SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 5927

Implementacija klasifikacijskog algoritma podizanja gradijenta stablima odluke s primjenom na predikciju ishoda nogometnih utakmica

Dominik Hrastić

Zagreb, lipanj 2019.

Sadržaj

1. Uvod

Strojno učenje grana je umjetne inteligencije koja se bavi oblikovanjem algoritama koji svoju učinkovitost poboljšavaju na temelju empirijskih podataka. Jedno je od danas najaktivnijih i najuzbudljivijih područja računarske znanosti, ponajviše zbog brojnih mogućnosti koje se protežu od raspoznavanja uzoraka i dubinske analize podataka do robotike, računalnog vida i bioinformatike. Postoji više vrsta strojnog učenja[12], no u ovom radu fokus će biti na nadziranom učenju, a bavit će se klasifikacijom koja se koristi za predviđanje diskretnih vrijednosti.
Razvojem IT industrije moguće je prikupiti sve više različitih podataka. Upravo zbog toga strojno učenje ima i sve veći utjecaj u sportu. Do sada, analiza podataka uspješno je primjenjena na sportove poput košarke i baseballa. Međutim, u nogometu je oduvijek bila ograničena upotreba analize podataka i strojnog učenja unatoč tome što je najpopularniji sport na svijetu. Naime, čak 4 od 10 ljudi prati nogomet te smatraju sebe navijačima[1]. Analiza podataka dugo nije imala utjecaj na nogomet prvenstveno zato jer se nogomet smatrao previše kaotičnim i fluidnim te se vjerovalo da je nemoguće predvidjeti ishod utakmica ili performansu igrača jer na nju utječe mnoštvo stvari. Tako je nedavno napravljeno i istraživanje kako na igrače i samu ekipu utječu objave na društvenoj mreži Twitter[18].
Predviđanje ishoda nogometnih utakmica ili ocjenjivanje izvedbe igrača često se oslanja na analizu i ocjenu nogometnih stručnjaka. Međutim, to često može voditi do subjektivnih zaključaka u kojima ima dosta odskakanja. Nadalje, stručnjaci ne izlažu kriterije kojima dolaze do zaključaka te samim time često nisu pouzdani. No, u zadnje vrijeme, razvojem tehnologije, moguće je pratiti mnoštvo stvari usred utakmice, na treningu ili u nekim simuliranim stresnim situacijama te se samim time može prikupiti sve više podataka koji se mogu analizirati. Od tehnologije koja omogućuje analizu i predviđanje izvedbe pojedinca do predviđanja ishoda utakmica i stvaranja novih strategija za protivničke ekipe, strojno učenje primjenjivo je na mnogo aspekata igre te nogomet stvarno postaje sport

vođen podacima (engl. *data driven* sport).
Tako se u ovom radu, analizom podataka opisanih u poglavlju 2 pokušava doći do što bolje predikcije ishoda nogometnih utakmica 1. engleske lige (EPL). Podaci su preuzeti sa stranice [10]. Kako bi se došlo do željenog rezultata implementirat će se klasifikacijski algoritam podizanja gradijenta stablima odluke.

2. Podaci

2.1. Korišteni podaci

Prva engleska nogometna liga (EPL) sastoji se od 20 ekipa. Svaka momčad odigra 38 utakmica po sezoni, sa svakom ekipom po 2 puta (jednom na domaćem, drugi put na gostujućem terenu). To znači da se u jednoj sezoni odigra ukupno 380 utakmica.
Podaci koji se koriste u ovom radu su iz sezone 2016./'17 i 2017./'18. Statističke kategorije koje se nalaze u navedenoj grupi su:

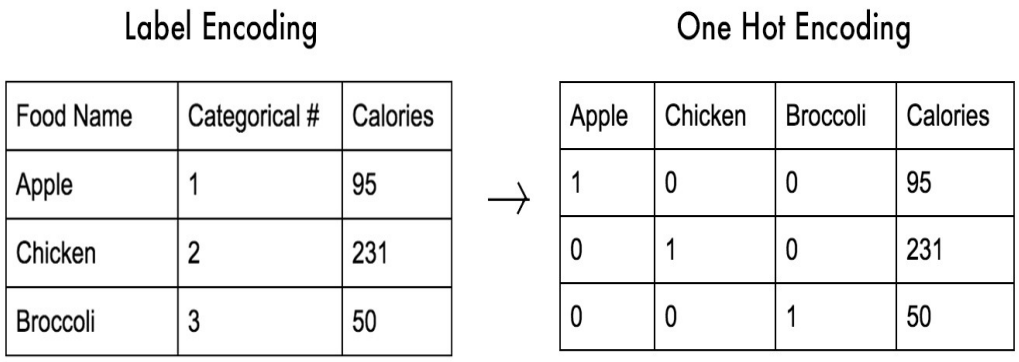
- *HomeTeam* - domaća momčad
- *AwayTeam* - gostujuća momčad
- HTHG (engl. *half time team goals*) - golovi domaće momčadi na poluvremenu
- HTAG (engl. *half time away goals*) - golovi gostujuće momčadi na poluvremenu
- HTR (engl. *half time result*) - rezultat utakmice na poluvremenu
- *Attendance* - broj navijača na tribinama
- *Referee* - sudac na utakmici
- HS (engl. *home team shots*) - udarci domaće ekipe
- AS (engl. *away team shots*) - udarci gostujuće ekipe
- HST (engl. *home shots on target*) - udarci domaće ekipe u okvir gola
- AST (engl. *away shots on target*) - udarci gostujuće ekipe u okvir gola
- HHW (eng. *home hit woodwork*) - udarci domaće ekipe u vratnice
- AHW (eng. *away hit woodwork*) - udarci gostujuće ekipe u vratnice
- HC (eng. *home corners*) - udarci iz kuta za domaću ekipu

- AC (eng. *away corners*) - udarci iz kuta za gostujuću ekipu
- HF (eng. *home fouls*) - prekršaji domaće ekipe
- AF (eng. *away fouls*) - prekršaji gostujuće ekipe
- HFKC (eng. *home free kicks conceded*) - broj slobodnih udaraca domaće ekipe
- AFKC (eng. *away free kicks conceded*) - broj slobodnih udaraca gostujuće ekipe
- HO (eng. *home offsides*) - broj zaleđa domaće ekipe
- AO (eng. *away offsides*) - broj zaleđa gostujuće ekipe
- HY (eng. *home yellow*) - broj žutih kartona domaće ekipe
- AY (eng. *away yellow*) - broj žutih kartona gostujuće ekipe
- HR (eng. *home red*) - broj crvenih kartona domaće ekipe
- AR (eng. *away red*) - broj crvenih kartona gostujuće ekipe
- HBP (eng. *home booking points*)
- ABP (eng. *away booking points*)

U originalnoj datoteci, uz sve navedene kategorije nalazili su se još i podaci o koeficijentima u raznim kladionicama koji su izbačeni iz podataka koji se koriste u analizi jer i bez njih postoji puno različitih atributa. Samim time i smanjujemo mogućnost prenaučenosti, o čemu će više riječi biti kasnije. Također, izbačeni su i završni rezultati utakmica iz očitog razloga.

2.2. Priprema podataka

Podaci su, za samu analizu, pripremljeni u programskom jeziku Python(verzija 3.5) koristeći biblioteke *pandas* i *sklearn*[17]. U pripremi podataka izbačeni su podatci o koeficijentima kladionica. Nadalje, u skupu podataka postoje i kategorički podatci koje algoritmi strojnog učenja ne znaju obrađivati te ih je potrebno prevesti u algoritmima prihvatljiviji oblik.
Kako tog problema ne bi bilo, značajke (engl. *features*) su kodirane procesom *One hot encoding*, dok su labele (engl. *labels*) kodirane procesom *label encoding* kao što je prikazano na slici 2.1. Oba procesa zamjenjuju kategoričke vrijednosti cjelobrojnim vrijednostima kako bi algoritam mogao raditi s njima.



Slika 2.1: Kodiranje varijabli procesom *One hot encoding*[7]

Labele su kodirane pomoću razreda **LabelEncoder** iz biblioteke *sklearn* dok značajke nisu kodirane sve, već samo one koje imaju kategoričke vrijednosti. Te značajke su: HomeTeam, AwayTeam, HTR i Referee. Implementacija kodiranja procesima *one hot encoding* i *label encoding* može se vidjeti u listingu 2.1.

Listing 2.1: Priprema podataka

```
# Label Encoding labels
label_encoder = LabelEncoder()
label_encoder = label_encoder.fit(Y.values.ravel())
label_encoded_y = label_encoder.\
    transform(Y.values.ravel())

# manually one hot encoding
# HomeTeam, AwayTeam, Referee and HTR columns
one_hot = pd.get_dummies(X['HomeTeam'])
X = X.drop('HomeTeam', axis=1)
X = X.join(one_hot)

one_hot1 = pd.get_dummies(X['AwayTeam'])
X = X.drop('AwayTeam', axis=1)
X = X.join(one_hot1, lsuffix='_home',
           rsuffix='_away')

one_hot2 = pd.get_dummies(X['HTR'])
X = X.drop('HTR', axis=1)
X = X.join(one_hot2)
```

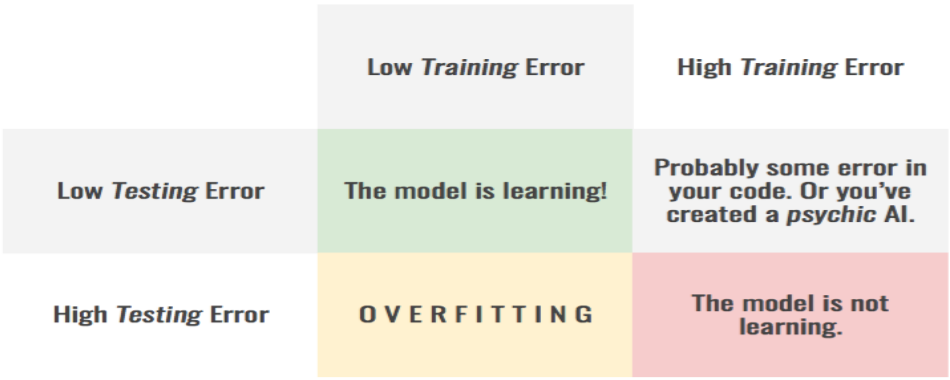
```
21
22      one_hot3 = pd.get_dummies(X['Referee'])
23      X = X.drop('Referee', axis=1)
24      X = X.join(one_hot3)
```

2.3. Prenaučenost

Prenaučenost (engl. *overfitting*) je jedan od najvećih problema strojnog učenja. Razni su razlozi zbog kojih algoritam može biti prenaučen no najčešći su prevelika složenost istih ili premalo podataka. U tom slučaju algoritmi ne generaliziraju već nauče očekivani ishod za svaku značajku. Upravo zbog tog svojstva, prenaučenost možemo uočiti. Naime, u slučaju prenaučenosti algoritam ima male pogreške na skupu za učenje, dok su velike pogreške na skupu za testiranje koje prije nije "vidio". Ovisnost modela o pogreškama skupa za učenje i testiranje može se vidjeti na slici 2.2.

Kako bi se provjerilo je li došlo do prenaučenosti u algoritmu, ulazni podaci se dijele na dva dijela pomoću funkcije *train_test_split* koja prvo promiješa podatke (engl. *shuffle data*) te zatim, ovisno o parametru *test_size* koji može biti u rasponu [0, 1], a predstavlja postotak, podijeli podatke na 2 dijela. Veći dio podataka uvijek se odvaja za učenje, dok se manji dio čuva za testiranje. Tim postupkom osiguravamo testiranje algoritma s njemu neviđenim podatcima. Na taj način možemo provjeriti dolazi li do prenaučenosti.

U ovom radu, podaci za testiranje čine 20% ukupnih podataka, dok podaci za učenje algoritma čine 80% ukupnih podataka. Kako bi se podaci uspjeli razdijeliti na ove postotke piše se funkcija *train_test_split* čiji izgled se može vidjeti u listingu 2.2.



Slika 2.2: Prenaučenost[8]

Listing 2.2: Razdvajanje podataka

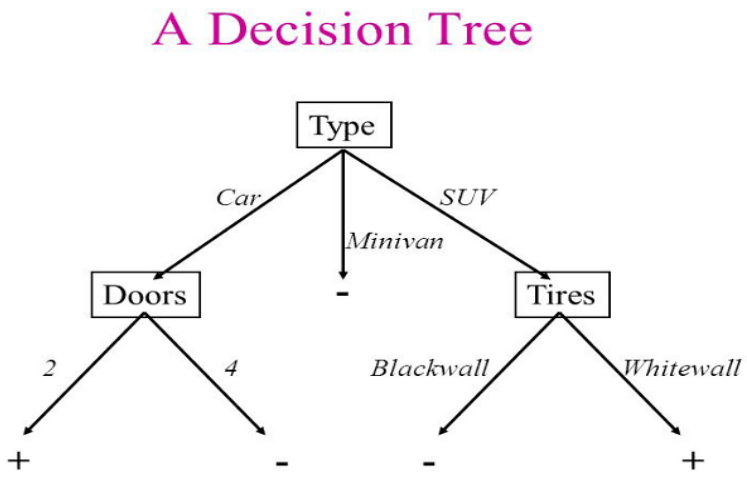
```
def train_test_split(X, y, test_size=0.5, seed=None):
    #get random Xs and ys
    np.random.seed(seed)

    #shuffle numbers 0 - number of columns
    #shuffle X and y array with that field
    x_id = np.arange(X.shape[0])
    np.random.shuffle(x_id)
    X, y = X[x_id], y[x_id]

    # Split the data with split index
    length = X.shape[0]
    split_index = length - int(test_size * length)
    X_train, X_test = X[:split_index], X[split_index:]
    y_train, y_test = y[:split_index], y[split_index:]
```

Funkcija train_test_split kao ulazne podatke prima značajke (X), labele (y) te u kojem omjeru se podaci žele podijeliti (*test_size*), odnosno koji postotak podataka će biti skup podataka za test. Zadana vrijednost varijable *test_size* je 0.5, no kao što je to već spomenuto u ovom radu će se u tu svrhu koristiti vrijednost 0.2.

U ovom radu, prenaučenost se izbjegava podešavanjem parametara, o čemu je više napisano u poglavlju 4.2.

3. Stabla odluke

Stablo ima mnogo analogija u stvarnom životu. Imalo je i velikog utjecaja na široko područje strojnog učenja zbog svoje mogućnosti da radi i s klasifikacijskim i regresijskim algoritmima. Stablo u IT-programiranju predstavlja strukturu podataka, po uzoru na pojam stabla u teoriji grafova. Točnije, ovdje se govori o *stablima odluke*. Stablo odluke ima strukturu dijagrama toka gdje svaki čvor predstavlja ispitivanje pojedinog atributa, a svaka grana pravilo. Grananje se odvija sve dok se ne dođe do odluke. Čvor odluke je zapravo list (engl. *leaf*) stabla odluke. Primjer stabla odluke može se vidjeti na slici 3.1.



Slika 3.1: Stablo odluke[2]

Stabla odluke mogu raditi i s klasifikacijskim i regresijskim algoritmima zbog čega i jesu uvelike upotrebljivi u svijetu strojnog učenja. Ovaj rad će se fokusirati na implementaciju klasifikacijskih stabala odluke s obzirom na skup podataka kojim se radi. Naime, klasifikacijski algoritam koristi se za skup podataka u kojem je predviđanje diskretna, odnosno nebrojčana vrijednost.

3.1. Izgradnja klasifikacijskog stabla odluke

```
ID3 (Examples, Target_Attribute, Attributes)
    Create a root node for the tree
    If all examples are positive, Return the single-node tree Root, with label = +.
    If all examples are negative, Return the single-node tree Root, with label = -.
    If number of predicting attributes is empty, then Return the single node tree Root,
    with label = most common value of the target attribute in the examples.
    Otherwise Begin
        A ← The Attribute that best classifies examples.
        Decision Tree attribute for Root = A.
        For each possible value, v_i, of A,
            Add a new tree branch below Root, corresponding to the test A = v_i.
            Let Examples(v_i) be the subset of examples that have the value v_i for A
            If Examples(v_i) is empty
                Then below this new branch add a leaf node with label = most common target value in the examples
            Else below this new branch add the subtree ID3 (Examples(v_i), Target_Attribute, Attributes – {A})
    End
    Return Root
```

Slika 3.2: ID3 algoritam[5]

Za izgradnju stabla odluke u ovom radu koristi se algoritam ID3. Kako bi se moglo izgraditi, stablo odluke mora imati svoj korijen (engl. *root*). Kao korijen stabla odabiremo atribut koji koji ima najveću informacijsku dobit. Informacijska dobit atributa je očekivana redukcija entropije uzrokovana podijelom primjera za učenje u skladu s tim atributom, tj. mjera kako pojedini atribut odjeljuje primjere za učenje u skladu s ciljnom klasifikacijom. Informacijska dobit računa se prema formuli:

$$IG(A, S) = H(S) - \sum_{t \in T} p(t)H(t) \tag{3.1}$$

gdje je:

- H(S) entropija skupa podataka S
- T skupovi nakon razdvajanja skup S po atributu A ($S = \cup_{t \in T} t$)
- p(t) omjer broja elemenata u t s ukupnim brojem elemenata
- H(t) entropija podskupa t

Implementacija računanja entropije može se vidjeti u listingu 3.1.

Listing 3.1: Stablo odluke

```
1 def entropy_calc(x):
2     entropy = 0
3     unique_labels = np.unique(x)
4     for label in unique_labels:
```

```
        label_len = len(x[x==label])
        p = label_len / len(x)
        entropy += -p * (math.log(p) / math.log(2))
    return entropy
```

Entropija u svijetu strojnog učenja predstavlja mjeru homogenosti primjera, tj. stupanj nečistoće podataka. Upravo zbog toga poželjno je da entropija primjera bude što bliže 0. Vrijednosti entropije kreću se u intervalu [0,1]. Smatra se niskom ako je bliže nuli. Računa se po formuli:

$$H(S) = \sum_{c \in C} -p(c) * \log_2 p(c) \tag{3.2}$$

gdje je:

- S skup podataka
- C skup klasa u S
- p(c) omjer broja elemenata u klasi c s ukupnim brojem elemenata

Nakon što se odabere atribut s najvećom informacijskom dobiti, taj atribut postavlja se za korijen stabla odluke. Njegov broj grana ovisi o broju vrijednosti tog atributa. Nakon grananja, svaki čvor predstavlja podskup skupa podataka koji odgovora vrijednosti tog atributa. Postupak biranja atributa s najvećom informacijom ponavlja se rekurzivno za svako grananje, odnosno za svaki čvor i nikad se ne vraća zbog razmatranja prethodnih čvorova što znači da ID3 spada u pohlepne algoritme. U ovom primjeru ID3 provodi pretraživanje "najbolji prvi" (engl. *greedy best-first search* pretraživajući prostor svih mogućih stabala odluke. Grananje prestaje kada se dođe do čvora u kojem su svi primjeri iz iste klase. Cijeli postupak vidljiv je na primjeru pseudokoda na slici 3.2.
Nadalje, stablo odluke ima niz parametara kojima se može poboljšati njegova izvedba. Neki od parametara su:

- *max_features* - broj značajki za razmotriti kad se traži najbolje grananje
- *max_leaf_nodes* - maksimalan broj listova stabla odluke
- *max_depth* - maksimalna dubina stabla odluke (najveća razina)

Ostali parametri mogu se naći na stranici [17].
U ovom radu koristit će se parametar *max_depth* koji predstavlja najveću razinu stabla, odnosno najdulji put od korijena do lista stabla. Njime se koristi kako bi se stablo ranije podrezalo i time sprječila prenaučenost. Implementacija čvora stabla odluke vidljiva je u listingu 3.2.

Listing 3.2: Čvor

```
class Node():
    """
    Represents a node of a decision tree.

    :param: attribute_index: Attribute index that
                             is used as threshold measurement
    :param: threshold: Value that attributes at
                    attribute index will be compared to
    :param: value: class prediction
    :param: left_tree_build: Left subtree
    :param: right_tree_build: Right subtree

    """
    def __init__(self, attribute_index=None, threshold=None,
                 value=None, left_tree_build=None,
                 right_tree_build=None):
        self.attribute_index = attribute_index
        self.threshold = threshold
        self.value = value
        self.left_tree_build = left_tree_build
        self.right_tree_build = right_tree_build
```

Nakon čvora stabla odluke, implementirano je samo stablo odluke postupkom opisanim u ovom poglavlju, algoritmom ID3. Kod implementacije može se vidjeti u listingu 3.3.

Listing 3.3: Stablo odluke

```
class DecisionTree(object):
    """
    Represents a decision tree, built with ID3 algorithm.

    :param: max_depth: Maximum depth of tree
    :param: min_info_gained: Minimum information gained
                             from a certain node
    :
    """
    def __init__(self, max_depth=5, min_info_gained=sys.float_info.min):
        self.root = None
```

```
12          self.max_depth = max_depth
13          self.min_info_gained = min_info_gained
14
15      def fit(self, attributes, results):
16          self.root = self._build_tree(attributes, results)
17
18
19      def _gained_info(self, results, left_results, right_results):
20          """
21          Calculate information gain.
22          """
23          # left tree possibility
24          p = len(left_results) / len(results)
25          # right tree possibility
26          q = 1 - p
27          entropy = entropy_calc(results)
28          left_entropy = entropy_calc(left_results)
29          right_entropy = entropy_calc(right_results)
30          gained_info = entropy - p * left_entropy - q * right_entropy
31
32          return gained_info
33
34      def _avg_results(self, results):
35          """
36          Calculate leaf value
37          """
38          avg = np.average(results, axis=0)
39          return avg
40
41      def _build_tree(self, attributes, results, current_depth=0):
42          """
43          Recursive ID3 algorithm that builds a tree.
44          """
45          most_info_gained = 0
46          best_criteria = None
47          best_attributes = None
48
49          #concatenate attributes and results
```

```
50          all_data = np.concatenate((attributes, results), axis=1)
51
52          n_samples, n_attributes = np.shape(attributes)
53          if current_depth <= self.max_depth:
54              #How much info is gained from each attribute?
55              for attribute_index in range(n_attributes):
56                  attribute_values = np.expand_dims(
57                      attributes[:, attribute_index], axis=1)
58                  unique_values = np.unique(attribute_values)
59
60                  for threshold in unique_values:
61                      #split tree on left and right side
62                      #left if sample value is less than threshold
63                      #right if value is greater or equal
64
65                      if isinstance(threshold, int) or \
66                              isinstance(threshold, float):
67
68                          split_func = lambda sample: \
69                              sample[attribute_index] >= threshold
70                      else:
71                          split_func = lambda sample:\
72                              sample[attribute_index] == threshold
73
74                      left = np.array([sample for sample in all_data
75                                       if split_func(sample)])
76                      right = np.array([sample for sample in all_data
77                                        if not split_func(sample)])
78
79                      left_tree, right_tree = np.array([left, right])
80
81                      if len(left_tree) > 0 and len(right_tree) > 0:
82
83                          left_results = left_tree[:, n_attributes:]
84                          right_results = right_tree[:, n_attributes:]
85
86                          info_gained = self._gained_info(
87                              results, left_results, right_results
```

```
                        )
                        if info_gained > most_info_gained:

                            most_info_gained = info_gained
                            best_criteria = {
                                "attribute_index": attribute_index,
                                "threshold": threshold
                            }
                            best_attributes =  {
                                "leftAttributes":
                                    left_tree[:, :n_attributes],
                                "leftResults":
                                    left_tree[:, n_attributes:],
                                "rightAttributes":
                                    right_tree[:, :n_attributes],
                                "rightResults":
                                    right_tree[:, n_attributes:]
                            }

        if most_info_gained > self.min_info_gained:

            left_tree_build = self._build_tree(
                best_attributes["leftAttributes"],
                best_attributes["leftResults"], current_depth+1
            )
            right_tree_build = self._build_tree(
                best_attributes["rightAttributes"],
                best_attributes["rightResults"], current_depth+1
            )

            return Node(attribute_index=best_criteria["attribute_index"],
                        threshold=best_criteria["threshold"],
                        left_tree_build=left_tree_build,
                        right_tree_build=right_tree_build)

        #determine the value if we are at leaf
        leaf = self._avg_results(results)
        return Node(value=leaf)
```

Nakon što je stablo naučeno, treba se moći predvidjeti ishod temeljem ulaznih podataka koji predstavljaju atribute. Ta funkcionalnost implementirana je metodom *predict*, koja je vidljiva u listingu 3.4.

Listing 3.4: Predviđanje stabla odluke

```
def _prediction(self, attributes, tree=None):
    """
    returns a prediction for a certain part of attributes. (0, 1, 2)
    """
    if tree is None:
        tree = self.root

    #if we are at leaf, return prediction
    if tree.value is not None:
        return tree.value

    attribute_value = attributes[tree.attribute_index]

    if attribute_value >= tree.threshold:
        path = tree.left_tree_build
    else:
        path = tree.right_tree_build

    return self._prediction(attributes, path)

def predict(self, attributes, subtree=None):
    predictions  = [self._prediction(part) for part in attributes]
    return predictions
```

4. Algoritam podizanja gradijenta

Ansambli su često korišteni mehanizmi pri izgradnji algoritama strojnog učenja. Ansambli kombiniraju više osnovnih klasifikatora (engl. *base learners*) u jedan **meta-klasifikator** te tako dolaze do boljeg učinka[14]. U ovom slučaju, osnovni klasifikatori su stabla odluke, o kojima je više napisano u poglavlju 3 te će se stoga, u daljnjem tekstu, umjesto pojma osnovni klasifikator koristiti pojam stabla odluke.
Ansambli se mogu implementirati na puno načina i kombinirati mnogo različitih algoritama strojnog učenja. U ovom radu implementiran je algoritam podizanja (engl. *boosting*). Točnije, implementirana je verzija algoritma podizanja - algoritam podizanja gradijenta (engl. *gradient boosting*).
Više o metodama ansambala te samom algoritmu podizanja može se pročitati u članku [3].

4.1. Podizanje stabala odluke

Algoritam podizanja stabala odluke temelji se na slijednom učenju algoritama na pogreškama prethodnih algoritama što znači da se stabla odluke ne grade zasebno, već slijedno. Svako novo stablo odluke "uči" na pogreškama koje su napravila prijašnja stabala odluke. Učeći na greškama prethodnih stabala, algoritmu treba manje vremena (iteracija) da se predvidi rezultat nego što bi trebalo jednom stablu odluke. Postoji više vrsta algoritma podizanja. Najpoznatiji, odnosno najčešće korišteni su AdaBoost (engl. *Adaptive Boosting*), algoritam podizanja gradijenta (engl. *Gradient Boosting*) i ekstremno podizanje gradijenta - XGBoost (*eXtreme Gradient Boosting*).
Razlika između ovih algoritama je u načinu "učenja" novih klasifikatora na pogreškama prijašnjih. Kod algoritma *AdaBoost* svaki klasifikator je učen sa slučajno odabranim podskupom skupa za učenje algoritma gdje se podskupovi mogu preklapati. Zatim se svakom primjeru pridjeljuje određena težina. Nakon što je kla-

sifikator učen, algoritam povećava težine primjera koji su krivo klasificirani[15]. S druge strane, algoritam podizanja gradijenta (engl. *Gradient Boosting*), kao što mu i samo ime govori, umjesto dodjeljivanja težina primjerima za učenje, koristeći gradijente u funkcijama gubitka, o čemu se više može pročitati u poglavlju 4.1.1.
Više o vrstama algoritma podizanja može se pročitati u članku [16].

4.1.1. Algoritam podizanja gradijenta

U algoritmu podizanjem gradijenta (engl. *Gradient Boosting*) stabla se grade slijedno. U svakoj rundi učenja izgradi se jedno stablo odluke te se predviđanje istog usporedi sa stvarnim rezultatom. Razliku između predikcije algoritma i stvarnog rezultata možemo predstaviti funkcijom gubitka. U klasifikacijskim algoritmima funkciju gubitka predstavlja "cijena" koja je plaćena za pogrešku u klasifikaciji, odnosno za netočnost predviđanja, a glavni cilj je tu funkciju minimizirati.
Ako X predstavlja prostor svih mogućih ulaznih podataka, a Y=[-1, 1] predstavlja prostor svih mogućih izlaznih podataka, želimo naći $f : X \mapsto R$ koja najbolje pridjeljuje vrijednosti $x \in X$ vrijednostima $y \in Y$[6].
Funkcije gubitka mogu biti razne i razlikuju se između regresijskih i klasifikacijskih problema. U ovom slučaju, budući da je problem klasifikacijski, kao funkciju gubitka uzima se unakrsna entropija (engl. *Cross Entropy*), poznatija i kao logaritamska funkcija gubitka (engl. *Log Loss*). Ako bi razmišljali o distribuciji kao alatu koji koristimo da kodiramo simbole, onda entropija mjeri broj bitova koje ćemo trebati ako bi koristili neki alat y. S druge strane, unakrsna entropija predstavlja broj bitova koje ćemo trebati ako kodiramo simbole iz skupa y krivim alatom[9].
Na primjer, pretpostavimo da je za neki primjer iz skupa za učenje točni ishod pobjeda domaćeg tima. Tada bi *one-hot* distribucija tog primjera iznosila kao što je prikazano u tablici 4.1.

Tablica 4.1: Točna distribucija vjerojatnosti

P(izjednačeno)	P(pobjeda domaći)	P(pobjeda gosti)
0.0	1.0	0.0

Gornji primjer može se imterpretirati tako da znači da je vjerojatnost izjednačenog ishoda 0%, vjerojatnost pobjede domaćeg tima je 100%, dok je vjerojatnost

pobjede gostujućeg tima 0%. Nadalje, pretpostavimo da je ovaj model predvidio distribuciju vjerojatnosti kao što je prikazano na slici 4.2.

Tablica 4.2: Predviđena distribucija vjerojatnosti

P(izjednačeno)	P(pobjeda domaći)	P(pobjeda gosti)
0.228	0.619	0.153

Kako bi se dobila funkcija gubitka unakrsne entropije, koristi se formula:

$$H(y,p) = -\sum_i y_i log(p_i) + (1-y_i) log(1-p_i) \tag{4.1}$$

gdje je:

- y_i - željena vjerojatnost
- p_i - predviđena vjerojatnost

Uvrštavanjem brojeva iz tablica 4.1 i 4.2 dobije se vrijednost 0.812, koja predstavlja koliko je predviđanje "krivo", odnosno koliko je daleko od točne distribucije vjerojatnosti[22]. Kako bi te pogreške bile što manje, traži se gradijent *Cross Entropy* funkcije gubitka, odnosno prva derivacije iste. Derivaciju izraza 4.1, računamo prema izrazu:

$$\frac{\partial H}{\partial p} = \frac{\partial(-\sum_i y_i log(p_i) + (1-y_i) log(1-p_i))}{\partial p} \tag{4.2}$$

Kako bi se vidio cijeli proces računanja derivacije, može se referirati na članak [21], a krajnji rezultat derivacije je izraz:

$$\frac{\partial H}{\partial p} = -\frac{y_i}{p_i} + \frac{1-y_i}{1-p_i} \tag{4.3}$$

Izraz 4.3 koristi se u kodu kako bi se odredio gradijent funkcije gubitka, što se može vidjeti u Listingu 4.1.

Listing 4.1: Računanje gradijenta

```
1 def gradient_calc(y, p):
2     # Avoid division by zero
3     p = np.clip(p, 1e-15, 1 - 1e-15)
4     return - (y / p) + (1 - y) / (1 - p)
```

Nakon što se zna koja se funkcija gubitka koristi te kako ju minimizirati, može se krenuti s implementiranjem algoritma podizanja gradijenta. U ovom radu implementiran je tako da predviđa gradijent funkcije gubitka. Dakle, prvo se

izračuna gradijent skupa stvarnih rezultata i predviđenih rezultata. Izračunatim gradijentom uči se stablo te se zatim predviđa gradijent funkcije gubitka kroz stablo odluke. Implementiran kod te funkcije vidljiv je u listingu 4.2.

Listing 4.2: Algoritam podizanja gradijenta

```
class XGBooster(object):
    def __init__(self, max_depth, n_estimators,
                 min_info_gained=sys.float_info.min):
        self.max_depth = max_depth
        self.n_estimators = n_estimators
        self.min_info_gained = min_info_gained


        self.learners = []

        for _ in range(self.n_estimators):
            learner = DecisionTree(
                max_depth=self.max_depth,
                min_info_gained=self.min_info_gained
            )

            self.learners.append(learner)

    def _most_frequent(self, List):
        occurence_count = Counter(List)
        return occurence_count.most_common(1)[0][0]

    def fit(self, X, y):
        mean = self._most_frequent(y)
        y = one_hot_encode(y)
        predictions = y.copy()
        if mean==2:
            predictions[:] = [0.0, 0.0, 1.0]
        if mean==1:
            predictions[:] = [0.0, 1.0, 0.0]
        if mean==0:
            predictions[:] = [1.0, 0.0, 0.0]

```

```
34          for estimator in range(self.n_estimators):
35              gradient = gradient_calc(y, predictions)
36              self.learners[estimator].fit(X, gradient)
37              updated_preds = self.learners[estimator].predict(X)
38              predictions -= updated_preds
```

Na kraju, nakon što je model završio s učenjem, on mora moći predvidjeti ishode testnih podataka(vidi poglavlje 2). U tu svrhu implementira se metoda *predict* koja je vidljiva u listingu 4.3.

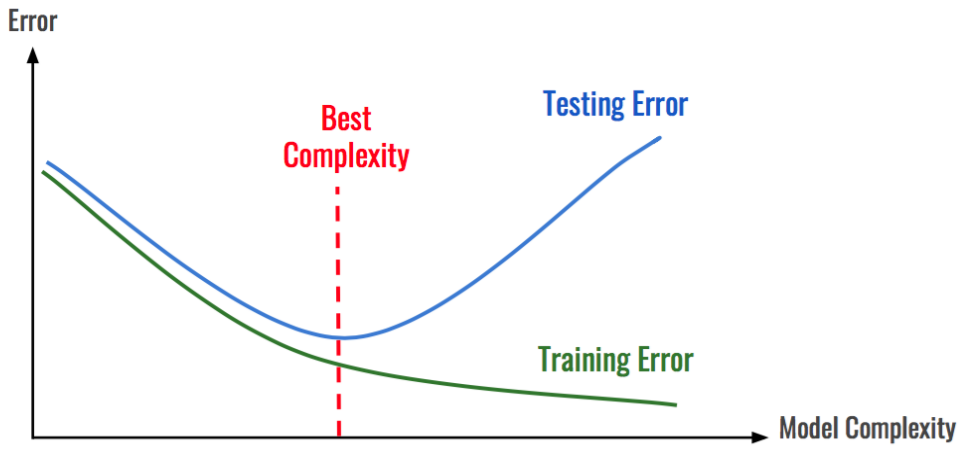
Listing 4.3: Predviđanje modela

```
1   predictions = np.array([])
2   # Make predictions
3   for tree in self.learners:
4       updated_preds = tree.predict(X)
5       updated_preds = np.asarray(updated_preds)
6       predictions = - updated_preds if not predictions.any() else predi
7
8   # Turn predictions into probability distribution
9   predictions = np.exp(predictions) / \
10                  np.expand_dims(
11                      np.sum(np.exp(predictions), axis=1), axis=1
12                  )
13  # choose a pred that maximizes probability
14  predictions = np.argmax(predictions, axis=1)
```

4.2. Parametri

Dodatni parametri koji su korišteni u ovom radu su *n_estimators*, *max_depth*[13]. Parametar **n_estimators** određuje koliko se osnovnih klasifikatora, odnosno u ovom slučaju stabala odluke treba izgraditi za model predviđanja. Ovaj parametar ne smije biti prevelik jer dodavanje stabala odluke nakon određene granice nema utjecaja ne predviđanja rezultata. Razlog tome je što se stabla odluke u alogritmu podizanja grade tako da svako novo stablo odluke popravlja pogreške prijašnjeg. Dakle, nakon određenog broja stabla odluke ne mogu se poboljšati.
Parametar **max_depth** određuje najveću dubinu stabla odluke.
Parametar max_depth zadužen je za podrezivanje stabla tijekom učenja (engl.

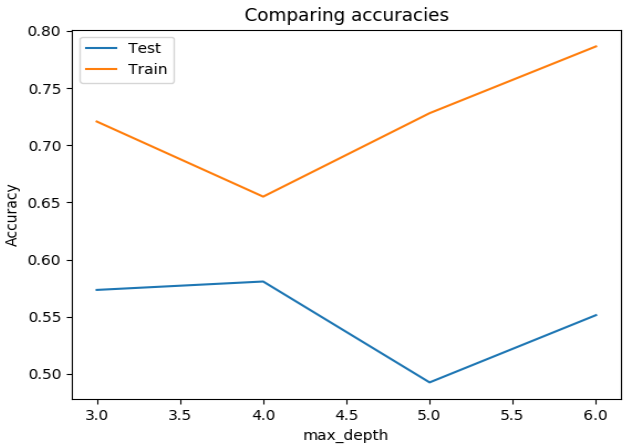
prepruning). *Prepruning* funkcionira tako da kod učenja stabla odluke, zaustavlja izgradnju stabla prije nego dođe do prenaučenosti istog, tj. prije nego što se stablo odluke potpuno prilagodi podacima skupa za učenje. Tim parametrom se model stabla pojednostavljuje. Model ne smije biti prejednostavan jer onda ne uči na danim podacima, a ne smije biti niti presložen jer onda ne zna generalizirati, odnosno dolazi do prenaučenosti.



Slika 4.1: Optimalna složenost modela[8]

4.2.1. Određivanje parametara

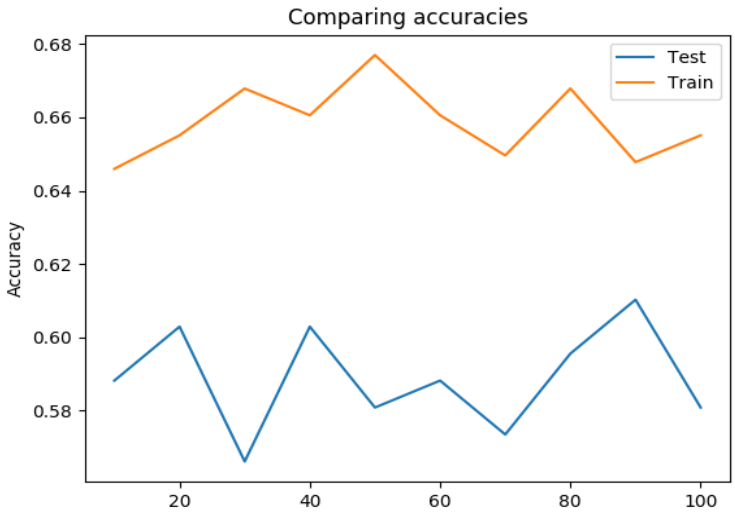
Parametri opisani u ovom poglavlju određuju se postupkom pokušaja i pogrešaka. Pokušava se naći kombinacija parametara koja daje najsličniju točnost predikcija na skupovima za učenje i testiranje, odnosno najsličniju pogrešku.
U prvom pokušaju, parametar *n_estimators* se postavlja na 100, parametar *test_size* postavlja se na 0.2, dok se parametar *max_depth* mijenja u intervalu [3,6] kako bi se došlo do onog koji daje najbolju točnost predviđanja.



Slika 4.2: Određivanje parametra *max_depth*

Iz predloženog grafa 4.2 vidi se da su vjerojatnosti skupa za učenje i skupa za testiranje najsličnije kada je parametar *max_depth* jednak 4, a kako vrijednost parametra raste sve se više razlikuju. Nadalje, iz grafa se može vidjeti da rastom parametra *max_depth* točnost predviđanja skupa za učenje raste prema 100%. To je bilo i za očekivati zato što se, kao što je opisano u poglavlju 4.2, njegovim smanjivanjem pojednostavljuje model stabla odluke. Posljedično, model uspijeva generalizirati ne prilagođavajući se u potpunosti podacima za učenje. Dakle, ako parametar *max_depth* raste, model postaje prenaučen.

Nakon što se uspješno uspio odrediti parametar *max_depth*, može se pristupiti određivanju parametra *n_estimators*. Određivanje parametra *n_estimators* odvija se tako da se, uz konstantan parametar *max_depth*, postupno povećava dok se ne dođe do broja koji daje najbolju točnost predviđanja za testni skup podataka. Odnos parametra *n_estimators* i točnosti predviđanja vidljiv je na slici 4.3.



Slika 4.3: Ovisnost točnosti izvođenja o parametru *n_estimators*

Kao što je vidljivo iz slike 4.3, točnost predikcija ovim algoritmom raste proporcionalno s parametrom *n_estimators*. Kao optimalna vrijednost uzima se broj 20 iz više razloga. Uz vrijednost parametra 20, algoritam ima dovoljno točno predviđa rezultate, a istovremeno su točnosti skupa podataka za učenje i testiranje slične. Također, povećavanjem parametra *n_estimators* raste i vrijeme izvođenja algoritma.

Dakle, u ovom radu optimalne vrijednosti svih parametara su:

- $test\_size = 0.2$
- $max\_depth = 4$
- $n\_estimators = 20$

Metodama opisanim u ovom poglavlju utvrdilo se da algoritam uz ove parametre donosi najbolje rezultate, a pri tome ne dolazi do prenaučenosti algoritma što je i bio krajnji cilj.

5. Uspoređivanje

Kako bi se utvrdilo kako implementirani algoritam radi, usporedit će ga se sa dva već postojeća algoritma:

- *XGBoost*
- Slučajna šuma (engl. *Random forest*)

Ovi algoritmi nisu implementirani u ovom radu nego su preuzete njihove gotove implementacije. Uspoređivat će se vrijeme izvođenja algoritama te njihova učinkovitost, odnosno točnost klasifikacije.

5.1. Usporedba s *XGBoost* algoritmom

Implementacija algoritma XGBoost za programski jezik Python preuzeta je sa stranice [24], tj. koristi se u obliku biblioteke *xgboost* programskog jezika Python dok se izvedba algoritma opisuje u znanstvenom članku [4]. Priprema podataka za analizu podataka XGBoost algoritmom ista je kao i za algoritam implementiran u ovom radu. Cijeli postupak opisan je u poglavlju broj 2.2.

5.1.1. Izbjegavanje prenaučenosti

Kako bi uopće mogli usporediti algoritme, potrebno je prvo provjeriti da li kod algoritma *XGBoost* dolazi do prenaučenosti te istu sprječiti. Biblioteka [24] nudi veoma opsežan izbor alata i parametara za tu svrhu. U ovom radu za sprječavanje prenaučenosti u *XGBoost* algoritmu koristit će se metoda ranijeg stajanja (engl. *early stopping*).
Biblioteka [24] nudi veliki broj metrika za evaluaciju tijekom učenja modela. Neke od njih su:

- *rmse* - root mean squared error
- *mae* - mean absolute error

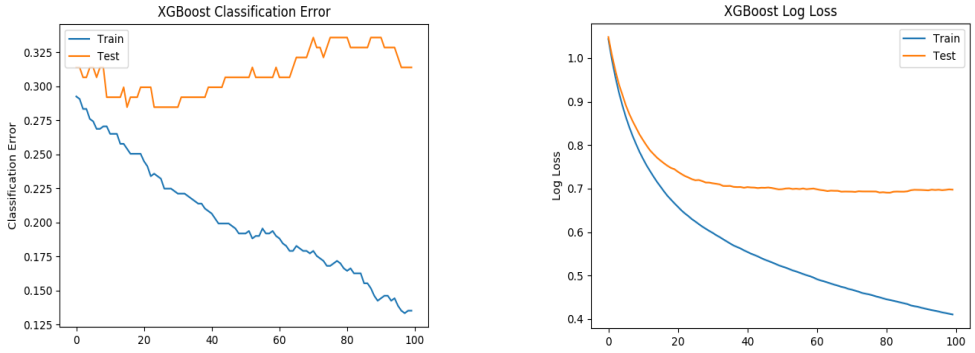
– *mlogloss* - multi-class log loss

– *merror* - multi-class classification error

Ove metrike omogućuju nam da pratimo učinkovitost učenja modela algoritmom *XGBoost*. Cijela lista može se naći na stranici [25], a u ovom radu koristi će se metrike *mlogloss* i *merror*.

Nakon što se podaci podijele na podatke za testiranje i podatke za učenje metodom *train_test_split* iz biblioteke *sklearn*, u varijablu *eval_set* postavljaju se skupovi podataka koje želimo evaluirati tijekom učenja. Pošto želimo vidjeti kako se tokom učenja ponašaju skupovi za učenje i testiranje te na kraju i njihovu usporedbu, upravo ta dva skupa postavljamo u *eval_set*. Kao željene metrike evaluacije podataka postavljamo *merror* koja mjeri pogrešku kod svake klasifikacije te *mlogloss* koja predstavlja funkciju gubitka kod klasifikacije tako da kvantificira cijenu pogreške predviđanja[20].

Kako bi se vizualizirala razlika obje metrike skupa podataka za učenje i testiranje crtaju se grafovi pomoću Pythonove *matplotlib* biblioteke. Grafovi prikazuju ovisnost funkcije gubitka o parametru *n_estimators*.

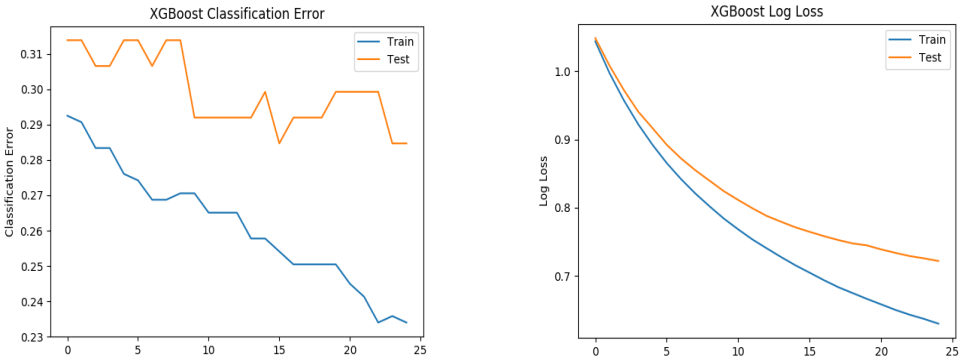


(a) merror **(b)** mlogloss

Slika 5.1: Evaluacija modela

Gledajući grafove procjenjuje se da porastom parametra *n_estimators* nakon što dosegne vrijednost 20, alogritam postaje prenaučen, odnosno previše se prilagođava podacima za učenje. Kako taj parametar ne bi trebali procjenjivati iz grafova, biblioteka [24] nudi parametar *early_stopping_rounds* koji to radi za nas.

Pokretanjem algoritma s parametrom *early_stopping_rounds* dobije se potvrda o optimalnoj vrijednosti parametra *n_paramters* dok učenje algoritma prestaje nakon što se zadana metrika prestane poboljšavati (*merror* u ovom slučaju).



(a) merror **(b)** mlogloss

Slika 5.2: Evaluacija modela uz rano zaustavljanje

Kako je vidljivo iz grafova, algoritam prestaje učiti nakon što parametar *n_estimators* dosegne vrijednost 25, dok vraća iteraciju broj 15 kao iteraciju u kojoj je *merror* najmanji, što se može vidjeti i na grafu. Kao konačnu potvrdu da ne dolazi do prenaučenosti algoritma uspoređuju se točnosti predviđanja na skupu podataka za učenje i test. Usporedba, prikazana u tablici 5.1, daje konačnu potvrdu da je spriječena prenaučenost algoritma.

Tablica 5.1: Usporedba točnosti predviđanja

max_depth	Test set accuracy	Training set accuracy
3	71.53%	74.59%

5.1.2. Vrijeme izvođenja

Nakon što se spriječila prenaučenost oba algoritma, mogu se usporediti vremena njihovih izvođenja. Kako bi se dobilo vrijeme izvođenja algoritma koristi se biblioteka *time* programskog jezika Python. Svaki algoritam će se pokrenuti pet puta kako bi se dobilo prosječno vrijeme izvođenja algoritma. Oba algoritma se izvode na prijenosnom računalu Dell G3 15, čije specifikacije su:

- Intel i7-8750H CPU @ 2.20GHz x 6
- 8 GB RAM-a
- SSD 256 GB

Performanse računala imaju utjecaj na vrijeme izvođenja.

Tablica 5.2: Vrijeme izvođenja algoritama

Broj izvođenja	Implementiran algoritam	*XGBoost*
1	26.0493 s	0.1269 s
2	26.0269 s	0.1308 s
3	25.8307 s	0.1283 s
4	25.9279 s	0.1272 s
5	25.9383 s	0.1282 s
Prosjek	25.9546 s	0.1283 s

Kao što je vidljivo iz tablice, razlika u vremenu izvođenja poprilično je velika– 25.8263 sekundi, što je bilo i za očekivati. Razlog tome je taj što je *XGBoost* algoritam implementiran tako da bude skalabilan u svim scenarijima te je deset puta brži nego bilo koji drugi postojeći algoritam na jednom računalu i sposoban je skalirati milijarde primjera u distribuiranim ili memorijom ograničenim postavkama[4].
Nadalje, kako bi se smanjilo vrijeme izvođenja, mogu se smanjiti parametri *n_estimators* i/ili *max_depth*. Međutim, pošto su određeni optimalni parametri za oba algoritma, smanjenjem istih može doći do podnaučenosti te pada točnost klasifikacije. Stoga se pristaje na sporije vrijeme izvođenja.

5.1.3. Točnost klasifikacije

Nakon što je algoritam pravilno naučen, koristi se testni skup podataka kako bi se dobila točnost predviđanja, odnosno točnost klasifikacija. Razlika točnosti klasifikacija dvaju algoritama može se dobiti usporedbom krajnjih rezultata predviđanja koju radimo uz različite podijelu na podatke za učenje i testiranje, odnosno s različitom vrijednosti parametra *test_size*. Usporedbu je moguće vidjeti u tablici 5.3.

Tablica 5.3: Točnosti klasifikacija algoritama

test_size	Implementiran algoritam	*XGBoost*
0.1	58.82%	68.12%
0.2	60.29%	71.53%
0.3	58.05%	68.45%
Prosjek	59.05%	69.37%

Iz tablice je vidljivo da točnost klasifikacije algoritma implementiranog u ovom radu manja od točnosti klasifikacije algoritma *XGBoost* za 10.32% što je i bilo za očekivati, budući da algoritam *XGBoost* koristi puno više regularizacijskih parametara[23]. Više o parametrima algoritma *XGBoost* može se pročitati u biblioteci [25].

5.2. Usporedba s algoritmom slučajne šume

Algoritam slučajna šuma (engl. *random forest*) je algoritam nadziranog učenja. Radi tako da izgradi više stabala odluke (ovisno o parametru *n_estimators*) te ih zatim "spoji" kako bi se dobila što bolja i stabilnija predviđanja. Takozvano spajanje zapravo se radi algoritmom pakiranja (engl. *bagging*)[19] koji je, uz algoritam podizanja, još jedna ansambl metoda. Za razliku od algoritma podizanja, opisanog u poglavlju 4, algoritam pakiranja ne gradi stabla odluke slijedno, već paralelno, a na kraju uzima najčešću vrijednost predviđanja od tih stabala odluke i dolazi do pravog predviđanja. Nadalje, algoritam može biti korišten i za klasifikacijske i regresijske probleme što čini većinu strojnog učenja danas.
U ovom radu koristi se razred *RandomForestClassifier* iz Pythonove biblioteke *sklearn* za implementaciju algoritma slučajne šume[17]. Priprema podataka za analizu podataka algoritmom slučajna šuma ista je kao i za algoritam implementiran u ovom radu. Cijeli postupak opisan je u poglavlju 2.2.

5.2.1. Izbjegavanje prenaučenosti

Prenaučenost se u ovom slučaju može izbjeći podešavanjem parametara *n_estimators* i *max_depth*. Kako bi pronašli optimalnu vrijednost parametra *max_depth*, parametar *n_estimators* postavlja se na vrijednost 100, a *max_depth* se mijenja dok se ne dobiju što bolje točnosti predviđanja bez da dolazi do prenaučenosti. Postupak je vidljiv u tablici 5.4.

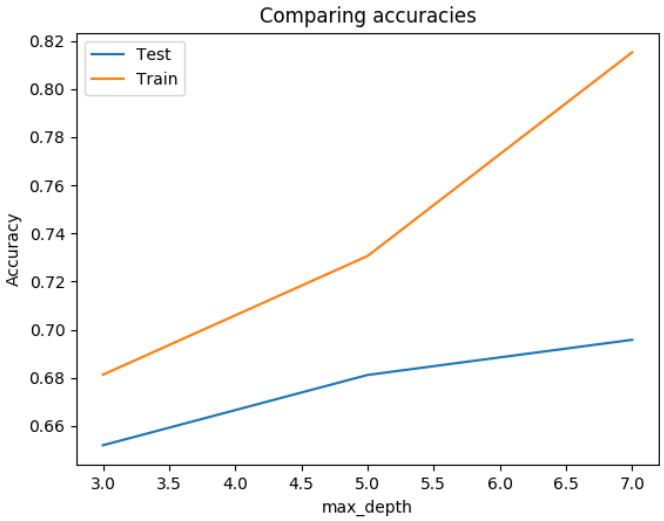
Tablica 5.4: Određivanje *max_depth* parametra

Test set accuracies				
max_depth	Test acc	Test acc(2)	Test acc(3)	Avg
–	70.80%	65.69%	69.34%	68.61%
7	70.07%	69.34%	69.34%	69.58%
5	68.61%	68.61%	67.15%	68.12%
3	64.96%	64.96%	65.69%	65.20%

Tablica 5.5: Određivanje *max_depth* parametra(2)

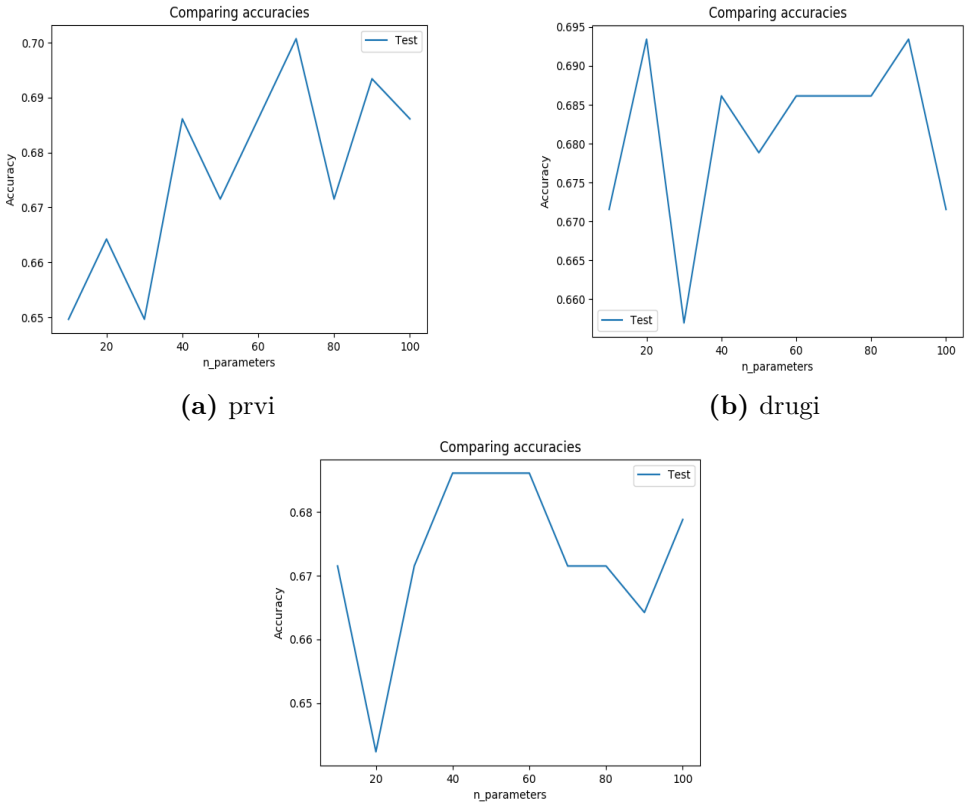
Training set accuracies				
max_depth	Train acc	Train acc(2)	Train acc(3)	Avg
–	100%	100%	100%	100%
7	79.83%	83.00%	81.72%	81.52%
5	72.21%	73.31%	73.67%	73.06%
3	67.64%	67.82%	68.92%	68.13%

Iz tablica 5.4 i 5.5 vidljivo je da dolazi do prenaučenosti algoritma ukoliko se ne obavi ranije podrezivanje (engl. *prepruning*), odnosno ukoliko se ne postavi parametar *max_depth*. To se može vidjeti po točnosti predviđanja kod skupa podataka za učenje u tablici 5.5 gdje, ako nema ranijeg podrezivanja, točnost predviđanja iznosi 100%. Nadalje, njegovim smanjivanjem dobivamo sve bolje rezultate. Optimalna vrijednost parametra u ovom slučaju je 5 zato jer daje najbolju točnost predviđanja te istovremeno spriječava prenaučenost, odnosno točnost skupa za učenje i testiranje je najbliža.
Kako bi se još bolje dočarala ovisnost prenaučenosti o parametru *max_depth* iscrtao se graf s podacima iz tablica 5.4 i 5.5, a koji prikazuju odnos parametra *max_depth* i prosjeka točnosti skupa podataka za učenje i testiranje. Graf je prikazan na slici 5.3. Slika prikazuje promjenu parametra *max_depth* do vrijednosti 7, budući da nakon te vrijednosti dolazi do prenaučenosti algoritma.



Slika 5.3: Odnos točnosti predviđanja i parametra *max_depth*

Dakle, zaključuje se da je optimalna vrijednost parametra *max_depth* 5 pošto ima dovoljno veliku točnost podataka za testiranje (68.12%), dok je najmanja razlika između točnosti podataka za učenje i testiranje (73.06%) što nam potvrđuje da ne dolazi do prenaučenosti što je i bio cilj.
Parametar *n_estimators* određuje se tako da se povećava, uz konstantnu vrijednost parametra *max_depth* dok se ne naiđe na vrijednost parametra, koja uz zadani parametar *max_depth* daje najbolju točnost predviđanja.



(a) prvi

(b) drugi

(c) treći

Slika 5.4: Ovisnost točnosti predviđanja o parametru *n_estimators*

Na slici 5.4 se lijepo vidi kako točnost predviđanja varira obzirom na parametar *n_estimators*. To je i logično pošto se podaci ne razdvajaju uvijek isto, nego ovisno o tome kako se izmješaju (engl. *shuffle*). Međutim, može se primjetiti da je točnost procjene uvijek poprilično dobra kada je *n_estimators* jednak broju 60. Stoga se odlučuje uzeti upravo tu vrijednost kao optimalnu.
Dakle, u algoritmu slučajne šume, optimalne vrijednosti parametara će biti:

- $max\_depth = 5$
- $n\_estimators = 60$

Metodama u ovom poglavlju utvrdilo se da algoritam uz ove parametre donosi najbolje rezultate, a pri tome ne dolazi do prenaučenosti algoritma što je i bio krajnji cilj.

5.2.2. Vrijeme izvođenja

Kako bi se dobilo vrijeme izvođenja oba algoritma koristi se biblioteka *time* programskog jezika Python. Algoritam slučajne šume će se pokrenuti pet puta kako

bi se dobilo prosječno virjeme izvođenja algoritma dok se za algoritam implementiran u ovom radu koriste vremena dobivena u poglavlju 5.1.2 (tablica 5.2). Nadalje, potrebno je ukazati da se oba algoritma izvode na prijenosnom računalu Dell G3 15 čije performanse imaju utjecaj ne vrijeme izvođenja. Potrebno je i reći da se oba algoritma izvode sa parametrima određenim u poglavljima 4.2.1 (implementiran algoritam) te 5.2.1 (algoritam slučajna šuma).

Tablica 5.6: Vrijeme izvođenja algoritama

Broj izvođenja	Implementiran algoritam	Random forest
1	26.0493 s	0.0379
2	26.0269 s	0.0375 s
3	25.8307 s	0.0367 s
4	25.9279 s	0.0376 s
5	25.9383 s	0.0408 s
Prosjek	25.9546 s	0.0381 s

Kako se može vidjeti iz tablice 5.6, razlika u izvođenju algoritama je opet poprilično velika – 25.917 sekundi. Opet je očekivana velika razlika pošto biblioteka [17] koristi mnoštvo optimizacija kod svojih algoritama te paralelizam.

5.2.3. Točnost klasifikacije

Točnost klasifikacija uspoređuje se tako da se usporede krajnje točnosti predviđanja. Usporedbe algoritama mogu se vidjeti u tablici 5.7.

Tablica 5.7: Usporedba točnosti klasifikacija algoritama

test_size	Implementiran algoritam	*Random Forest*
0.1	58.82%	66.67%
0.2	60.29%	63.77%
0.3	58.05%	68.12%
Prosjek	59.05%	66.19%

Dakle, implementirani algoritam predviđa ishode utakmica s točnosti od 59.05% dok algoritam slučajna šuma predviđa točno u 66.19%. Razlika u točnosti klasifikacija je 7.12% što je definitivno prihvatljiva vrijednost. Smatram da bi se ove

vrijednosti, općenito performanse, implementiranog algoritma mogle poboljšati uz malo više vremena za rad na ovom radu.

6. Zaključak

Iz ovog rada može se zaključiti da se implementiranim algoritmom može predvidjeti ishod utakmice 1. engleske nogometne lige (EPL) s točnošću od 59.05%, uz parametre određene u poglavlju 4.2.1:

- $max\_depth = 4$
- $n\_parameters = 20$

Usporedbama s algoritmima *XGBoost* i slučajnom šumom (engl. *Random forest*) zaključeno je da postoje velike razlike u vremenu izvođenja. Implementiranom algoritmu za izvođenje navedenim parametrima treba prosječno 25.9546 sekundi, algoritmu *XGBoost* treba 0.1283 sekundi, dok se algoritam slučajna šuma izvodi u prosječnom vremenu od 0.0381 sekundi. Razlike u vremenu izvođenja su velike zbog toga jer cilj ovog rada nije bio implementirati optimalan i najefikasniji algoritam već prikazati unutarnje procese istog na transparentan način i svima dostupan način.
Nadalje, usporedbom točnosti klasifikacije algoritama, dobiveni su rezultati prikazani u tablici 6.1.

Tablica 6.1: Točnosti klasifikacija algoritama

Implementirani algoritam	*XGBoost*	*Random Forest*
59.05%	69.37%	66.19%

Iako oba algoritma imaju bolje točnosti predviđanja od implementiranog algoritma, točnost od 59.05% je puno bolja od slučajnog predviđanja, koja iznosi 33.33% (pobjeda, gubitak, izjednačeno). Nadalje, uz više vremena za rad na ovom projektu mogli bi se postići još i bolji rezultati te se ostavlja prostor za rad na ovom projektu u budućnosti. Poboljšanje točnosti klasifikacije algoritma mogla bi se dobiti uvođenjem novih regularizacijskih parametara (npr. *shrinkage parameter*). Vrijeme izvođenja algoritma moglo bi se smanjiti tako da svako sta-

blo odluke uči s nasumičnim skupom podataka, o čemu se više može pročitati u znanstvenom radu [11].

Literatura

[1] Ira Boudway. Soccer is the world's most popular sport and still growing. 2018-06-12. Accessed: 2019-06-03.

[2] Rajesh S. Brid. Decision trees - a simple way to visualize a decision. 10 2018. Accessed: 2019-05-08.

[3] Peter Bühlmann. Bagging, boosting and ensemble methods. *Handbook of Computational Statistics*, 01 2012. doi: 10.1007/978-3-642-21551-3_33.

[4] Tianqi Chen i Carlos Guestrin. Xgboost: A scalable tree boosting system. 2016. doi: https://arxiv.org/pdf/1603.02754.pdf.

[5] Wikipedia contributors. Id3 algorithm. `https://en.wikipedia.org/wiki/ID3_algorithm`, 2019. Accessed: 2019-05-05.

[6] Wikipedia contributors. Loss function for classification. `https://en.wikipedia.org/wiki/Loss_functions_for_classification`, 2019. Accessed: 2019-06-01.

[7] Michael DelSole. What is one hot encoding and how to do it. 4 2018. Accessed: 2019-04-29.

[8] Julien Despois. Memorizing is not learning! - 6 tricks to prevent overfitting in machine learning. 3 2018. Accessed: 2019-05-06.

[9] Rob DiPietro. A friendly introduction to cross-entropy loss. 5 2016. Accessed: 2019-06-02.

[10] football data.co.uk. Data files: England. `http://football-data.co.uk/englandm.php`. Accessed: 2019-05-15.

[11] Andrew L. Friedman i Samantha Miles. Developing stakeholder theory. 12 2002. doi: https://doi.org/10.1111/1467-6486.00280.

[12] Hunter Heidenreich. What are types of machine learning. *Towards Data Science*, 12 2018.

[13] Aarshay Jain. Complete machine learning guide to parameter tuning in gradient boosting (gbm) in python. 2 2016. Acccessed: 2019-05-10.

[14] Evan Lutins. Ensemble methods in machine learning: What are they and why use them? 8 2017. Accessed: 2019-05-20.

[15] Chris McCorMick. Adaboost tutorial. 12 2013. Accessed: 2019-05-20.

[16] Alexey Natekin i Alois Knoll. Gradient boosting machines, a tutorial. *Front Neurobot*, 12 2013. doi: 10.3389/fnbot.2013.00021.

[17] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, i E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12: 2825–2830, 2011.

[18] John Price, Neil Farrington, i Lee Hall. Changing the game? The impact of Twitter on relationships between football clubs, supporters and the sports media. *Soccer and Society*, 14, 07 2013. doi: 10.1080/14660970.2013.810431.

[19] Joseph Roca. Ensemble methods: bagging, boosting and stacking. Accessed: 2019-05-20.

[20] Srishti Saha. Understanding the log loss function of xgboost. Accessed: 2019-05-19.

[21] Sefik Ilkin Serengil. A gentle introduction to cross-entropy loss function. 12 2017. Accessed: 2019-06-04.

[22] Naoki Shibuya. Demystifying cross-entropy. 10 2018. Accessed: 2019-05-10.

[23] Synced. Tree boosting with xgboost—why does xgboost win "every" machine learning competition? 10 2017. Accessed:2019-06-05.

[24] xgboost developers. Xgboost python package. `https://xgboost.readthedocs.io/en/latest/python/index.html`, . Accessed: 2019-04-05.

[25] xgboost developers. Xgboost paramters. `https://xgboost.readthedocs.io/en/latest//parameter.html`, . Accessed: 2019-05-15.

Implementacija klasifikacijskog algoritma podizanja gradijenta stablima odluke s primjenom na predikciju ishoda nogometnih utakmica

Sažetak

U ovom radu implementiran je klasifikacijski algoritam podizanja gradijenta stablima odluke, što je zapravo vrsta algoritma podizanja. Algoritam slijedno gradi stabla odluke koja uči predviđajući gradijent funkcije gubitka, a kako bi se izgradila stabla odluke koristi se algoritam ID3. Cilj algoritma je predvidjeti rezultate utakmica u 1. engleskoj nogometnoj ligi (EPL). Nakon što je algoritam implementiran, sprječila se prenaučenost podešavanjem parametara. Za parametar *max depth*, koji služi za ranije podrezivanje stabla, određena je optimalna vrijednost 4, dok je za parametar *n_parameters*, koji određuje broj stabala odluke, određena optimalna vrijednost 20.
Dobivene točnosti predikcija ovim algoritmom kreću se oko 59%, što je lošije od već implementiranih algoritama, *XGBoost* i slučajne šume, no opet bolje od slučajnog predviđanja.

Ključne riječi: *boosting*, stabla odluke, ID3, podizanje gradijenta, nogomet, 1. engleska liga

Implementation of Gradient Boosted Decision Trees Classification Algorithm with Application to Football Games Outcomes Prediction

Abstract

In this work, gradient boosted decision trees classification is implemented, which is a type of *boosting* algorithm. Algorithm sequentially builds decision trees that are trained on predicting the gradient of the loss function. For building decision trees ID3 algorithm is used. The Goal of this algorithm is to predict scores of football games in England Premiership League. After this algorithm has been implemented, overfitting was avoided by tuning parameters. For parameter *max_depth*, which purpose is to preprune a tree, the optimal value is 4, while for parameter *n_parameters*, which determines size of a tree, optimal value is 20. Prediction accuracies with this algorithm are around 59%, which is worse than prediction accuracy with one of the already implemented algorithms, such as *XGBoost* and random forest, but again better than prediction by chance.

Keywords: *boosting*, decision trees, ID3, gradient boosting, football/soccer, EPL