Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Martin Volk

Učinkovita implementacija odločitvenega drevesa z logistično regresijo v listih

diplomsko delo

univerzitetni študijski program prve stopnje
računalništvo in informatika

Mentor: prof. dr. Igor Kononenko

Ljubljana 2015

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Odločitveno drevo z logistično regresijo v listih je model strojnega učenja, ki se uspešno uporablja na področju športne analitike. Kljub dobrim lastnostim ima model tudi svoje pomanjkljivosti. Največja je ta, da se zelo hitro prilagodi vhodnim podatkom, kar pripelje do manj natančnih napovedi. Cilj diplomske naloge je implementacija algoritma za gradnjo omenjenega modela, ki bo omogočil robustno rezanje drevesa ter robustno izbiro ustrezne podmnožice atributov, ki nastopajo v logističnih modelih v listih drevesa. Implementacija algoritma mora biti tudi učinkovita, zaradi zelo zahtevne domene saj učne množice tipično vsebujejo okoli 500.000 učnih primerov.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Martin Volk, z vpisno številko **63100320**, sem avtor diplomskega dela z naslovom:

Učinkovita implementacija odločitvenega drevesa z logistično regresijo v listih

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom prof. dr. Igorja Kononenka,
- 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 na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 28. avgusta 2015 Podpis avtorja:

Zahvaljujem se mentorju prof. dr. Igorju Kononenku in as. mag. Petru Vračarju za mentorstvo in usmerjanje pri izdelavi diplomske naloge ter družini za podporo med študijem.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
AIC	Akaike information criterion	informacijski kriterij Akaike
FAM	first AIC minimum	prvi najmanjši AIC
UCI	University of California, Irvine	Univerza v Kaliforniji, Irvine
CSV	Comma-separated values	vrednosti, ločene z vejico

Povzetek

Namen diplomske naloge je bila implementacija algoritma za gradnjo odločitvenih dreves z logistično regresijo v listih. Ker se takšni algoritmi uporabljajo za analizo športnih tekem, kjer je količina podatkov zelo velika, je največja pomanjkljivost takšnih algoritmov poraba pomnilniškega prostora, ki je potreben za gradnjo modela. V nalogi sta na začetku predstavljena algoritem za gradnjo odločitvenih dreves in algoritem za izdelavo modela logistične regresije. Temu sledi opis algoritma za gradnjo dreves z logistično regresijo v listih, na katerem temelji naša rešitev. Na koncu je predstavljena naša implementacija in rezultati testov.

Ključne besede: odločitveno drevo, logistična regresija.

Abstract

The purpose of this thesis was implementation of an algorithm for building decision trees with logistic regression in the leaves. Since such algorithms are used for sport data analysis, where the amount of data to analyse is large, the biggest drawback of these algorithms is the consumption of computer memory required to build the model. First, an algorithm for building decision trees and an algorithm for building logistic regression models are described. This is followed by a description of an algorithm for building decision trees with logistic regression in the leaves, on which our solution is based. At the end our implementation is presented together with results of the tests.

Keywords: decision tree, logistic regression.

Poglavje 1

Uvod

Strojno učenje je veja umetne inteligence, ki se ukvarja s sistemi, ki se lahko učijo iz podatkov. Prve aplikacije strojnega učenja so se uporabljale predvsem v medicini, bančništvu in v raziskovalnih institucijah. Ker se je v zadnjih desetletjih uporaba računalnikov zelo razširila, se je povečala tudi količina podatkov, ki si shranjeni v elektronski obliki. To je zelo pripomoglo k razvoju strojnega učenja in njegovi širši uporabi. Uporaba odločitvenih in priporočilnih sistemov je danes tako razširjena, da se uporabljajo skoraj povsod. Primeri takšnih sistemov so filtri neželene elektronske pošte, priporočilni sistem v spletni trgovini, ipd.

Pogost problem v strojnem učenju je klasifikacijskega tipa, pri čemer se iz učnih podatkov, kjer imajo posamezni primeri označen razred, v katerega spadajo, naučimo klasificirati nov primer. Za reševanje klasifikacijskih problemov imamo na voljo več algoritmov, med katerimi sta tudi odločitveno drevo, ki ima začetke v strojnem učenju, in logistična regresija, ki ima začetke v statistiki. Oba algoritma sta opisana v poglavju 2. Da bi dobili boljše klasifikatorje, je včasih potrebno različne algoritme združiti. Primer take združitve je tudi odločitveno drevo z logistično regresijo v listih.

1.1 Motivacija

Navadno odločitveno drevo ima v listih konstantne oznake razredov, kar pomeni, da ima vsak izmed listov določen razred, kateremu bo pripadal nov primer. Listi običajno vsebujejo primere, ki ne pripadajo samo enemu razredu. Na bolj zapletenih domenah je težko poiskati pogoj, ki bi popolnoma razdelil učno množico na podmnožice, ki bi vsebovale samo primere, ki pripadajo enemu razredu. Zaradi tega se list drevesa označi kot razred, kateremu pripada večina primerov. Množico podatkov v listu lahko še naprej delimo in dobimo večje drevo, katerega listi so bolj čisti, ampak lahko drevo postane preveliko in zato tudi težje razumljivo. Vprašljiva je tudi uporabnost takšnega drevesa, saj se lahko preveč prilagodi učnim podatkom in bi na netrivialnih primerih doseglo slabše rezultate. Da bi ohranili drevo majhno in hkrati povečali njegovo točnost, lahko v listih drevesa zgradimo model logistične regresije. Tako v listih drevesa nimamo več konstantnih oznak razredov, temveč se v listih nahaja funkcija logistične regresije, ki vsak primer v listu klasificira posebej.

1.2 Sorodna dela

Med bolj znane algoritme, ki v listih drevesa namesto konstantne vrednosti uporabijo regresijo, je M5, ki ga je leta 1992 razvil Quinlan [8]. Prednost tega algoritma pred navadnim regresijskim drevesom je v tem, da je zgrajeno drevo manjše. V določenih primerih lahko algoritem zgradi navadno regresijsko drevo. To se zgodi, če je model v listu le povprečna vrednost ciljne spremenljivke vseh učnih primerov. Tako kot vsi drevesni modeli se tudi ta gradi po metodi deli in vladaj. Učna množica se razdeli na več podmnožic in dobljene podmnožice se delijo naprej, dokler ne dosežemo zaustavitvenega kriterija. V prvem delu algoritma se izračuna standardni odklon ciljne spremenljivke v učni množici. Za izbiro atributa, po katerem se deli učna množica, se uporabi izraz 1.1, kjer so T učna množica in T_i podmnožice učne množice, ki jih dobimo z izbranim atributom, sd pa je standardni odklon. Al-

goritem izbere takšen atribut, da je sprememba napake največja (standardni odklon se najbolj zmanjša).

$$\Delta error = sd(T) - \sum_i \frac{|T_i|}{|T|} \times sd(T_i) \tag{1.1}$$

Ko imamo zgrajeno regresijsko drevo, je potrebno v vozliščih zgraditi modele linearne regresije. Ker v naslednjem koraku sledi rezanje drevesa, je potrebno regresijo zgraditi v vseh vozliščih, saj se lahko vsako notranje vozlišče po rezanju spremeni v list. Atributi, ki se uporabijo pri gradnji modela linearne regresije v vozlišču, so enaki tistim, ki so uporabljeni pri gradnji poddrevesa tega vozlišča. Rezanje poteka tako, da se v vseh notranjih vozliščih preveri napako poddrevesa in napako linearnega modela. Če je napaka poddrevesa večja, se drevo v tistem vozlišču reže. Zadnji korak je glajenje, ki je potrebno zaradi nezveznosti med linearnimi modeli sosednjih listov. Za glajenje se uporablja izraz 1.2, ki ga je izpeljal Quinlan. Najprej v listu drevesa izračunamo predvideno vrednost, nato pa to vrednost združimo z vrednostjo, ki jo izračuna linearen model v očetu lista. Postopek se ponavlja, dokler ne dosežemo korena drevesa. S tem dosežemo, da so sosednji linearni modeli med seboj zvezni, kar poveča točnost celotnega drevesa.

$$p' = \frac{np + kq}{n + k} \tag{1.2}$$

p' je vrednost napovedi, ki se bo poslala očetu vozlišča, p je vrednost napovedi, ki je prišla od sina vozlišča, q je vrednost napovedi trenutnega vozlišča, število primerov, ki doseže sina vozlišča, je označeno z n in k je konstanta, ki ima privzeto vrednost 15.

Wang in Witten [10] sta leta 1997 predlagala izboljšavo algoritma M5, ki sta ga poimenovala M5'. Prva razlika med njima je ta, da so pri M5' vsi atributi binarni ali numerični. Vsak nominalni atribut s k vrednostmi je potrebno pred gradnjo drevesa pretvoriti v $k - 1$ binarnih atributov. Tukaj se lahko pojavi problem pri izbiri atributa za delitev, saj bo atribut, ki ima veliko različnih vrednosti, vedno izbran prvi. Težavo sta odpravila tako, da sta pomnožila spremembo napake s faktorjem β, ki se z večanjem vrednosti

atributa eksponentno zmanjšuje. Druga izboljšava je v obravnavanju manjkajočih vrednosti v podatkih. Posodobila sta izraz za izračun spremembe napake pri izbiri atributa.

$$\Delta error = \frac{m}{|T|} \times \beta(i) \times \left[sd(T) - \sum_{j \in D,L} \frac{|T_j|}{|T|} \times sd(T_j) \right] \tag{1.3}$$

m je število primerov brez manjkajočih vrednosti za določen atribut, T_D in T_L sta podmnožici, ki nastaneta pri delitvi po izbranem atributu. Opazimo lahko, da imamo sedaj samo dve množici, saj so vse delitve binarne. Ker se lahko pri nekaterih primerih pojavi manjkajoča vrednost atributa, po katerem delimo, je potrebno za takšne primere uporabiti poseben pristop, saj manjkajoče vrednosti ne moremo primerjati z delitvenim pogojem. Najprej se na podmnožici razdeli primere, pri katerih je vrednost atributa znana, nato pa se izračuna povprečna vrednost ciljne spremenljivke v obeh množicah. Primer z manjkajočo vrednostjo se nato dodeli podmnožici glede na to, ali je njegova ciljna spremenljivka večja ali manjša od izračunanega povprečja.

Algoritem M5' sta Wang in Witten kasneje s sodelavci spremenila tako, da sta ga lahko uporabila tudi za klasifikacijo [3]. Da se algoritem lahko uči na podatkih, ki imajo diskretne vrednosti razreda, je potrebno iz učne množice narediti za vsak razred svojo učno množico. V primeru da lahko razred zavzame 5 vrednosti, dobimo 5 učnih množic, pri katerih razred zavzame vrednosti 1 ali 0, odvisno od tega, ali primer pripada določenemu razredu ali ne. Za vsako učno množico se nato zgradi svoje odločitveno drevo. Ker je v listih drevesa še vedno linearna regresija, nam posamezno drevo vrne samo verjetnost, s katero določen primer pripada razredu. Vsak nov primer je potrebno klasificirati na vseh drevesih. Razred, za katerega dobimo največjo verjetnost, je razred, kateremu pripada nov primer. Ena izmed slabosti takšnega modela je velika računska zahtevnost, saj je potrebno zgraditi toliko modelov, kot je razredov, poleg tega pa je potrebno vsak nov primer klasificirati na vseh drevesih. Težava se pojavi tudi pri interpretaciji drevesa, saj je rešitev, ko imamo za vsak razred svoje drevo, težko interpretirati kot celoto.

Do sedaj opisane rešitve v listih drevesa uporabljajo linearno regresijo, ki je bolj primerna za probleme, ko je razred zvezen. Če imamo diskretni razred, bi bilo bolj smiselno v listih uporabiti klasifikator, kot sta npr. naivni Bayes [2] in logistična regresija. Obstaja več algoritmov, ki ustrezajo temu opisu. En izmed njih je LMT [6], ki je podrobno opisan v razdelku 2.3.

1.3 Struktura diplomske naloge

V poglavju 2 se nahaja teoretična podlaga, ki je pomembna za razumevanje ostalih poglavij. Podrobno sta opisana odločitveno drevo in logistična regresija, na koncu pa še algoritem LMT, ki v listih odločitvenega drevesa uporablja logistično regresijo. Sledi poglavje 3, kjer je opisana naša implementacija drevesa z logistično regresijo v listih. Naša implementacija temelji an obstoječem algoritmu LMT, ki so ga avtorji implementirali v sistem Weka. Ker je implementacija v Weki zelo neučinkovita, smo algoritem implementirali znova in se bolj posvetili učinkovitosti. V zadnjem delu poglavja 3 je opisana tudi naša metoda za izbiro atributa za delitev drevesa, ki jo imenujemo *bootstrapping*. V prvem delu poglavja 4 so prikazani rezultati primerjave učinkovitosti naše implementacije z Weko, v drugem delu pa rezultati testov metode *bootstrapping*. Sklepne ugotovitve se nahajajo v poglavju 5.

Poglavje 2

Teoretična podlaga

To poglavje podrobno opiše dve metodi strojnega učenja, odločitveno drevo in logistično regresijo, ki sta sestavni del našega algoritma. Vsebina podpoglavij 2.1 in 2.2 je potrebna za razumevanje naše rešitve, ki temelji na algoritmu LMT, ki je podrobno opisan v podpoglavju 2.3.

2.1 Odločitveno drevo

Odločitveno drevo je ena izmed najstarejših in največkrat uporabljenih metod nadzorovanega strojnega učenja. Poznamo dve vrsti odločitvenih dreves, ki se razlikujeta v tem, kakšnega tipa je ciljna spremenljivka, ki jo napovedujeta. Če je ciljna spremenljivka zvezna, ki lahko zavzame realne vrednosti med 0 in 1, takšnemu drevesu rečemo regresijsko drevo. V primeru, da je ciljna spremenljivka diskretna in lahko zavzema vnaprej določeno število različnih vrednosti, imenujemo takšno drevo klasifikacijsko drevo. Ker se v tem delu osredotočamo na probleme klasifikacijskega tipa, so opisana le klasifikacijska drevesa. Med najbolj znane algoritme za gradnjo odločitvenih dreves spadajo ID3, ki ga je razvil Quinlan [7], C4.5, ki je naslednik algoritma ID3, in CART, katerega avtor je Breiman [1]. Quinlanov C4.5 in Breimanov CART sta bila razvita neodvisno en od drugega, zato lahko med njima najdemo veliko razlik.

Glavna ideja gradnje odločitvenega drevesa je v tem, da učno množico

podatkov razdelimo po enem izmed atributov in tako dobimo več podmnožic, ki jih v primeru, da vsebujejo dovolj učnih primerov, razdelimo naprej, dokler niso vse množice dovolj majhne ali dovolj čiste.

2.1.1 Izbira atributa

Za izbiro atributa, po katerem v določenem vozlišču razdelimo učno množico, obstaja več pristopov, ki pa imajo skupen cilj: Razdeliti množico na podmnožice tako, da bo v vsaki podmnožici čim večji delež primerov z enakim razredom. Algoritem ID3 kot mero za izbiro atributa uporablja informacijski prispevek. Ta nam pove koliko informacije prispeva atribut k določitvi razreda. Informacijski prispevek temelji na entropiji oziroma na meri nečistoče, ki izhaja iz informacijske teorije.

$$Gain(S, A) = H(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} H(S_v) \tag{2.1}$$

Enačba 2.1 se uporablja za izračun informacijskega prispevka. $Gain(S, A)$ je informacijski prispevek atributa A pri učni množici S. $H(S)$ je entropija učne množice, $H(S_v)$ pa entropija podmnožice, pri kateri ima atribut A vrednost v. Slaba lastnost informacijskega prispevka je v tem, da imajo večvrednostni atributi večji informacijski prispevek. Quinlan je to slabost algoritma ID3 odpravil pri C4.5. Definiral je razmerje informacijskega prispevka tako, da je informacijski prispevek normaliziral z entropijo vrednosti atributa (izraz 2.2). Težava pri razmerju informacijskega prispevka je v tem, da precenjuje atribute, ki imajo zelo majhno entropijo ($H(A)$). Pri izbiri se zato upošteva samo atribute, ki imajo informacijski prispevek večji od povprečnega informacijskega prispevka vseh atributov [5].

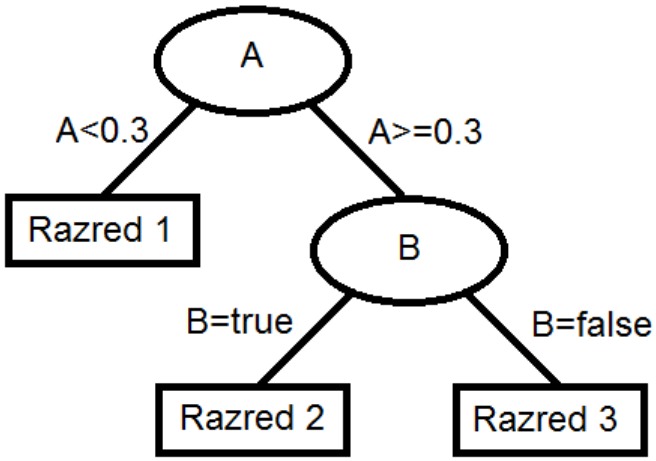
$$GainR(S, A) = \frac{Gain(S, A)}{H(A)} \tag{2.2}$$

Breiman je za izbiro atributa v svojem algoritmu uporabil gini indeks [1].

$$Gini(A) = \sum_j p_{\cdot j} \sum_k p^2_{k|j} - \sum_k p^2_k \tag{2.3}$$

2.1.2 Zaustavitveni pogoj

Naslednje pomembno vprašanje pri gradnji odločitvenega drevesa je, kdaj naj se ustavimo. Če učno množico vsakega vozlišča delimo, dokler ne pripadajo vsi primeri v listu istemu razredu, bo takšno drevo slabo generaliziralo problem, saj se bo preveč prilagodilo učnim podatkom. Poleg tega bo drevo zelo veliko in se bo gradilo dlje časa, kakor v primeru, da prej zaustavimo gradnjo. Enostavna pogoja zaustavitve sta omejena globina drevesa (drevesu omejimo, koliko je največje dovoljeno število vozlišč od korena do lista), ali pa omejimo najmanjše število učnih primerov v vozlišču. Obstajajo tudi bolj napredni načini. Tak je naprimer statistični test hi-kvadrat, ki ustavi gradnjo drevesa, ko med atributi in razredom ni več statistično pomembne povezave. Tak pristop uporablja algoritem C4.5.



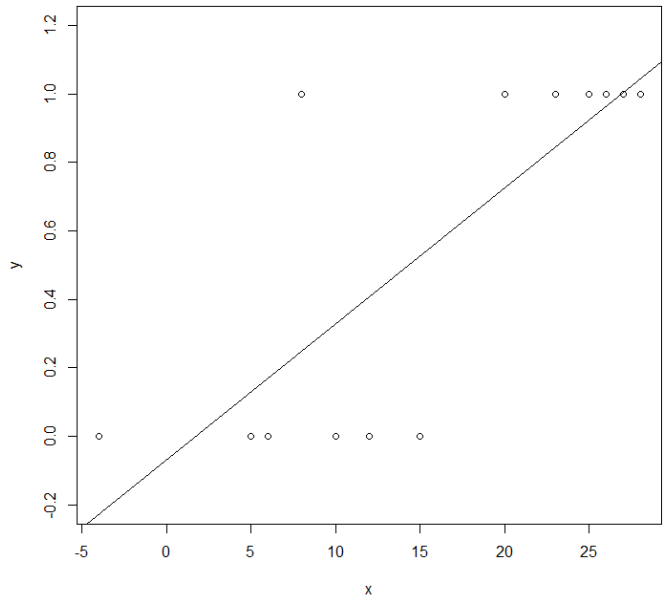
Slika 2.1: Primer odločitvenega drevesa

Na sliki 2.1 je primer odločitvenega drevesa. V korenskem vozlišču učno množico razdelimo na 2 podmnožici glede na vrednost atributa A. V primeru, da je ta vrednost manjša od 0,3, se bo primer klasificiral v prvi razred. V nasprotnem primeru pa se bo preverjal tudi binarni atribut B, na podlagi katerega bo narejena končna odločitev glede razreda.

2.2 Logistična regresija

2.2.1 Primerjava z linearno regresijo

Linearna regresija se kot klasifikator ne obnese preveč dobro, saj je namenjena napovedovanju numeričnega (zveznega) razreda. Za primer vzemimo umetno generirano učno množico, ki ima en numeričen atribut in razred z dvema vrednostima. Rezultat uporabe linearne regresije na takšni množici nam prikazuje slika 2.2. Na sliki lahko opazimo, da linearna regresija napoveduje vse vrednosti med 0 in 1, ki v našem primeru ne obstajajo, saj imamo binaren razred, ki lahko zavzame samo vrednost 0 ali 1. Poleg tega pa napoveduje tudi vrednosti, ki so večje od 1, in celo takšne, ki so negativne.



Slika 2.2: Linearna regresija kot klasifikator

Z vpeljavo logistične regresije se lahko tem težavam izognemo, saj logistična funkcija (2.4) vedno zavzame vrednosti med 0 in 1.

$$F(x) = \frac{1}{1 + e^{-\beta x}} \tag{2.4}$$

V zgornji enačbi β nastopa kot vektor parametrov in x kot vektor neodvisnih spremenljivk ali atributov. βx lahko drugače zapišemo kot $\beta_0 + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_n x_n$. Do logistične funkcije pridemo tako, da na začetku izberemo en razred, ki bo bazni razred. V našem primeru imamo samo 2 razreda (0 in 1), zato bomo kot bazni razred izbrali 1. V primeru, da bi imeli več razredov, bi vsak razred potreboval svojo logistično funkcijo (2.4). Tako bi imeli za vsak razred svoj vektor parametrov β. Če verjetnost, da bo primer pripadal baznemu razredu, p delimo z verjetnostjo, da ne bo pripadal baznemu razredu $1 - p$, dobimo razmerje verjetnosti:

$$\frac{p}{1-p} \tag{2.5}$$

Če zgornje razmerje (2.5) logaritmiramo z naravnim logaritmom, dobimo naslednjo enačbo:

$$ln\left(\frac{p}{1-p}\right) = \beta x \tag{2.6}$$

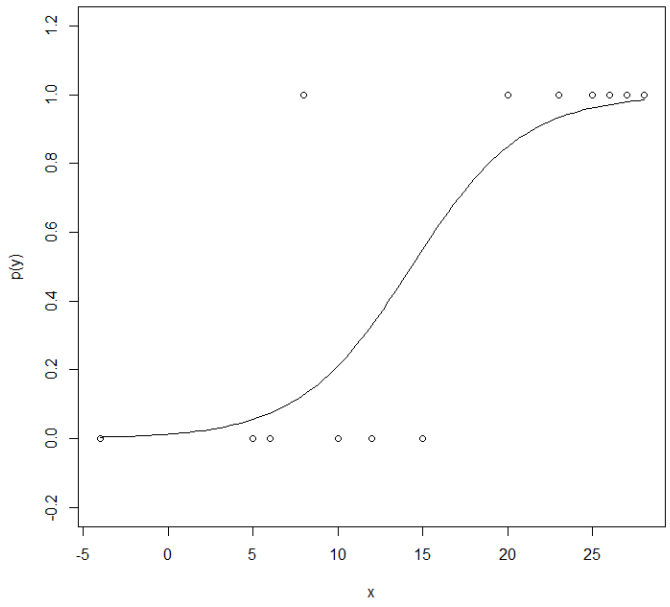
$$\frac{p}{1-p} = e^{\beta x} \tag{2.7}$$

Če se v enačbi (2.6) znebimo naravnega logaritma, dobimo enačbo (2.7), iz katere nato izpeljemo verjetnost p, ki predstavlja verjetnost, da nek primer spada v bazni razred, ki je v našem primeru 1. V primeru multinominalne logistične regresije je potrebno za vsak razred izračunati svojo verjetnost p_j (2.9), kjer je $F_j(x)$, logistična funkcija za razred j in J število razredov. Vsota vseh verjetnosti mora biti enaka 1.

$$p = \frac{1}{1 + e^{-\beta x}} \tag{2.8}$$

$$p_j = \frac{e^{F_j(x)}}{\sum_{k=1}^{J} e^{F_k(x)}} \tag{2.9}$$

Če na istih podatkih, ki smo jih uporabili pri linearni regresiji (slika 2.2), zgradimo model logistične regresije, dobimo krivuljo (slika 2.3), ki je omejena na vrednosti med 0 in 1. Za razliko od linearne regresije, ki je napovedovala vrednosti odvisne spremenljivke y, ta napoveduje verjetnost, da bo spremenljivka y pri določenem x zavzela vrednost 1. Lastnosti te krivulje so določene s parametri v vektorju β, katerih približke lahko izračunamo s pomočjo algoritma LogitBoost [4].



Slika 2.3: Primer logistične regresije

2.2.2 LogitBoost

V tem razdelku je opisan algoritem LogitBoost [4], katerega psevdokoda je na sliki 2.4. Na omenjeni sliki je opis algoritma, ki se uporablja pri multinominalni logistični regresiji, ko imamo več kot 2 razreda. Glavna razlika med binarno in multinominalno verzijo algoritma je v tem, da imamo pri multinominalni verziji za vsak razred svojo funkcijo f_{mj} in F_j in je zato potrebno narediti koraka 2.a.i in 2.a.ii za vsak razred. Na začetku določimo vrednosti

uteži $w_{ij} = 1/n$, kjer je i učni primer in j razred, kateremu pripada utež. Vrednost n je število vseh učnih primerov. $p_j(x)$ so trenutne verjetnosti, da učni primer x pripada razredu j in jih na začetku nastavimo na $p_j(x) = 1/J$, kjer je J število razredov. Algoritem se izvaja M iteracij, kjer se v vsaki iteraciji sprehodimo za vsak razred čez vse učne primere. Za vsak učni primer nato izračunamo vrednosti z_{ij}, kjer je y^*_{ij} enako 1, če primer pripada razredu j in 0 sicer. Nato izračunamo novo utež w_{ij}. S pomočjo utežene regresije, v kateri nastopa z_{ij} kot od x_i odvisna spremenljivka, izračunamo funkcije $f_{mj}(x)$. Na koncu iteracije LogitBoosta prištejemo f_{mj} vsaki funkciji $F_j(x)$ ter posodobimo verjetnosti $p_j(x)$.

1. Start with weights $w_{ij} = 1/n,\ i = 1, \ldots, n,\ j = 1, \ldots, J,\ F_j(x) = 0$ and $p_j(x) = 1/J\ \forall j$
2. Repeat for $m = 1, \ldots, M$:
   (a) Repeat for $j = 1, \ldots, J$:
      i. Compute working responses and weights in the jth class

$$z_{ij} = \frac{y^*_{ij} - p_j(x_i)}{p_j(x_i)(1 - p_j(x_i))}$$

$$w_{ij} = p_j(x_i)(1 - p_j(x_i))$$

      ii. Fit the function $f_{mj}(x)$ by a weighted least-squares regression of z_{ij} to x_i with weights w_{ij}

   (b) Set $f_{mj}(x) \leftarrow \frac{J-1}{J}(f_{mj}(x) - \frac{1}{J}\sum_{k=1}^{J} f_{mk}(x)),\ \ F_j(x) \leftarrow F_j(x) + f_{mj}(x)$

   (c) Update $p_j(x) = \frac{e^{F_j(x)}}{\sum_{k=1}^{J} e^{F_k(x)}}$
3. Output the classifier $\underset{j}{\operatorname{argmax}}\ F_j(x)$

Slika 2.4: Algoritem LogitBoost [6]

2.2.3 Izbira atributov

Za izbiro atributov ponuja LogitBoost dve možnosti. Prva je ta, da enostavno zgradimo logistično regresijo na vseh atributih. To pomeni, da za vsako regresijsko funkcijo $f_{mj}(x)$ uporabimo vse atribute. S tem načinom bo LogitBoost prej zgradil logistično regresijo, ampak bo večja verjetnost, da se bo model preveč prilegal učnim podatkom. Drugi način je, da za vsak atri-

but izračunamo svojo linearno funkcijo in nato v $F_j(x)$ dodamo samo tisto funkcijo, ki najbolj zmanjša napako modela.

2.3 Odločitveno drevo z logistično regresijo

Obstaja več algoritmov, ki v listih odločitvenega drevesa uporabljajo logistično regresijo. Najbolj uspešen in dodelan je algoritem *Logistic model tree* ali na kratko LMT, ki je podrobno opisan v tem razdelku.

LMT je algoritem, ki zgradi odločitveno drevo z logistično regresijo v listih [6]. V nadaljevanju so opisani posamezni deli algoritma, kot je gradnja drevesa, izbira atributov, rezanje drevesa in uporaba algoritma LogitBoost. Nazadnje so opisane tudi izboljšave algoritma, ki so jih avtorji algoritma uporabili za hitrejše izvajanje [9].

2.3.1 Gradnja drevesa

LMT zgradi drevo na način, kot to počne M5'. Najprej zgradi navadno klasifikacijsko drevo z algoritmom C4.5 in v vsakem vozlišču sproti zgradi model logistične regresije. Logistično regresijo zgradi v vsakem vozlišču zato, ker je vsako izmed vozlišč kandidat, da postane list po tem, ko drevo porežemo. Model logistične regresije v listih je zgrajen s pomočjo algoritma LogitBoost tako, da upošteva tudi parametre očeta trenutnega vozlišča. Na logistični model v vozlišču imajo torej vpliv tudi parametri, ki jih vsebuje oče tega vozlišča. Če pogledamo sliko 2.4, vidimo, da to lahko dosežemo tako, da prenašamo funkcije $F_j(x)$, uteži w_{ij} in verjetnosti $p_j(x)$ naprej v sinove vozlišča in v sinovih dodajamo nove linearne funkcije v $F_j(x)$. Tako imajo sinovi vozlišča določene globalne parametre, ki jih podedujejo od očeta in svoje lokalne parametre, ki jih z LogitBoostom pridobi vsak sin sam. Prednost tega pristopa je v tem, da so si sosednji listi drevesa v parametrih bolj podobni kot v primeru, da bi vsako logistično regresijo zgradili od začetka. Druga prednost je tudi manjša računska zahtevnost, saj je potrebno v $F_j(x)$ dodati manj funkcij. Ko je drevo zgrajeno, je potrebno uporabiti algoritem za reza-

nje drevesa, saj se drevo lahko preveč prilagodi učnim podatkom. Ker ima LMT v listih logistično regresijo, je verjetnost prevelike prilagoditve veliko večja kot pri navadnem odločitvenem drevesu. Za rezanje LMT uporablja algoritem, ki ga uporablja tudi CART in je opisan kasneje. Rezanje poteka hitro, saj so v vseh vozliščih že zgrajeni logistični modeli.

2.3.2 Izbira atributa v notranjem vozlišču

Za izbiro atributa, po katerem razdelimo učno množico v notranjih vozliščih, ponuja LMT dve metodi. Prva metoda je tista, ki jo uporablja tudi C4.5, torej uporaba informacijskega prispevka kot mere za ocenjevanje atributa. Druga metoda, ki jo lahko uporabimo, se osredotoča na spremenljivko z_{ij} (slika 2.4), ki predstavlja napako trenutno naučenega modela na učnih podatkih. Za razliko od slednje metode bo prva izbrala atribut, ki nam pove največ informacije, kar za drugo ne drži vedno, zato je tudi kot privzeta metoda izbrana prva. Posamezno vozlišče se deli, če ustreza naslednjim kriterijem:

- Vozlišče, ki se deli, mora vsebovati najmanj 15 učnih primerov. To je več, kot je običajno pri navadnih odločitvenih drevesih, vendar je v listih potrebno zgraditi logistično regresijo, ki za dober model potrebuje več primerov.

- Vozlišče se deli, če po deljenju nastaneta vsaj 2 podmnožici, od katerih ima vsaj ena več kot 2 učna primera. Ta pogoj nadgradi prejšnjega tako, da v primeru, ko vsi učni primeri pripadajo enemu razredu, množice ne delimo.

- Logistična regresija se zgradi v vozlišču, če to vsebuje najmanj 5 primerov, ker jih potrebujemo zaradi prečnega preverjanja pri določanju števila iteracij LogitBoosta.

2.3.3 Izbira atributov v logističnem modelu

Kot že omenjeno, LMT za izdelavo logistične regresije uporablja algoritem LogitBoost. Preden se LogitBoost začne izvajati, je potrebno določiti število iteracij LogitBoosta (M na sliki 2.4). To lahko storimo na več načinov. Prvi način je, da kot parameter algoritmu podamo vnaprej določeno število iteracij, ki naj se izvedejo. To ni priporočljivo, saj lahko izberemo premajhno število iteracij in dobimo posledično slabši model. Drugi način je, da to storimo s pomočjo prečnega preverjanja. To storimo tako, da podatke razdelimo 5-krat na učno in testno množico in nad vsako učno množico izvedemo vnaprej določeno število iteracij LogitBoosta. Po petih testiranjih izberemo število iteracij, kjer je v povprečju najmanjša napaka na testnih podatkih. Ker je potrebno to storiti v vsakem vozlišču, preden se zgradi model logistične regresije, se izvajanje algoritma lahko zelo podaljša, zato lahko uporabimo tretjo rešitev. Do te so avtorji LMT algoritma prišli tako, da so poizkušali izračunati število iteracij LogitBoosta samo v korenskem vozlišču in nato uporabiti vedno isto število iteracij. Eksperimenti, ki so jih naredili avtorji algoritma LMT, so pokazali, da se klasifikacijska točnost ne zmanjša preveč, močno pa se zmanjša čas, potreben za učenje. Atributi se izberejo samodejno s pomočjo LogitBoosta, kot je opisano v poglavju 2.2.3. Dodajanje novih atributov poteka, dokler se napaka modela ne zmanjšuje več.

2.3.4 Rezanje drevesa

Tako kot pri navadnih odločitvenih drevesih je tudi pri LMT potrebno drevo porezati, saj se velika drevesa lahko preveč prilagodijo učnim podatkom. Pri rezanju navadnih odločitvenih dreves ponavadi kaznujemo velikost drevesa in spodbujamo klasifikacijsko točnost, zato poizkušamo poiskati nekakšno ravnovesje med tema dvema količinama. Pri drevesih z logistično regresijo v listih pa je potrebno paziti, saj je logistični model v listih veliko bolj kompleksen kot navaden list z oznako razreda. Pri LMT se velikokrat zgodi, da najboljše rezultate dobimo, če drevo porežemo do korenskega vozlišča, kar se

skoraj nikoli ne zgodi pri navadnem odločitvenem drevesu, ki bi v tem primeru postal večinski klasifikator. Algoritmi za rezanje navadnih odločitvenih dreves se lahko uporabijo tudi za drevesa LMT, ampak le v primeru, da se v sinovih vozlišča upoštevajo parametri očeta, kar za LMT drži. V primeru, da bi v vsakem vozlišču zgradili logistično regresijo od začetka, trditev da je manjše drevo manj kompleksno ne drži vedno, saj lahko nižja vozlišča ali listi vsebujejo manj parametrov kot vozlišča višje v drevesu. LMT za rezanje uporablja algoritem, ki ga uporablja CART [1].

Tako kot algoritem rezanja pri M5', tudi ta upošteva napako pri učenju in velikost drevesa, poleg tega pa definira še parameter α, ki je odvisen od naše množice podatkov. Parameter α izračunamo s pomočjo prečnega preverjanja. LMT uporablja 5-kratno prečno preverjanje, kar pomeni, da je potrebno pred gradnjo končnega drevesa zgraditi pet dreves, s pomočjo katerih nato izračunamo ta parameter in na koncu še drevo na vseh učnih podatkih. Čeprav to zelo poveča računsko zahtevnost algoritma, je rezanje drevesa zelo pomembno, saj tako izboljšamo točnost modela. Označimo začetno neporezano drevo s T_{max}, množico listov drevesa T s $\tilde{T}$ in napako pri učenju z $R(T)$. Cilj algoritma je zmanjšati vrednost naslednje enačbe:

$$R_\alpha = R(T) + \alpha|\tilde{T}| \tag{2.10}$$

Na velikost R_α vpliva napaka pri učenju in število listov, pomnoženo s parametrom α. V primeru, da je α enako 0, se velikost drevesa ne kaznuje in bo algoritem pri rezanju gledal samo napako pri učenju. V nasprotnem primeru, če je parameter enak 1, kar je tudi največja vrednost, ki jo lahko zasede, bo drevo porezano do korenskega vozlišča. Začnemo z drevesom T_{max}. V vsakem listu izračunamo točnost poddrevesa in najprej porežemo poddrevesa, ki ne pripomorejo k točnosti drevesa. Nato sledi rezanje slabih poddreves. V določenem vozlišču t postane zamenjava poddrevesa T_t z listom $\tilde{t}$ smiselna, če je napaka poddrevesa $R_\alpha(T_t)$ večja ali enaka $R_\alpha(\tilde{t})$, ali ko je parameter α večji od α_t:

$$\alpha_t = \frac{R(\tilde{t}) - R(T_t)}{|\tilde{T_t}| - 1}, \tag{2.11}$$

To pomeni, da rešimo enačbo 2.11 v vsakem listu in odrežemo poddrevo v vozlišču, kjer je ta vrednost najmanjša. To ponavljamo, dokler obstajajo vozlišča, katerih α_t je večji od α, ali dosežemo korensko vozlišče. Po vsakem rezanju je treba vrednosti α_t ponovno izračunati, saj se točnosti poddreves spremenijo.

2.3.5 Izboljšave algoritma

V tem razdelku so opisane izboljšave algoritma, ki so jih avtorji algoritma implementirali v Weko. Gre za izboljšave, s katerimi so dosegli hitrejše izvajanje algoritma in se pri tem bistveno ne zmanjša klasifikacijska točnost modela. Prva takšna izboljšava je že opisana v razdelku 2.3.3. Gre za izračun iteracij LogitBoosta samo v korenskem vozlišču, saj je prečno preverjanje za izbiro optimalnega števila iteracij lahko precej računsko zahtevno. Druga izboljšava je bila tudi narejena na izračunu števila iteracij algoritma LogitBoost. Pogosto se zgodi, da je število potrebnih iteracij majhno, pri izračunu pa se izvede vseh 200 iteracij. Ko izvajamo iteracije LogitBoosta, napaka modela najprej pada, nato pa začne naraščati zaradi prevelikega prilagajanja učnim podatkom. Da bi se algoritem izvedel bolj hitro, lahko določimo število iteracij, v katerih se opazuje spremembo napake. Če se napaka ne zmanjša v določenem številu iteracij, lahko izvajanje LogitBoosta prekinemo, saj se napaka zelo verjetno ne bo več zmanjšala in bo celo začela naraščati. Ta izboljšava zelo poveča hitrost izdelave modela na podatkih, kjer je optimalno število iteracij majhno.

Dve dodatni izboljšavi sta bili dodani v algoritem kasneje in sta opisani v [9]. Tudi ti izboljšavi se nanašata na algoritem LogitBoost. Pri prvi, ki se imejuje *Weight Trimming* (WT), gre za to, da pri gradnji modela v LogitBoostu upoštevamo samo primere, katerih uteži nosijo $100 \times (1 - \beta)\%$ vseh uteži. Primeri z majhnimi utežmi se torej ne upoštevajo pri gradnji logistične regresije. V začetku so sicer vse uteži enake, ampak se med izvajanjem algoritma LogitBoost uteži določenih primerov močno zmanjšajo. Izkaže se, da je takih primerov lahko zelo veliko, zato se vsaka naslednja iteracija LogitBo-

osta lahko izvede hitreje od prejšnje. Testi so pokazali, da se klasifikacijska točnost modelov, zgrajenih na tak način, ne poslabša preveč.

Pri zadnji izboljšavi gre za uporabo informacijskega kriterija Akaike (AIC), s pomočjo katerega ni potrebno računati števila iteracij LogitBoosta s prečnim preverjanjem. Prav tako niso več potrebne omejitve za maksimalno število iteracij, ki je bilo prej 200. AIC je definiran kot:

$$AIC = -\frac{2}{N} loglik + 2\frac{d}{N} \tag{2.12}$$

kjer je N število učnih primerov, d število parametrov in $loglik$ funkcija največjega verjetja, ki jo uporablja LogitBoost pri izbiri atributov. Za parameter d je bilo določeno število iteracij LogitBoosta. Drugi člen izraza se tako povečuje s povečevanjem iteracij, medtem ko se prvi člen zmanjšuje, saj se logistični model z vsakim parametrom bolje prilega podatkom in je posledično manjša napaka na učnih podatkih. Optimalno število iteracij je torej tisto, ki najbolj zmanjša vrednost AIC. Postopek je torej podoben kot pri iskanju optimalnega števila iteracij s prečnim preverjanjem, ampak to storimo samo enkrat. Omejitev največjega števila iteracij je še vedno 200 in v primeru, da se AIC po 50 iteracijah več ne zmanjša, se ustavimo. Ko so avtorji algoritem testirali, so ugotovili, da se vrednost AIC, po doseženem minimumu ne zmanjšuje več, zato ni več potrebno določiti največjega števila iteracij, saj lahko algoritem zaustavimo takoj, ko dosežemo prvi minimum. Metodo so poimenovali FAM. Podobno kot pri prejšnji izboljšavi, ko upoštevamo samo primere z največjimi utežmi, se tudi tukaj hitrost zmanjšuje nižje, ko smo v drevesu. Tako lahko to metodo uporabimo v vsakem vozlišču drevesa in s tem še bolj pospešimo gradnjo, saj bo v nižjih vozliščih potrebno manj iteracij kot v višjih. Rezultati so pokazali, da lahko z zadnjima dvema izboljšavama zelo pohitrimo izvajanje algoritma LMT. V nekaterih primerih se je hitrost povečala kar 55-krat, medtem ko se klasifikacijka točnost ni preveč zmanjšala.

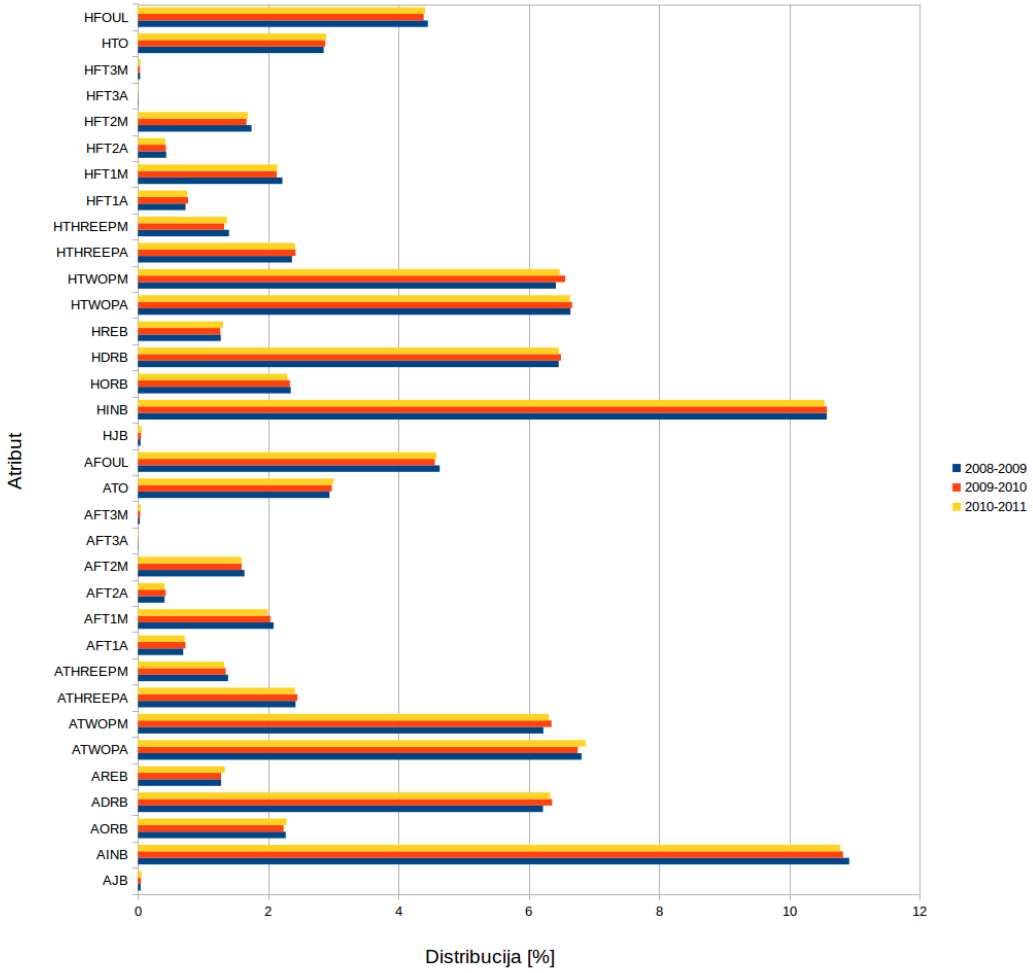
2.4 Učni podatki

V tem razdelku so opisani učni podatki s športno domeno, na katerih smo zgradili modele z našo implementacijo algoritma. Na razpolago smo imeli tri učne množice s košarkarsko tematiko. Vsaka učna množica predstavlja eno sezono košarkarske lige NBA. Podatkovne množice so zelo velike. Vsaka izmed njih namreč vsebuje več kot 500.000 učnih primerov, kar je prikazano v tabeli 2.1.

Vsak učni primer je sestavljen iz 19 numeričnih atributov, enega nominalnega atributa, ki ima 34 različnih vrednosti, in razreda, ki je prav tako nominalen in zaseda 34 različnih vrednosti. 16 numeričnih atributov opisuje statistične lastnosti domače in tuje ekipe, kot je na primer odstotek uspešnih metov gostujoče ekipe. Preostali trije numerični atributi so podatek o četrtini tekme, čas do konca tekme in trenutna razlika v točkah. Zadnji atribut je nominalen in nam pove prejšnji dogodek na tekmi. Razredni atribut nam pove naslednji dogodek na tekmi, torej dogodek, ki ga poskušamo napovedati. Dogodek je lahko uspešen met za 3 točke domače ekipe, prekršek gostujoče ekipe nad domačo ekip, ipd. Distribucije razredov so prikazane na sliki 2.5. Problem pri analizi teh podatkov je njihova velikost. Poleg zelo velike količine učnih primerov vsebujejo tudi zelo veliko atributov. Dodatne težave povzroči nominalni atribut, ki ima kar 34 različnih vrednosti, saj ga je potrebno pri izdelavi logističnega modela pretvoriti v binarno obliko. To povzroči večjo porabo pomnilnika, ki ga potrebuje algoritem. Zaradi razreda, ki lahko zaseda 34 različnih vrednosti, ima veliko dela tudi algoritem LogitBoost, saj mora za vsak učni primer izračunati verjetnosti za vsak razred posebej.

sezona	število učnih primerov
2008-2009	567967
2009-2010	569185
2010-2011	566393

Tabela 2.1: Število učnih primerov v posamezni podatkovni množici



Slika 2.5: Distribucija razredov za vse učne množice

Poglavje 3

Implementacija

V tem poglavju je opisana naša implementacija algoritma LMT. Algoritem je bil implementiran v programskem jeziku C++. Avtorji algoritma so LMT implementirali v sistem Weka v programskem jeziku Java. Implementacija v Weki je zelo neučinkovita, kar prikazujejo tudi rezultati v poglavju 4.1, vendar je bila kljub temu zelo pomembna oporna točka za našo implementacijo. Glavni namen naše implementacije je bila večja pomnilniška učinkovitost, saj je bila to edina možnost, da zgradimo model na naših podatkih s športno domeno. Implementirali smo tudi nov način za izbiro atributa, ki ga algoritem uporablja za delitev drevesa v vozliščih.

3.1 Weka

Weka je odprtokodna programska oprema, namenjena strojnemu učenju, ki je bila razvita na univerzi Waikato na Novi Zelandiji. Projekt se je od začetka leta 1993 zelo razširil in je aktiven še danes. Sprva je bila Weka napisana v različnih programskih jezikih, kasneje pa je bila v celoti prepisana v Javo.

Weka ponuja veliko množico algoritmov za strojno učenje, tako nadzorovano kot nenadzorovano. Poleg tega ima tudi nekaj orodij za vizualizacijo podatkov. Ima tudi grafični vmesnik, ki omogoča, da lahko rezultate različnih algoritmov primerjamo med seboj. Tako nam ni potrebno algoritmov poga-

njati iz ukazne vrstice.

Ena izmed lastnosti sistema je tudi lasten format zapisa podatkov. Podatke preberemo iz datoteke tipa ARFF, ki je v bistvu CSV datoteka, ki ima na začetku dodan opis atributov. Pri naši implementaciji je bil tudi uporabljen ta format, saj so tudi podatkovne množice s športno domeno shranjene v tem formatu.

3.2 Implementacija

Algoritem LMT je zelo zahteven tako v količini časa kot tudi v količini pomnilnika, ki je potreben za gradnjo modela. Na začetku smo naredili nekaj testov algoritma v sistemu Weka, ampak se je že na manjših zbirkah podatkov pokazala pomanjkljivost te implementacije. Predvsem je težave povzročala velika poraba glavnega pomnilnika. Ko smo poizkusili zgraditi model na košarkarski množici podatkov, ki vsebuje več kot pol milijona učnih podatkov, je bilo očitno, da algoritem LMT, ki je implementiran v Weki, ni sposoben zgraditi modela na običajnem računalniku. Za implementacijo algoritma LMT smo izbrali programski jezik C++. V tem razdelku je opisana naša implementacija algoritma in primerjava z implementacijo v javi. Opisane so predvsem podatkovne strukture, zaradi katerih je naša implementacija bolj učinkovita. Implementirali smo osnovni algoritem LMT brez WT in AIC izboljšav, uporabili pa smo izboljšavo, ki na začetku gradnje drevesa s prečnim preverjanjem oceni optimalno število iteracij algogirtma LogitBoost. Poleg samega algoritma je bilo potrebno implementirati celoten sistem za strojno učenje, saj je sistemov za strojno učenje v programskem jeziku C++ zelo malo. Sistem je bil tako implementiran v celoti, od branja podatkov iz datoteke do samega ogrodja za testiranje modela.

3.2.1 Shranjevanje podatkov

Pri izbiri podatkove strukture za shranjevanje podatkov smo želeli doseči to, da bi podatki porabili čim manj prostora. V Weki so podatki shranjeni tako,

da je vsak učni primer svoj objekt. Stuktura in povezave med objekti so v Weki večje, predvsem zaradi tega, ker Weka omogoča več različnih opravil nad podatki, kot je naprimer vizualizacija. To je tudi glavni razlog za neučinkovitost implementacije v Weki.

Naša implementacija večine teh dodatnih atributov ne potrebuje, zato podatke shranjujemo v objekt tipa *vector*. Vsak element vektorja je kazalec na polje, ki vsebuje podatke za en učni primer. Bolj enostavna rešitev, ki bi porabila več pomnilniškega prostora, bi bila ta, da bi uporabili kar vektor vektorjev. Ravno zaradi varčevanja s pomnilnikom smo se odločili, da uporabimo bolj primitivno stukturo. Vsak vektor ima namreč nekaj dodatnih atributov, kot je na primer dolžina vektorja, ki pa nas ne zanima, saj imajo vsi učni primeri enako dolžino, ki jo poznamo vnaprej. Polja, na katere kažejo elementi vektorja, so polja spremenljivk tipa *double*, ki so decimalna števila z dvojno natančnostjo. Vse vrednosti v učni množici lahko namreč shranimo v tej obliki. Vrednosti nominalnih atributov, ki so zapisane z besedo, lahko preslikamo v številsko vrednost. Da ločimo med številskimi in nominalnimi atributi, imamo poleg vektorja s podatki tudi vektor, ki nam pove, kakšega tipa so posamezni atributi.

3.2.2 Gradnja drevesa

Ko imamo podatke shranjene v našo podatkovno strukturo, sledi gradnja odločitvenega drevesa in gradnja modelov logistične regresije v vozliščih. Postopek gradnje drevesa je rekurziven, kar pomeni, da se vsaka podmnožica podatkov lahko razdeli na manjše podmnožice, kjer se celoten postopek gradnje ponovi. Za namen gradnje drevesa uporabimo rekurzivno funkcijo *buildTree()*. Funkcija *buildTree()* najprej nastavi vrednosti potrebnih spremenljivk, pretvori nominalne atribute v binarne in nastavi matriko linearnih regresij. Če se nahajamo v korenskem vozlišču, se ustvari nova, prazna matrika, sicer se matrika prebere iz očeta tega vozlišča. Pri pretvarjanu podatkov v numerično obliko, ko nominalne atribute pretvorimo v binarno obliko, se ustvari kopija podatkov. Te podatke pred novim klicem funkcije *buildTree()*

izbrišemo. Tako sprostimo pomnilnik, ki ga zasedajo podatki, ki jih v nadaljevanju ne bomo več potrebovali.

```
void LMTNode::buildTree(Data &d, ...){
        // inicializiraj spremenljivke
        // pretvori podatke v numericno obliko
        // ustvari matriko linearnih regresij
        // LogitBoost
        // izbrisi numericne podatke
        // razdeli podatke na podmnozice
        // v vsakem sinu vozlisca poklici buildTree(...)
}
```

Funkcija buildTree()

Ko ustvarimo matriko linearnih regresij, sledi izdelava modela logistične regresije s pomočjo algoritma LogitBoost. Ta algoritem časovno zavzame največji del izvajanja. Poleg časovne zahtevnosti ima tudi relativno veliko pomnilniško zahtevnost. Na začetku izvajanja se namreč ustvarijo tri matrike z numeričnimi podatki, ki so shranjeni kot decimalna števila z dvojno natančnostjo. Matrika ima dimenzije število učnih primerov $\times$ število razredov, kar v primeru naših košarkarskih učnih množic pomeni 560 000 $\times$ 34. Števila v matriki morajo biti tipa *double*, saj dobimo v primeru enojne natačnosti ali *float* drugačne rezultate, ki so posledica napake zaradi premajhne natančnosti. Končni rezultat algoritma LogitBoost je posodobljena matrika linearnih regresij. Ta matrika predstavlja model logistične regresije, zato je prisotna v vsakem vozlišču drevesa. Dimenzije matrike so število razredov $\times$ število atributov v binarni obliki. V našem primeru je to 34 x 53, kar pomeni matriko z nekaj več kot 1800 celicami. Na prvi pogled se to ne zdi veliko, vendar se je treba zavedati, da so elementi matrike objekti, ki so v bistvu linearne regresije. Poleg tega je med gradnjo drevesa takšna matrika prisotna v vsakem vozlišču, ki pa jih je pri privzetih nastavitvah algoritma nekaj čez 91 000. Večina vozlišč je seveda kasneje, v zadnjem delu algoritma porezana, vendar mora biti kljub temu v vsakem vozlišču prisoten

logističen model, saj ne vemo, katera vozlišča bodo na koncu ostala. Izkazalo se je, da ravno vse te matrike linearnih regresij porabijo največ pomnilniškega prostora. Število linearnih regresij, ki jih mora program hraniti, dokler se ne začne rezanje drevesa, je zaradi velikosti matrike (1800 celic) in velikosti drevesa (91 000 vozlišč) zelo veliko. Če ti dve števili zmnožimo, ugotovimo, da je linearnih regresij več kot 160 milijonov. Linearno regresijo smo na začetku definirali kot razred s konstruktorjem in destruktorjem, ki je vseboval eno celoštevilko spremenljivko tipa *int* in dve decimalni spremenljivki tipa *double*. Med testiranjem smo opazili, da algoritem za izdelavo modela še vedno porabi preveč pomnilnika, zato smo linearno regresijo predefinirali v C-jevsko strukturo, spremenljivki, ki hranita naklon in presek, pa smo spremenili v tip $float$, ki zasede polovico manj pomnilnika kot *double*. Vrednosti teh spremenljivk zasedajo dovolj velike vrednosti, da natančnost shranjevanja nima vpliva na rezultat. Velikost ene linearne regresije v pomnilniku je torej 12 bajtov. Če to zmnožimo s številom vseh linearnih regresij v drevesu, dobimo velikost logističnih modelov v vseh vozliščih, ki znaša približno 2 GB.

Zadnji del funkcije *buildTree()* skrbi za razdelitev podatkov na podmnožice. Poleg osnovnega algoritma, ki za izbiro atributa, po katerem razdelimo množico, uporablja mero za količino informacije, je bila implementirana še druga metoda, s katero smo poizkušali na drugačen način razdeliti množico podatkov. Ta metoda je opisana v razdelku 3.2.3. Ko razdelimo podatke na podmnožice, se za vsako izmed njih ustvari novo vozlišče, na katerem se kliče metoda *buildTree()*. Ker gre za rekurzivni klic metode, v katero se kot parameter poda podmnožico podatkov, je zelo smiselno, da podatke podamo kot referenco. S tem se izognemo dodatnemu kopiranju že tako velike množice podatkov. Z vsakim dodatnim polnim nivojem drevesa bi se količina povečala za eno celotno množico podatkov iz korenskega vozlišča. Pri velikih podatkovnih množicah, kot je na primer naša množica košarkarskih podatkov, je lahko teh nivojev, veliko preden se drevo poreže.

Pri implementaciji rezanja drevesa nismo dosegli velikih sprememb glede učinkovitosti, saj sam postopek ni pomnilniško zahteven. Dodana je bila

možnost, da se rezanje izklopi in možnost, da ročno nastavimo največjo dovoljeno globino drevesa. Ti dve dodatni nastavitvi algoritma sta bili implementirani zaradi naše metode za izbiro atributov, ki je opisana v naslednjem razdelku.

3.2.3 Izbira attributov z metodo bootstrapping

Pri navadnih odločitvenih drevesih se običajno za izbiro atributa, ki bi najbolje razdelil podatke v vozlišču, uporablja razmerje informacijskega prispevka. Uporablja ga tudi algoritem LMT. V našo implementacijo smo dodali drugačen način za izbiro atributa pri gradnji drevesa, ki za razliko od obstoječega načina ne uporablja količine informacije kot mere za ocenjevanje kakovosti atributa.

Kot je opisano v prejšnjih poglavjih, se v vsakem vozlišču drevesa zgradi model logistične regresije, ki je predstavljen v obliki matrike, katere elementi so linearne regresije. Vsaka linearna regresija vsebuje tri podatke. Presečišče z ordinatno osjo, naklon in atribut, kateremu linearna regresija pripada. Nov način izbire atributov deluje tako, da poleg prvega modela logistične regresije zgradi še tri dodatne modele z metodo *bootstrapping*. Pri bootstrappingu gre za to, da iz obstoječe množice podatkov, ki jo v našem primeru predstavljajo vsi podatki v vozlišču, naključno s ponavljanjem vlečemo podatke. Nova množica ima enako število učnih primerov, vendar je zaradi naključne izbire s ponavljanjem iz začetne množice v povprečju izbranih 63.2 % različnih učnih primerov, kar pomeni, da se nekateri podatki ponovijo. Na takšen način dobimo tri različne množice podatkov, na katerih zgradimo tri modele logističnih regresij. Za vsak model nato naredimo matriko koeficientov modela.

Koeficienti so bili omenjeni v razdelku 2.2.1 kot vektor β v enačbi logistične funkcije 2.4. V primeru večrazrednih podatkov imamo za vsak razred svojo funkcijo, oziroma za vsak razred svoj vektor koeficientov, ki jih lahko skupaj združimo v matriko koeficientov. Koeficiente izračunamo s pomočjo matrike linearnih regresij. Vrstice matrike koeficientov prestavljajo razrede,

stolpci pa atribute. Vrednosti v posamezni vrstici torej predstavljajo koeficiente za določen razred. Prvi koeficient je vsota presečišč, kjer linearne regresije, ki pripadajo temu razredu, presečejo ordinatno os. Vrednosti ostalih koeficientov dobimo tako, da seštejemo vse naklone, ki pripadajo linearni regresiji za par razred - atribut. Vse vrednosti na koncu zmnožimo z ulomkom $\frac{st.razredov-1}{st.razredov}$.

Ko imamo pripravljene vse tri matrike koeficientov, najprej odstranimo prvi stolpec v vseh matrikah, saj ne pripada nobenemu atributu. Prvi stolpec vsebuje namreč vsote presečišč z ordinato. Med istoležnimi elementi matrik nato izračunamo razdalje in jih seštejemo. Tako dobimo matriko razdalj med koeficienti. Razdalje med koeficienti, ki pripadajo istemu atributu, nato seštejemo in delimo s številom razredov. Tako dobimo za vsak atribut svojo vrednost, ki je povprečna razdalja med njegovimi koeficienti. Atribut z največjim povprečnim odstopanjem je tisti, ki ga izberemo za delitev. Vrednosti povprečij nato normaliziramo, tako da jih delimo s povprečjem izbranega atributa. Tako ima izbran atribut normalizirano vrednost 1, vrednosti vseh ostalih atributov pa zasedajo vrednosti med 0 in 1. Te vrednosti potrebujemo pri preverjanju dodatnega pogoja, ki preverja, za koliko izbrani atribut odstopa od drugouvrščenega. Če je razlika med povprečnimi vrednostimi atributov premajhna, pomeni, da atribut ni bistveno boljši kandidat za delitev od ostalih. Zaradi tega je bil implementiran dodaten pogoj, kateremu kot parameter podamo najmanjšo dovoljeno razliko med normaliziranima vrednostima izbranega in drugouvrščenega atributa. Če je razlika med njima premajhna, se gradnja drevesa v tem vozlišču ustavi.

Za lažje razumevanje postopka je narejen enostaven primer izračuna koeficientov in izbire atributa. Podatkovna množica v našem primeru vsebuje 4 atribute in 3 razrede. V tabeli 3.1 so prikazane linearne regresije za prvi model. Tabeli linearnih regresij za preostala dva modela imata enake dimenzije. Razlikujeta se le v vrednostih, zato ju nismo vključili v besedilo.

V tabeli 3.2 so koeficienti, ki smo jih izračunali iz podatkov v tabeli 3.1. Izračun preseka z ordinato za razred A naredimo tako, da enostavno

seštejemo vse preseke, ki pripadajo razredu A v tabeli linearnih regresij. Izračun koeficientov pri atributih je nekoliko drugačen. Za atribute nas zanimajo nakloni v tabeli 3.1. Sešteti je potrebno vse naklone, ki pripadajo paru atribut - razred. Seštevanju sledi množenje z vrednostjo $\frac{st.razredov-1}{st.razredov}$, ki je v našem primeru $\frac{2}{3}$. Vrednost koeficienta za par razred A - atribut 2 smo dobili tako, da smo vsoto naklonov, ki je enaka 1.6163, pomnožili z $\frac{2}{3}$ in dobili 1.07753.

Enako storimo tudi za preostala dva modela in tako dobimo matriko vseh koeficientov, ki je prikazana s tabelo 3.3. Ker nas zanimajo samo koeficienti atributov, smo iz tabele odstranili preseke. Tako smo dobili enotno tabelo za vse tri modele. V naslednjem koraku izračunamo razdalje med pari razred - atribut za vse modele po enačbi:

$$R(r,a) = \sum_{i=0}^{st.modelov} \sum_{j=1}^{st.modelov} |K(r,a,i) - K(r,a,j)| \tag{3.1}$$

V enačbi 3.1 predstavlja R(r,a) razdaljo med koeficienti, ki pripadajo paru razred - atribut. K(r,a,i) in K(r,a,j) predstavljata vrednost koeficienta za par razred - atribut pri modelu i in j. Izračunamo torej vsoto razdalj med koeficienti in tako dobimo zgornji del tabele 3.4. Razdalje za posamezne atribute nato povprečimo s številom razredov in tako dobimo srednji del tabele. Pri povprečnih vrednostih razdalj lahko opazimo, da koeficienti atributa 2 najbolj odstopajo. Za deljenje drevesa bi zato v tem vozlišču izbrali atribut 2. Pred tem je potrebno preveriti še dodaten pogoj. Povprečna razdalja med koeficienti atributa 2 mora biti za določen odstotek večja od drugouvrščenega atributa, ki je v našem primeru atribut 3. Povprečne vrednosti je zato potrebno normalizirati, tako da jih delimo z povprečno vrednostjo, ki pripada atributu 2. Vrednosti lahko vidimo v zadnjem delu tabele 3.4. Razlika med normaliziranima vrednostima za atribut 2 in atribut 3 je torej 0.042263. V primeru, da na začetku izvajanja algoritma nastavimo parameter mejne vrednosti, ki ga bomo poimenovali T, na manjšo vrednost, kot je ta razlika, se bo drevo v tem vozlišču razdelilo na dve poddrevesi po atributu 2. Nasprotno

	presek z ordinato	naklon	atribut
razred A	0	0	1
	-5.10063	1.6163	2
	28.3803	-10.616	3
	9.03815	-13.2926	4
razred B	-27.6588	4.59006	1
	-1.26344	0.785913	2
	10.4807	-2.14981	3
	1.43999	-0.913929	4
razred C	7.46505	-1.22583	1
	3.8492	-1.38452	2
	-56.2579 1	1.5445	3
	-33.4264 2	1.3442	4

Tabela 3.1: Linearne regresije prvega modela

	presek z ordinato	atribut 1	atribut 2	atribut 3	atribut 4
razred A	32.31782	0	1.07753	-7.07735	-8.86176
razred B	-17.00155	3.06004	0.523942	-1.4332	-0.609286
razred C	-78.37005	-0.817223	-0.923014	7.69632	14.2295

Tabela 3.2: Izračunani koeficienti za prvi model

se deljenje drevesa v tem vozlišču zaključi, če je parameter T večji od razlike normaliziranih vrednosti.

	razred	atribut 1	atribut 2	atribut 3	atribut 4
model 1	A	0	1.07753	-7.07735	-8.86176
	B	3.06004	0.523942	-1.4332	-0.609286
	C	-0.817223	-0.923014	7.69632	14.2295
model 2	A	0	1.07753	-7.07735	-8.86176
	B	3.06004	0.523942	-1.4332	-0.609286
	C	-0.817223	-0.923014	7.69632	14.2295
model 3	A	0	0	-9.62683	-7.25466
	B	1.58599	-0.965226	-0.902842	-1.3842
	C	-0.380271	-4.60218	4.79426	14.0822

Tabela 3.3: Koeficienti vseh treh modelov brez preseka z ordinato

razred	atribut 1	atribut 2	atribut 3	atribut 4
A	0	2.15506	5.09895	3.21419
B	2.94811	2.97834	1.06072	1.54982
C	0.873903	7.35833	5.80411	0.294482
avg	1.274	4.16391	3.98793	1.68617
norm	0.305964	1	0.957737	0.404948

Tabela 3.4: Razdalje med koeficienti

Poglavje 4

Rezultati

V tem poglavju so predstavljeni rezultati testov naše implementacije. V prvem podpoglavju je prikazana primerjava naše imeplementacije in implementacije v Weki, v drugem podpoglavju pa smo se posvetili predvsem rezultatom testiranja z našo metodo izbire atributov.

Glavni testi so bili narejeni na košarkarskih podatkovnih množicah, naredili pa smo tudi nekaj testov na podatkovnih množicah iz repozitorija podatkovnih množic za strojno učenje Univerze v Kaliforniji ali krajše UCI (University of California, Irvine). Testi na podatkih UCI so bili narejeni predvsem zato, da smo preverili, kako se različne začetne nastavitve algoritma obnesejo na različnih podatkih. Za razliko od podatkovnih množic iz repozitorija UCI, ki imajo manjše število učnih primerov, imajo košarkarski podatki zelo veliko učnih primerov (več kot 500 000), poleg tega pa imajo tudi veliko število atributov.

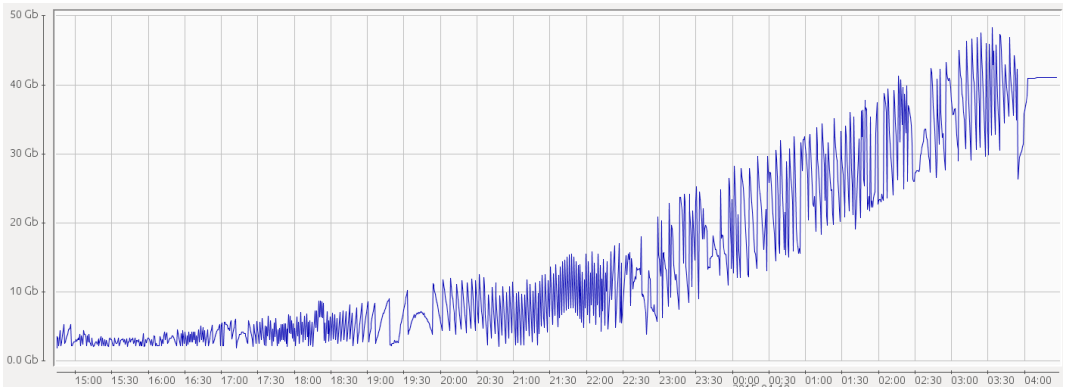
4.1 Učinkovitost v primerjavi z Weko

Pri primerjavi naše implementacije z Weko nas je zanimala predvsem razlika v porabi časa in pomnilnika. Naredili smo test na košarkarski podatkovni množici za sezono 2008/2009. Zanimala nas je predvsem največja poraba pomnilnika, saj je od te količine odvisno, ali bo nek računalnik sposoben zgra-

diti model v razumnem času. Največjo porabo pomnilnika algoritem doseže v trenutku, ko zgradi celotno drevo in preden začne z rezanjem drevesa. Ker nas torej ne zanima poraba pomnilnika po rezanju drevesa, smo obe implementaciji testirali brez rezanja drevesa. Tako smo prihranili čas za testiranje, ne da bi vplivali na rezultate testa. Zaradi velike porabe pomnilnika Weke smo namreč za testiranje potrebovali amazonov strežnik r3.2xlarge, ki ponuja 61 GB glavnega pomnilnika. Da so bili časovni rezultati primerljivi, smo morali tudi našo implementacijo algoritma testirati na amazonovem strežniku. Tako sta bili obe implementaicji testirani na enakem procesorju in smo lahko naredili tudi primerjavo časa, ki je potreben za izvajanje.

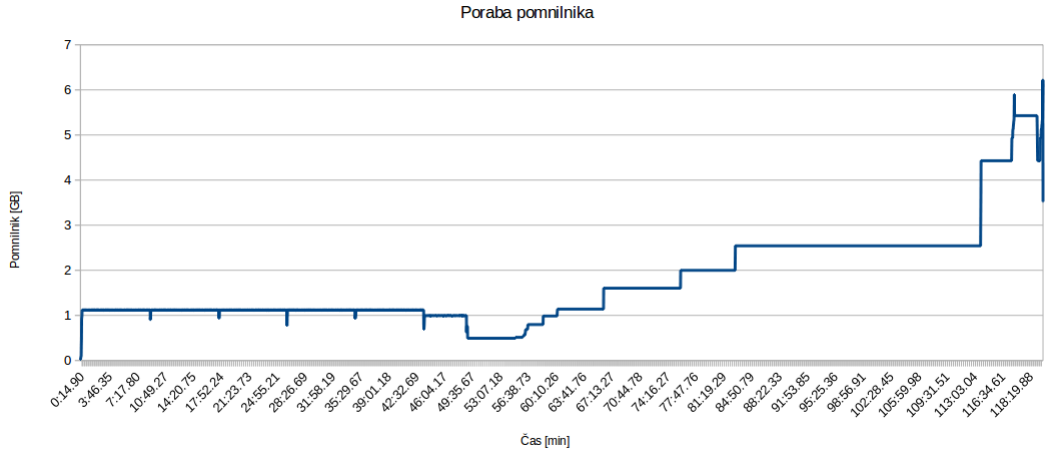
Na sliki 4.1 imamo graf porabe pomnilnika za algoritem v Weki. Graf je bil narejen s pomočjo orodja Java Monitoring & Management Console. Iz grafa lahko razberemo, da je program dosegel najvišjo poraba pomnilnika pri približno 48 GB. Poraba kasneje pade na približno 27 GB. Predvidevamo lahko, da se to zgodi, ko se vsi rekurzivni klici pri gradnji drevesa zaključijo in se sprosti prostor, ki so ga zasedale spremenljivke. Velikost samega modela je tako 27 GB, vendar za izdelavo tega modela potrebujemo vseh 48 GB. Na koncu lahko opazimo, da se poraba pomnilnika ponovno dvigne nad 27 GB, kar je posledica branja in testiranja na ločeni testni množici podatkov. Ker nas zanima predvsem čas gradnje drevesa, pri časovni primerjavi tega dela grafa ne bomo upoštevali. Glede časa za izdelavo modela lahko na grafu opazimo, da se je gradnja začela približno ob 14.45 in končala ob 3.55 naslednji dan, kar pomeni, da je algoritem za gradnjo drevesa potreboval 13 ur in 10 minut.

Poraba pomnilnika za našo implementacijo algoritma je prikazana na grafu na sliki 4.2. Takoj lahko opazimo, da je graf bolj gladek. Če bi sliko povečali, bi lahko opazili rahlo nazobčanost, ki je posledica LogitBoosta. Na začetku vsake iteracije algoritma LogitBoost se namreč ustvarijo matrike, ki zasedejo določen del pomnilnika in se pred koncem iteracije odstranijo iz pomnilnika. Bolj kot to nas seveda zanima učinkovitost implementacije. Opazimo lahko, da naša implementacija za gradnjo modela potrebuje nekaj



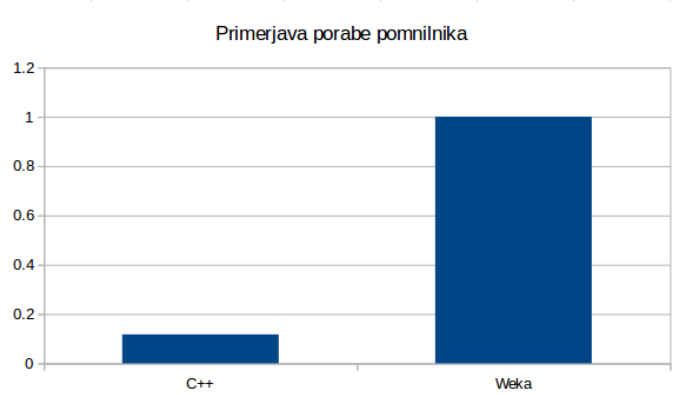
Slika 4.1: Poraba pomnilnika - Weka

več kot 5 GB pomnilnika. Časovno je algoritem za rešitev problema potreboval manj kot 2 uri časa.

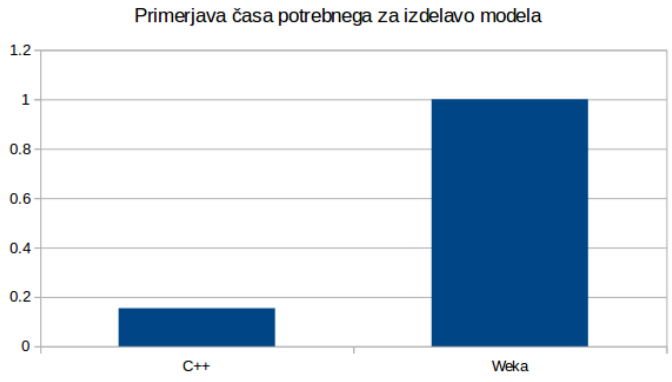


Slika 4.2: Poraba pomnilnika - naša implementacija

S stolpičnimi diagrami 4.3 in 4.4 je prikazano razmerje porabe pomnilnika in časa za obe implementaciji. Po porabi pomnilnika je naša implementacija približno 9-krat bolj učinkovita, po času, potrebnem za izdelavo modela, pa približno 6.5-krat bolj učinkovita.



Slika 4.3: Primerjava porabe pomnilnika



Slika 4.4: Primerjava časa, potrebnega za izdelavo modela

4.2 Bootstrapping

Našo metodo, ki za izbiro atributa uporablja *bootstrapping*, smo primerjali z osnovno implementacijo algoritma, ki za izbiro atributa uporablja razmerje informacijskega prispevka. Nekaj osnovnih testov smo naredili na podatkovnih množicah iz repozitorija UCI, glavni testi pa so bili narejeni na podatkovnih množicah s košarkarskimi podatki.

Na voljo smo imeli tri množice košarkarskih podatkov, vsako za svojo sezono ameriške košarkarske lige NBA. Vse tri množice podatkov vsebujejo približno 560 000 učnih primerov. Testi so bili narejeni tako, da smo na eni množici zgradili model in ga testirali z drugo. Vsaka podatkovna množica je bila enkrat uporabljena za učenje in enkrat za testiranje. Za model, zgrajen na podatkih za sezono 2008/2009 smo tako za testiranje vedno uporabili sezono 2009/2010, za testiranje sezone 2009/2010 smo uporabili sezono 2010/2011 in model za sezono 2010/2011 smo testirali s podatki sezone 2008/2009. Pri podatkih z repozitorija UCI smo za testiranje uporabili 10-kratno prečno preverjanje, kar pomeni da smo 10-krat uporabili 90 % podatkov za izdelavo modela in preostalih 10 % podatkov za testiranje.

Pri testiranju modelov, zgrajenih na športni domeni, smo poleg klasifikacijske točnosti izračunali tudi Brierjevo mero. Klasifikacijska točnost nam pove odstotek testnih primerov, ki so bili klasificirani v pravilen razred. Pomanjkljivost te mere je v tem, da ne upošteva verjetnosti, da bo določen primer pripadal razredu. Brierjeva mera te verjetnosti upošteva in je izračunana z enačbo (4.1). V enačbi je N je število testnih primerov, R je število razredov, f_{ti} je verjetnost, da primer pripada trenutnemu razredu in o_{ti} predstavlja pravilni razred, ki mu primer pripada.

$$BS = \frac{1}{N} \sum_{t=1}^{N} \sum_{i=1}^{R} (f_{ti} - o_{ti})^2 \tag{4.1}$$

Vrednosti, ki jih lahko zaseda f_{ti}, so med 0 in 1, medtem ko o_{ti} zaseda samo vrednost 0, če primer ne pripada trenutnemu razredu, ali vrednost 1,

če primer pripada razredu. Pri Brierjevi meri je pomembno upoštevati, da vrednost 0 pomeni najboljšo možno vrednost in 2 najslabšo vrednost. Manjša kot je Brierjeva mera, boljši je klasifikator.

Tabeli 4.1 in 4.2 prikazujeta rezultate testov na množicah podatkov z repozitorija UCI. Opazimo lahko, da so bila skoraj vsa drevesa, zgrajena z metodo *bootstrapping*, porezana do korenskega vozlišča. Druga lastnost, ki jo lahko opazimo, je ta, da se klasifikacijske točnosti na določenih podatkovnih množicah razlikujejo, če primerjamo obe metodi izbire atributa. Ta razlika je posledica tega, da so bili podatki naključno premešani pred začetkom gradnje modela. To pomeni, da je razlika v klasifikacijski točnosti pri modelih z enim vozliščem nepomembna, saj so zgrajeni modeli identični, le podatki za testiranje so se razlikovali.

Algoritem smo na košarkarskih podatkih testirali z več različnimi parametri. Za osnovne teste smo uporabili razmerje informacijskega prispevka kot metodo za izbiro atributov. Rezultati so prikazani v tabeli 4.3. Vsi ostali testi so za izbiro atributa pri deljenu drevesa uporabili metodo *bootstrapping*. Pri prvi iteraciji testov z našo metodo smo parameter T, omenjen v poglavju 3.2.3, izklopili in pustili algoritmu, da sam poreže celotno drevo. Rezultati tega testa so predstavljeni v tabeli 4.4. Drevesa so bila v primerjavi z osnovnimi testi veliko manjša, zaradi česar smo naredili še 2 dodatni iteraciji testov. Izklopili smo rezanje drevesa in nastavili parameter T na vrednost 0.1 v prvem in 0.15 v drugem testu. V tabeli 4.5 opazimo, da velikosti dreves znotraj posameznih testov zelo odstopajo. Prav tako so velike razlike pri majhni spremembi parametra T za iste podatkovne množice. S tem testom smo odkrili eno slabost našega pristopa, kjer uporabljamo parameter T. Težava je v tem, da je razlika med razdaljama prvo- in drugouvrščenega atributa v korenskem vozlišču premajhna. Zato ima zgrajen model za sezono 2009/2010 pri vrednosti $T = 0.15$ samo eno vozlišče, pri vrednosti $T = 0.1$ pa 8909 vozlišč. Zgodi se namreč to, da je razlika med razdaljama atributov v sinovih korenskega vozlišča večja kot parameter T, vendar je v korenskem vozlišču razlika manjša od T. Posledično do deltive v korenskem vozlišču ne

podatk. množica	klasifikacijska točnost	čas [s]	število vozlišč
Balance-scale	89.92	0.149	9
Credit-g	75.9	0.959	1
diabetes	77.21	0.154	1
glass	69.16	0.158	17
Heart-statlog	82.59	0.055	1
ionosphere	93.16	0.276	3
iris	95.33	0.022	1
Kr-vs-kp	99.69	4.907	15
letter	92.04	370	69
lymph	85.81	0.102	1
segment	96.45	4.64	5
sonar	78.85	0.279	1
vehicle	82.98	1.54	3
vowel	94.24	5.31	9
zoo	92.08	0.117	1

Tabela 4.1: Rezultati testov na podatkovnih množicah UCI z razmerjem informacijskega prispevka

podatk. množica	klasifikacijska točnost	čas [s]	število vozlišč
Balance-scale	87.68	0.09	1
Credit-g	74.8	0.518	1
diabetes	77.08	0.098	1
glass	64.95	0.094	1
Heart-statlog	83.33	0.029	1
ionosphere	88.6	0.127	1
iris	96.66	0.023	1
Kr-vs-kp	96.99	4.81	1
letter	83.94	1076	77
lymph	84.46	0.076	1
segment	95.45	6.92	1
sonar	75	0.148	1
vehicle	77.06	0.847	1
vowel	84.34	3.89	1
zoo	93.07	0.107	1

Tabela 4.2: Rezultati testov na podatkovnih množicah UCI z metodo *bootstrapping*

sezona	klas. točnost	Brier	čas [s]	število vozlišč
2008-2009	53.165	0.569	31724	65
2009-2010	53.0211	0.5711	25341	85
2010-2011	53.0996	0.5697	25583	73

Tabela 4.3: Rezultati testov na košarkarskih podatkovnih množicah (razmerje informacijskega prispevka)

pride in dobimo drevo z enim vozliščem. Zaradi tega smo algoritmu dodali še dodaten parameter, ki določa največjo dovoljeno globino drevesa. Naredili smo še 4 teste, kjer smo izklopili parameter T in omejili velikost drevesa na 1, 2, 3 in 4 dodatne nivoje. Rezultate teh testov prikazuje tabela 4.6. Opazimo, da je klasifikacijska točnost modelov najboljša, ko imajo drevesa 9 vozlišč. Z naraščanjem drevesa se klasifikacijska točnost ponovno zmanjšuje. Opazimo tudi, da se Brierjeva mera zelo rahlo povečuje z naraščanjem dreves in je najboljša pri manjših drevesih.

Zelo zanimiva je tudi primerjava strukture drevesa, ki ga zgradi osnovni LMT in LMT z uporabo metode *bootstrapping*. Drevo, zgrajeno z osnovnim algoritmom LMT je v korenskem vozlišču razdeljeno na poddrevesa po nominalnem atributu. Drevesa, zgrajena z metodo *bootstrapping*, ki smo jim omejili globino, tega atributa niso uporabila. Kljub temu se klasifikacijska točnost dreves, zgrajenih z metodo *bootstrapping*, ki imajo 9 vozlišč, od osnovnega LMT ne razlikuje več kot za 1.5%, razlika v Brierjevi meri pa je nekje med 0.01 in 0.02. K temu seveda veliko pripomorejo logistični modeli v listih.

sezona	klas. točnost	Brier	čas [s]	število vozlišč
2008-2009	52.6952	0.5735	58683	1
2009-2010	52.5128	0.575	127978	3
2010-2011	52.6927	0.5731	129772	1

Tabela 4.4: Rezultati testov na košarkarskih podatkovnih množicah (*bootstrapping* z rezanjem)

sezona	klas. točnost	Brier	čas [s]	število vozlišč
	$T = 0.1$			
2008-2009	43.4523	0.752	12138	2935
2009-2010	51.5665	0.586	25689	8909
2010-2011	52.6927	0.573	4095	1
	$T = 0.15$			
2008-2009	52.6952	0.574	3629	1
2009-2010	52.6306	0.575	4181	1
2010-2011	52.6927	0.573	4099	1

Tabela 4.5: Rezultati testov na košarkarskih podatkovnih množicah (*bootstrapping* s parametrom T)

sezona	klas. točnost	Brier	čas [s]	število vozlišč
	globina drevesa = 1			
2008-2009	36.1389	0.968	3143	3
2009-2010	52.5128	0.573	5915	3
2010-2011	52.6203	0.573	5745	3
	globina drevesa = 2			
2008-2009	36.1389	0.968	3790	5
2009-2010	52.5128	0.575	7174	5
2010-2011	52.6631	0.574	6867	5
	globina drevesa = 3			
2008-2009	51.6061	0.586	3899	9
2009-2010	52.4998	0.576	8289	9
2010-2011	52.4986	0.575	8232	9
	globina drevesa = 4			
2008-2009	51.6061	0.586	4563	15
2009-2010	52.38	0.577	9458	15
2010-2011	52.54	0.587	9027	15

Tabela 4.6: Rezultati testov na košarkarskih podatkovnih množicah (*bootstrapping* z omejeno globino drevesa)

Poglavje 5

Sklepne ugotovitve

V sklopu diplomske naloge smo implementirali algoritem, ki zgradi odločitveno drevo z logistično regresijo v listih in ga testirali na košarkarskih učnih množicah. Po pregledu obstoječe literature smo se odločili, da bomo implementirali algoritem LMT, saj je glede na raziskavo avtorjev algoritma v primerjavi s podobnimi algoritmi dosegel najboljše rezultate. Za programski jezik implementacije smo izbrali C++. Glavni razlog za neučinkovito implementacijo algoritma v Weki je v tem, da so osnovni razredi v Weki zelo veliki in zato njihovi objekti zasedejo preveč pomnilnika.

Odločitev, da algoritem implementiramo od začetka brez obstoječega sistema za strojno učenje, se je izkazala za zelo primerno. S tem smo dosegli, da je naša implementacija 9-krat bolj varčna s porabo pomnilnika in zgradi model 6,5-krat hitreje od implementacije v Weki. Naša implementacija torej omogoča, da zgradimo napovedni model na učni množici s športno domeno v razumnem času z uporabo danes povprečnega osebnega računalnika. Za dosego istega cilja z Weko bi potrebovali dražjo opremo in veliko več časa.

Poleg varčnosti in hitrosti smo želeli doseči tudi boljšo točnost napovedi. To smo poskušali doseči z implementacijo nove metode za izbiro atributa v vozliščih drevesa, ki uporablja metodo *bootstrapping*. Čeprav se nova metoda v praksi kljub različnim izboljšavam ni izkazala za boljšo od osnovne metode, so bili rezultati v primerjavi z osnovno metodo relativno dobri. Predvsem

je presenetljivo to, da je kljub zelo različni izbiri atributov za delitev klasifikacijska točnost naše metode v najboljšem primeru manjša od klasifikacijske točnosti osnovnega algoritma zgolj za 0.5 % do 1.5 %, odstopanje Brierjeve mere pa je med 0.01 in 0.02.

Kljub zelo učinkoviti implementaciji algoritma je na voljo še veliko prostora za razne izboljšave. Algoritem bi lahko s stališča učinkovitosti še izboljšali, vendar je bilo za nas dovolj to, da lahko na povprečnem osebnem računalniku zgradimo model na veliki količini podatkov. Poleg dodatne učinkovitosti bi lahko poiskali drugačne načine za gradnjo drevesa, računanje logističnih regresij ali rezanje drevesa. Zelo dobro bi bilo izboljšati našo metodo *bootstrapping* za izbiro atributa. Trenutna pomanjkljivost je v zaustavitvenem pogoju pri gradnji drevesa z uporabo razlike med prvim in drugouvrščenim atributom. V primeru, ko imamo več atributov, ki so približno enako dobri, se gradnja drevesa ustavi, čeprav je lahko veliko atributov, ki so zelo slabi v primerjavi z njimi. Bolj primerno bi bilo, če bi primerjali razliko prvega atributa z povprečjem razdalj do vseh ostalih atributov.

Glavna pridobitev naše implementacije je lažje in hitrejše testiranje novih nadgradenj algoritma tudi na večjih podatkovnih množicah, kar bi v primeru nadaljnih raziskav prišlo zelo prav.

Literatura

[1] Leo Breiman, Jerome Friedman, Charles J Stone, and Richard A Olshen. *Classification and regression trees*. CRC press, 1984.

[2] Bojan Cestnik, Igor Kononenko, Ivan Bratko, et al. Assistant 86: A knowledge-elicitation tool for sophisticated users. In *EWSL*, pages 31–45, 1987.

[3] Eibe Frank, Yong Wang, Stuart Inglis, Geoffrey Holmes, and Ian H Witten. Using model trees for classification. *Machine Learning*, 32(1):63–76, 1998.

[4] Jerome Friedman, Trevor Hastie, Robert Tibshirani, et al. Additive logistic regression: a statistical view of boosting (with discussion and a rejoinder by the authors). *The annals of statistics*, 28(2):337–407, 2000.

[5] Igor Kononenko and Marko Robnik Šikonja. *Inteligentni sistemi*. Založba FE in FRI, 2010.

[6] Niels Landwehr, Mark Hall, and Eibe Frank. Logistic model trees. *Machine Learning*, 59(1-2):161–205, 2005.

[7] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[8] John R Quinlan et al. Learning with continuous classes. In *Proceedings of the 5th Australian joint Conference on Artificial Intelligence*, volume 92, pages 343–348. Singapore, 1992.

[9] Marc Sumner, Eibe Frank, and Mark Hall. Speeding up logistic model tree induction. In *Knowledge Discovery in Databases: PKDD 2005*, pages 675–683. Springer, 2005.

[10] Yong Wang and Ian H Witten. Inducing model trees for continuous classes. In *Proceedings of the Ninth European Conference on Machine Learning*, pages 128–137, 1997.