SVEUČILIŠTE U SPLITU
PRIRODOSLOVNO MATEMATIČKI FAKULTET

SEMINARSKI RAD

Implementacija stabla odluke za probleme klasifikacije i regresije

Fabijan Šarić

Sadržaj

Uvod

Stabla odluke (eng. Decision Trees) su moćan alat u strojnom učenju koji se koristi za rješavanje problema klasifikacije i regresije.

Problem klasifikacije predstavlja zadatak pridruživanja nekog objekta nekoj kategoriji ili klasi. Klasifikacija može biti binarna ili višeklasna. Kod binarne klasifikacije postoje samo dvije moguće kategorije kojima objekt može pripadati (npr. predviđanje ima li osoba određenu bolest ili nema), dok kod višeklasne klasifikacije postoji više mogućih klasa kojima objekt može pripadati (npr. predviđanje pozicije koju nogometaš igra).

Sa druge strane problem regresije je predviđanje numeričkih vrijednosti na osnovu ulaznih podataka. Za razliku od problema klasifikacije, gdje su se predviđale diskretne vrijednosti klasa, kod regresijskih problema predviđa se kontinuirana numerička vrijednost.

U nastavku rada, fokus će biti na implementaciji stabla odluke (eng. Decision Tree) u svrhu rješavanja problema binarne klasifikacije i problem regresije. Kao programski jezik za implementaciju izabran je programski jezik Java. Nakon implementacije modeli će biti testirani pomoću standardnih metoda za validaciju klasifikacijskih modela.

1. Rad sa podacima – paket datatools.

Prije implementacije samog algoritma, potrebno je obratiti pažnju na rad sa podacima. Kako bi se omogućilo što lakše učitavanje i obrada podataka, stvoren je paket (eng. package) datatools, koji sadrži klase koje znatno olakšavaju korisniku rad sa podacima. Među njima se nalaze sljedeće klase:

- DataSeries
- DataSet
- DataOps

U nastavku poglavlja će biti opisana svaka od ovih klasa.

1.1. DataSeries

Klasa DataSeries nasljeđuje klasu ArrayList i implementira sučelje DescriptiveStatistics (koje će biti detaljnije opisano u nastavku poglavlja), koje sadrži apstraktne statističke metode koje su morale biti implementirane u klasi DataSeries, a to su metode za izračun aritmetičke sredine, modalne i medijalne vrijednosti, varijance, frekvencija, korelacije itd. Svaki DataSet sadrži i polje label, koje označava atribut podataka koji su spremljeni u seriju. Učitavanje jednostavne serije podataka je prikazano na slici (Slika 1).

```
// Podaci koji se spremaju u seriju.
Object[] obj = { 21.2, 22.6, 21.9, 21, 21, 21.3 };

// Stvaranje serije podataka.
DataSeries testSeries = new DataSeries(obj);

// Expected.
double sum = testSeries.sum();
double mean = testSeries.mean();
double median = testSeries.median();
double mode = testSeries.mode();
double min = testSeries.min();
double max = testSeries.max();
```

Slika 1 Stvaranje objekta klase DataSeries i izračun nekih statističkih vrijednosti. h

Zamišljeno je da objekti ove klase služe za spremanje nizova podataka i izvršavanje operacija nad njima. Budući da klasa nasljeđuje klasu ArrayList, omogućeno je dinamičko

dodavanje i brisanje elemenata. Osim toga moguće je i filtriranje podataka u novu seriju prema nekom kriteriju, kao što prikazuje slika (Slika 2).

```
// Podaci koji se spremaju u seriju.
Object[] obj = { "Anna", "Simon", "Theo" };

// Stvaranje serije podataka.
DataSeries testSeries = new DataSeries(obj);

// Odabir indeksa prema kriteriju.
Integer[] indices = testSeries.indicesWhere(x -> x.toString().length() == 5);

/*
* Stvaranje nove serije koja sadrži sve elemente iz
* originalne serije podataka koji se nalaze na
* odabranim indeksima.
* */
DataSeries filtered = testSeries.getAll(indices);
```

Slika 2 Odabir podataka iz objekta klase DataSeries

Moguće je odjednom učitati i cijeli niz podataka. Osim navedenih metoda postoje još i brojne druge. Klasa je napravljena da bude slična klasi Series biblioteke NumPy programskog jezika Python.

1.2. DataSet

Klasa DataSet ima svrhu spremanja više serija podataka (DataSeries objekata), s time da svaka odnjih mora imati jedinstvenu oznaku atributa (eng. label). Moguće je dohvatiti red koji se nalazi na zadanom indeku, dodati novi stupac sa atributom, dohvatiti dimeznije (broj redaka i stupaca) ili ukloniti stupac koji ima određenu oznaku. U svrhu statistike moguće je i generirati matricu korelacije. Podaci se mogu dodavati ručno ili se mogu učitati iz CSV (eng. Comma Separated Values) datoteke pomoću metode readCSV. Potrebno je da struktura datoteke bude organizirana na način da prvi redak datoteke sadrži oznake ('labels') stupaca i da svaki sljedeći redak sadrži podataka onoliko koliko se oznaka stupaca nalazi u prvom retku datoteke. Također je potrebno i da se za razdvajanje podataka koristi jedinstveni simbol (preferira se ',' ili ';'). Prilikom učitavanja automatski se prepoznaju podaci koji su spremljeni u svakom stupcu, iako su svi elementi unutar strukture tipa 'Object'. To se radi u svrhu naknadnog pretvaranja podataka. Metoda prima dva obavezna parametra, a to su putanja do datoteke na disku i separator (simbol za razdvajanje podataka u datoteci). Ako se kao

parametar proslijedi i skup oznaka stupaca, iz datoteke će biti učitani samo stupci čije se oznake nalaze u tome skupu. Primjer učitavanja iz datoteke prikazan je na slici (Slika 3).

```
// Stvaranje skupa podataka.
DataSet dataSet = new DataSet();

// Filteri za odabrane podatke.
HashSet<String> filter = new HashSet<>();
filter.add("Name");
filter.add("Country");
filter.add("Assists");
filter.add("Goals");
filter.add("Market Value");

// Čitanje podataka iz datoteke.
dataSet.fromCSV(TestFiles.PLAYERS_FILE, ";", filter);

// Dohvaćanje vrijednosti.
int samples = dataSet.getShape().rows();
int features = dataSet.getShape().columns();

// Novi DataSet koji sadrži samo numeričke vrijednosti
// iz originalnog DataSet-a.
DataSet numericalOnly = dataSet.numerical();
```

Slika 3 Učitavanje skupa podataka iz CSV datoteke.

U primjeru koji je prikazan na slici se dogodilo sljedeće. Prvo je stvoren prazan skup podataka (DataSet). Zatim je stvoren skup koji sadrži odabrane atribute iz datoteke koji će biti učitani, nakon čega je pročitana datoteka. Umjesto samog teksta koji bi predstavljao putanju do datoteke, stvorena je klasa sa statičkim poljima, TestFiles, u kojoj su kao statička polja spremljene sve putanje do datoteka koje služe za testiranje podataka. Kada su učitani svi podaci iz datoteke, u varijable su spremljeni podaci o broju redaka i stupaca DataSet-a (za spremanje podataka o strukturi definirana je pomoćna record klasa pod nazivom DataShape). Na kraju je stvoren novi skup podataka koji sadrži samo stupce sa numeričkim vrijednostima originalnog stupca.

1.3. Sučelje DescriptiveStatistics

Ovo sučelje predstavlja predložak za sve klase koje bi trebale imati mogućnost računanja statističkih parametara. Dio koda ovog sučelja je prikazan na slici ispod (Slika 4).

```
public interface DescriptiveStatistics {

    /**
     * Provjerava jesu li svi elementi zadanog skupa podataka
     * kategoričkog tipa. Ako jesu vraća true inače vraća false.
     * */
    boolean isCategorical();

    /**
     * Provjerava jesu li svi elementi zadanog skupa podataka
     * numeričkog tipa. Ako jesu vraća true inače vraća false.
     * */
    boolean isNumerical();

    /**
     * Računa sumu svih elemenata zadanog skupa podataka.
     * @throws ArithmeticException ukoliko svi elementi skupa nisu numerički podaci
     * */
    double sum() throws ArithmeticException;

    /**
     * Računa aritmetičku sredinu svih elemenata zadanog skupa podataka.
     * @throws ArithmeticException ukoliko svi elementi skupa nisu numerički podaci
     * */
    double mean() throws ArithmeticException;

    /**
     * Računa medijan svih elemenata zadanog skupa podataka.
     * @throws ArithmeticException ukoliko svi elementi skupa nisu numerički podaci
     * */
    double median() throws ArithmeticException;

    /**
     * Računa modalnu vrijednost svih elemenata zadanog skupa podataka.
     * @throws ArithmeticException ukoliko svi elementi skupa nisu numerički podaci
     * */
    double mode() throws ArithmeticException;
```

Slika 4 Dio sučelja DescriptiveStatistics

U ovom projektu klasa DataSeries implementira ovo sučelje i time onda svaka serija podataka ima mogućnost izvršavanja statističkih metoda. Na slici iznad (Slika 4) prikazan je samo dio sučelja, pa osim navedenih metoda sučelje sadrži i dodatne metode koje na slici nisu prikazane.

1.4. DataOps

Klasa DataOps sadrži pomoćne statičke metode čija je svrha prvenstveno izbjegavanje ponavljanja istog programskog koda pri radu sa podacima. Neke od glavnih metoda koje se nalaze u ovoj klasi su:

- **convert** - Prima literal u obliku Stringa i pretvara ga u željeni tip podatka (koji može biti String, bool, int, double). Vraća podatak pretvoren u 'Object', jer se svi podaci u DataSet spremaju kao 'Object'.
- **resloveType** - Prima literal u obliku Stringa i procjenjuje koji je to tip podatka, odnosno vraća njegov tip podatka (koji može biti String, bool, int, double).
- **corrMatrixToStr** – Vraća ispisanu matricu korelacije u obliku Stringa.
- **splitData** – služi za podjelu skupa podataka na dva skupa koji služe za treniranje i testiranje modela strojnog učenja.

2. Stabla odluke

Stablo odluke je jedan od najpopularnijih modela strojnog učenja. Generalno, algoritmi strojnog učenja se mogu podijeliti na:

- **Algoritme nadziranog učenja –** svaki podatak iz skupa podataka ima svoje ulazne oznake (skup X) i izlazni podatak (y). Ovakvim algoritmima model se treniranjem pokušava naučiti da za ulazne podatke pogađa ciljni izlaz. Ovakvi algoritmi rješavaju klasifikacijske i regresijske probleme. Jedan od primjera modela nadziranog učenja može biti model koji predviđa tržišnu vrijednost nogometaša na osnovu postignutih golova, asistencija i dobivenih kartona predviđa cijenu igrača.
- **Algoritme nenadziranog učenja –** kod ovakvih algoritama, podaci nemaju ciljnu (izlaznu vrijednost). Ovakvi algoritmi rješavaju probleme grupiranja podataka, smanjenja dimenzionalnosti podataka itd.
- **Algoritme poduprtog učenja** – model se uči na osnovu interakcija sa okolinom. Model, na osnovu odluka koje je donio, od okoline kao povratnu informaciju dobiva nagradu ili kaznu, pa se uči na osnovu povratnih informacija. Primjer toga mogu biti računalni protivnici u videoigrama koji se uče prema odigranim potezima.

Stabla odluke spadaju u kategoriju nadziranog učenja (eng. supervised learning) jer je cilj ovog algoritma naučiti model kako povezati ulazne značajke s odgovarajućim izlaznim vrijednostima.

2.1. Općenito

Pogledajmo sljedeće podatke koji govore o isplativosti proizvodnje nekih proizvoda (Tablica 1). Podaci se sastoje od 3 stupca, a to su:

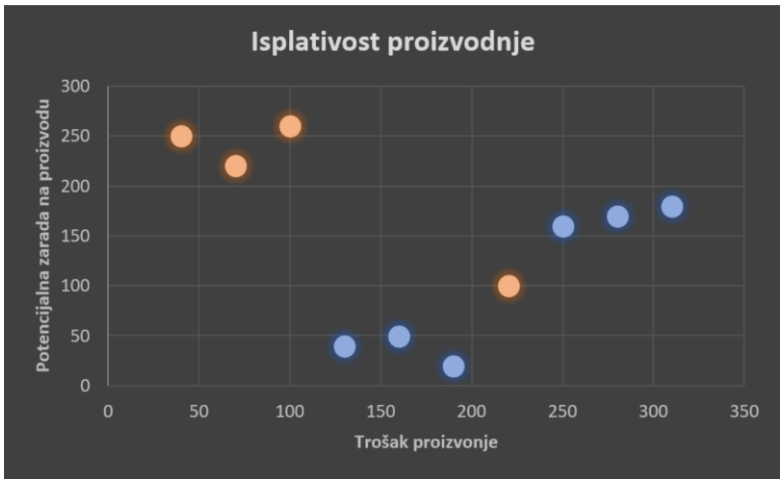
- Troškovi proizvodnje
- Zarada
- Isplativost

Model bi je potrebno istrenirati da na osnovu podataka o troškovima proizvodnje i potencijalnoj zaradi na tom proizvodu svrstava proizvod u kategorije isplativosti (0 – nije isplativ, 1 - isplativ).

Troškovi proizvodnje	Zarada	Isplativost
40	250	1
70	220	1
100	260	1
130	40	0
160	50	0
190	20	0
220	100	1
250	160	0
280	170	0
310	180	0

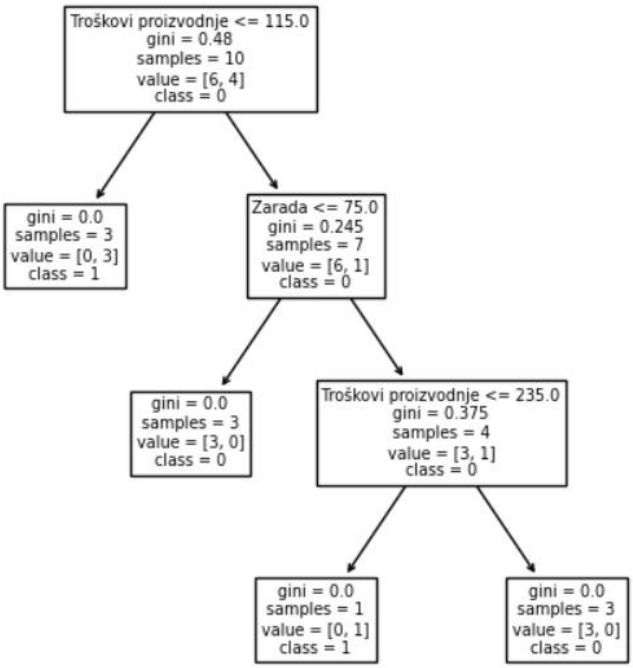
Tablica 1 - Podaci o proizvodnji

Ako bismo htjeli vizualizirati ove podatke najbolje je koristiti raspršeni dijagram (eng. scatter plot diagram) na kojemu će kao koordinate točaka biti prikazani podaci o troškovima proizvodnje i zaradi, a boja točaka će ovisiti o isplativosti (prikazuje Slika 5).



Slika 5 Vizualizacija podataka o proizvodnji.

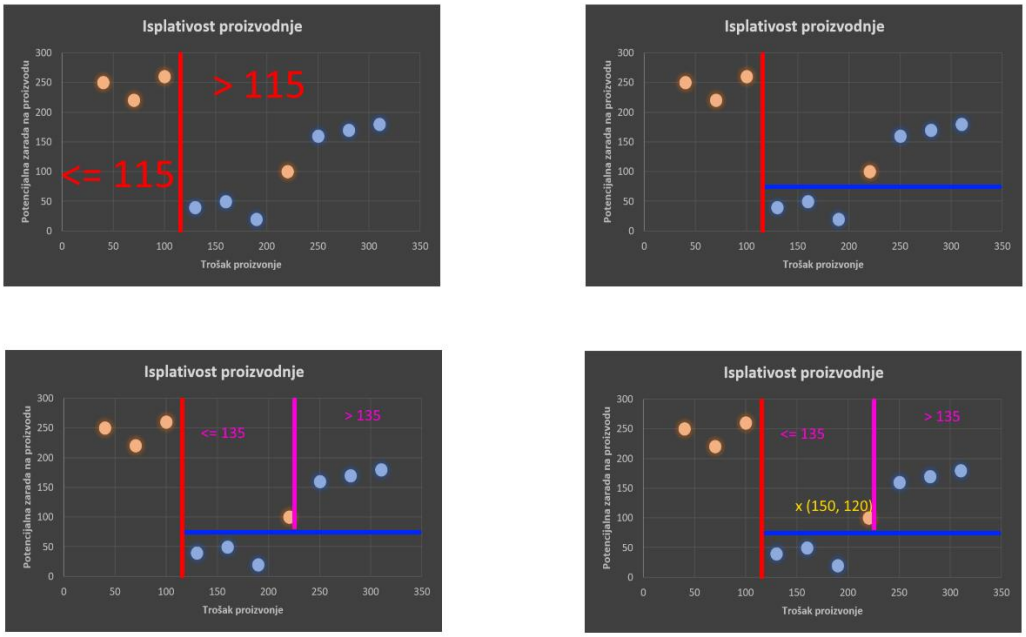
Recimo da bismo sada htjeli proizvesti novi proizvod za kojega bi troškovi proizvodnje iznosili 150, a potencijalna zarada na tom proizvodu bila bi 120. Za ovakav skup podataka bi klasifikacijski model stabla odluke bio od velike koristi. Recimo da smo istrenirali takav model stabla odluke na podacima iz Tablice 1. Takav model stabla odluke je prikazan na slici ispod (Slika 6).



Slika 6 - Stablo odluke za klasifikaciju isplativosti proizvodnje

Pokušajmo objasniti ovakav model. Model u stvari predstavlja binarno stablo koje ako slijedimo možemo doći do klasifikacije konkretnog podatka. Pogledajmo sada naš podatk kojega želimo klasificirati (troškovi 150 i zarada 120). Krećemo od korijena stabla čiji je prag 150, a atribut „Troškovi proizvodnje“ (struktura čvora će detaljnije biti objašnjena kasnije). Troškovi proizvodnje našeg podatka iznose 150, što je veće od praga, pa kao sljedeći korak uzimamo desnog sljedbenika trenutnog čvora (korijena). Ako sada promotrimo naš trenutni čvor, njegov atribut je „Zarada“, a prag 75. Potencijalna zarada našeg proizvoda iznosi 120, što je većeod prga trenutnog čvora, pa trenutni čvor našeg stabla opet postaje njegov desni sljedbenik. Sada vidimo da u trenutnom čvoru više nema niti praga niti atributa, što znači da je njegov čvor list stabla, a tada donosimo odluku o klasifikaciji podatka. Budući da je klasa ovog čvora 0, zaključujemo da proizvod možemo svrstati u kategoriju neisplativih proizvoda.

Promotrimo sada korijen stabla. Svojstvo „samples“ nam prikazuje da se u korijenu stabla nalazi točno 10 podataka i to su podaci iz skupa za treniranje modela (odnosno podaci iz Tablice 1). Način odlučivanja još bolje prikazuje Slika 7.



Slika 7 - Klasifikacija novog podatka pomoću stabla odluke

Ako malo bolje promotrimo sliku, vidimo da ona odgovara modelu. Nakon prvog koraka uočavamo da su svi proizvodi kojima je trošak proizvodnje manji ili jednaki 115 klasificirani kao isplativi. Ako bismo se zaustavili na ovom koraku, mogli bismo klasificirati ostale proizvode kao neisplative, a time vidimo da bi onda neki proizvodi bili pogrešno klasificirani. Ako bismo nastavili sa izgradnjom stabla, vidimo da su svi podaci lijevog sljedbenika jednoznačno klasificirani i tu onda nema potrebe za daljnjom klasifikacijom. Ali zato se klasifikacija može nastaviti u njegovom desnom sljedbeniku. Obično se kao uvjet prestanka izgradnje stabla odluke postavi njegova maksimalna dubina do koje izgradnja može ići ili najmanji broj podatak u čvoru za koji se može nastaviti izgradnja. Algoritam izgradnje stabla odluke će biti detaljnije opisan u kasnijim poglavljima.

2.2. Prednosti i nedostatci

Stabla odluke imaju dosta prednosti. Jedna od prednosti bila bi jednostavnost interpretacije ovih stabala (jer je model u stvari skup jednostavnih pravila po kojima se dolazi do zaključaka), a osim toga stabla odluke se mogu vrlo jasno vizualizirati. Velika prednost ovog modela je i to da ne zahtjeva veliku količinu pripreme podataka za treniranje modela, a nakon što je model istreniran, predviđanja se vrše u logaritamskom vremenu.

Osim navedenih prednosti, stabla odluke imaju i određene nedostatke. Glavni nedostatak ovakvih modela je nestabilnost iz razloga što mala promjena na podacima može promijeniti cijelo stablo. Također, ukoliko osoba koja ih koristi ne razumije jasno što radi, lako može doći do problema prenaučenosti (eng. overfitting). Prenaučenost se javlja kada model odlično radi na skupu podataka za treniranje, a jako loše na skupu podataka za testiranje.

3. Implementacija stabla odluke

3.1. Odabir algoritma

Postoje brojni algoritmi čija je svrha što efektivnija izgradnja stabala odluke. Među najpopularnijima su sljedeći algoritmi:

- ID3
- C4.5
- C5.0
- CART

Za izgradnju stabla odluke koristiti će se algoritam CART (eng. Classification And Regression Tree). Ovaj algoritam brojne prednosti, a glavna je to što se može koristiti za rješavanje i klasifikacijskih i regresijskih problema. Osim toga, ovaj algoritam koristi i biblioteka scikit-learn. To je popularna biblioteka programskog jezika Python za strojno učenje. Nedostatak ovog algoritma je to što podržava samo numeričke vrijednosti podataka, pa je kategoričke podatke potrebno na neki način kodirati prije treniranja ovakvog modela.

Inače se ovakva stabla izgrađuju rekurzivno, ali u ovom radu će stabla biti izgrađena pomoću pretrage po dubini koja može zamijeniti rekurziju.

Algoritam CART sastoji se od sljedećih koraka:

1. Odabir varijable i praga koji najbolje razdvajaju podatke.
2. Podjela trenutnih podataka na dva skupa prema odabranom pragu i varijabli.
3. Za svaki od tih skupova stvoriti novi čvor i te čvorove dodati kao lijevog i desnog sljedbenika u stablu.
4. Rekurzivno ponavljati navedene korake sve do uvjeta prekida.

U kasnijim dijelovima poglavlja biti će detaljnije opisana implementacija.

3.2. Implementacija čvorova.

Čvor (eng. Node) je osnovna jedinica stabla odluke. Svaki čvor ima svoj način odluke ovisno o vrijednosti atributa i praga koji su dobiveni minimizacijom gini indeksa nečistoće ili minimizacijom srednje kvadratne pogreške. Osim toga čvor stabla sadrži i podatke koji se u

njemu nalaze, te referencu na lijevog i desnog sljedbenika. Kako bismo implementirali čvorove, stvorena je klasa TreeNode (Slika 8).

```
public class TreeNode {

    public final String featureName;
    public final double treshold;
    public final double index;
    public final DataSet X;
    public final DataSeries y;
    private TreeNode left;
    private TreeNode right;

    public TreeNode(String feature, DataSet X, DataSeries y, double treshold, double index) {
        this.X = X;
        this.y = y;
        this.featureName = feature;
        this.treshold = treshold;
        this.index = index;
        this.left = null;
        this.right = null;
    }

    public TreeNode getLeft() {
        return left;
    }

    public void setLeft(TreeNode left) {
        this.left = left;
    }

    public TreeNode getRight() {
        return right;
    }

    public void setRight(TreeNode right) {
        this.right = right;
    }

    public boolean isLeaf() {
        return this.left == null && this.right == null;
    }
}
```

Slika 8 - Klasa TreeNode

3.3. Klasa DecisionTree

Radi implementacije stabla odluke stvorena je apstraktna klasa DecisionTree koja sadrži sljedeća polja:

- **root –** instanca klase TreeNode koja predstavlja korijen stabla odluke
- **maxDepth –** predstavlja maksimalnu dubinu stabla
- **minSamplesSplit –** predstavlja minimalni broj podataka za koji se čvorovi mogu rastaviti.

Također sadrži sljedeće metode:

- **getRoot** i **setRoot –** budući da u Javi ne postoji mogućnost korištenja svojstva, polja se enkapsuliraju pomoću metoda.
- **findNode –** metoda koja u stablu odluke traži list prema zadanim podacima. Pretraga je slijepa budući da ovisi o uvjetima u svakom čvoru.
- **depth –** računa i vraća dubinu stabla odluke.
- **minimized –** računa minimiziranu vrijednost gini indeksa ili srednje kvadratne pogreške i vraća izračunatu vrijednost te atribut i prag na kojem se nalazi minimizirana vrijednost.
- **valueOfSplit –** metoda je apstraktna. Ideja je da u za zadani stupac sa atributom, stupac sa izlaznim podatkom i zadani prag pronalazi gini vrijednost ili vrijednost srednje kvadratne pogreške. Ona se treba implementirati u svakoj klasi koja nasljedi klasu DecisionTree, ovisno o tome što računa.
- **fit –** gradi (trenira)stablo odluke.
- **PredictOne** – apstraktna metoda koja predviđa podatke samo za jednu seriju. Potrebno ju je implementirati ovisno o klasifikaciji ili regresiji jer za klasifikaciju predviđenu vrijednost računa kao modalnu vrijednost podataka u listu, dok za regresiju predviđenu vrijednost računa kao aritmetičku sredinu podataka u čvoru.
- **Predict** - na osnovu metode predictOne predviđa vrijednosti za cijeli skup podataka.

Ovu klasu, ovisno o problemu, nasljeđuju klase DecisionTreeClassifier i DecisionTreeRegressor i prema potrebi implementiraju apstraktne metode.

3.4. Klasa Minimizer i određivanje pragova

Klasa Minimizer služi za praćenje podataka o vrijednosti koja se minimiziraju, a to mogu biti gini indeks nečistoće ili vrijednost srednje kvadratne pogreške. Podaci koji se nalaze u instancama ove klase odgovaraju na sljedeća pitanja:

- Koliki je iznos minimiziranog Gini indeksa?
- U kojemu se od atributu među atributima originalnih podataka nalazi minimizirani indeks nečistoće?
- Na kojem se pragu nalazi minimizirani Gini indeks?

Programski kod ove klase prikazuje Slika 9.

```java
public class Minimizer {

    public final double treshold;
    public final String feature;
    public final double index;


    public Minimizer(String feature, double treshold, double index) {
        this.treshold = treshold;
        this.index = index;
        this.feature = feature;
    }

    /**
     * Traži pragove (tresholds) za zadane podatke. Svi podaci moraju
     * biti numeričkog tipa.
     * */
    static Double[] findTresholds(DataSeries dataSeries) {

        DataSeries sorted = dataSeries.sorted().unique();
        ArrayList<Double> tresholds = new ArrayList<>();

        for (int i = 0; i < sorted.count() - 1; i++) {

            Double temp = DataOps.toDouble(sorted.get(i));
            Double next = DataOps.toDouble(sorted.get(i + 1));
            Double mean = (temp + next) / 2;

            if (!tresholds.contains(mean)) {
                tresholds.add(mean);
            }
        }

        return tresholds.toArray(new Double[0]);
    }
}
```

Slika 9 - Klasa Minimizer

Osim navedenih polja (konstanti), klasa Minimizer sadrži statičku metodu findTresholds, koja za zadanu seriju podataka određuje potencijalne pragove prema kojima se mogu određivati uvjeti čvorova. Pragovi se određuju na način da za svaki atribut stvorimo skup X koji će sadržavati jedinstvene vrijednosti stupca sa zadanim atributom uzlazno sortirane. Svaki prag se dobije na sljedeći način $prag_i = \frac{X_i + X_{i+1}}{2}, za\ svaki\ i < |X|$. Traženi prag se dobije minimizacijom vrijednosti gini indeksa ili srednje kvadratne pogreške (detaljnije u kasnijim dijelovima poglavlja).

3.5. Minimizacija

Cilj minimizacije je pronaći prag koji minimizira Gini indeks ili srednju kvadratnu pogrešku (ovisno o problemu). To se radi na način da se prvo za svaki stupac skupa primjera odaberu potancijalni pragovi, a zatim za svaki od tih pragova pomoću metode valueOfSplit (ovisi o problemu kojim se bavi) koja za zadani prag računa vrijednost Gini indeksa ili srednje kvadratne pogreške. Onaj prag koji ima najmanju izračunatu vrijednost uzima se kao prag odluke (stvara se instanca klase Minimizator) koja sadrži podatke o minimiziranom pragu,

oznaci atributa u kojoj se nalazi i minimiziranoj vrijednosti. Minimizaciju vrši metoda minimized koja se nalazi u klasi DecisionTree (Slika 10).

```
/**
  * Traži optimalni (minimizirani) indeks u zadanom trenutku. */
 public Minimizer minimized(DataSet X, DataSeries y) {

     // Postavljamo minimum.
     Minimizer min = new Minimizer("", 0, Double.MAX_VALUE);

     // Pretražujemo sve značajke (features).
     for (String dataLabel : X.getLabels()) {

         // Dohvaćamo svaki stupac (feature) DataSet-a.
         DataSeries column = X.getColumn(dataLabel);

         // Pragovi (za podjelu).
         Double[] tresholds = Minimizer.findTresholds(column);

         // Za svaki treshold.
         for (double treshold: tresholds) {

             // Računamo gini indeks trenutne podjele podataka
             Minimizer d = this.valueOfSplit(column, y, treshold);

             // Tražimo najmanji indeks u skupu podataka.
             if (d.index < min.index) {
                 min = d;
             }
         }
     }

     // Vraćamo podatak sa najmanjim indeksom.
     return min;
 }
```

Slika 10 - Minimizacija vrijednosti

3.6. Izgradnja stabla odluke.

Većina algoritama stablo izgrađuju rekurzivno. Umjesto takve metode, u ovom primjeru iskoristiti ćemo metodu pretrage po dubini (eng. Depth First Search).

Koraci su sljedeći:

1. Izračunavamo minimizirane vrijednosti za korijen stabla. Zatim korijen stabla dodajemo u stog.
2. Sve dok stog nije prazan, dohvaćamo element sa vrha stoga.
    a. Izračunavamo njegovu dubinu i broj podataka.
    b. Ako dubina i broj podataka ne prelaze zadane uvjete i vrijednost u čvoru nije potpuno minimizirana, rastavljamo čvor na lijevog i desnog sljedbenika, koje dodajemo u stog.

Stablo odluke se gradi pomoću metode fit (prikazuje Slika 11).

```java
/**
     * Metoda koja služi za nadzirano treniranje i izgradnju
     * stabla odluke.
     * */
    public void fit(DataSet xTrain, DataSeries yTrain) {

        // 'Granica' koja prati dodavanje čvorova.
        Stack<TreeNode> nodeStack = new Stack<>();

        // Izračun optimalnog indeksa za korijen.
        Minimizer gdRoot = this.minimized(xTrain, yTrain);

        // Postavljanje korijena stabla (prvog čvora odluke).
        this.setRoot(new TreeNode(gdRoot.feature, xTrain, yTrain, gdRoot.treshold, gdRoot.index));

        // Dodajemo korijen u granicu.
        nodeStack.push(this.getRoot());

        // Dodavanje se odvija sve dok nešto postoji unutar granice.
        while (!nodeStack.isEmpty()) {

            // Dohvaćanje čvora iz granice.
            TreeNode tempNode = nodeStack.pop();

            // Dohvaćanje broja podataka.
            int currentSamples = tempNode.y.count();

            // Trenutna dubina stabla
            int currentDepth = this.depth();

            // Ako vrijede uvjeti razdvajanja...
            if (currentSamples >= this.minSamplesSplit && currentDepth < this.maxDepth && !tempNode.
featureName.equals("")) {

                // Dijelimo podatke trenutnog čvora na dva dijela.
                DataSplit dataSplit = DataSplit.makeSplit(tempNode.X, tempNode.y, tempNode.treshold,
tempNode.featureName);

                // Računamo gini podatke za oba dijela.
                Minimizer minLeft = this.minimized(dataSplit.xLeft(), dataSplit.yLeft());
                Minimizer minRight = this.minimized(dataSplit.xRight(), dataSplit.yRight());

                // Stvaramo čvorove (lijevi i desni).
                TreeNode leftNode = new TreeNode(minLeft.feature, dataSplit.xLeft(), dataSplit.yLeft(),
minLeft.treshold, minLeft.index);
                TreeNode rightNode = new TreeNode(minRight.feature, dataSplit.xRight(), dataSplit.yRight
(), minRight.treshold, minRight.index);

                // Dodajemo ih trenutnom.
                tempNode.setLeft(leftNode);
                tempNode.setRight(rightNode);

                // Dodajemo ih u granicu kako bi se postupak mogao nastaviti.
                nodeStack.push(leftNode);
                nodeStack.push(rightNode);
            }
        }
    }
```

Slika 11 Metoda koja izgrađuje stablo

3.7. Predviđanje vrijednosti

Predviđanje vrijednosti u stablu odluke se bazira na slijepu pretragu čvorova u stablu. Metoda slijepe pretrage prikazana je na slici ispod (Slika 12).

```
protected TreeNode findNode(DataSeries x, String[] labels) {

        TreeNode temp = this.getRoot();

        double data;

        while (!temp.isLeaf()) {

            int featureIndex = List.of(labels).indexOf(temp.featureName);
            data = Double.parseDouble(x.get(featureIndex).toString());

            if (data <= temp.treshold) {
                temp = temp.getLeft();
            }
            else {
                temp = temp.getRight();
            }
        }

        return temp;
    }
```

Slika 12 Traženje listova u stablu odluke

Nakon što je pronađen traženi list stabla, poziva se metoda predictOne koja se posebno implementira za problem klasifikacije i problem regresije, jer za problem klasifikacije vraća modalnu vrijednost, dok za problem regresije vraća vrijednost aritmetičke sredine podataka u listu stabla.

3.8. Klasifikacijsko stablo odluke i gini indeks nečistoće

Radi implementacije klasifikacijskog stabla odluke, stvorena je klasa DecisionTreeClassifier koja nasljeđuje klasu DecisionTree i implementira metode valueOfSplit i predictOne. Također sadrži metodu gini koja služi za računanje gini indeksa nečistoće.

Gini indeks nečistoće (eng. Gini impurity) je mjera nečistoće podataka koja pomaže pri odabiru optimalne podjele u čvorovima stabla kako bi se postigla što bolja klasifikacija ili predikcija ciljne varijable. Koristi se isključivo za problem klasifikacije. Formula za izračun Gini indeksa čvora I s K klasa je sljedeća: $Gini(I) = 1 - \sum_{i=1}^{K} {p_{I,j}}^2$, gdje $p_{I,j}$ predstavlja udio označava vjerojatnost da podatak u čvoru I pripada klasi j. Cilj algoritma je minimizacija Gini indeksa za zadane podatke i odabir podatka i praga koji minimiziraju Gini indeks. Taj podatak i prag će biti uzeti kao atribut i prag prema kojemu se donosi pravilo odluke čvora. Na primjer za skup podataka $A = \{ 1, 1, 0, 1 \}$, $Gini(A)$ iznositi će $Gini(A) =$

$1 - \left(\frac{3}{5}\right)^2 - \left(\frac{2}{5}\right)^2 = 0.48$, a za skup $B = \{\, 1, 1, 1, 1 \,\}$, $Gini(B)$ iznositi će 0. Što je izračunati indeks bliži 0, to znači da čvor nema nečistoća, odnosno da je u potpunosti čist. Cijela klasa prikazana je na slici ispod (Slika 13).

```java
public final class DecisionTreeClassifier extends DecisionTree{

    /**
     * Konstruktor koji prima podatke koji prikazuju maksimalnu moguću
     * dubinu stabla (maxDepth) i minimalnu količinu podataka koji se mogu
     * nalaziti u rastavljenom čvoru (minSamplesSplit).
     * */
    public DecisionTreeClassifier(int maxDepth, int minSamplesSplit) {
        super(maxDepth, minSamplesSplit);
    }


    /**
     * Metoda koja služi za računanje Gini indeksa na nekom
     * skupu podataka. */
    public static double gini(DataSeries dataSeries) {

        double gini = 1;

        for (Integer f: dataSeries.frequencies().values()) {
            double localP = (double)f / dataSeries.count();
            gini -= Math.pow(localP, 2);
        }

        return gini;
    }


    @Override
    public Minimizer valueOfSplit(DataSeries series, DataSeries y, double treshold) {

        DataSeries below = y.getAll(series.indicesWhere(x -> DataOps.toDouble(x) < treshold));
        DataSeries above = y.getAll(series.indicesWhere(x -> DataOps.toDouble(x) > treshold));

        int belowCount = below.count();
        int aboveCount = above.count();
        int total = belowCount + aboveCount;

        double belowPart = (double) belowCount / total;
        double abovePart = 1 - belowPart;
        double gini = belowPart * gini(below) + abovePart * gini(above);

        return new Minimizer(series.getLabel(), treshold, gini);
    }


    @Override
    public double predictOne(DataSeries x, String[] labels) {

        TreeNode leaf = this.findNode(x, labels);
        return leaf.y.mode();
    }
}
```

Slika 13 Klasa DecisionTreeClassifier

3.9. Regresijsko stablo odluke

Klasa DecisionTreeRegression nasljeđuje klasu DecisionTree, kao i klasa DecisionTreeClassifier implementira metode valueOfSplit i predictOne. Razlika je u tome što se sada ne minimizira gini indeks nečistoće nego srednja kvadratna pogreška. Klasa DecisionTreeRegressor je prikazana na slici ispod (Slika 14).

```
public class DecisionTreeRegressor extends DecisionTree{

    /**
     * Konstruktor koji prima podatke koji prikazuju maksimalnu moguću
     * dubinu stabla (maxDepth) i minimalnu količinu podataka koji se mogu
     * nalaziti u rastavljenom čvoru (minSamplesSplit).
     *
     * @param maxDepth maksimalna dubina stabla.
     * @param minSamplesSplit minimalni broj podataka koji se može rastaviti.
     */
    public DecisionTreeRegressor(int maxDepth, int minSamplesSplit) {
        super(maxDepth, minSamplesSplit);
    }


    @Override
    public Minimizer valueOfSplit(DataSeries series, DataSeries y, double treshold) {

        DataSeries below = y.getAll(series.indicesWhere(x -> DataOps.toDouble(x) < treshold));
        DataSeries above = y.getAll(series.indicesWhere(x -> DataOps.toDouble(x) > treshold));

        int belowCount = below.count();
        int aboveCount = above.count();
        int total = belowCount + aboveCount;

        double belowPart = (double) belowCount / total;
        double abovePart = 1 - belowPart;

        double mse = belowPart * below.variance() + abovePart * above.variance();

        return new Minimizer(series.getLabel(), treshold, mse);
    }


    @Override
    public double predictOne(DataSeries x, String[] labels) {

        TreeNode leaf = this.findNode(x, labels);
        return leaf.y.mean();
    }
}
```

Slika 14 Klasa DecisionTreeRegressor

4. Testiranje i validacija modela

U ovom će poglavlju biti prikazano kako koristiti implementirane modele za probleme klasifikacije i regresije na prikupljenim podacima. Osim toga biti će prikazano kako evaluirati svaki od tih modela, te kako izbjeći prenaučenost (overfitting).

4.1. Validacija klasifikacijskog modela

Nakon implementacije, na red dolazi testiranje modela. To je jako važan korak jer nam govori koliko dobro model radi.

Prvi korak prilikom upotrebe modela je učitavanje podataka. Za problem klasifikacije iskoristiti ćemo podatke sa stranice Kaggle, točnije dataset Diabetes (https://www.kaggle.com/datasets/akshaydattatraykhare/diabetes-dataset) koji služi za dijagnosticiranje šećerne bolesti kod pacijenata na osnovu dobi, razine inzulina itd.

```
// Učitavanje DataSet-a.
DataSet dataSet = new DataSet();
dataSet.fromCSV(TestFiles.DIABETES_FILE, ";");

// Stupac koji predviđamo.
String feature = "Outcome";

// Nasumična podjela podataka.
DataSet[] dataSets = DataOps.splitData(dataSet, 0.2, 3);

// Skup primjera za treniranje modela.
DataSet XTrain = dataSets[0];

// Skup značajki za treniranje modela.
DataSeries yTrain = XTrain.getColumn(feature);
XTrain.dropColumn(feature);

// Skup za testiranje modela.
DataSet XTest = dataSets[1];

// Skup značajki za testiranje točnosti modela.
DataSeries yTest = XTest.getColumn(feature);
XTest.dropColumn(feature);

// Stvaranje i treniranje modela.
DecisionTreeClassifier model = new DecisionTreeClassifier(5, 20);
model.fit(XTrain, yTrain);

// Predviđanje vrijednosti.
DataSeries predicted = model.predict(XTest);

// Izračun točnosti.
double acc = ModelMetrics.accuracy(yTest, predicted);
System.out.println("Accuracy: " + acc);
```

Slika 15 Testiranje klasifikacijskog modela

Nakon što smo učitali podatke u DataSet, odabrati ćemo oznaku koju predviđamo. To će za ovaj slučaj biti oznaka 'Outcome'. Zatim ćemo podatke podijeliti na dva skupa. Prvi skup će sadržavati podatke za treniranje, a drugi skup će sadržavati podatke na kojima ćemo testirati model. Skup za treniranje će sadržavati oko 80% podataka, a skup za testiranje ostalih 20%. Podaci za oba skupa će biti odabrani nasumično. Potrebno je podatke za treniranje i testiranje modela podijeliti i na seriju oznaka i skup primjera.

Zatim stvaramo model sa željenim hiperparametrima (o tome malo kasnije) i treniramo ga na podacima za treniranje. Sada je model prilagođen podacima za treniranje i može se koristiti za klasifikaciju pacijenata u kategorije oboljelih ili zdravih pacijenata.

Međutim, uvjek nam je potrebna povratna informacija koliko je taj naš model zapravo dobar. Za validaciju modela strojnog učenja postoje razne metode koje se koriste oviso o samom modelu kojeg se procjenjuje. Za problem klasifikacije koristi se metrika točnosti (eng. accuracy) koja govori o omjeru broja točno klasificiranih podataka i broja svih podataka. Za naš model točnost iznosi oko 74%. Cijeli postupak prikazan je na slici ispod (Slika 15).

4.2. Testiranje regresijskog modela

Za razliku od klasifikacijskog modela, budući da se kod regresijskog modela predviđaju kontinuirane vrijednosti, mjera točnosti nam neće dati dobre rezultate.

Mjere koje se najčešće koriste za validaciju regresijskih modela su sljedeće:

- Srednja aplsolutna pogreška (eng. Mean Absolute Error)

$$MAE = \frac{1}{n}\sum_{i=1}^{n} | y^{(i)} - y_{pred}^{(i)} |$$

- Srednja kvadratna pogreška (eng. Mean Squared Error)

$$MSE = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - y_{pred}^{(i)}\right)^2$$

- Korijen iz srednje kvadratne pogreške (eng. Root Mean Squared Error)

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - y_{pred}^{(i)}\right)^2}$$

Gdje je

- $y^{(i)}$ – stvarni podatak.
- $y_{pred}^{(i)}$ – predviđeni podatak.
- n – broj primjeraka.

Uzmimo na primjer srednju kvadratnu pogrešku. Što je vrijednost te pogreške manja, možemo zaključiti da je model bolji. Recimo npr. da srednja kvadratna pogreška iznosi 10. Nije svejedno predviđamo li cijenu mliječnih proizvoda za koju je to stvarno velika pogreška ili cijenu nekretnine gdje je to tek jako mala i prihvatljiva pogreška.

Na slici ispod (Slika 16) prikazano je testiranje regresijskog modela (na podacima https://www.kaggle.com/datasets/ialabISEP/footballsoccerstatistics?select=dataset_football_cleaned.csv). Skup podataka ima podatke o raznim igračima, ali mi ćemo odabrati samo neke atribute tih podataka, a to su:

- Dob igrača
- Cijena
- Broj poziva u reprezentaciju
- Liga u kojoj igra
- Broj golova
- Broj asistencija

Među učitanim podacima ima kategoričkih vrijednosti, pa je njih potrebno kodirati prije samog treniranja modela. Nadalje je postupak sličan kao i kod klasifikacije. Potrebno je podijeliti podatke na skupove za treniranje i testiranje itd.

Npr. Srednja apsolutna pogreška kod ovog skupa podataka je jako velika (iznosi 1955904.51). Na prvu možemo reći kako je model jako loš, no međutim ovdje postoji nekoliko parametara koji to opovrgavaju. Prvi je veličina samog DataSet-a koji sadrži 12177 primjeraka. Drugi razlog mogu biti same cijene nogometaša, ako znamo da se današnje cijene nogometaša kreću u desetcima i stotinama milijuna, možemo zaključiti da naš model čak i nije toliko loš, te ako pogriješi za oko 2 milijuna po igraču, to i nije toliko velika pogreška. Cijeli postupak je prikazan na slici ispod (Slika 16).

```java
// Odabir podataka koji će biti učitani
HashSet<String> filters = new HashSet<>();
filters.add("age");
filters.add("price");
filters.add("selections");
filters.add("league");
filters.add("goal_champ");
filters.add("assist_champ");

// Učitavanje skupa podataka.
DataSet dataSet = new DataSet();
dataSet.fromCSV(TestFiles.FOOTBALL_FILE, ";", filters);

// Kodiranje kategoričkih vrijednosti.
for (String label: dataSet.getLabels()) {
    DataSeries column = dataSet.getColumn(label);
    if (column.isCategorical()) {
        column.codeValues(column.autoCoding());
    }
}

// Atribut kojega predviđamo.
String feature = "price";

// Razdvajamo podatke u skupove za treniranje i testiranje.
DataSet[] setSplit = DataOps.splitData(dataSet, 0.3, 2);

// Skup za treniranje.
DataSet Xtrain = setSplit[0];
DataSeries yTrain = Xtrain.getColumn(feature);
Xtrain.dropColumn(feature);

// Skup za testiranje.
DataSet Xtest = setSplit[1];
DataSeries yTest = Xtest.getColumn(feature);
Xtest.dropColumn(feature);

// Stvaranje i testiranje regresijskog modela.
DecisionTreeRegressor model = new DecisionTreeRegressor(5, 3);
model.fit(Xtrain, yTrain);

// Predviđene vrijednosti modela.
DataSeries yPred = model.predict(Xtest);

// Rezultati validacije modela.
System.out.println("MAE: " + ModelMetrics.meanAbsoluteError(yTest, yPred));
System.out.println("MSE: " + ModelMetrics.meanSquaredError(yTest, yPred));
System.out.println("RMSE: " + ModelMetrics.rootMeanSquaredError(yTest, yPred));
```

Slika 16 Testiranje regresijskog modela.

4.3. Prenaučenost i odabir najboljih hiperparametara modela

Prilikom stvaranja modela stabla odluke, konstruktoru možemo proslijediti dva podatka, a to su maksimalna dubina stabla i minimialni broj podataka u čvoru za koje je dopušteno rastavljanje. Ti arumenti se zovu hiperparametri. Postavlja se pitanje kako pronaći najbolje vrijednosti hiperparametara.

Fokusirajmo se za sada na maksimalnu dubinu stabla. Ako odaberemo previše malu dubinu stabla, riskiramo da se model neće dobro naučiti i neće dobro predviđati podatke jer će donositi previše malen broj odluka prilikom procjene. Ako se ipak odlučimo za prevelik broj

riskiramo prenaučenost (eng. overfitting), odnosno da će model perfektno raditi na skupu za treniranje, a jako loše na skupu za testiranje (odnosno previše će se prilagoditi podacima iz skupa za treniranje).

Jedan od načina je traženje svih mogućih kombinacija hiperparametara (max_depth, min_split_samples) i odabir one koja ima najveću točnost. U ovom primjeru ćemo promatrati klasifikacijski model iz prošlog poglavlja (Slika 15). Način odabira najboljih hiperparametara prikazuje Slika 17.

```
// Spremanje svih mogućih kombinacija.
HashMap<Double, Integer[]> combinations = new HashMap<>();

for (int maxDepth = 0; maxDepth < 10; maxDepth++) {
    for (int minSamples = 2; minSamples < 10; minSamples++) {

        // Stvaranje i treniranje modela.
        DecisionTreeClassifier model = new DecisionTreeClassifier(maxDepth, minSamples);
        model.fit(XTrain, yTrain);

        // Predviđanje vrijednosti.
        DataSeries predicted = model.predict(XTest);

        // Rezultati validacije modela.
        double accuracy = ModelMetrics.accuracy(yTest, predicted);

        combinations.put(accuracy, new Integer[] { maxDepth, minSamples });
    }
}

// Ispis svih parova.
for (Double acc: combinations.keySet()) {
    System.out.println("acc: " + acc + " - " + Arrays.toString(combinations.get(acc)));
}
```

Slika 17 - Traženje najboljih hiperparametara

Zaključak

Stabla odluke se nalaze među najpopularnijim modelima strojnog učenja i spadaju u kategoriju nadziranog učenja budući da se prilikom treniranja model „uči“ da prilagođava izlazne podatke podacima iz skupa za treniranje. Prednost im je što su lako prilagodljiva za bilo kakvu distribuciju podataka i jednostavna su za interpretaciju, jer svaki čvor predstavlja „if-else“ pravilo kojim se dolazi do zaključaka.

Stabla odluke mogu riješavati probleme klasifikacije, odnosno pridruživanje skupa ulaznih podatak diskretnoj klasi izlaznih podataka, i regresije tj. predviđanje kontinuiranih vrijednosti. Najpopularniji algoritam za izgradnju stabla odluke je CART (eng. Classification And Regression Trees) upravo zato jer podržava i klasifikacijska i regresijska stabla odluke. Kakvo god stablo da izgrađuje, ovaj algoritam radi na principu minimizacije. Ako se radi o klasifikaciji, minimizira se Gini indeks nečistoće podataka, a ako se radi o regresiji, minimizira se srednja kvadratna pogreška.

U strojnom učenju važno je i dobiti povratnu informaciju o modelu putem validacije. Ako testiramo klasifikacijski model promatramo točnost, a ako testiramo regresijski model promatramo srednju apsolutnu pogrešku, srednju kvadratnu pogrešku ili korijen iz srednje kvadratne pogreške.

Također, važan faktor je i odabir ispravnih hiperparametara kako ne bi došlo do loše naučenosti (eng. underfitting) ili prevelike naučenosti na skupu za treniranje (eng. overfitting).

Popis slika

Literatura

[1] https://scikit-learn.org/stable/modules/tree.html ATM FORUM, User-Network Interface (UNI) Specification, http://www.atmforum.com, 4. 4. 2010.

[2] https://www.zemris.fer.hr/education/ml/nastava/ag20022003/3-stabla-odluke.pdf.

[3] *https://www.unite.ai/hr/%C5%A1to-je-stablo-odlu%C4%8Divanja/*

[4] https://www.geeksforgeeks.org/decision-tree-implementation-python/