SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 4848

Postupci ansambla stabala odluke

Darin Dašić

Zagreb, lipanj 2017.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA
ODBOR ZA ZAVRŠNI RAD MODULA

Zagreb, 6. ožujka 2017.

ZAVRŠNI ZADATAK br. 4848

Pristupnik: **Darin Dašić (0036485752)**
Studij: Računarstvo
Modul: Računarska znanost

Zadatak: **Postupci ansambla stabala odluke**

Opis zadatka:

Ansambli stabala odluke koriste se za vrlo točnu klasifikaciju uzoraka ili predviđanje vrijednosti ciljnog atributa u većem broju područja znanosti i tehnologije (npr. računalni vid, bioinformatika, medicina).
U ovom završnom radu, potrebno je proučiti danas dostupne postupke ansambla stabala odluke (random forest, rotation forest, AdaBoost+C4.5, extremely randomized trees, itd.) te implementirati u programskom jeziku Java barem jedan od postupaka ansambla stabala odluke.
Osim implementacije samog postupka ansambla stabala odluke, potrebno je podržati i učitavanje podataka te prikaz rezultata analize kroz sučelje implementiranog programa.
Točnost i brzinu algoritma potrebno je usporediti sa sličnim implementacijama u programskim paketima Weka, RapidMiner ili KEEL.
Vrednovanje implementacije potrebno je provesti na temelju više otvorenih skupova podataka iz repozitorija UC Irvine Machine Learning (UCI).

Zadatak uručen pristupniku: 10. ožujka 2017.
Rok za predaju rada: 9. lipnja 2017.

Mentor:

Doc. dr. sc. Alan Jović

Djelovođa:

Doc. dr. sc. Tomislav Hrkać

Predsjednik odbora za završni rad modula:

Prof. dr. sc. Siniša Srbljić

Zahvaljujem se mentoru doc. dr. sc. Alanu Joviću na savjetima i pomoći pri izradi ovog Završnog rada.

SADRŽAJ

1. Uvod

Strojno učenje jest programiranje računala na način da optimiziraju neki kriterij uspješnosti temeljem podatkovnih primjera ili prethodnog iskustva (Alpaydin, 2009). Strojno učenje koristi algoritme koji grade model za rješavanje kompleksnih problema bez izravnog programiranja rješenja. Ako gledamo na strojno učenje kao područje računarske znanosti, ono je zapravo disciplina koja uključuje matematiku i statistiku, ali i inženjerstvo podataka. Samo područje je proizašlo iz statistike i raspoznavanja uzoraka, a danas je zasigurno najpopularniji pojam u računarskoj znanosti ali i u cijeloj tehnološkoj industriji.

U ovom radu razmotrit ćemo metode ansambala stabala odluke kao metodu strojnog učenja te ćemo pobliže razmotrit jednu verziju: slučajne šume. Glavna ideja ansambla odluke je, umjesto izgradnje samo jednog modela, napraviti složeni model koji se zasniva na više osnovnih modela (engl. *base models*). Umjesto da iz skupa podataka za učenje izgradimo samo jedan model, taj skup podataka je često bolje ponovno uzorkovati. Iz svakog novo dobivenog uzorka, skupa za učenje, izgraditi poseban model, zatim kombinirati sve modele te završiti s jednim moćnijim modelom. Ovaj postupak je donekle kontra intuitivan. Naizgled bi najbolji model bio onaj koji je učen nad cijelim originalnim skupom podataka. No, stabla odluke su sklona prenaučenosti (engl. *overfitting*) te uvođenje neke vrste šuma u podatke dovodi do izrade različitih stabala. Svako posebno stablo neće uvijek biti vrlo precizno kao osnovni klasifikator. Međutim u ansamblu kao skupu modela, zajedno daju značajno bolje rezultate.

Pod tehnike za izgradnju ansambla odluke spadaju *bagging*, *boosting* te *stacking*. Metoda *stacking* je jedina koja u pravilu ne koristi osnovne modele istog tipa. Svaka od tih tehnika u većini slučajeva proizvodi modele koji su precizniji od osnovnih. Ansambli se koriste u klasifikacijskim problemima te ih je moguće izgraditi i za regresijske probleme.

Od spomenutih tehnika, *boosting* uglavnom daje najbolje rezultate. Metoda *boosting* koristi neke iste tehnike kao i poznata statistička tehnika aditivnih modela, što je rezultiralo i daljnjom optimizacijom algoritma. Metoda *boosting* se

bazira na ideji da u slučaju više eksperta jedan pokriva područje u kojemu drugi nije toliko precizan. Tako da u skupu više eksperta ili osnovnih modela, zajedno pokrivaju široku domenu problema. Osnovni modeli u *boostingu* su istog tipa, npr. stabla odluke. Izgradnja ansambla metodom *boosting* se bazira na iteraciji osnovnih modela u kojoj se svaki novi model izgrađuje u odnosu na performanse prijašnjih iteracija. Svaka nova iteracija stvara model koji se fokusira na popravku grešaka onih instanci koje je prošli model pogrešno klasificirao. Svaka ta instanca dobiva jače težine, koje sljedeća iteracija modela pokušava točno klasificirati sa većim prioritetom. Ansambl je izgrađen prolaskom kroz sve potrebne iteracije ili ranijim prekidom te svaki model ima svoju težinu. To znači da nisu svi osnovni modeli jednaki, već imaju težine koje uvjetuju koliko njihov glas vrijedi u konačnom glasanju cijelog ansambla.

Neke od implementacija metode *boosting* su algoritmi *Gradient boosting* i *AdaBoost C4.5* . *AdaBoost* koristi stablo odluke C4.5 kao osnovni klasifikator što omogućuje implementaciju težinskog treniranja samog stabla. U početku, svaka instanca u setu za treniranje ima jednaku težinu ($1/N,\ N = \text{broj instanci}$). Učenje počinje izgradnjom stabla prve iteracije, a zatim se osvježavaju vrijednosti težina pojedinih instanci. Nakon toga se gradi model sljedeće iteracije i postupak ponavlja. Sljedeća formula prikazuje osvježavanje težina točno klasificiranih instanci.

$$weight \leftarrow weight \times e/(1-e)$$

Gdje *e* predstavlja pogrešku modela na težinskim instancama (u rasponu od 0 do 1). Nakon što su osvježene sve vrijednosti , težine se normaliziraju, tj. suma svih težina će biti ista kao i prije. U slučaju da je greška *e* veća ili jednaka 0.5, briše se zadnja iteracija modela te se zaustavlja izgradnja daljnjih iteracija. Postupa se isto u slučaju kada je greška jednaka 0. Sljedeća formula koja definira težine pojedinih modela pojašnjava zašto se izgradnja zaustavlja u slučaju $e = 0$.

$$weight = -\log\frac{e}{1-e}$$

Metoda *bagging* se bazira na statističkoj metodi ponovnog uzorkovanja, *bootstrap* (*bagging* je skraćenica za *bootstrap aggregation*, što u prijevodu znači *bootstrap* uzorkovanje). Uzorkovanje *bootstrap* iz populacije vadi uzorak u kojemu se neke instance iz populacije ponavljaju, dok drugih nema.

Iz originalnog skupa za učenje vadi se uzorak jednake veličine kao i originalni. U tom uzorku neke instance za učenje će se ponavljati dok drugih neće biti. To je na neki način slično težinama instanci u *boosting* metodi. Instance koje se više puta pojavljuju tretiraju se kao instance s većim težinama, dok instance kojih više nema, s manjim jer se ne uzimaju u obzir prilikom izgradnje stabla. Međutim, ponovno uzorkovane instance se uzimaju potpuno slučajno, za razliku od težina u *boostingu*. Svaki osnovni model je iste vrste te pri glasanju težina nema. Sljedeća formula predstavlja predikciju metodom *bagging*.

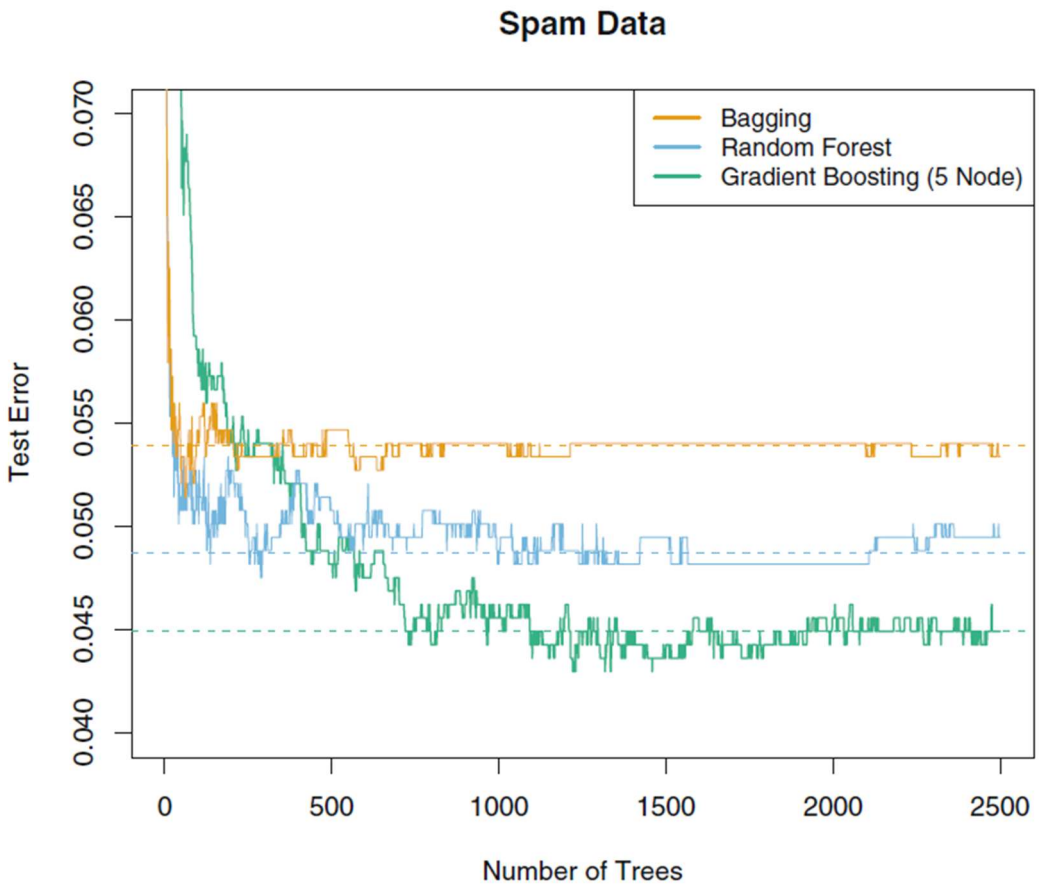
$$\hat{f}_{bag}(x) = \frac{1}{B}\sum_{b=1}^{B} \hat{f}^{*b}(x)$$

Gdje je *B*, broj ponovnih uzorkovanja, $\hat{f}^{*b}$ osnovni model izgrađen na temelju *bootstrap* uzorka. Primjenom *bagging* metode smanjuje se varijanca u odnosu na osnovnog klasifikatora dok pristranost ostaje jednaka.

Pristranost je mjera koliko model precizno klasificira podatke za učenje. Što je pristranost manja, to je model precizniji na skupu za učenje. Varijanca ovisi o podacima za učenje te se mjeri nad skupom za testiranje. Ukoliko model ima visoku varijancu, onda je osjetljiv na promjene u podacima za učenje te će davati loše rezultate pri klasifikaciji novih instanci.

Pristranost i varijanca ovise jedna o drugoj te obje pridonose pogrešci modela. Modeli sa malom pristranošću najčešće imaju visoku varijancu te obratno. To jedan od glavnih problema prilikom izgradnje modela, izgraditi što bolji model na temelju odnosa varijance i pristranosti. Analiza međusobne ovisnosti se naziva dekompozicija varijance i pristranosti (engl. *bias-variance decomposition*) kojoj je cilj pronaći optimalnu hipotezu (Witten, 2011).

Neki od algoritama koji se baziraju na *bagging* metodi su slučajne šume, ekstremno slučajne šume te rotacijske šume. Iz slike 1.1. (Hastie,2009) mogu se vidjeti različite performanse nekih navedenih algoritama. Možemo zaključiti da su slučajne šume značajno poboljšanje *bagging* metode, te da naspram *gradient boosting* algoritma brže dolaze do svojeg optimalnog modela. *Gradient boosting* treba puno više iteracija do optimalnog modela ali daje bolje rezultate i pri manjem broju iteracija.



Slika 1.1 Ovisnost pogreške o broju stabala

Zajednička mana ovih algoritma je njihova nejasna interpretacija. Ansambli se do jedne razine ponašaju kao crne kutije. Pošto se sastoje od velikog broja osnovnih modela, unatoč njihovoj preciznosti nije baš intuitivno shvatiti koji točno faktori pridonose preciznosti algoritma. U sljedećim poglavljima ćemo detaljno razmotriti algoritam slučajnih šuma.

2. Algoritam slučajne šume

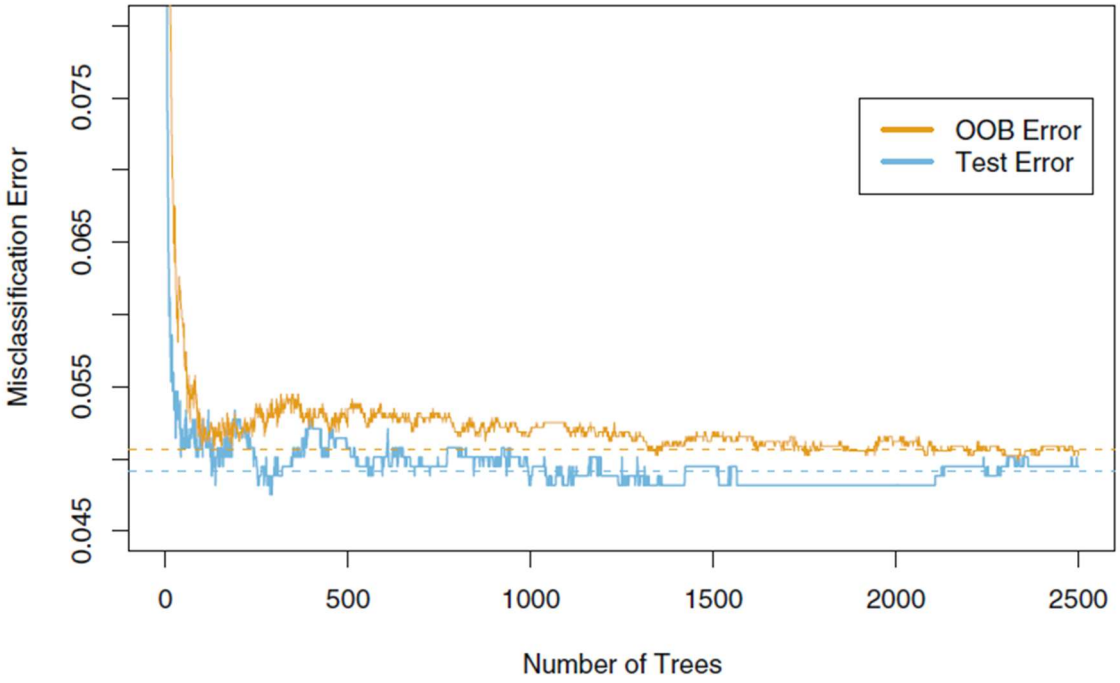
Algoritam slučajnih šuma se bazira na modificiranoj verziji *bagging* metode. Razvio ga je Breiman u svom radu *Random Forests* (Breiman 2001). Algoritam kombinira različite postojeće ideje o slučajnostima iz statistike. Već spomenutu *bagging* metodu te odabir slučajnog vektora atributa iz cijelog skupa atributa (Dietterich,2000). Obje tehnike pokušavaju dovesti određeni šum u podatke i algoritam, što rezultira sa preciznijim modelom pri obradi novog skupa podataka.

Algoritam slučajnih šuma gradi velik broj različitih stabala koja glasaju u slučaju klasifikacije ili pronalazi srednju vrijednost pri regresiji. Iz skupa za učenje, *bootstrap* uzorkovanjem dobivamo novi uzorak koji je po veličini jednak originalnom. Zatim se nad svakim uzorkom izgradi jedno modificirano stablo CART, prema pseudokodu na slici 2.1.

1. Za b=1 do B , gdje je B broj stabala
    a. Izvadi *bootstrap* uzorak Z* veličine *N* iz skupa za učenje
    b. Izgradi stablo slučajne šume T_b trenirano na uzorku bootstrap, rekurzivno ponavljajući sljedeće korake za svaki čvor u stablu
        i. Izaberi *m* slučajnih varijabli od *p* mogućih.
        ii. Pronađi najbolju varijablu i vrijednost za podjelu među odabranih *m.*
        iii. Podijeli čvor u lijevi i desni.
2. Spremi skup stabala $\{T_b\}_1^B$.
3. Prilikom predviđanja
    a. Za regresiju : $\hat{f}_{rf}^B(x) = \frac{1}{B}\sum_{b=1}^{B} T_b(x)$
    b. Za klasifikaciju: $\hat{c}_{rf}^B(x) = najviše\ glasova\ \{\hat{c}_b(x)\}_1^B$, gdje je $\hat{c}_b(x)$ predikcija pojedinog stabla

Slika 2.1 Algoritam slučajne šume

Pri uzorkovanju, otprilike 33% originalnih podataka se ne iskoristi. Taj skup podataka se naziva „out-of-bag“ (OOB) skup, te pomoću njega je moguće provesti unakrsnu validaciju svakog izgrađenog stabla. Takva unakrsna validacija je dobra procjena greške slučajne šume. Također testiranje se izvodi paralelno uz izgradnju stabla.



Slika 2.2 Ovisnost broja stabala i greške OOB skupa te skupa za testiranje

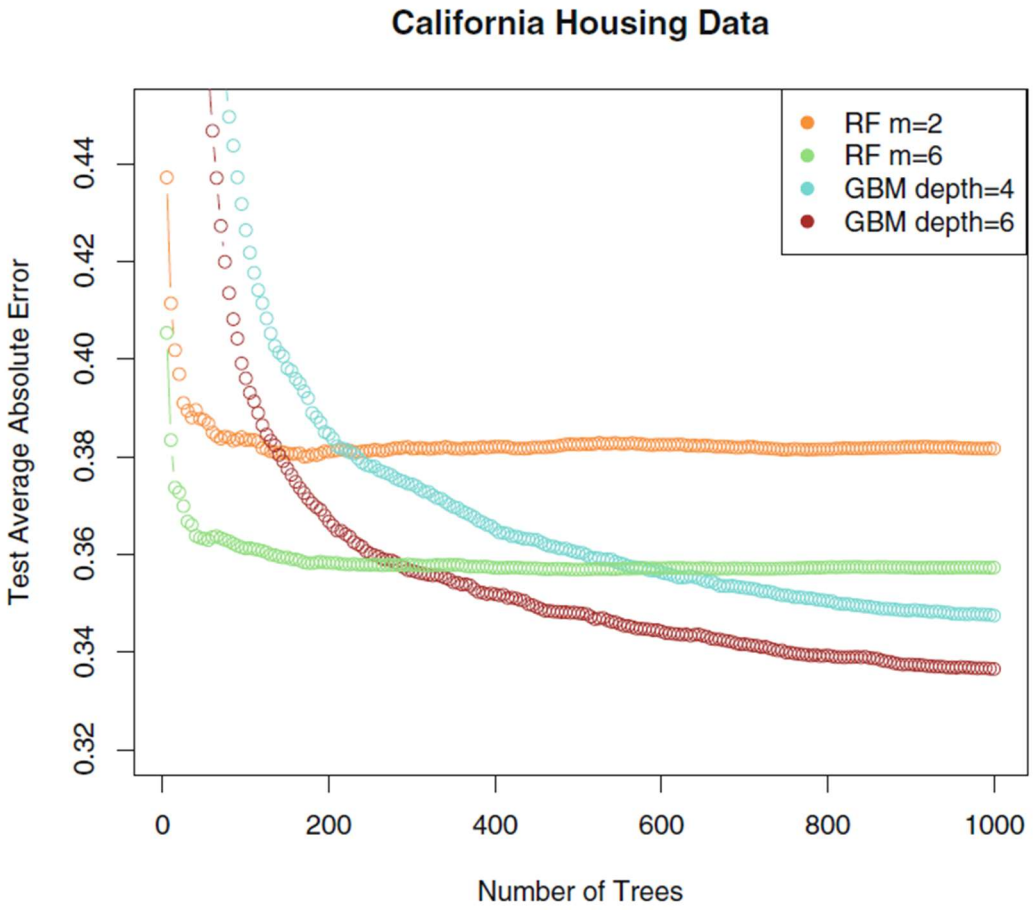
Za vrijeme stabilizacije *OOB* greške moguće je prekinuti daljnju izgradnju budućih stabala. Na slici 2.2 (Hastie, 2009) vidljiva je ovisnost greške naspram izgrađenih stabla te kako se ponaša *OOB* greška u odnosu na onu iz *k*-struke unakrsne validacije.

Postupak uzorkovanja i izgradnje stabala ponavljamo onoliko puta koliko će biti stabla u slučajnoj šumi, dok je $N = 100$ pretpostavljena vrijednost. Svaki takav uzorak se razlikuje, stoga nastaje šum naspram originalnog skupa podataka. Zato se svako izgrađeno stablo međusobno razlikuje. Kao i kod metode *bagging*, glavni razlog zašto je uvođenje šuma pozitivno je smanjenje varijance.

Stabla odluke i regresijska stabla, koja se koriste kao osnovni modeli u slučajnim šumama, sklona su prenaučenosti. U slučaju prenaučenosti, model ima nisku vrijednost pristranosti te donosi loše odluke pri klasifikaciji novih podataka. Stoga, završavamo s modelom s visokom varijancom. S druge strane, stabla odluke mogu prepoznati kompleksne veze između atributa i ciljne klase što je ključno jer je ansambl dobar samo ukoliko su mu osnovni modeli sposobni

proizvodit dobre rezultate. Svako stablo izgrađeno na uzorku *bootstrap* je identično distribuirano, stoga je očekivanje samog ansambla jednako kao i kod svakog pojedinog stabla. Kao rezultat, pristranost slučajnih šuma je jednaka kao i kod samih stabala, dok smanjenje varijance uvjetuje bolje rezultate. Cilj algoritma je što više smanjiti korelaciju među pojedinim stablima kako bi se pokrila što veća varijacija podataka bez da se poveća varijanca ukupnog ansambla.

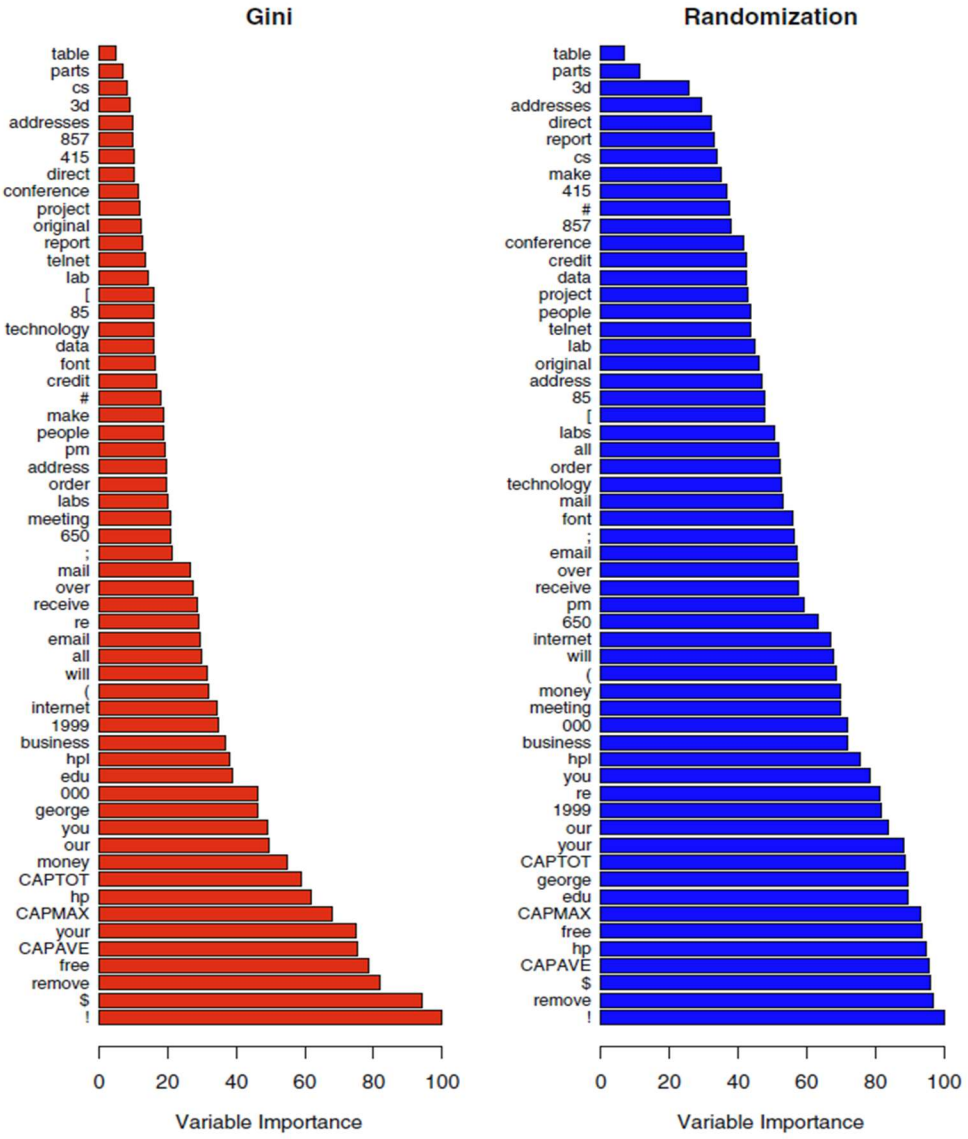
Zato se dodatno uvodi postupak slučajnog odabira atributa pri izgradnji stabla. Pri izgradnji svakog čvora, slučajno se odabere $m \leq p$ ulaznih varijabla (*p* je ukupan broj atributa) kao razmatranih za podjelu čvora. Obično se kao *m* uzima $\lfloor \sqrt{p} \rfloor$ pri klasifikaciji, te $\lfloor p / 3 \rfloor$ za regresiju. Ovi parametri mogu biti proizvoljno odabrani te se koriste kao parametri za podešavanje algoritma čije će najbolje vrijednosti zavisiti o problemu, vidjeti sliku 2.3 (Hastie 2009). Dodatno, svako stablo se gradi do kraja, bez podrezivanja, a najmanji broj instanci pri podijeli je 2.



Slika 2.3 Usporedba odabira parametra m naspram preciznosti algoritma slučajnih šuma (RF) i gradient boostinga (GBM)

Slučajne šume, budući da se baziraju na stablima odluke, mogu računati procjenu važnosti pojedinih atributa. Osim kriterija za odabir podijele čvora pomoću indeksa Gini, moguće je obaviti procjenu važnosti atributa pomoću skupa OOB. Prilikom izgradnje pojedinog stabla potrebno je izračunati preciznost na temelju OOB skupa. Zatim, za *i*-ti atribut slučajno permutirati sve vrijednosti te onda ponovno izračunati preciznost stabla. Smanjenje preciznosti se sprema, te se pronalazi srednja vrijednost za sva stabala. Što je veće smanjenje preciznosti, to se uvjetuje veća važnost određenog atributa.

Na slici 2.4. (Hastie,2009) vidi se rezultat izračuna važnosti atributa. Na slici lijevi graf (Gini) prikazuje vrijednosti izračunate pomoću indeksa Gini, dok desni graf (*Randomization*) prikazuje vrijednosti izračunate slučajnom permutacijom vrijednosti atributa iz *OOB* skupa.



Slika 2.4 Procjena važnosti atributa pomoću indeksa Gini te Randomization

Vidljivo je da obje metode pridaju slične vrijednosti pojedinim atributima. Važna razlika je uniformna razdioba za metodu *OOB* naspram indeksa Gini. Uz to, metodom *OOB* ne možemo znati kako bi se ponašali atributi ako bismo jedan od atributa potpuno uklonili, budući da bi u tom slučaju stablo bilo izgrađeno na drugačiji način.

3. Eksperimentalno vrednovanje algoritma

U ovom poglavlju ispitana je moja implementacija algoritma slučajne šume te je uspoređena sa implementacijom iz paketa WEKA. Programski paket WEKA je dostupan na sljedećoj poveznici: http://www.cs.waikato.ac.nz/ml/weka/downloading.html.

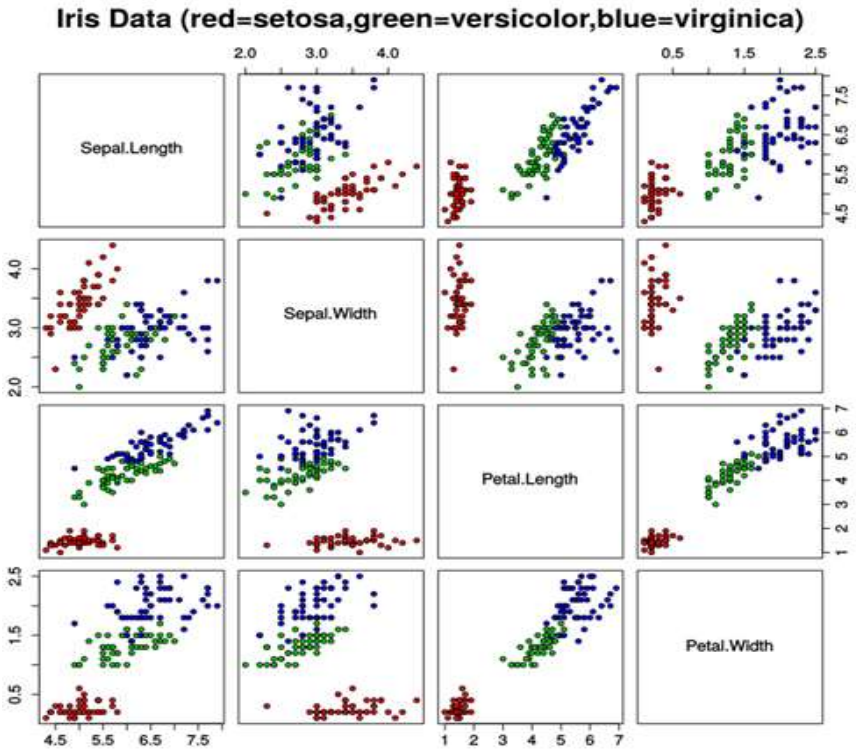
Svi skupovi podataka za ispitivanje su dostupni sa sljedeće poveznice : http://storm.cis.fordham.edu/~gweiss/data-mining/datasets.html, a moguće ih je pronaći i na UCI repozitoriju za strojno učenje : http://archive.ics.uci.edu/ml/

Ispitivanja su provedena na računalu sljedećih karakteristika (Slika 3.1) :

Processor:	Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz 2.70 GHz
Installed memory (RAM):	8.00 GB (7.85 GB usable)
System type:	64-bit Operating System, x64-based processor

Slika 3.1 Karakteristike računala

Prvi skup podataka koji je testiran je poznati Fisherov skup podataka cvjetova irisa, slika 3.2 (Nicoguaro, 2016).



Slika 3.2 Skup podataka iris

Skup se sastoji od 150 instanci, 4 kontinuirana atributa te ciljne klase. U programu je pokrenuta 10-struka unakrsna validacija. Također, sa istim parametrima je pokrenuta i Wekina implementacija. Moji rezultati znaju podosta varirati zbog svih slučajnosti nad kojima se algoritam temelji. Priložene slike će prikazivati rezultat jedne unakrsne validacije , dok će dodatno biti ispisana i srednja vrijednost te standardna devijacija uzorka za određeni broj izvođenja.

Pošto je skup podataka irisa dosta mali, evaluaciju sam pokrenuo 100 puta te srednja vrijednost preciznosti iznosi 95.3131% dok standardna devijacija uzorka 0.7371%. Slika 3.3 prikazuje rezultat jednog izvođenja unakrsne evaluacije nad skupom iris. Na slici 3.4 je isječak rezultata izvođenja u Wekinom programu.

```
Confusion matrix:
  1.  2.  3.    <-- classified as
  47   3   0  | 1. = Iris-versicolor
   2  48   0  | 2. = Iris-virginica
   2   0  48  | 3. = Iris-setosa

 Correctly classified instances:             143      95.3333 %
 Incorrectly classified instances:             7       4.6667 %

 Time taken to build a model: 0.56 s
```

Slika 3.3 Moji rezultati za skup podataka iris

```
weka.classifiers.trees.RandomTree -K 0 -M 1.0 -V 0.001 -S 1 -do-not-check-capabilities

Time taken to build model: 0.02 seconds

Correctly Classified Instances         143               95.3333 %

Incorrectly Classified Instances         7               4.6667 %

=== Confusion Matrix ===

 a  b  c   <-- classified as

 50  0  0 |  a = Iris-setosa

 0 47  3 |  b = Iris-versicolor

 0  4 46 |  c = Iris-virginica
```

Slika 3.4 Wekini rezultati za skup podataka iris

U ovom testu sam dobio identične rezultate što se tiče preciznosti. Ako pogledamo matricu konfuzije, koja nam jednostavno prikazuje sve rezultate unakrsne validacije, vidi se da iako je postotak točne klasifikacije jednak, algoritmi se ponašaju drugačije. Wekin model miješa samo *Iris-versicolor* te *Iris-virginicu* dok moje rješenje zna te klase klasificirati i kao *Iris-setosa.* Glavna razlika u

rezultatima je vrijeme izvođenja. Nažalost, moj algoritam je vrlo spor naspram rješenja iz WEKA paketa. Zasigurno njihov algoritam sadrži kvalitetnu heuristiku prilikom izgradnje stabala dok moj iscrpno pretražuje sve moguće podjele.

Sljedeći skup podataka naziva „Glass Identification Database“, sadrži 214 instanci, te 10 atributa. Na temelju kemijskih elemenata pokušava se klasificirati u jednu od 7 vrsta stakla. Svih 10 atributa su kontinuirane vrijednosti.

```
Confusion matrix:
  1.  2.  3.  4.  5.  6.    <-- classified as
   7   2   0   0   0   0  | 1. = tableware
   2  57   2   4   9   2  | 2. = 'build wind non-float'
   0   3   9   0   0   1  | 3. = containers
   0   4   0   5   8   0  | 4. = 'vehic wind float'
   0  11   0   1  57   1  | 5. = 'build wind float'
   0   2   0   0   1  26  | 6. = headlamps

 Correctly classified instances:             161      75.2336 %
 Incorrectly classified instances:            53      24.7664 %

 Time taken to build a model: 5.35 s
```

Slika 3.5 Moji rezultati za skup vrsta stakla

Algoritam je evaluiran 10-strukom unakrsnom validacijom, koja je pokrenuta 10 puta. Srednja vrijednost preciznosti iznosi 74.070% dok je standardno odstupanje uzorka 1.2709%. Nad ovim skupom podataka moj algoritam je ostvario značajno slabije rezultate od Wekinog.

```
Time taken to build model: 0.07 seconds

Correctly Classified Instances         171               79.9065 %
Incorrectly Classified Instances        43               20.0935 %

=== Confusion Matrix ===

 a  b  c  d  e  f  g   <-- classified as
61  7  2  0  0  0  0 |  a = build wind float
 8 62  2  0  2  1  1 |  b = build wind non-float
 8  3  6  0  0  0  0 |  c = vehic wind float
 0  0  0  0  0  0  0 |  d = vehic wind non-float
 0  1  0  0 11  0  1 |  e = containers
 0  1  0  0  0  7  1 |  f = tableware
 1  4  0  0  0  0 24 |  g = headlamps
```

Slika 3.6 Wekini rezultati za skup vrsta stakla

Sljedeće testiranje je provedeno na skupu podataka „Pima Indians Diabetes Database“ . Skup podataka se sastoji od 768 instanci, 8 atributa te ciljne klase. Svi atributi su kontinuirani. Ciljna klasa se sastoji od dvije različite vrijednosti: da li osoba boluje od dijabetesa ili ne. Pošto su sljedeća dva skupa podataka dosta velika, nije izračunata srednja vrijednost i devijacija.

```
Confusion matrix:
  1.  2.    <-- classified as
 421  79  | 1. = tested_negative
 125 143  | 2. = tested_positive

 Correctly classified instances:             564      73.4375 %
 Incorrectly classified instances:           204      26.5625 %

 Time taken to build a model: 35.61 s
```

Slika 3.7 Moji rezultati za skup podataka dijabetes

Time taken to build model: 0.16 seconds

Correctly Classified Instances 582 75.7813 %

Incorrectly Classified Instances 186 24.2188 %

=== Confusion Matrix ===

a b <-- classified as

421 79 | a = tested_negative

107 161 | b = tested_positive

Slika 3.8 Wekini rezultati za skup podataka dijabetes

Što se preciznosti tiče , obje implementacije su ostvarile niske vrijednosti međutim Wekin algoritam ostvaruje značajno bolje rezultate nego moja implementacija.

Zadnji skup podataka pod naslovom „Image Segmentation data“ se sastoji od 1607 instanci. Svaka instanca sadrži 19 kontinuiranih atributa te ciljnu klasu. Postoji sedam različitih ciljnih klasa s jednakim brojem instanci.

Iz rezultata sa slika 3.9 i 3.10 vidi se da je Wekin algoritam krivo klasificirao samo 8 slika (od 510), dok moje rješenje 20. U postocima su oba algoritma dali vrlo impresivne rezultate ali ponovno WEKA ima signifikantno bolju razinu točnosti, 98.4% naspram 96.1%.

Očigledno, Wekina implementacija je superiornija od moje, posebice u vremenu izvođenja ali za neke skupove i moja implementacija daje vrlo dobre rezultate.

```
 Correctly classified instances             490       96.0784 %
 Incorrectly classified instances            20        3.9216 %

 Confusion matrix:
1.  2.  3.  4.  5.  6.  7.    <-- classified as
75   0   0   0   0   0   0  | 1. = sky
 0  73   0   0   0   0   0  | 2. = path
 1   0  76   1   0   0   2  | 3. = foliage
 3   3   0  60   0   0   3  | 4. = cement
 0   0   0   1  70   0   1  | 5. = brickface
 0   0   0   0   0  77   0  | 6. = grass
 0   0   3   1   1   0  59  | 7. = window

Time taken to build a model: 303.88 s
```

Slika 3.9 Moji rezultati za segmentaciju slika

Time taken to build model: 0.66 seconds

Correctly Classified Instances 502 98.4314 %

Incorrectly Classified Instances 8 1.5686 %

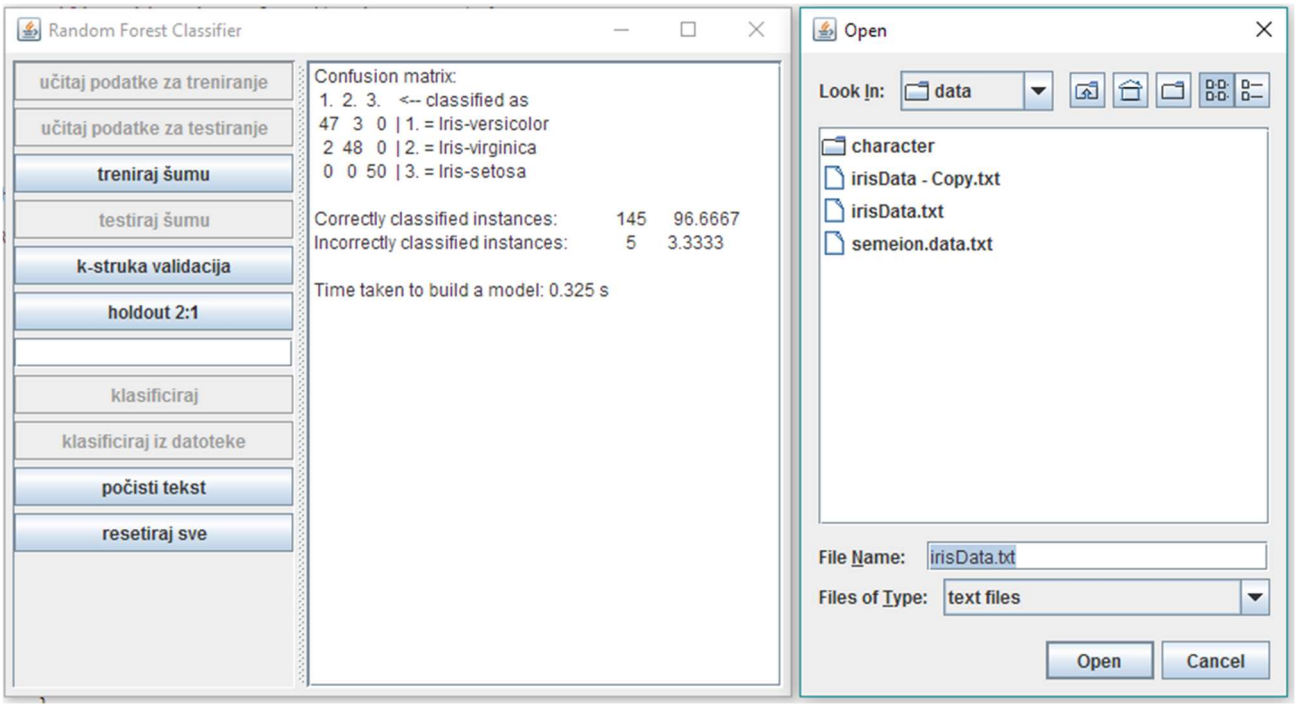
=== Confusion Matrix ===

```
 a  b  c  d  e  f  g  <-- classified as
64  0  0  0  0  0  0 |  a = brickface
 0 81  0  0  0  0  0 |  b = sky
 0  0 71  0  2  0  0 |  c = foliage
 0  0  0 74  5  0  0 |  d = cement
 0  0  1  0 59  0  0 |  e = window
 0  0  0  0  0 80  0 |  f = path
 0  0  0  0  0  0 73 |  g = grass
```

Slika 3.10 Wekini rezultati za segmentaciju slika

4. Opis programskog rješenja

Programsko rješenje je ostvareno u programskom jeziku java. Program se sastoji od jednostavnog grafičkog sučelja koje omogućava korisniku odabir željenog seta podataka za testiranje navigacijom kroz datotečni sustav operacijskog sustava. Datoteke koje su pogodne za učitavanje su formata .csv , no učitavanje će raditi i za druge formate ako su pravilno formatirane. Očekuje se da od prvog reda , svaki red predstavlja jednu instancu, gdje je svaka vrijednost odvojena od druge zarezom. Zadnji atribut se tretira kao ciljna klasa.



Slika 4.1 Grafičko sučelje programa

Nakon što se podaci učitaju, korisnik bira hoće li napraviti evaluaciju algoritma ili trenirati jednu slučajnu šumu. Korisnik može birati između unakrsne validacije sa zadržavanjem (2:1) ili *10*-struke unakrsne validacije. Svaka evaluacija ispisuje u grafičko sučelje broj točno klasificiranih instanci, broj netočno klasificiranih instanci, postotni udio, matricu konfuzije te vrijeme potrebno za izgradnju jedne slučajne šume.

Ako je korisnik izabrao učenje jedne slučajne šume, nakon toga mu se omogućuje odabir nove datoteke koja sadrži set podataka za testiranje. Ukoliko korisnik želi primijeniti algoritam za klasifikaciju svojih instanci, skup instanci može

učitati iz datoteke ili ručno unositi vrijednosti svakog atributa odvojeno zarezom.

Sam program se sastoji od 5 razreda. Razred `Data.java` se koristi za upravljanje podacima, a predstavlja skup instanci. Omogućuje lakše baratanje s instancama, dohvat vrijednosti atributa, dohvat mogućih vrijednosti atributa i slično. Konstruktor prima listu lista stringova, gdje svaka lista stringova predstavlja jednu instancu, a string predstavlja vrijednost pojedinog atributa.

```
package zavrsnirad;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Map.Entry;

public class Data {

	Integer numberOfFeatures;
	Integer size;
	HashSet<Integer> discreteFeatures;
	ArrayList<ArrayList<String>> data;

	public Data(ArrayList<ArrayList<String>> data) {


		this.data = data;
		this.size = data.size();
		if(data.size()>0)
			this.numberOfFeatures = data.get(0).size()-1;
		this.discreteFeatures = new HashSet<>();

	}

	public HashSet<Integer> findDiscreteValues(Double criteria) {…}

	public static boolean isNumeric(String str) {…}

	public boolean isDiscrete(Integer id) {…}

	public int getSize() {…}

	public String getLabel(int i) {…}

	public Integer getNumberOfFeatures(){…}

	public String getFeature(int i, Integer feature) {…}

	public ArrayList<String> getObservation(int i) {…}

	public HashSet<String> getValuesForSplit(Integer feature) {…}

	public String findLabel() {…}

	public String isPure(){…}

	public HashSet<String> getPossibleLabels(){…}

	public Data getSubdata(ArrayList<Integer> dataindexes){…}

}
```

Slika 4.2 Isječak iz razreda Data

Ukratko o bitnijim metodama. Metoda `findDiscreteValues` prolazi kroz listu instanci te klasificira svaki atribut kao diskretni ili kontinuirani. Svaki atribut čije vrijednosti sadrže znakove i slova su automatski identificirane kao diskretne. Za slučaj brojeva, gleda se koliko unikatnih vrijednosti postoje naspram broja instanci. Ako ih je više od kriterija koji je pretpostavljen kao 10% onda je atribut označen kao kontinuiran, a u suprotnom diskretan. Metoda `isDiscrete` provjerava da li je zadani atribut diskretan ili ne. Metoda `getFeature` vraća vrijednost atributa `feature` za instancu *i*. Metoda `getValuesForSplit` za zadani atribut vraća sve postojeće unikatne vrijednosti atributa. Metoda `findLabel` prolazi kroz skup instanci te pronalazi koja se klasa najviše pojavljuje te ju vraća. U slučaju da ih je više, vraća se slučajnim odabirom. Metoda `isPure` se koristi za ispitivanje da li su sve instance jednake klase. Metoda `getSubdata` za zadanu listu indeksa vraća novi objekt `Data` koji je podskup objekta nad kojim je pozvana metoda.

Razred `DataLoader.java` služi za učitavanje podataka iz datoteke. Sadrži dvije metode, a njena glavna metoda `readCSV` prima putanju do datoteke u string formatu te učitava podatke i sprema ih u listu instanci, gdje je svaka instanca lista stringova. Takav zapis se koristi pri konstrukciji objekta razreda `Data`.

```
public class DataLoader {

	private static final String DEFAULT_SEPARATOR = ",";

	public static ArrayList<ArrayList<String>> readCSV(String csvFile) throws
IOException, ParseException, NumberFormatException{…}

	private static ArrayList<String> parseLine(String readLine, int i) throws
ParseException {…}
```

Slika 4.3 Isječak razreda DataLoader

Sljedeći razred je osnovni model slučajnih šuma, `RandomTree.java` . Razred predstavlja modificiran algoritam stabla CART za potrebe slučajne šume. Stablo se gradi tako da se slučajno uzme *m* atributa iz skupa svih atributa. Za svaki od izabranih atributa se prođe kroz sve moguće podijele te računa indeks Gini. Ovdje leži razlog sporosti algoritma naspram implementacije iz WEKA paketa.

```
public class RandomTree {

	Integer maxDepth, minSize, numberOfFeatres;
	HashSet<String> labels;
	HashSet<Integer> discreteFeatures;
	Node rootNode;

	public RandomTree(Integer minSize, Integer numberOfFeatures, HashSet<String>
labels,
			HashSet<Integer> discreteFeatures) {

		this.minSize = minSize;
		this.numberOfFeatres = numberOfFeatures;
		this.labels = labels;
		this.discreteFeatures = discreteFeatures;

	}

	public void fit(Data dataset) {…}

	public Double getPredictionAccuracy(Data testData) {…}

	public String predict(ArrayList<String> observation) {…}

	public Double calculateGini(ArrayList<Data> subnodes, HashSet<String> labels) {…}

	public Wrapper findBestSplit(Data dataset, HashSet<Integer> validFeatures) {…}

	public ArrayList<Data> getSplit(Data dataset, Integer feature,String value){…}

	public class Node {…}

	private class Wrapper {…}
}
```

Slika 4.4 Isječak razreda RandomTree

Indeks Gini se dobiva pomoću metode `calculateGini`. Metoda mjeri frekvenciju pojedinih klasa u podijeljenim podacima te vraća težinski zbroj svakog čvora.

```
   public Double calculateGini(ArrayList<Data> subnodes, HashSet<String> labels) {

		double sum = subnodes.get(0).getSize() + subnodes.get(1).getSize();

		double giniIndex = 0.0;
		for (Data subnode : subnodes) {

			double subnodeWeight = subnode.getSize() / sum;
			ArrayList<String> dataLabels = new ArrayList<>();
			double size = subnode.getSize();

			for (int i = 0; i < size; i++) {
				dataLabels.add(subnode.getLabel(i));
			}
			for (String label : labels) {
				double ratio = Collections.frequency(dataLabels, label)
  /size;
				giniIndex += subnodeWeight * ratio * ratio;
			}
		}
		return giniIndex;
	}
```

Slika 4.5 Isječak metode za izračun indeksa Gini

Izgradnja samog stabla počinje izgradnjom čvora korijena stabla. Čvor je posebno ugniježđeni razred `Node` unutar razreda `RandomTree.java`.

```
public class Node {

		Node left;
		Node right;
		String label;
		Integer featureIndex;
		String featureValue;
		boolean discrete;
		boolean isTerminal;

		public Node(Data dataset, Integer numberOfFeaturesForSplit) {…}

		public String predict(ArrayList<String> observation) {…}

		}
```

Slika 4.6 Isječak razreda Node

Konstrukcija počinje u korijenu te se rekurzivno nastavlja do samih listova, odnosno terminalnih čvorova. Svaki čvor sadrži pokazivač na lijevi i desni čvor, oznaku atributa, vrijednost za grananje, da li je gledani atribut diskretan te da li je čvor list. Ukoliko je list, sadrži i oznaku kojoj klasi pripada čvor. Konstrukcija razreda `Node` se oslanja na metodu `findBestSplit`.

```
	public Wrapper findBestSplit(Data dataset, HashSet<Integer> validFeatures) {
		Wrapper best = null;
		ArrayList<Data> subnodes;
		double tempGiniIndex;
		double bestGiniIndex = -1;
		HashSet<Integer> evaluatedFeatures = new HashSet<>(validFeatures);

		for (Integer feature : validFeatures) {
			HashSet<String> splitValues = dataset.getValuesForSplit(feature);
			for (String value : splitValues) {
				subnodes = getSplit(dataset, feature, value);
				tempGiniIndex = calculateGini(subnodes, labels);

				if (tempGiniIndex > bestGiniIndex) {
					bestGiniIndex = tempGiniIndex;
					best = new Wrapper(feature, value, subnodes);
				}
			}
		}

 // if all feature values are same for different labels, check other attributes
while (best == null || evaluatedFeatures.size() < numberOfFeatres) {…}

		return best;
	}
```

Slika 4.7 Isječak metode findBestSplit

Ta metoda prima slučajan vektor atributa i traži najbolju podjelu. Informacije o podijeli vraća u obliku omotača pod nazivom `Wrapper`. To je jednostavan omotač koji sadrži indeks atributa, vrijednost tog atributa te podatke podijeljene na lijevi i desni podskup. Sa slike 4.7 se vidi kako funkcionira metoda `findBestSplit`.

Konstrukcija cijelog stabla se odvija rekurzivno u dubinu pronalaskom najbolje podijele. Kada u jednom čvoru završe instance sa jednakom ciljnom klasom, čvor se označava listom i izgradnja prestaje. Izgradnja također prestaje kada u jednom čvoru ostane samo jedna instanca, te u slučaju nemogućnosti pronalaska najbolje podjele. Tada se uzima klasa s najvećom frekvencijom kao predikcija klase lista.

Glavna klasa, koja predstavlja slučajnu šumu, se nalazi u `RandomForest.java`. Konstrukcija šume započinje zadavanjem željenog broja stabala u šumi. Nakon toga pomoću metode predajemo set podataka za treniranje šume. Pomoću razreda Data doznajemo korisne informacije o skupu za treniranju te ih spremamo u članske varijable slučajne šume. Zatim se za svako stablo vadi *bootstrap* uzorak pomoću metode `getBootstrapDataSample`. Na temelju *bootstrap* uzorka uči se jedno stablo pomoću metode fit. Nakon izgradnje, *N* stabala, slučajna šuma je spremna za predikciju novih rezultata. Šuma radi predikcije pomoću glasanja. Algoritam provuče set atributa za predikciju kroz svako stablo u sumi te zbraja glasove. Ona ciljna klasa sa najviše glasova je klasa za koju će slučajne šume glasati.

Za predviđanje jednog seta atributa koristi se metoda `predictLabel`. Za slučaj kada nas zanima rezultat svih stabala koristeći metodu `predictMap` dobivamo mapu koja za svaku ciljnu klasu vraća broj stabala koji su glasali za nju. Za slučaj izračuna unakrsne validacije stabala na temelju OOB skupa prilikom učenja šume, koristi se metoda `trainCrossValidation`.

Implementirane su dvije metode koje evaluiraju algoritam. Obje rade evaluaciju na temelju unakrsne validacije. Metoda `evaluateWithKfoldCV` koristi k-struku unakrsnu validaciju dok metoda `evaluateWithHoldOut` unakrsnu sa zadržavanjem dijela podataka za treniranje. Unakrsna validacija sa zadržavanjem, dijeli podatke u omjeru 2:1 gdje se veći podskup koristi za učenje.

K-struka unakrsna validacija dijeli set podataka na k jednakih dijelova. Svaki k-ti dio predstavlja jedan skup za testiranje, dok ostatak predstavlja skup za treniranje.

Gradi se k slučajnih šuma i računaju matrice konfuzije sa svaki od k šuma pomoću metode `confusionMatrix`. Sve matrice se zbrajaju pomoću metode `addConfMat` kako bi se dobila konačna matrica konfuzija za unakrsnu validaciju. Uz to, računa se prosječno vrijeme potrebno za izgradnju jednog modela slučajne šume.

```
public class RandomForest {

	int numberOfTrees;
	ArrayList<RandomTree> trees;
	Data trainingData;
	HashSet<String> labels;
	Integer numberOfFeatures;
	HashSet<Integer> discreteFeatures;

	public RandomForest(int numberOfTrees) throws NumberFormatException,
IOException, ParseException{

		this.numberOfTrees = numberOfTrees;
		trees = new ArrayList<>();
	}
	public void fit(Data trainingData){

		this.trainingData=trainingData;
		this.labels = trainingData.getPossibleLabels();
		this.numberOfFeatures = trainingData.getNumberOfFeatures();
		this.discreteFeatures = trainingData.findDiscreteValues( 0.1 );

		for(int i=0;i<numberOfTrees;i++){

			RandomTree ft = new RandomTree(999, 2, numberOfFeatures,
labels,discreteFeatures);
			ft.fit(getBootstrapDataSample(trainingData));
			trees.add(ft);
		}
	}

	public HashMap<String, Integer> predictMap(ArrayList<String> observation){…}

	public static double evaluateWithHoldOut(Data trainingData) throws
NumberFormatException, IOException, ParseException{…}

	public static double evaluateWithKfoldCV (Data trainingDataa, int kFolds)
			throws NumberFormatException, IOException, ParseException {…}

	public Double trainCrossValidation(Data trainingData){…}

	public String predictLabel(ArrayList<String> observation){…}

	public HashMap<String,HashMap<String,Integer>> confussionMatrix(Data
testData){…}

	public static HashMap<String,HashMap<String,Integer>>
addConfMat(HashMap<String,HashMap<String,Integer>>
m1,HashMap<String,HashMap<String,Integer>> m2, HashSet<String> lbls){…}

	public static Data getBootstrapDataSample(Data population){…}
}
```

Slika 4.8 Isječak razreda RandomForest

5. Zaključak

Algoritam slučajne šume daje vrlo impresivne rezultate naspram potrebnih parametara za podešavanje. Kod ostalih algoritama, koji znaju biti bolji no ne značajno, često zna biti mukotrpan posao pronalazak optimalnih parametara za model. Stoga vrijeme potrebno za izgradnju je puno kraće te je često pametno koristi slučajne šume za brzu analizu skupa podataka.

Uglavnom nije potrebno predprocesuirati podatke. Algoritam je sposoban sam zamijeniti i nedostajuće vrijednosti. Radi s kategoričkim i kontinuiranim atributima, bez potrebe za skaliranjem.

Sama optimizacija brzine izvođenja dolazi prilikom optimizacije preciznosti. Pronalaskom slučajnog podprostora svih atributa za izgradnju pojedinog čvora, smanjuje se prostor pretrage rješenja za podjelu. Zauzvrat, uz potrebnu de-korelaciju međusobnih stabala, algoritam je u mogućnosti raditi s velikim brojem atributa.

Slučajne šume su također sposobne provesti implicitan odabir značajnih atributa, što je uvijek poželjno znati za interpretaciju skupa podataka. Moguće je provesti uz kriterij Gini ili slučajnu permutaciju vrijednosti atributa iz *OOB* skupa kako bi se utvrdili značajni atributi.

Naravno, algoritam ima i svoje nedostatke. Ponekad za rješavanje problema nije dovoljna sama klasifikacija nego i interpretacija rezultata. Algoritam se ponaša kao crna kutija što rezultira nemogućnosti interpretacije rješenja.

U slučaju da želimo model kojemu je brzina predviđanja podatka ključna, često je bolje izabrati neki drugi model. Razlog tome su mnogobrojna stabla odluke u samoj šumi. To rezultira s većim vremenom potrebnim za donošenje predikcije. Ako želimo precizniji model, koristit ćemo što više stabala, no to uvjetuje sporiju predikciju.

U slučaju da nam je potrebna velika preciznost, te vrijeme izgradnje modela nije presudno, bolje bi se bilo odlučiti za neki drugi model baziran na *boostingu* ili iz ostalih skupa algoritama iz strojnog učenja.

Glavni problem s kojim sam se susreo prilikom implementacije svog programskog rješenja jest brzina izgradnje modela. Paket WEKA, te implementacija u scikit.learn za programski jezik python, sadrže programska rješenja sa signifikantno boljim vremenima izgradnje modela.

Unakrsnom validacijom algoritma u odnosu na onu iz paketa WEKA, dobivao sam nešto manju razinu preciznosti, ali ponekad i veću. Ovisno o slučajnosti, moji rezultati su podosta varirali dok njihova nisu. Isprva se činilo da promjenom vrijednosti sjemenke slučajnosti, njihovi rezultati će se isto mijenjati. Međutim ispostavilo se da su se rijetko kada mijenjali, a kada je i došlo do promjene standardno odstupanje je bilo vrlo malo u odnosu na moje.

Nažalost, nisam pronašao koje optimizacijske algoritme koriste prilikom izgradnje modela. To je zasigurno najveći nedostatak moje implementacije. Zasigurno njihova metoda koja traži kriterij podijele ne računa indeks Gini na isti način kao i ja. Njihovo kraće vrijeme izgradnje modela je uvjetovano drugačijim računanjem kriterija, a samim time odabir atributa i vrijednosti za podjelu su drugačiji. Što znači da se uzrok manjoj preciznosti i sporosti krije u istom problemu.

Daljnja moguća implementacija će svakako biti prilagodba modela za regresijske probleme. Nadalje, odabir značajnih atributa te zamjena nedostajućih vrijednosti su važne sposobnosti slučajnih šuma koje bi svakako bilo pametno implementirati. Naravno, glavni cilj je riješiti navedene probleme. Također, planiram implementirati i neke algoritme zasnovane na principu *boosting*.

LITERATURA

(Alpaydin, 2009) Alpaydin, E., Introduction to Machine Learning. Second Edition. Cambridge: The MIT Press. 2009 .
http://cs.du.edu/~mitchell/mario_books/Introduction_to_Machine_Learning_-_2e_-_Ethem_Alpaydin.pdf

(Breiman, 2001) Breiman, L., Random Forests. Machine Learning, Volume 45, Issue 1, pp. 5–32,
2001.https://www.stat.berkeley.edu/~breiman/randomforest2001.pdf

(Dietterich, 2000) Dietterich, T.G., An Experimental Comparison of Three Methods for Constructing Ensembles of Decision Trees:
Bagging, Boosting, and Randomization. Machine Learning, Volume 40, Issue 2, pp. 139–157, 2000.
http://www.csd.uwo.ca/faculty/ling/cs860/papers/mlj-randomized-c4.pdf

(Hastie 2009) Hastie, Tibshirani , Friedman, T., R., J., The Elements of Statistical Learning: Data Mining, Inference, and Prediction – Second Edition. Stanford: Springer. 2009.
https://statweb.stanford.edu/~tibs/ElemStatLearn/printings/ESLII_print10.pdf

(Nicoguaro, 2016) Nicoguaro - Own work, CC BY 4.0,
https://commons.wikimedia.org/w/index.php?curid=46257808

(Witten, 2011) Witten, IH., Frank, E., Hall. M.,, Data Mining: Practical Tools and Techniques. Third Edition. Burlington: Morgan Kaufmann Publishers, 2011.

Postupci ansambla stabala odluke

Sažetak

Ansambli stabala odluke su modeli strojnog učenja zasnovani na višestrukim osnovnim modelima, stablima odluke. Postoje dvije glavne metode za izgradnju ansambla: *boosting* i *bagging*. *Boosting* iterativno gradi stabla tako da nove iteracije ovisi o prošloj iteraciji, dok stabla izgrađena *bagging* metodom ovise o *bootstrap* uzorku za učenje. Prilikom glasanja konačnog ansambla, *boosting* provodi težinsko glasanje dok *bagging* ne. Jedna implementacija *bagging* metode su slučajne šume, koje dodatno uvode slučajnost prilikom odabira atributa pri izgradnji stabla. U ovom radu opisana je izgradnja slučajne šume, ostvarena je programska implementacija metode te je provedena eksperimentalna usporedba vrednovanja rezultata između ovog ostvarenja i alata Weka na nekoliko skupova podataka.

Ključne riječi: slučajne; šume; stablo; odluke, klasifikacija; strojno; učenje; ansambl; boosting; bagging

Decision Tree Ensemble Methods

Abstract

Ensemble methods are based on combining multiple base models. There are two main methods for building an ensemble: boosting and bagging. Base models are constructed iteratively in boosting, where each iteration depends on the performance of the previous one. In bagging, base models are constructed upon bootstrap sample from the training data. When the ensemble makes a decision in boosting each base model casts a weighted vote, whereas in bagging each vote counts the same. Random forests are modification of the bagging method where each CART tree is additionally randomized by selecting a random subspace of the attributes for the split. In this work, random forest algorithm is discussed, an implementation is provided and the results of the implementation are compared to those of WEKA.

Keywords: random; forest; decision; tree; classification; machine; learning; ensemble; boosting; bagging