opazimo, da se z dodajanjem procesnih enot čas izvajanja algoritma približa idealni linearni pohitritvi.

Nesimetrična porazdelitev razredov

Razrede v podmnožicah segmentation in lymphography smo nesimetrično porazdelili in shranili v novi podmnožici podatkov (tabeli 5.17 in 5.18). Na podmnožicah z enakomerno in nesimetrično porazdelitvijo razredov smo z



Slika 5.3: Primerjava časov izvajanja gozda porazdeljenih odločitvenih dreves z naključni gozdovi na podatkovni množici lymphography.



Slika 5.4: Primerjava časov izvajanja gozda porazdeljenih odločitvenih dreves z naključnimi gozdovi na podatkovni množici segmentation.

	scikit	DiscoMLL			
računalnik	MacBook	Vihar	Vihar	Vihar	Vihar
virtualni stroji	1	1	3	6	9
CPE	2	2	6	12	18
branje	0:15:48	03:20:28	01:13:34	00:39:36	00:25:49
učenje	05:45:23				
napovedovanje	00:04:47	01:32:19	00:31:21	00:15:50	00:11:00
zapisovanje	00:00:09	00:00:48	00:00:48	00:00:48	00:00:48
skupni čas	06:06:07	04:53:35	01:45:43	00:56:14	00:37:37

pohitritve	0%	20%	71%	85%	90%
		0%	64%	81%	87%
			0%	47%	64%
				0%	33%

Tabela 5.16: Primerjava časov izvajanja in pohitritev gozda porazdeljenih odločitvenih dreves z naključnimi gozdovi na podatkovni množici segmentation.

različico FDDT zgradili model in primerjali klasifikacijsko točnost algoritma na testnih podatkih (tabela 5.19). Pričakovano je model zgrajen na podmnožici z nesimetrično porazdelitvijo razredov dosegel nižjo klasifikacijsko točnost na podatkovni množici segmentation, presenetila pa nas je višja klasifikacijska točnost na podatkovni množici lymphography.

Velikost lokalne podmnožice podatkov

Implementirane različice porazdeljenih naključnih gozdov imajo lokalno podmnožico podatkov v glavnem pomnilniku v času gradnje modela in se s tem razlikujejo od ostalih implementiranih algoritmov. Različice porazdeljenih naključnih gozdov temeljijo na enakem algoritmu za gradnjo odločitvenih dreves, zato mejna velikost lokalne podmnožice podatkov velja za vse implementirane različice porazdeljenih naključnih gozdov. Pomembno je, da algoritem lahko obdela 64 MB podmnožico podatkov, saj porazdeljeni datotečni sistem Disco na takšno velikost razdeli velike datoteke pri nalaganju. Z al-

Enakomerna porazdelitev razredov			
razred	metastases	malign lymph	fibrosis
podmnožica 1	55,47%	41,81%	2,73%
podmnožica 2	55,49%	41,76%	2,75%

Nesimetrična porazdelitev razredov			
razred	metastases	malign lymph	fibrosis
podmnožica 1	12,80%	86,56%	0,64%
podmnožica 2	88,26%	7,38%	4,35%

Tabela 5.17: Enakomerna in nesimetrična porazdelitev podmnožic lymphography. Vrednosti predstavljajo odstotek primerov z danim razredom.

Enakomerna porazdelitev razredov							
razred	SKY	CEMENT	WINDOW	BRICKFACE	FOLIAGE	PATH	GRASS
podmnožica 1	14,29%	14,28%	14,29%	14,29%	14,28%	14,29%	14,28%
podmnožica 2	14,29%	14,29%	14,28%	14,28%	14,29%	14,29%	14,29%

Nesimetrična porazdelitev razredov							
razred	SKY	CEMENT	WINDOW	BRICKFACE	FOLIAGE	PATH	GRASS
podmnožica 1	2,54%	2,55%	2,54%	23,05%	23,11%	23,12%	23,07%
podmnožica 2	29,05%	29,04%	29,04%	3,27%	3,19%	3,18%	3,23%

Tabela 5.18: Enakomerna in nesimetrična porazdelitev podmnožic segmentation. Vrednosti predstavljajo odstotek primerov z danim razredom.

podatkovna množica	segmentation		lymphography	
porazdelitev	enakomerna	nesimetrična	enakomerna	nesimetrična
klasifikacijska točnost	0,9109	0,8024	0,9508	0,9981

Tabela 5.19: Klasifikacijske točnosti različice gozd porazdeljenih odločitvenih dreves z zgrajenim modelom na podmnožicah z enakomerno in nesimetrično porazdelitvijo razredov.

goritmom za gradnjo odločitvenih dreves smo obdelali lokalno podmnožico podatkov segmentation velikosti 128 MB, kar predstavlja zgornjo mejo na dani specifikaciji delovnih vozlišč.

5.3.3 Porazdeljeni naključni gozdovi

Različico porazdeljenih naključnih gozdov (v nadaljevanju imenujemo to različico DRF, angl. Distributed Random Forest) primerjamo z naključnimi gozdovi v paketu scikit. Algoritmoma smo nastavili globino drevesa na 50, minimalno število primerov v listu pa na 5. Algoritmu v paketu scikit smo nastavili število odločitvenih dreves na 50 in entropijo za mero za ocenjevanje atributov. Različici DRF smo nastavili število odločitvenih dreves na lokalni podmnožici podatkov na 20, informacijski prispevek za mero za ocenjevanje atributov in parameter razlika na 0.3, ki določi razliko med prvo in drugo najverjetnejšo napovedjo.

Podatkovna množica lymphography

Različica DRF je dosegla višjo klasifikacijsko točnost kot algoritem v paketu scikit na podatkovni množici lymphography (tabela 5.20). Pričakovali smo, da bo različica DRF dosegla višjo točnost v primerjavi z različico FDDT, ki zgradi eno odločitveno drevo na lokalni podmnožici podatkov. Nismo pa pričakovali, da bo različica DRF dosegla višjo točnost v primerjavi z algoritmom v paketu scikit. Menimo, da je razlog za višjo točnost v podobnih podmnožicah podatkov, kjer je veliko enakih primerov. Zato gradnja modela na celotni množici podatkov ne doprinese k točnosti klasifikatorja. Različica DRF je pri napovedih v povprečju uporabila 15.92 dreves. Različica DRF z enim delovnim vozliščem se je izvedla 26% počasneje kot naključni gozdovi v paketu scikit na podatkovni množici lymphography (tabela 5.21). Na grafu (slika 5.5) opazimo, da se čas izvajanja algoritma z dodajanjem procesnih enot zmanjšuje v skladu z idealno linearno pohitritvijo. Čas izvedbe različice DRF z 6 procesnimi enotami je skoraj enak času izvedbe algoritma v paketu scikit z 2 procesnima enotama.

podatkovna množica	segmentation		lymphography	
paket	scikit	DiscoMLL	scikit	DiscoMLL
klasifikacijska točnost	0,9939	0,8928	0,9567	0,9679

Tabela 5.20: Primerjava klasifikacijskih točnosti različice porazdeljenih naključnih gozdov z naključnimi gozdovi v paketu scikit.

	scikit	DiscoMLL			
računalnik	MacBook	Vihar	Vihar	Vihar	Vihar
virtualni stroji	1	1	3	6	9
CPE	2	2	6	12	18
branje	6:40:53	07:28:28	02:33:36	01:18:34	00:54:48
učenje	02:18:43				
napovedovanje	00:04:03	04:41:28	01:42:12	00:53:08	00:38:17
zapisovanje	00:00:18	00:04:11	00:04:11	00:04:11	00:04:11
skupni čas	09:03:57	12:14:07	04:19:59	02:15:53	01:37:16

pohitritve	0%	-26%	52%	75%	82%
		0%	65%	81%	87%
			0%	48%	63%
				0%	28%

Tabela 5.21: Primerjava časov izvajanja in pohitritev različice porazdeljenih naključnih gozdov z algoritmom naključni gozdovi v paketu scikit na podatkovni množici lymphography.

Podatkovna množica segmentation

Algoritem v paketu scikit je dosegel občutno višjo klasifikacijsko točnost kot algoritem v paketu DiscoMLL na podatkovni množici segmentation (tabela 5.20). Pričakovali smo, da bo kljub manj zmogljivi funkciji za iskanje številskih razcepov, različica DRF dosegla višjo točnost od različice FDDT, ki zgradi eno odločitveno drevo na lokalni podmnožici podatkov. Različica DRF je za napoved v povprečju uporabila 43.09 dreves. Različica DRF se izvede 62% počasneje od algoritma v paketu scikit na podatkovni množici segmentation (tabela 5.22). Razlog je v računanju velikega števila informa-

cijskih prispevkov pri iskanju najboljšega razcepa za številske atribute. Na grafu (slika 5.6) opazimo, da se čas izvedbe algoritma zmanjšuje z dodajanjem procesnih enot. Čas izvedbe različice DRF z 6 procesnimi enotami je skoraj enak času izvedbe algoritma v paketu scikit z 2 procesnima enotama.

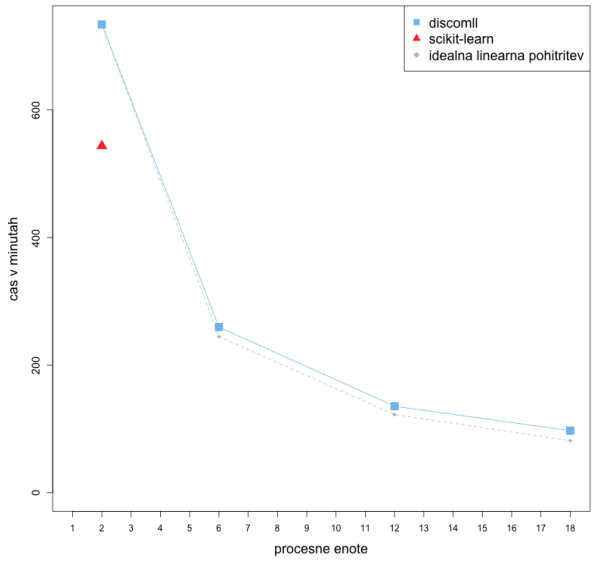
	scikit	DiscoMLL			
računalnik	MacBook	Vihar	Vihar	Vihar	Vihar
virtualni stroji	1	1	3	6	9
CPE	2	2	6	12	18
branje	0:15:48	11:47:28	04:01:42	02:05:48	01:29:27
učenje	05:45:23				
napovedovanje	00:04:47	04:08:04	01:30:52	00:44:44	00:34:45
zapisovanje	00:00:09	00:05:50	00:05:50	00:05:50	00:05:50
skupni čas	06:06:07	16:01:22	05:38:24	02:56:22	02:10:02

pohitritve	0%	-62%	8%	52%	64%
		0%	65%	82%	86%
			0%	48%	62%
				0%	26%

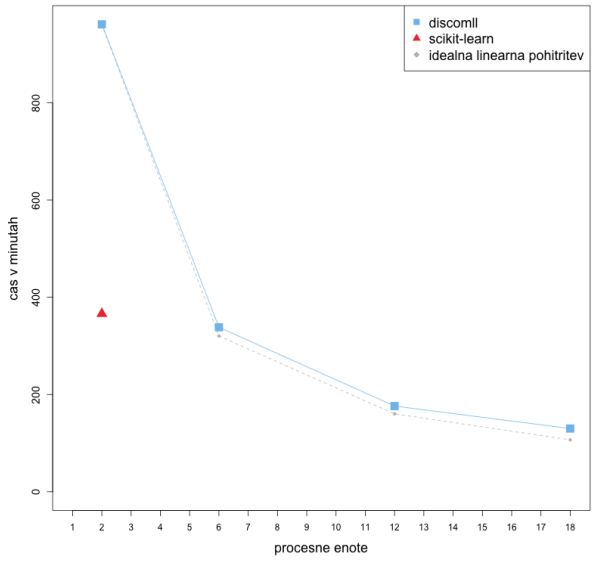
Tabela 5.22: Primerjava časov izvajanja in pohitritev različice porazdeljenih naključnih gozdov z algoritmom naključni gozdovi v paketu scikit na podatkovni množici segmentation.

Nesimetrična porazdelitev razredov

Z različico DRF smo zgradili modele na učnih množicah z različno porazdelitvijo razredov (tabeli 5.17 in 5.18). Opazili smo, da ima porazdelitev razredov manj vpliva na različico DRF v primerjavi z različico FDDT, saj so klasifikacijske točnosti bolj usklajene (tabela 5.23). To smo pričakovali, saj pri različici DRF zgradimo raznolik ansambel, na katerega ima porazdelitev razredov manj vpliva.



Slika 5.5: Primerjava časov izvajanja porazdeljenih naključnih gozdov z naključnimi gozdovi na podatkovni množici lymphography.



Slika 5.6: Primerjava časov izvajanja porazdeljenih naključnih gozdov z naključnimi gozdovi na podatkovni množici segmentation.

podatkovna množica	segmentation		lymphography	
porazdelitev	enakomerna	nesimetrična	enakomerna	nesimetrična
klasifikacijska točnost	0,8967	0,8800	0,9606	0,9671

Tabela 5.23: Klasifikacijske točnosti različice porazdeljenih naključnih gozdov z zgrajenim modelom na podmnožicah z enakomerno in nesimetrično porazdelitvijo razredov.

5.3.4 Porazdeljeni uteženi naključni gozdovi

Različico porazdeljenih uteženih naključnih gozdov (v nadaljevanju imenujemo to različico DWRF, angl. Distributed Weighted Random Forest) primerjamo z naključnimi gozdovi v paketu scikit. Algoritmoma smo nastavili globino drevesa na 50, minimalno število primerov v listu pa na 5. Algoritmu v paketu scikit smo nastavili število odločitvenih dreves na 50 in entropijo za mero za ocenjevanje atributov. Različici DWRF smo nastavili število odločitvenih dreves na lokalni podmnožici podatkov na 20, informacijski prispevek za mero za ocenjevanje atributov, parameter k razvrščanja v skupine pa algoritem nastavi samodejno in je pri podatkovni množici segmentation 4, pri podatkovni množici lymphography pa 3.

Podatkovna množica lymphography

Različica DWRF doseže višjo klasifikacijsko točnost kot algoritem v paketu scikit na podatkovni množici lymphography (tabela 5.24). Različica DWRF uporablja natančnejše napovedovanje v primerjavi z različico DRF, saj za napoved uporabi le drevesa z visokim zaupanjem na podobnih primerih. Čeprav je različica DWRF računsko zahtevnejša od različice DRF, na podatkovni množici lymphography ne doseže višje točnosti. Različica DWRF z enim delovnim vozliščem se izvede 69% počasneje od algoritma v paketu scikit (tabela 5.25). Razlog je v računsko zahtevnejši gradnji modela in napovedovanju. Na grafu (slika 5.7) opazimo, da se čas izvedbe algoritma zmanjšuje z dodajanjem procesnih enot, čeprav pa vidno odstopa od idealne linearne pohitritve.

podatkovna množica	segmentation		lymphography	
paket	scikit	DiscoMLL	scikit	DiscoMLL
klasifikacijska točnost	0,9939	0.9247	0,9567	0.9667

Tabela 5.24: Primerjava klasifikacijskih točnosti različice porazdeljenih uteženih naključnih gozdov z naključnimi gozdovi v paketu scikit.

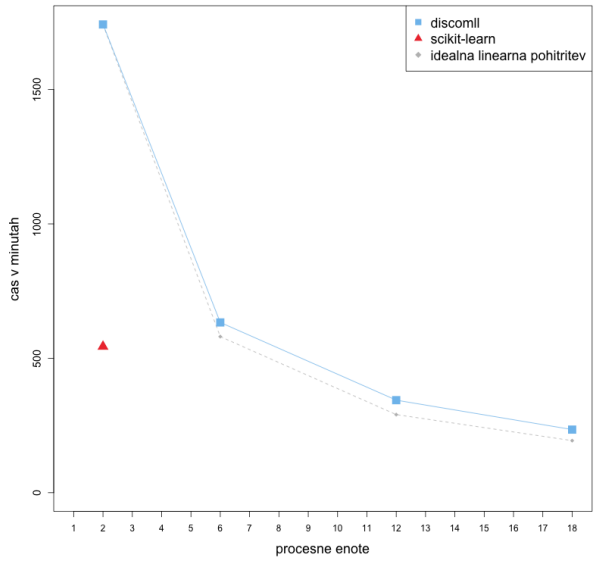
	scikit	DiscoMLL			
računalnik	MacBook	Vihar	Vihar	Vihar	Vihar
virtualni stroji	1	1	3	6	9
CPE	2	2	6	12	18
branje	6:40:53	07:29:53	02:43:28	01:23:40	00:57:22
učenje	02:18:43				
napovedovanje	00:04:03	21:30:15	07:48:06	04:18:35	02:55:38
zapisovanje	00:00:18	00:01:54	00:01:54	00:01:54	00:01:54
skupni čas	09:03:57	29:02:02	10:33:28	05:44:09	03:54:54

pohitritve	0%	-69%	-14%	37%	57%
		0%	64%	80%	87%
			0%	46%	63%
				0%	32%

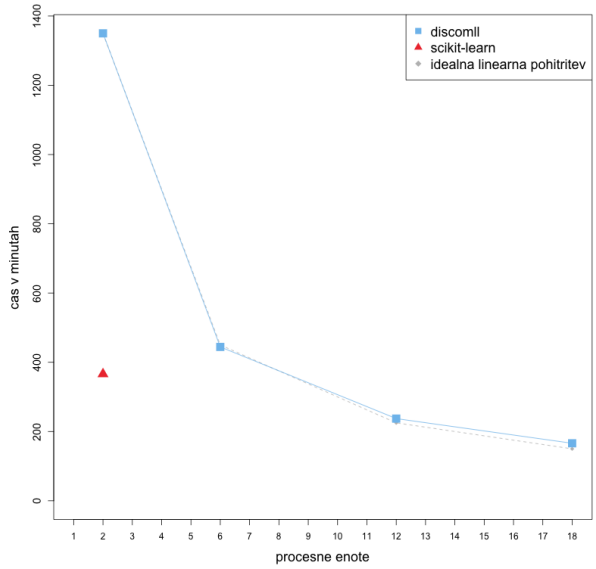
Tabela 5.25: Primerjava časov izvajanja in pohitritev različice porazdeljenih uteženih naključnih gozdov z algoritmom naključni gozdovi v paketu scikit na podatkovni množici lymphography.

Podatkovna množica segmentation

Na podatkovni množici segmentation algoritem v paketu scikit doseže višjo klasifikacijsko točnost (tabela 5.24). Različica DWRF doseže višjo klasifikacijsko točnost v primerjavi z različico DRF. Različica DWRF se na enem delovnem vozlišču izvede 73% počasneje od algoritma v paketu scikit (tabela 5.26). Na grafu (slika 5.8) opazimo, da se z dodajanjem procesnih enot algoritem na začetku približa potem pa počasi oddaljuje od idealne linearne pohitritve.



Slika 5.7: Primerjava časov izvajanja porazdeljenih uteženih naključnih gozdov z naključnimi gozdovi na podatkovni množici lymphography.



Slika 5.8: Primerjava časov izvajanja porazdeljenih uteženih naključnih gozdov z naključnimi gozdovi na podatkovni množici segmentation.

	scikit	DiscoMLL			
računalnik	MacBook	Vihar	Vihar	Vihar	Vihar
virtualni stroji	1	1	3	6	9
CPE	2	2	6	12	18
branje	0:15:48	12:13:48	04:09:03	02:12:43	01:31:48
učenje	05:45:23				
napovedovanje	00:04:47	10:15:14	03:14:27	01:43:47	01:13:13
zapisovanje	00:00:09	00:01:02	00:01:02	00:01:02	00:01:02
skupni čas	06:06:07	22:30:04	07:24:32	03:57:32	02:46:03

pohitritve	0%	-73%	-18%	35%	55%
		0%	67%	82%	88%
			0%	47%	63%
				0%	30%

Tabela 5.26: Primerjava časov izvajanja in pohitritev različice porazdeljenih uteženih naključnih gozdov z algoritmom naključni gozdovi v paketu scikit na podatkovni množici segmentation.

Nesimetrična porazdelitev podatkov

Z različico DWRF smo zgradili modele na učnih množicah z različno porazdelitvijo razredov (tabeli 5.17 in 5.18). Porazdelitev razredov ima podoben vpliv na različico DWRF kot na različico DRF na podmnožici lymphography (tabela 5.27). Opazimo pa večje odstopanje pri podmnožici segmentation, kjer pri enakomerni porazdelitvi algoritem doseže visoko točnost, pri nesimetrični pa nizko (podobno kot različica FDDT, ki zgradi eno odločitveno drevo na lokalni podmnožici podatkov). Pričakovali smo, da bo porazdelitev razredov imela manj vpliva na različico DWRF, saj uporablja natančnejše napovedovanje. Možen razlog za odstopanje točnosti pri podmnožici segmentation je prenizko nastavljen parameter k pri razvrščanju z voditelji ali pa premajhen vzorec primerov med katerimi iščemo medoide.

podatkovna množica	segmentation		lymphography	
porazdelitev	enakomerna	nesimetrična	enakomerna	nesimetrična
klasifikacijska točnost	0,9258	0,8066	0,9673	0,9607

Tabela 5.27: Klasifikacijske točnosti različice porazdeljenih uteženih naključnih gozdov z zgrajenim modelom na podmnožicah z enakomerno in nesimetrično porazdelitvijo razredov.

5.3.5 Povzetek rezultatov različic naključnih gozdov

Različica FDDT, ki zgradi eno odločitveno drevo na lokalni podmnožici podatkov, je hiter klasifikator v primerjavi z ostalimi implementirani različicami porazdeljenih naključnih gozdov. Na podatkovni množici lymphography doseže malo manjšo točnost kot naključni gozdovi ali naivni Bayes v paketu scikit. Implementirane različice porazdeljenih naključnih gozdov dosežejo nižjo točnost na podatkovni množici segmentation, ki ima le številske atribute. Različice uporabljajo enako funkcijo za iskanje najboljšega razcepa s številskimi atributi, pri kateri smo sprejeli kompromis med točnostjo in hitrostjo izvedbe. Čeprav pa vse implementirane različice dosežejo občutno višjo točnost kot algoritem naivni Bayes na podatkovni množici segmentation. Izboljšava funkcije za iskanje najboljšega razcepa s številskimi atributi bi zvišala klasifikacijsko točnost vsem implementiranim različicam porazdeljenih naključnih gozdov na podatkovnih množicah s številskimi atributi.

Različica DRF izboljša točnost klasifikacije na podatkovni množici lymphography in je odpornejša na lokalne podmnožice z nesimetrično porazdelitvijo razredov. Gre za računsko zahtevnejšo različico porazdeljenih naključnih gozdov, saj pri gradnji modela zgradimo več odločitvenih dreves in jih uporabimo pri napovedovanju.

Različica DWRF je kljub manj zmogljivi funkciji za iskanje najboljšega razcepa s številski atributi dosegla primerljivo točnost z naključnimi gozdovi v paketu scikit na podatkovni množici segmentation. Ta različica je pri gradnji modela in napovedovanju računsko najzahtevnejša. Trenutno algoritem izbere 500 neizbranih primerov in jih napove. To število predstavlja

omejitev in možnost za izboljšavo. Različica DWRF hrani simetrične pare v matriki podobnosti, kar poveča porabo glavnega pomnilnika. Možna izboljšava je hranjenje enega izmed parov, ampak bi s tem podaljšali izvajanje algoritma razvrščanje z voditelji. Pri napovedovanju izračunamo podobnosti med vsakim primerom in medoidi. Pri tem računsko zahtevnost napovedovanja zmanjšamo z uporabo gozda, ki vsebuje najpodobnejši medoid. Napovedovanje bi lahko pohitrili, saj trenutno z matričnim množenjem izračunamo podobnost testnega primera z vsemi medoidi v gozdu. Za pohitritev bi lahko z matričnim množenjem izračunali podobnost tesnega primera z vsemi medoidi v vseh gozdovih.

5.3.6 Logistična regresija

Algoritem logistična regresija ovrednotimo na podatkovnih množicah ionosphere in sonar, ki imata številske atribute in binarni razred.

Podatkovna množica ionosphere

Algoritmu logistična regresija v paketu DiscoMLL smo nastavili parameter za maksimalno število iteracij na 15, paket Knime pa ne omogoča nastavitve tega parametra. Algoritma sta dosegla enako klasifikacijsko točnost na podatkovni množici ionosphere (tabela 5.28). Algoritem v paketu DiscoMLL se je izvajal 29% hitreje na enem delovnem vozlišču v primerjavi z algoritmom v paketu Knime (tabela 5.29). Na grafu (slika 5.9) opazimo, da se z dodajanjem procesnih enot čas izvajanja zmanjšuje v skladu z idealno linearno pohitritvijo.

podatkovna množica	ionosphere		sonar	
paket	Knime	DiscoMLL	Knime	DiscoMLL
klasifikacijska točnost	0,9608	0,9608	0,9416	0,9416

Tabela 5.28: Primerjava klasifikacijskih točnosti algoritma logistična regresija.

	Knime 2	DiscoMLL			
računalnik	Vihar	Vihar	Vihar	Vihar	Vihar
virtualni stroji	1	1	3	6	9
CPE	2	2	6	12	18
branje	00:53:25	4:01:02	1:26:57	0:45:51	0:31:03
učenje	05:00:11				
napovedovanje	00:06:23	00:15:42	00:05:27	00:02:51	00:01:56
zapisovanje	00:02:43	00:02:07	00:02:07	00:02:07	00:02:07
skupni čas	06:02:42	04:18:51	01:34:31	00:50:49	00:35:06

pohitritve	0%	29%	74%	86%	90%
		0%	63%	80%	86%
			0%	46%	63%
				0%	31%

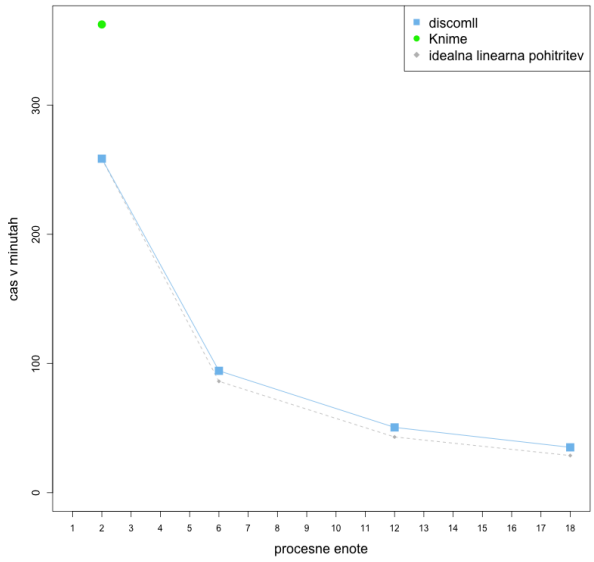
Tabela 5.29: Primerjava časov izvajanja in pohitritev z algoritmom logistična regresija na podatkovni množici ionosphere.

Podatkovna množica sonar

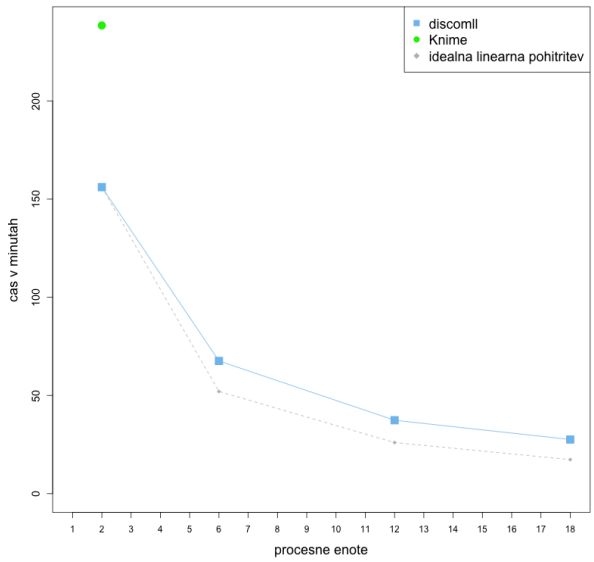
Algoritmu logistična regresija v paketu DiscoMLL smo nastavili parameter za maksimalno število iteracij na 10, paket Knime pa ne omogoča nastavitve tega parametra. Algoritma sta dosegla enako klasifikacijsko točnost na podatkovni množici sonar (tabela 5.28). Algoritem v paketu DiscoMLL se je izvajal 35% hitreje kot algoritem v paketu Knime (tabela 5.30). Na grafu (slika 5.10) opazimo večje odstopanje od idealne linearne pohitritve pri 6 procesnih enot. Z dodajanjem več procesnih enot se čas izvajanja zmanjšuje, ampak odstopa od idealne linearne pohitritve.

Podatkovna množica z velikim številom atributov

Z algoritmom logistična regresija v paketu DiscoMLL smo našli zgornjo mejo pri 3.500 atributih na delovnem vozlišču z dano specifikacijo (tabela 5.31). Razlog za obdelavo manjšega števila atributov v primerjavi z ostalimi predstavljenimi algoritmi, je uporaba zunanjega produkta pri izračunu Hessove matrike. Pri zunanjem produktu vektorja, ki zasede 0.1 MB glavnega po-



Slika 5.9: Primerjava časov izvajanja algoritma logistična regresija na podatkovni množici ionosphere.



Slika 5.10: Primerjava časov izvajanja algoritma logistična regresija na podatkovni množici sonar.

	Knime 2	DiscoMLL			
računalnik	Vihar	Vihar	Vihar	Vihar	Vihar
virtualni stroji	1	1	3	6	9
CPE	2	2	6	12	18
branje	00:50:26	2:23:05	1:02:12	0:34:15	0:25:06
učenje	02:55:56				
napovedovanje	00:10:33	00:12:23	00:05:05	00:02:43	00:02:09
zapisovanje	00:01:47	00:00:40	00:00:40	00:00:40	00:00:40
skupni čas	3:58:42	2:36:08	1:07:57	0:37:38	0:27:55

pohitritve	0%	35%	72%	84%	88%
		0%	56%	76%	82%
			0%	45%	59%
				0%	26%

Tabela 5.30: Primerjava časov izvajanja in pohitritev z algoritmom logistična regresija na podatkovni množici sonar.

mnilnika, poraba naraste na 90 MB. Velikost modela pri logistični regresiji je $št.atributov + 1$.

podatkovna množica	
št. atributov	3.500
primeri	100
razredi	2
min. vrednost	0
max vrednost	0,99
št. decimalnih mest	2
velikost podatkov	1,6 MB

zmogljivost	
št. atributov	3.500
št. iteracij	3
učenje	00:18:01
napovedovanje	00:00:27
velikost modela	77 KB

Tabela 5.31: Podatkovna množica z velikim številom številskih atributov in mejna zmogljivost algoritma logistična regresija.

5.3.7 Linearni SVM

Algoritem linearni SVM ovrednotimo na podatkovnih množicah ionosphere in sonar, ki imata številske atribute in binarni razred. Algoritmu linearni SVM

v paketu DiscoMLL smo nastavili parameter nu za prilagoditev klasifikatorja na 0.1, algoritmu v paketu scikit pa smo pustili privzete parametre.

Podatkovna množica ionosphere

Algoritem v paketu scikit je dosegel višjo klasifikacijsko točnost kot algoritem v paketu DiscoMLL na podatkovni množici ionosphere (tabela 5.32). Menimo, da je razlog za višjo točnost algoritma zmogljivejša implementacija algoritma v paketu scikit. Pri primerjavi časov izvajanja (tabela 5.33) opazimo učinkovitost preprostejše različice algoritma linearni SVM v paketu DiscoMLL, saj se je algoritem izvedel 36% hitreje na manj zmogljivem delovnem vozlišču v primerjavi z algoritmom v paketu scikit. Na grafu (slika 5.11) opazimo, da z dodajanjem procesnih enot čas izvedbe počasi odstopa od idealne linearne pohitritve.

podatkovna množica	ionosphere		sonar	
paket	scikit	DiscoMLL	scikit	DiscoMLL
klasifikacijska točnost	0,9607	0,9483	0,9426	0,9241

Tabela 5.32: Primerjava klasifikacijskih točnosti algoritma linearni SVM.

Podatkovna množica sonar

Algoritem v paketu scikit je dosegel višjo klasifikacijsko točnost kot algoritem v paketu DiscoMLL (tabela 5.32). Algoritem v paketu scikit se je izvedel 35% hitreje na podatkovni množici sonar od algoritma DiscoMLL z enim delovnim vozliščem (tabela 5.34). Opazimo, da algoritem v paketu scikit zgradi model hitreje na podatkovni množici sonar kot na podatkovni množici ionosphere. Menimo, da je razlog v različnem številu primerov, saj jih ima podatkovna množica sonar 2.980.000, podatkovna množica ionosphere pa 5.587.500. Kljub veliki razliki v številu primerov, algoritem DiscoMLL zgradi model v podobnem času na obeh podatkovnih množicah. Menimo, da je razlog v različnem številu atributov, saj jih ima podatkovna množica

	scikit	DiscoMLL			
računalnik	MacBook	Vihar	Vihar	Vihar	Vihar
virtualni stroji	1	1	3	6	9
CPE	1	2	6	12	18
branje	0:08:34	00:13:11	00:05:20	00:02:51	00:02:10
učenje	00:29:27				
napovedovanje	00:00:04	00:10:59	00:04:19	00:02:02	00:01:46
zapisovanje	00:00:05	00:00:23	00:00:23	00:00:23	00:00:23
skupni čas	00:38:10	00:24:33	00:10:02	00:05:16	00:04:19

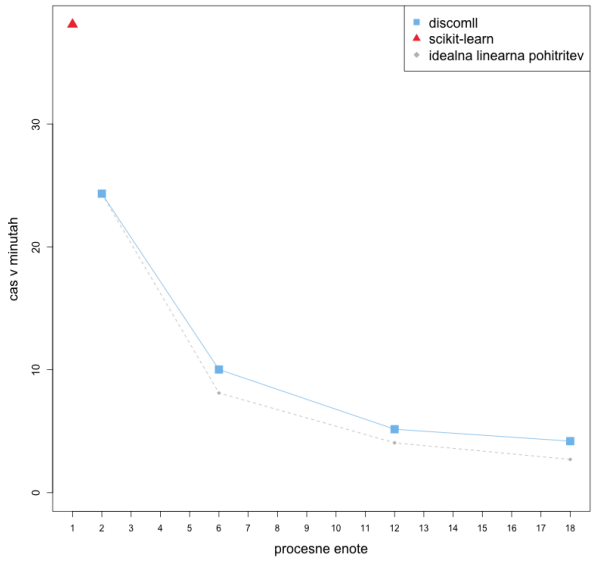
pohitritve	0%	36%	74%	86%	89%
		0%	59%	79%	82%
			0%	48%	57%
				0%	18%

Tabela 5.33: Primerjava časov izvajanja in pohitritev z algoritmom linearni SVM na podatkovni množici ionosphere.

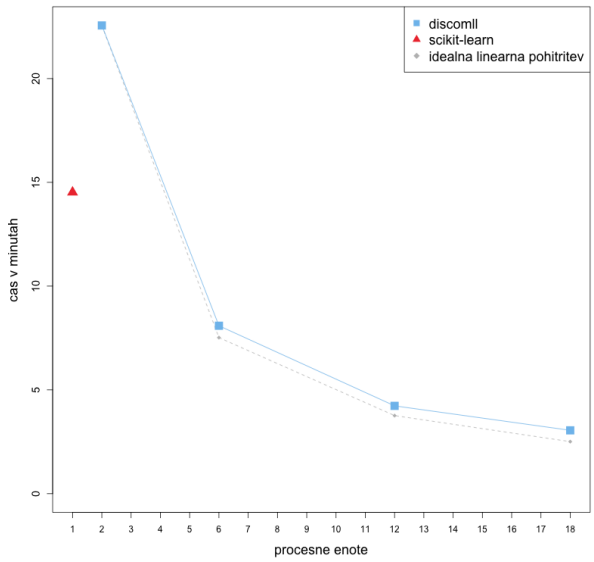
sonar 60, podatkovna množica ionosphere pa 34. Algoritem v paketu DiscoMLL porabi več časa za gradnjo modela na podatkovni množici sonar z manj atributi zaradi izračuna zunanjega produkta z večjim vektorjem. Na grafu (slika 5.12) opazimo, da se čas izvedbe algoritma v paketu DiscoMLL z dodajanjem procesnih enot približa idealni linearni pohitritvi.

Podatkovna množica z velikim številom atributov

Pri algoritmu linearni SVM smo našli zgornjo mejo pri 5.000 atributih, ki jih lahko obdelamo na delovnem vozlišču z dano specifikacijo (tabela 5.35). Algoritem pri izračunu statistik uporablja zunanji produkt. Velikost modela je $št.atributov + 1$.



Slika 5.11: Primerjava časov izvajanja algoritma linearni SVM na podatkovni množici ionosphere.



Slika 5.12: Primerjava časov izvajanja algoritma linearni SVM na podatkovni množici sonar.

	scikit	DiscoMLL			
računalnik	MacBook	Vihar	Vihar	Vihar	Vihar
virtualni stroji	1	1	3	6	9
CPE	1	2	6	12	18
branje	0:07:01	00:12:36	00:04:25	00:02:19	00:01:37
učenje	00:07:47				
napovedovanje	00:00:02	00:10:07	00:03:31	00:01:51	00:01:15
zapisovanje	00:00:02	00:00:13	00:00:13	00:00:13	00:00:13
skupni čas	00:14:52	00:22:56	00:08:09	00:04:23	00:03:05

pohitritve	0%	-35%	45%	71%	79%
		0%	64%	81%	87%
			0%	46%	62%
				0%	30%

Tabela 5.34: Primerjava časov izvajanja in pohitritev z algoritmom linearni SVM na podatkovni množici sonar.

5.3.8 Razvrščanje z voditelji

Podatkovna množica segmentation

Algoritmoma v paketih scikit in DiscoMLL smo nastavili parametra za število gruč na 7 in maksimalno število iteracij na 10. V tabeli 5.36 primerjamo povprečne vrednosti centroidov na podatkovni množici segmentation, kjer opazimo, da algoritma najdeta podobne centroide. Izmerili smo $95,67\%$ ujemanje označenih gruč med algoritmoma. Algoritem v paketu scikit se izvede 90% hitreje kot algoritem v paketu DiscoMLL (tabela 5.37). To je pričakovano, saj ima algoritem scikit podatke v glavnem pomnilniku, algoritem v paketu DiscoMLL pa izvede več poslov MapReduce pri katerih bere in piše vmesne rezultate na disk. Na grafu (slika 5.13) opazimo, da se algoritem povsem približa idealni linearni pohitritvi z dodajanjem procesnih enot.

podatkovna množica	
št. atributov	5.000
primeri	100
razredi	2
min. vrednost	0
max vrednost	0,99
št. decimalnih mest	2
velikost podatkov	2,3 MB

zmogljivost	
št. atributov	5.000
učenje	00:12:21
napovedovanje	00:00:56
velikost modela	112 KB

Tabela 5.35: Podatkovna množica z velikim številom številskih atributov in mejna zmogljivost algoritma linearni SVM.

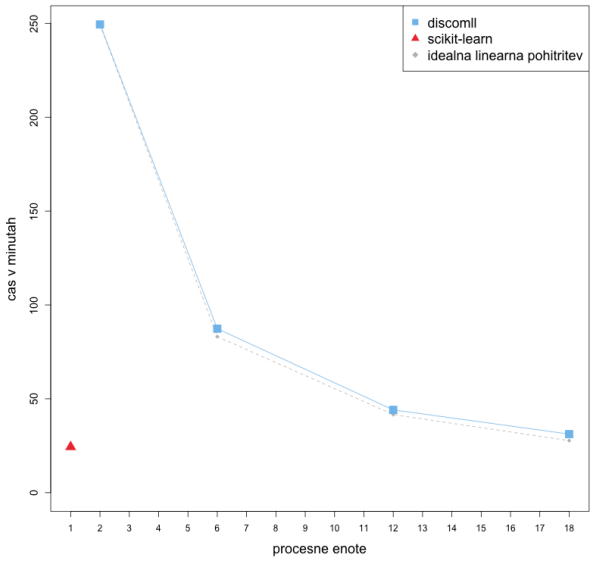
podatkovna množica segmentation							
centroid	1	2	3	4	5	6	7
DiscoMLL	13,34	23,48	25,63	27,40	41,05	47,20	88,69
scikit	13,25	24,47	25,36	27,30	42,20	47,29	100,01

podatkovna množica linear					
centroid	1	2	3	4	5
DiscoMLL	1,29	1,57	1,80	2,02	2,29
scikit	1,28	1,57	1,80	2,03	2,31

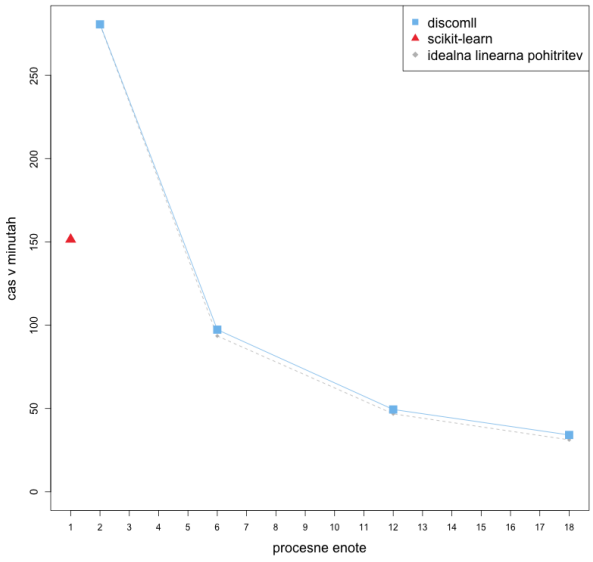
Tabela 5.36: Povprečne vrednosti najdenih centroid z algoritmom razvščanje z voditelji na podatkovnih množicah segmentation in linear.

Podatkovna množica linear

Algoritmoma v paketih scikit in DiscoMLL smo nastavili parametra za število gruč na 5 in maksimalno število iteracij na 10. V tabeli 5.36 primerjamo povprečne vrednosti centroidov na podatkovni množici linear, kjer opazimo, da algoritma najdeta podobne centroide. Izmerili smo $97,47\%$ ujemanje označenih gruč med algoritmoma. Na podatkovni množici linear se je algoritem v paketu scikit izvedel 46% hitreje od algoritma v paketu DiscoMLL z enim delovnim vozliščem, kar je manjša pohitritev kot na podatkovni množici segmentation. Na grafu (slika 5.14) opazimo, da se algoritem povsem približa idealni linearni pohitritvi z dodajanjem procesnih enot.



Slika 5.13: Primerjava časov izvajanja algoritma razvrščanje z voditelji na podatkovni množici segmentation.



Slika 5.14: Primerjava časov izvajanja algoritma razvrščanje z voditelji na podatkovni množici linear.

	scikit	DiscoMLL			
računalnik	MacBook	Vihar	Vihar	Vihar	Vihar
virtualni stroji	1	1	3	6	9
CPE	1	2	6	12	18
branje	0:16:00	3:46:15	01:18:02	00:39:18	00:26:57
učenje	00:04:38				
napovedovanje	00:02:20	00:21:50	00:07:57	00:03:15	00:02:46
zapisovanje	00:01:38	00:01:36	00:01:36	00:01:36	00:01:36
skupni čas	00:24:36	04:09:41	01:27:35	00:44:09	00:31:19

pohitritve	0%	-90%	-72%	-44%	-21%
		0%	65%	82%	87%
			0%	50%	64%
				0%	29%

Tabela 5.37: Primerjava časov izvajanja in pohitritev z algoritmom razvrščanje z voditelji na podatkovni množici segmentation.

Podatkovna množica z velikim številom atributov

Z algoritmom razvrščanje z voditelji smo našli zgornjo mejo pri 200.000 atributih z dano specifikacijo delovnega vozlišča (tabela 5.39). Velikost modela je $št.atributov * št.gruč.$

5.3.9 Linearna regresija

Podatkovna množica linear

Algoritma linearna regresija v paketih DiscoMLL in Knime dosežeta enak srednji kvadrat napake na podatkovni množici linear (tabela 5.40). Algoritem v paketu DiscoMLL se je na enem delovnem vozlišču izvedel 54% hitreje v primerjavi z algoritmom v paketu Knime (tabela 5.41). Na grafu (slika 5.15) opazimo, da se z dodajanjem procesnih enot čas izvajanja algoritma oddaljuje od idealne linearne pohitritve.

	scikit	DiscoMLL			
računalnik	MacBook	Vihar	Vihar	Vihar	Vihar
virtualni stroji	1	1	3	6	9
CPE	1	2	6	12	18
branje	1:47:26	4:15:30	1:27:31	0:44:08	0:30:02
učenje	00:25:29				
napovedovanje	00:16:34	00:24:20	00:08:51	00:04:22	00:02:59
zapisovanje	00:02:17	00:01:09	00:01:09	00:01:09	00:01:09
skupni čas	02:31:46	04:40:59	01:37:31	00:49:39	00:34:10

pohitritve	0%	-46%	36%	67%	77%
		0%	65%	82%	88%
			0%	49%	65%
				0%	31%

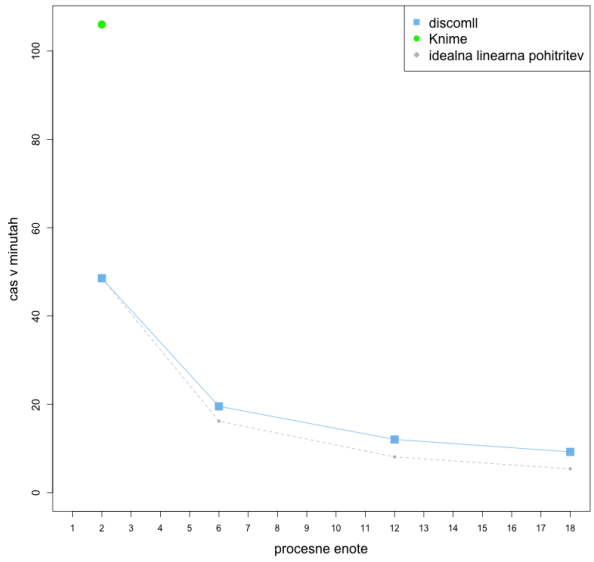
Tabela 5.38: Primerjava časov izvajanja in pohitritev z algoritmom razvrščanje z voditelji na podatkovni množici linear.

Podatkovna množica fraction

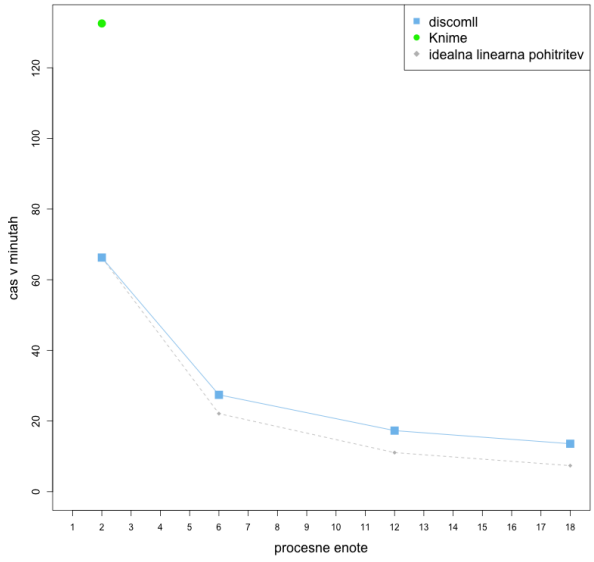
Algoritma linearna regresija v paketih DiscoMLL in Knime dosežeta enak srednji kvadrat napake na podatkovni množici fraction (tabela 5.40). Algoritem v paketu DiscoMLL se je na enem delovnem vozlišču izvedel 50% hitreje v primerjavi z algoritmom v paketu Knime (tabela 5.42). Na grafu (slika 5.16) opazimo, da se z dodajanjem procesnih enot čas izvajanja algoritma oddaljuje od idealne linearne pohitritve.

Podatkovna množica z velikim številom atributov

Z algoritmom linearna regresija smo našli mejo pri 5.000 atributih na delovnih vozliščih z dano specifikacijo (tabela 5.43). Algoritem pri izračunu parametrov uporablja zunanji produkt, ki zahteva veliko glavnega pomnilnika. Velikost modela pri linearni regresiji je določena s $št.atributov + 1$.



Slika 5.15: Primerjava časov izvajanja algoritma linearna regresija na podatkovni množici linear.



Slika 5.16: Primerjava časov izvajanja algoritma linearna regresija na podatkovni množici fraction.

podatkovna množica	
št. atributov	500.000
primeri	100
min. vrednost	0
max vrednost	0,99
št. decimalnih mest	2
velikost podatkov	234 MB

zmogljivost	
št. atributov	200.000
št. gruč	10
št. iteracij	3
učenje	0:17:43
napovedovanje	00:05:04
velikost modela	53 MB

Tabela 5.39: Podatkovna množica z velikim številom številskih atributov in mejna zmogljivost algoritma razvrščanje z voditelji.

podatkovna množica	linear		fraction	
paket	Knime	DiscoMLL	Knime	DiscoMLL
srednji kvadrat napake	6,08	6,08	0,08	0,08

Tabela 5.40: Srednji kvadrat napake z linearno regresijo na podatkovnih množicah linear in fraction.

5.3.10 Lokalno utežena linearna regresija

Paket Knime z namestitvijo razširitve nudi algoritem lokalna utežena linearna regresija, ki pa nam je ni uspelo pognati na 3 GB učni in testni množici podatkov. Algoritem je računsko prezahteven za velike podatkovne množice. Za preizkus pravilnosti algoritma v paketu DiscoMLL smo model zgradili na podmnožici podatkov in napovedali 1000 testnih primerov. Meritve časa smo izvedli z napovedjo 2 testnih primerov. Za primerjavo dodamo podaljšanje izvajanja pri napovedi 4 in 6 testnih primerov. Algoritem ima enake omejitve pri obdelavi velikega števila atributov kot logistična regresija.

Podatkovna množica linear

Učna podmnožica linear vsebuje 404.000 primerov (84 MB), testna podmnožica pa 1000 primerov (209 KB). Algoritmoma smo nastavili Gaussovo utežno funkcijo in algoritmu v paketu DiscoMLL parameter tau na 10. Algoritma sta dosegla enak srednji kvadrat napake na podatkovni množici linear (tabela 5.44). Čase izvedbe algoritma v DiscoMLL z različnim številom pro-

	Knime 2	DiscoMLL			
računalnik	Vihar	Vihar	Vihar	Vihar	Vihar
virtualni stroji	1	1	3	6	9
CPE	2	2	6	12	18
branje	01:13:00	00:22:23	00:08:03	00:04:13	00:02:47
učenje	00:15:26				
napovedovanje	00:11:46	00:22:51	00:08:10	00:04:10	00:02:55
zapisovanje	00:05:50	00:03:41	00:03:41	00:03:41	00:03:41
skupni čas	01:46:02	00:48:55	00:19:54	00:12:04	00:09:23

pohitritve	0%	54%	81%	89%	91%
		0%	59%	75%	81%
			0%	39%	53%
				0%	22%

Tabela 5.41: Primerjava časov izvajanja in pohitritev z algoritmom linearna regresija na podatkovni množici linear.

cesnih enot primerjamo v tabeli 5.45. Za vsaka dva dodatna testna primera, algoritem porabi približno 4 minute na podatkovni množici linear. Na grafu (slika 5.17) opazimo, da se z dodajanjem procesnih enot čas izvedbe algoritma oddaljuje od idealne linearne pohitritve.

Podatkovna množica fraction

Učna podmnožica fraction vsebuje 645.000 primerov (83 MB), testna podmnožica pa 1000 primerov (127 KB). Algoritmoma smo nastavili Gaussovo utežno funkcijo in algoritmu v paketu DiscoMLL parameter tau na 10. Algoritma sta dosegla enak srednji kvadrat napake na podatkovni množici fraction (tabela 5.44). Čase izvedbe algoritma v DiscoMLL z različnim številom procesnih enot primerjamo v tabeli 5.46. Za vsaka dva dodatna testna primera, algoritem porabi približno 7 minut na podatkovni množici fraction. Čas za napoved dveh testnih primerov se podaljša v primerjavi s podatkovno množico linear (14.500.000 primerov, 20 atributov), saj ima podatkovna množica fraction (23.166.666 primerov, 13 atributov) več primerov.

	Knime 2	DiscoMLL			
računalnik	Vihar	Vihar	Vihar	Vihar	Vihar
virtualni stroji	1	1	3	6	9
CPE	2	2	6	12	18
branje	01:36:19	00:28:56	00:10:12	00:05:24	00:03:45
učenje	00:15:32				
napovedovanje	00:13:13	00:31:21	00:11:18	00:05:53	00:04:00
zapisovanje	00:07:52	00:06:11	00:06:11	00:06:11	00:06:11
skupni čas	02:12:56	01:06:28	00:27:41	00:17:28	00:13:56

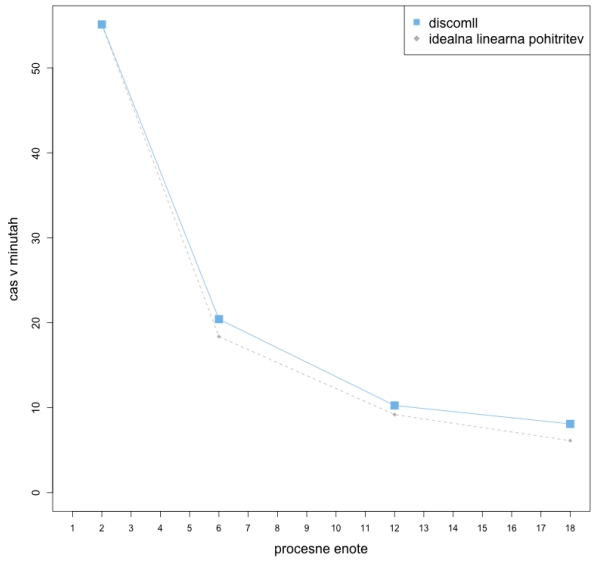
pohitritve	0%	50%	79%	87%	90%
		0%	58%	74%	79%
			0%	37%	50%
				0%	20%

Tabela 5.42: Primerjava časov izvajanja in pohitritev z algoritmom linearna regresija na podatkovni množici fraction.

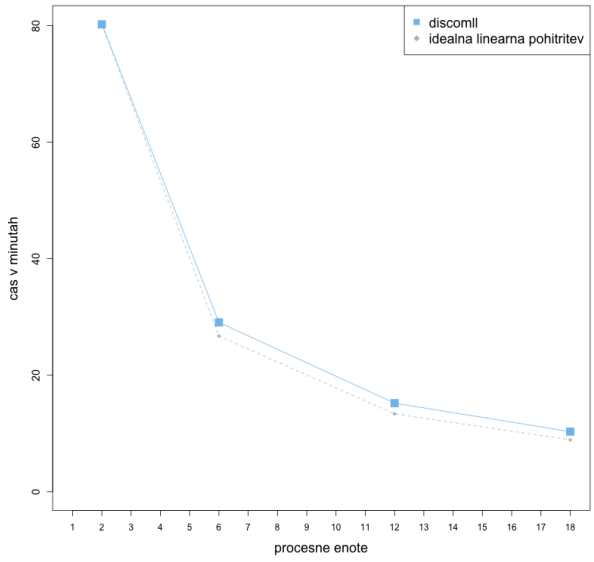
Na grafu (slika 5.18) opazimo, da se z dodajanjem procesnih enot zmanjšuje čas izvajanja algoritma v skladu z idealno linearno pohitritvijo.

5.4 Povzetek rezultatov

V tabeli 5.47 primerjamo točnost algoritmov na danih podatkovnih množicah. Algoritem naivni Bayes doseže primerljivo točnost v primerjavi z algoritmi v paketih Knime in scikit. Implementirane različice porazdeljenih naključnih gozdov dosežejo višjo klasifikacijsko točnost kot algoritem naivni Bayes (izjema je različica FDDT na podatkovni množici lymphography), kljub manj zmogljivi funkciji za iskanje razcepov s številskimi atributi. Različici DRF in DWRF dosežeta višjo klasifikacijsko točnost kot algoritem v paketu scikit na podatkovni množici lymphography. Logistična in linearna regresija sta deterministična algoritma in vrneta enake napovedi kot algoritma v paketu Knime. Algoritem linearni SVM v paketu DiscoMLL doseže nižjo točnost kot algoritem v paketu scikit. Menimo, da je razlog v manj zmogljivi im-



Slika 5.17: Primerjava časov izvajanja algoritma lokalno utežena linearna regresija na podatkovni množici linear.



Slika 5.18: Primerjava časov izvajanja algoritma lokalno utežena linearna regresija na podatkovni množici fraction.

podatkovna množica	
št. atributov	5.000
primeri	100
min. vrednost	0,00
max vrednost	0,99
št. decimalnih mest	2
velikost podatkov	2,3 MB

zmogljivost	
št. atributov	5.000
učenje	00:11:55
napovedovanje	00:00:56
velikost modela	112 KB

Tabela 5.43: Podatkovna množica z velikim številom številskih atributov in mejna zmogljivost algoritma linearna regresija.

podatkovna množica	linear		fraction	
paket	Knime	DiscoMLL	Knime	DiscoMLL
srednji kvadrat napake	4,73	4,73	0,08	0,08

Tabela 5.44: Srednji kvadrat napake z lokalno uteženo linearno regresijo na podatkovnih množicah linear in fraction.

plementaciji algoritma. Algoritma k-means nismo dodali med primerjave, saj smo primerjali centroide in oznake gruč z algoritmom v paketu scikit. Algoritma sta našla podobne centroide in podobno označila gruče na podatkovnih množicah segmentation in linear. Algoritem lokalno utežena linearna regresija v paketih Knime in DiscoMLL doseže skoraj enak srednji kvadrat napake. Algoritmi v paketu DiscoMLL dosežejo podobne točnosti kot algoritmi v paketih Knime in scikit.

V tabeli 5.48 podamo zgornjo mejo števila atributov, ki jih lahko algoritmi obdelajo na dani specifikaciji delovnih vozlišč. Algoritma naivni Bayes in k-means lahko obdelata veliko število atributov. Ostali algoritmi uporabljajo zunanji produkt, ki porabi veliko glavnega pomnilnika. Pri logistični in lokalno uteženi regresiji zunanjega produkta ne seštejemo implicitno, saj ga pred tem množimo z določenim faktorjem. Zaradi tega algoritma potrebujeta še več glavnega pomnilnika in lahko obdelata manjše število atributov v primerjavi z ostalimi implementiranimi algoritmi. Različice porazdeljenih naključnih gozdov uporabljajo enak algoritem za gradnjo odločitvenih dreves in lahko obdelajo lokalno podmnožico podatkov velikosti 128 MB.

računalnik	Vihar	Vihar	Vihar	Vihar	Vihar	Vihar
virtualni stroji	1	3	6	9	9	9
CPE	2	6	12	18	18	18
št. primerov	2	2	2	2	4	6
branje	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01
učenje	00:55:09	00:20:40	00:10:24	00:08:07	00:12:26	00:17:21
napovedovanje						
zapisovanje	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01
skupni čas	00:55:11	00:20:42	00:10:26	00:08:09	00:12:28	00:17:23

pohitritve	0%	62%	81%	85%
		0%	50%	61%
			0%	22%

Tabela 5.45: Primerjava časov izvajanja in pohitritev algoritma lokalno utežena linearna regresija na podatkovni množici linear.

Različica DWRF porabi več glavnega pomnilnika kot različica FDDT, saj DWRF vzdržuje matriko podobnosti. Z različico FDDT bi lahko obdelali malo večjo množico podatkov, ne pa veliko večje, saj omejimo velikost matrike podobnosti.

računalnik	Vihar	Vihar	Vihar	Vihar	Vihar	Vihar
virtualni stroji	1	3	6	9	9	9
CPE	2	6	12	18	18	18
št. primerov	2	2	2	2	4	6
branje	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01
učenje	01:20:21	00:29:05	00:15:19	00:10:30	00:18:13	00:25:22
napovedovanje						
zapisovanje	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01
skupni čas	01:20:23	00:29:07	00:15:21	00:10:32	00:18:15	00:25:24

pohitritve	0%	64%	81%	87%
		0%	47%	64%
			0%	31%

Tabela 5.46: Primerjava časov izvajanja in pohitritev algoritma lokalno utežena linearna regresija na podatkovni množici fraction.

Klasifikacijska točnost

Algoritem		lymphography	segmentation
Naivni Bayes	scikit	0,8276	0,8046
Naivni Bayes	DiscoMLL	0,9516	0,8046
Naivni Bayes	Knime	0,9511	0,7574
Naključni gozdovi	scikit	0,9567	0,9940
FDDT	DiscoMLL	0,9443	0,8952
DRF	DiscoMLL	0,9679	0,8928
DDWF	DiscoMLL	0,9667	0,9247

Klasifikacijska točnost

Algoritem		ionosphere	sonar
Logistična regresija	Knime	0,9608	0,9416
	DiscoMLL	0,9608	0,9416
Linearni SVM	scikit	0,9607	0,9426
	DiscoMLL	0,9483	0,9241

Srednji kvadrat napake

Algoritem		linear	fraction
Linearna regresija	Knime	6,0821	0,0833
	DiscoMLL	6,0821	0,0833
LOESS	Knime	4,7289	0,0841
	DiscoMLL	4,7284	0,0841

Tabela 5.47: Primerjava točnosti algoritmov.

	tip atributov	št. atributov
Naivni Bayes	kategorični	60.000
Naivni Bayes	številski	120.000
Logistična regresija	številski	3.500
Linearni SVM	številski	5.000
k-means	številski	200.000
Linearna regresija	številski	5.000
LOESS	številski	3.500

Tabela 5.48: Zgornja meja števila atributov, ki jih lahko obdelajo implementirani algoritmi na dani specifikaciji delovnih vozlišč.

Poglavje 6

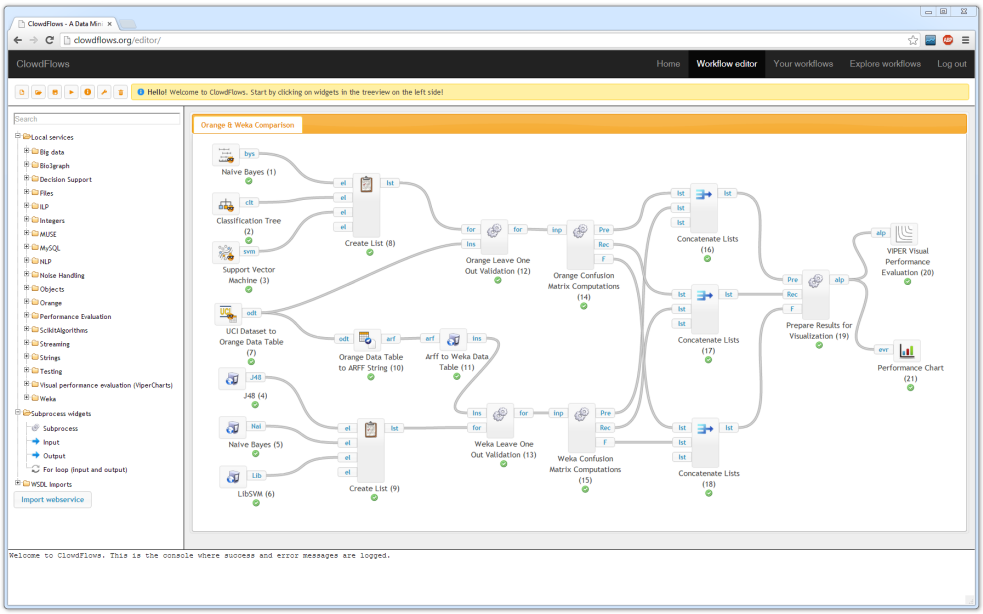
Platforma ClowdFlows

Platforma ClowdFlows [8] je odprtokodna oblačna platforma za sestavljanje, izvajanje in deljenje interaktivnih delotokov (angl. workflows) podatkovnega rudarjenja. Temelji na principu storitveno usmerjenega odkrivanja znanj iz podatkov z interaktivnimi delotoki. Za razliko od ostalih platform za podatkovno rudarjenje, se platforma ClowdFlows izvaja v spletnem brskalniku in jo lahko uporabimo tudi na mobilnih napravah. Platforma ClowdFlows raziskovalcem olajša predstavitev dela in rezultatov, saj za delo potrebujejo le povezavo do spleta in spletni brskalnik. Platformo so razvili na Institutu Jožef Stefan, na Odseku za tehnologije znanja.

6.1 Arhitektura in opis osnovnih gradnikov

Platforma ClowdFlows je sestavljena iz urejevalnika delotokov (grafični uporabniški vmesnik) in strežniške aplikacije, ki upravlja z delotoki in gosti več javno dostopnih delotokov. Urejevalnik je implementiran z jezikom za označevanje hiperteksta in s programskim jezikom javascript ter se izvaja v brskalniku. Strežniški del je napisan v programskem jeziku python in uporablja spletno ogrodje Django.

Urejevalnik delotokov (slika 6.1) sestavljata risalna površina in shramba vizualnih gradnikov (angl. widget repository), ki vsebuje seznam komponent,



Slika 6.1: Potek dela na platformi ClowdFlows za primerjavo algoritmov dveh različnih knjižnic za strojno učenje (Weka in Orange).

ki jih lahko dodamo na risalno površino. Shramba ima mnogo vgrajenih vizualnih gradnikov, med katere spadajo algoritmi za klasifikacijo iz orodja Orange [23] in algoritmi za klasifikacijo in združevanje iz orodja Weka [24]. Knjižnica algoritmov Orange je vključena v platformo ClowdFlows, saj je prav tako implementirana v programskem jeziku python, algoritmi orodja Weka pa so vključeni z uporabo spletnih storitev. Platforma vključuje več orodij za podatkovno rudarjenje, na primer orodja za rudarjenje iz podatkovnih tokov v realnem času, orodja za vizualno ocenjevanje uspešnosti algoritmov z interaktivnimi grafi in tako dalje. Shrambo vizualnih gradnikov lahko uporabniki razširijo z uvozom spletnih storitev.

Strežniška stran implementira metode za sestavljanje in izvajanje delotokov in vsebuje relacijsko podatkovno bazo delotokov, vizualnih gradnikov in podatkov. Urejevalnik dostopa do metod za manipulacijo z delotoki z zaporedjem asinhronih zahtev HTTP. Vsako zahtevo obravnava strežnik in izvede vizualne gradnike, shrani spremembe v podatkovno bazo in vrne re-

zultate klientu. Strežnik lahko upravlja z več zahtevami istočasno in lahko hkrati izvede mnogo delotokov in vizualnih gradnikov.

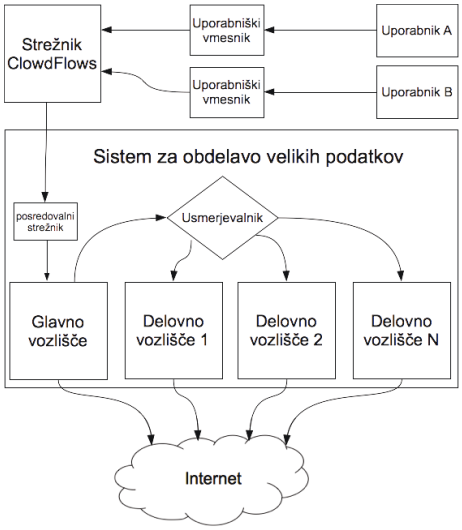
Strežnik hrani podatke v podatkovni bazi. Platforma ClowdFlows je neodvisna od različice podatkovne baze, v javno dostopni različici pa se uporablja MySQL. Podatke za obdelavo lahko naložimo na strežnik ali pa podamo le vir podatkov, kar je odvisno od uporabljene spletne storitve in vizualnega gradnika.

Kadar uporabnik odpre javno dostopen delotok, se ustvari kopija delotoka v uporabnikovi privatni shrambi delotokov. Uporabnik lahko izvede delotok in pregleda rezultate ali pa delotok spremeni in spremembe deli z ostalimi. Vsak delotok lahko javno objavimo v obliki novega javnega delotoka, kar omogoča sodelovanje.

6.2 Obdelava velikih podatkov

Platformo ClowdFlows smo združili z ogrodjem Disco in s tem dodali podporo za obdelavo velikih paketnih podatkov (slika 6.2). Oba sistema sta napisana v programskem jeziku python, kar omogoča dobro integracijo. Ogrodje Disco poganja več medsebojno povezanih virtualnih strojev s povezavo do spleta. Ta je potrebna za neposreden prenos podatkov na delovna vozlišča. Strežnik ClowdFlows komunicira z glavnim vozliščem preko posredovalnega strežnika. Ob izvedbi delotoka strežnik ClowdFlows pošlje ogrodju Disco parametre za posel s spletnimi povezavami do podatkov. Delovna vozlišča prenesejo podatke preko spleta, jih obdelajo in po končani obdelavi shranijo na porazdeljen datotečni sistem. Strežnik ClowdFlows rezultate prebere z DDFS in jih shrani v datoteko, ki jo nato lahko prenese uporabnik. Pri komunikaciji med platformo ClowdFlows in ogrodjem Disco se prenašajo le parametri o poslu in rezultat po končani obdelavi.

Za uporabnika platforme ClowdFlows je združitev sistemov neopazna, saj lahko z uporabo vizualnega programiranja obdelajo velike paketne podatke. Za vsak implementiran algoritem v knjižnici DiscoMLL (poglavje 4) smo



Slika 6.2: Združitev platforme ClowdFlows s sistemom za obdelavo velikih podatkov.

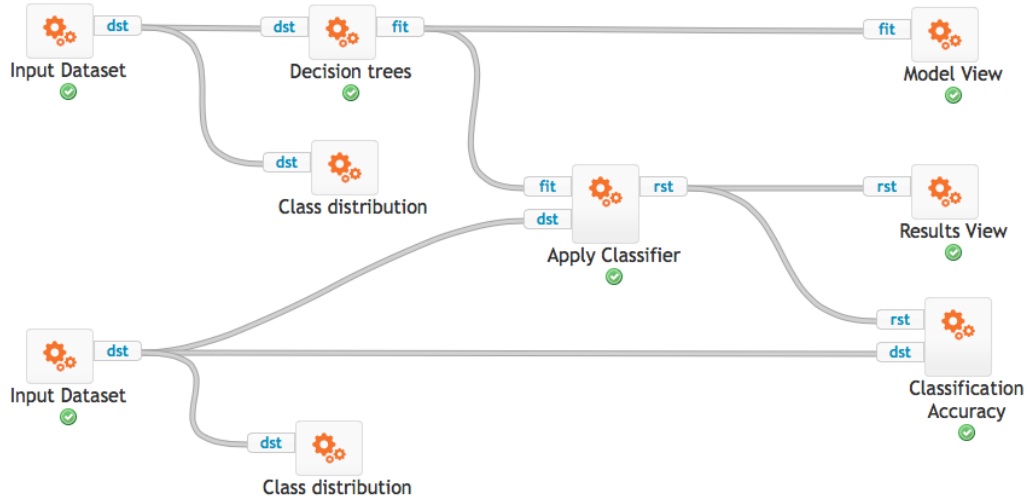
dodali gradnik ki zgradi model. Za napovedovanje smo dodali gradnik, ki izračuna napovedi z modelom. Ker platforma ClowdFlows ne hrani velikih paketnih podatkov, jih ob zahtevi prenesejo delovna vozlišča in jih podajo algoritmu. Za določitev spletnih virov podatkov smo dodali gradnik *Input dataset*, za prenos datoteke z rezultati k uporabniku pa gradnik *Results View*. Ta vse izhodne datoteke z rezultati zapiše v končno datoteko in uporabniku poda spletno povezavo za prenos.

Primer paketne obdelave podatkov s ClowdFlows

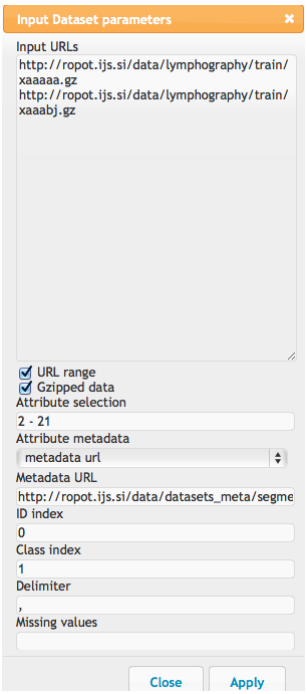
V primeru opišemo obdelavo velikih paketnih podatkov s platformo ClowdFlows. Pred začetkom obdelave je dobro podatke razdeliti na več kosov, saj je število kosov enako številu opravil map posameznega posla. Zaradi izvajanja več vzporednih opravil map se bo posel hitreje izvedel. Ob napaki na delovnemu vozlišču se tako ne izgubijo vsi podatki, ampak le tisti v kosu. V operacijskih sistemih Unix podatke razdelimo z ukazom *split -a* 5 *-l* 100000 *ime_datoteke*. Parameter *a* določi dolžino imena kosa in

kose poimenuje z *xaaaaa*, *xaaaab*, in tako dalje, parameter l pa določi število vrstic, ki jih bo vseboval posamezen kos. Za zmanjšanje prenosa preko spleta podatke zgostimo z ukazom *gzip x∗*. Podatki morajo biti dostopni preko spletne povezave, da jih lahko preberejo delovna vozlišča ogrodja Disco, zato jih je potrebno naložiti na datotečni strežnik.

Delotok za obdelavo velikih paketnih podatkov sestavlja več gradnikov, ki jih izbere uporabnik iz shrambe vizualnih gradnikov, in jih poveže kot prikazuje slika 6.3. Gradnik Input Dataset (slika 6.4) določi parametre, ki se uporabijo za pridobitev in razčlenitev podatkov. V tem primeru uporabnik poda spletno povezavo do prvega in zadnjega kosa podatkov, ki jih je ustvaril z ukazom split, ter označi parameter URL range, ki ustvari vse vmesne kombinacije z imeni kosov. Ker so podatki zgoščeni je potrebno označiti parameter Gzipped data. Gradnik omogoča izbiro atributov, izbiro indeksa razreda in tako dalje. Določimo lahko povezavo do datoteke z metapodatki o atributih, ki vsebuje imena in tipe atributov. To je potrebno pri algoritmih, ki omogočajo obdelavo podatkovni množic s številskimi in s kategoričnimi atributi. V tem delotoku gradnik Input Dataset določi parametre za učno in testno množico podatkov. Gradnik Class distribution izmeri in izpiše porazdelitev razredov za vsak kos podatkovne množice. Gradnik Decision trees zgradi model iz učne množice in ga poda gradniku Apply Classifier, ki ga uporabi na testni množici. Izbira parametrov gradnika za gradnjo modela je odvisna od algoritma, ki ga uporablja gradnik. Na primer, gradnik algoritma razvrščanje z voditelji sprejme parameter za število centroidov. Gradnik Model View omogoča izpis modela za vsak algoritem. Gradnik Results View zapiše rezultat obdelave v datoteko in uporabniku poda povezavo za prenos. Gradnik Classification accuracy izračuna in prikaže klasifikacijsko točnost algoritma.



Slika 6.3: Delotok z algoritmom gozd porazdeljenih odločitvenih dreves in ostalimi gradniki za obdelavo velikih paketnih podatkov.



Slika 6.4: Gradnik Input dataset določi parametre posla.

Poglavje 7

Zaključek

V magistrskem delu smo se podrobneje seznanili s paradigmo MapReduce, na kateri temelji večina orodij za obdelavo velikih podatkov (poglavje 2). Osredotočili smo se na opis implementacije paradigme MapReduce v ogrodju Disco, ki omogoča dobro vključitev v obstoječe aplikacije, v katerih je paketna obdelava podatkov le ena izmed funkcionalnosti. Ogrodje Disco smo primerjali s konkurenčnim ogrodjem Hadoop, ki je več uporabljeno ogrodje na področju obdelave velikih paketnih podatkov (poglavje 3). V poglavju 4 smo predstavili implementirane algoritme s paradigmo MapReduce, ki jih opisujejo v [12]. Algoritmom smo dodali tri različice porazdeljenih naključnih gozdov, ki gradijo model na lokalnih podmnožicah podatkov. Dve različici uporabita podoben pristop kot pristopa MReC4.5 [3] in COMET [4], ki nista odprtokodna. Predstavimo novi algoritem porazdeljeni uteženi naključni gozdovi, ki pri gradnji modela izračuna statistike in jih uporabi kot uteži pri napovedovanju. Kodo algoritmov smo objavili v sklopu knjižnice DiscoMLL v [14] in s tem omogočili ostalim raziskovalcem, da svoje algoritme primerjajo z algoritmi v knjižnici DiscoMLL. V poglavju 5 smo ovrednotili implementirane algoritme s primerjavo s paketoma Knime ali scikit. Pri tem smo primerjali pohitritve implementiranih algoritmov z dodajanjem več delovnih vozlišč in opisali njihove omejitve. Ugotovili smo, da implementirani algoritmi dosežejo podobno točnost v primerjavi z algoritmi v paketih Knime ali scikit. Imple-

mentirani algoritmi niso omejeni s številom primerov v učni ali testni množici in se približajo idealni linearni pohitritvi z dodajanjem delovnih vozlišč. V poglavju 6 opišemo oblačno platformo za sestavljanje, izvajanje in deljenje interaktivnih delotokov podatkovnega rudarjenja. Knjižnico DiscoMLL smo vključili v platformo ClowdFlows in smo omogočili obdelavo velikih paketnih podatkov z vizualnim programiranjem. Za obdelavo velikih paketnih podatkov ni potrebno nameščati ogrodja Disco in knjižnice DiscoMLL, ampak jih lahko obdelamo na spletnem portalu ClowdFlows.

V prihodnje bi lahko v knjižnico DiscoMLL vključili več algoritmov strojnega učenja za obdelavo velikih paketnih podatkov ali knjižnici DiscoMLL razširili funkcionalnost (na primer, s prečnim preverjanjem). Pri ovrednotenju algoritmov smo pri različicah porazdeljenih naključnih gozdov, omenili težave s funkcijo za iskanje najboljšega razcepa s številskimi atributi. Funkcijo bi bilo dobro izboljšati, saj bi to izboljšalo točnost vseh različic porazdeljenih naključnih gozdov. V magistrski nalogi smo predstavili nov algoritem porazdeljeni uteženi naključni gozdovi, ki bi ga lahko s podrobnejšo analizo še izboljšali in naredili odpornejšega na nesimetrično porazdeljene razrede v lokalnih podmnožicah podatkov.

Literatura

[1] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[2] P.N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining*. Pearson International Edition. Pearson Addison Wesley, 2006.

[3] G. Wu, H. Li, X. Hu, Y. Bi, J. Zhang, and X. Wu. MReC4.5: C4.5 ensemble classification with MapReduce. In *ChinaGrid Annual Conference, 2009. ChinaGrid'09. Fourth*, pages 249–255. IEEE, 2009.

[4] J. Basilico, M. Munson, T. Kolda, K. Dixon, and W. Kegelmeyer. COMET: A recipe for learning and using large ensembles on massive data. In *Data Mining (ICDM), 2011 IEEE 11th International Conference on*, pages 41–50. IEEE, 2011.

[5] B. Panda, J. Herbach, S. Basu, and R. Bayardo. PLANET: Massively parallel learning of tree ensembles with MapReduce. *Proceedings of the VLDB Endowment*, 2(2):1426–1437, 2009.

[6] P. Mundkur, V. Tuulos, and J. Flatow. Disco: A computing platform for large-scale data analytics. In *Proceedings of the 10th ACM SIGPLAN workshop on Erlang*, pages 84–89. ACM, 2011.

[7] T. White. *Hadoop: the definitive guide*. "O'Reilly Media, Inc.", 2009.

[8] J. Kranjc, V. Podpečan, and N. Lavrač. Clowdflows: A cloud based scientific workflow platform. In *Proceedings of Machine Learning and Knowledge Discovery in Databases*, pages 816–819. Springer, 2012.

[9] H. Karloff, S. Suri, and S. Vassilvitskii. A model of computation for MapReduce. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 938–948. Society for Industrial and Applied Mathematics, 2010.

[10] R. Lämmel. Google's MapReduce programming model—revisited. *Science of computer programming*, 70(1):1–30, 2008.

[11] Disco, massive data - minimal code. `http://discoproject.org`. Accessed: 2014-10-10.

[12] C. Chu, S. Kim, Y. Lin, Y. Yu, G. Bradski, A. Ng, and K. Olukotun. MapReduce for Machine Learning on Multicore. In *NIPS*, volume 6, pages 281–288, 2006.

[13] M. Kearns. Efficient noise-tolerant learning from statistical queries. *Journal of the ACM (JACM)*, 45(6):983–1006, 1998.

[14] R. Orač. Disco Machine Learning Library. `https://github.com/romanorac/discomll`. Accessed: 2014-10-10.

[15] T. Oliphant. Python for scientific computing. *Computing in Science & Engineering*, 9(3):10–20, 2007.

[16] T.F. Chan, G.H. Golub, and R.J. LeVeque. Updating formulae and a pairwise algorithm for computing sample variances. In H. Caussinus, P. Ettinger, and R. Tomassone, editors, *COMPSTAT 1982 5th Symposium held at Toulouse 1982*, pages 30–41. Physica-Verlag HD, 1982.

[17] G. Fung and O. Mangasarian. *Incremental Support Vector Machine Classification*, chapter 15, pages 247–260.

[18] L. Breiman. Random Forests. *Machine Learning*, 45(1):5–32, 2001.

[19] L. Breiman, J. Friedman, C. Stone, and R. Olshen. *Classification and Regression Trees*. CRC Press, 1984.

[20] I. Kononenko. On biases in estimating multi-valued attributes. In *IJCAI*, volume 95, pages 1034–1040, 1995.

[21] J. Gower. A general coefficient of similarity and some of its properties. *Biometrics*, pages 857–871, 1971.

[22] M. Robnik-Šikonja. Not enough data? Generate it! Technical report, University of Ljubljana, Faculty of Computer and Information Science, 2014.

[23] J. Demšar, B. Zupan, G. Leban, and T. Curk. Orange: From Experimental Machine Learning to Interactive Data Mining. In *Proceedings of the PKDD*, pages 537–539, 2004.

[24] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. Witten. The WEKA Data Mining Software: An Update. *SIGKDD Explor. Newsl.*, 11(1):10–18, November 2009.