

Technická univerzita v Košiciach  
Fakulta elektrotechniky a informatiky

# Bagging

Práca z predmetu  
Strojové učenie

**Technická univerzita v Košiciach**  
**Fakulta elektrotechniky a informatiky**

**Bagging**

**Práca z predmetu**  
**Strojové učenie**

Študijný program: Inteligentné systémy  
Študijný odbor: Informatika  
Školiace pracovisko: Katedra kybernetiky a umelej inteligencie (KKUI)

**Košice 2024**

**Arsenii Milenchuk**

## Zadanie práce

Cieľom zadania je naprogramovať bagging pomocou suboru RS

- Nie je dovolené využívať naprogramované funkcie algoritmu z nejakej knižnice. Ani metriky ako Presnosť, Správnosť, Návratnosť, F1 miera nemôžete implementovať z knižnice, ale musíte sami naprogramovať.
- Algoritmus má byť naprogramovaný všeobecne. Nemá byť prispôsobený na konkrétne dáta, ale na ľubovoľnú dátovú množinu.
- Dátová množina pre experimenty má byť reálna/z praxe. Dáta z cvičenia nestačia. Dáta majú mať aspoň 4 popisné atribúty (+ cieľový atribút) a aspoň 20 záznamov. V odôvodnených a prekonzultovaných prípadoch to môže byť aj inak.
- Kód má obsahovať komentáre v slovenčine.
- Dokumentácia má obsahovať:
  - Teoretický popis algoritmu
  - Popis postupu a jednotlivých funkcií
  - Popis dát (vysvetlenie, popis atribútov, počet záznamov, zdroj)
  - Vyhodnotenie (metriky, interpretácia výsledkov, záver)
- Dokumentácia má byť napísaná v šablóne záverečných prác (aby naučili s ňou pracovať).
- Všetko čo ste robili má byť popísané v dokumentácii.
- Pozor na správne citovanie zdrojov.
- Hodnotí sa aj množstvo práce a úsilia.
- Na obhajobu zadania je 1 pokus.

# Contents

<b>1</b>	<b>Teoretický popis algoritmu</b>	<b>1</b>
1.1	Definícia baggingu . . . . .	1
1.2	Historický kontext baggingu . . . . .	2
1.3	Základné princípy baggingu . . . . .	3
1.4	Druhy vzorkovania pri baggingu . . . . .	5
1.5	Rozchodovacie stromy C4.5 . . . . .	6
1.5.1	Základné princípy a fungovanie . . . . .	6
1.5.2	Delenie priestoru príznakov v algoritme C4.5 . . . . .	8
1.6	Metaforické vysvetlenie baggingu . . . . .	8
1.7	Teoretický zaver . . . . .	9
<b>2</b>	<b>Popis postupu a jednotlivých funkcií</b>	<b>10</b>
2.1	Knižnice . . . . .	10
2.2	ReadCSV . . . . .	10
2.2.1	__init__ . . . . .	11
2.2.2	_prepare_data . . . . .	12
2.3	DataSplitter . . . . .	13
2.3.1	split_data . . . . .	14
2.3.2	make_portion . . . . .	15
2.4	BasicTreeMethod . . . . .	16
2.4.1	_entropy . . . . .	17
2.4.2	_conditional_entropy . . . . .	18
2.4.3	_info_gain . . . . .	19
2.4.4	_shannon_entropy . . . . .	19
2.4.5	_normalize_information_gain . . . . .	20
2.4.6	_find_count_pos_and_neg . . . . .	21
2.4.7	_find_count_for_number . . . . .	21
2.4.8	_find_pos_neg . . . . .	22

---

2.5	Node . . . . .	22
2.6	Leaf . . . . .	24
2.7	CN45 . . . . .	25
2.7.1	make_tree . . . . .	26
2.7.2	_separate_space . . . . .	28
2.7.3	make_prediction . . . . .	29
2.7.4	_give_new_space_numeric . . . . .	30
2.7.5	_give_new_space_attr . . . . .	32
2.7.6	_calculate_gain_numeric . . . . .	33
2.7.7	_calculate_gain . . . . .	34
2.8	BaggingCN45 . . . . .	35
2.8.1	make_prediction . . . . .	36
2.9	mlMetrics . . . . .	37
2.9.1	Accuracy . . . . .	37
2.9.2	Confusion matrix . . . . .	38
2.9.3	Precision . . . . .	38
2.9.4	F1 Score . . . . .	38
2.9.5	Reccal . . . . .	38
<b>3</b>	<b>Popis dát</b>	<b>40</b>
3.1	Employee.csv . . . . .	40
3.2	apple_quality.csv . . . . .	41
3.3	startupdata.csv . . . . .	42
<b>4</b>	<b>Vyhodnotenie</b>	<b>45</b>
4.1	Testovanie CN4.5 . . . . .	45
4.2	Testovanie Bagging . . . . .	47
4.3	Zmena parametrov bagging . . . . .	50
4.4	Zaver . . . . .	53

---

**Zoznam použitej literatúry****55**

## List of Figures

1–1	Leo Breiman . . . . .	3
1–2	Princíp delenia údajov v bagginge . . . . .	4
1–3	Priklad stromu . . . . .	6
1–4	Hlasovanie . . . . .	9
4–1	Vplyv veľkosti testovacej vzorky na presnosť CN4.5 nad 100 iterácií Employee.csv . . . . .	45
4–2	Vplyv veľkosti testovacej vzorky na presnosť CN4.5 nad 100 iterácií apple_quality.csv . . . . .	46
4–3	Vplyv veľkosti testovacej vzorky na presnosť CN4.5 nad 100 iterácií startupdata.csv . . . . .	47
4–4	Metriky pomocou Bagging spolu s piatimi stromomy Employee.csv	48
4–5	Metriky pomocou jedného stromu Employee.csv . . . . .	49
4–6	Metriky pomocou Bagging spolu s jedným stromom Employee.csv .	49
4–7	Použitie rôzneho počtu stromov v algoritme Employee.csv . . . . .	50
4–8	Použitie rôzneho množstva bootstrapu Employee.csv . . . . .	51
4–9	Použitie bootstrapu s hodnotou 5 pre rôzny počet stromov Employee.csv	52
4–10	Rozdiel medzi používaním Bagging a jedným rozhodovacím stromom na množine údajov apple_quality.csv . . . . .	53
4–11	Rozdiel medzi používaním Bagging a jedným rozhodovacím stromom na množine údajov startupdata.csv . . . . .	54

# 1 Teoretický popis algoritmu

## 1.1 Definícia baggingu

Bagging, alebo bootstrap agregácia, je jeden z ansámblových meta-algortimov v strojovom učení. Tento metóda má za cieľ zlepšiť stabilitu a znížiť rozptyl predikcií algoritmov, ktoré sa často používajú na riešenie úloh klasifikácie a regresie. Tu je podrobnejšie vysvetlenie jeho princípov a mechanizmov:

- **Bootstrap:**

Názov "bootstrap" pochádza zo štatistickej metódy bootstrappingu, ktorá zahŕňa viacnásobné vytváranie vzoriek z jednej pôvodnej sady údajov. Toto sa dosahuje náhodným výberom prvkov s možnosťou opätovného zaradenia tých istých prvkov do každej novej vzorky. Tento prístup umožňuje vytvoriť rôznorodé tréningové dátové sady z obmedzeného množstva pôvodných údajov.

- **Agregácia:**

Agregácia výsledkov v baggingu sa deje prostredníctvom priemerovania výstupov jednotlivých modelov (pre regresiu) alebo použitím metódy hlasovania (pre klasifikáciu). Toto priemerovanie pomáha znížiť rozptyl predpovedí, pretože náhodné chyby v predpovediach jednotlivých modelov sa často vzájomne kompenzujú.

- **Viacnásobné učenie:**

V baggingu sa každá bootstrapová dátová sada používa na tréning novšej verzie základného modelu. Takto, ak je pôvodný model náchylný na preučenie kvôli vysokému citlivosti na dáta tréningu, použitie mnohých takýchto modelov, ktoré sú tréňované na rôznych dátových sadách, môže výrazne zlepšiť schopnosť ansámbľu generalizovať.



## 1.2 Historický kontext baggingu

Bagging bol predstavený Leo Breimanom v roku 1994 a predstavuje významný krok vývoja v oblasti strojového učenia. Tu je podrobnejšie vysvetlenie jeho historického vývoja a mechanizmov:

- **Vznik metódy:**

Leo Breiman rozvinul koncept baggingu (bootstrap agregácia) ako spôsob, ako zvýšiť spoľahlivosť modelov strojového učenia tým, že vytvorí viacero verzií modelu a následne ich kombinuje. Breiman identifikoval, že tento prístup môže pomôcť znížiť variácie v predikciách, ktoré často vznikajú z citlivosti modelu na malé zmeny v tréningových dátach.

- **Princípy a motivácia:**

Hlavná myšlienka baggingu je založená na znižovaní rozptylu predikcií bez zvyšovania bias (systémové zaujatie), čo sa dosahuje prostredníctvom tréningu jednotlivých modelov na rôznych bootstrap vzorkách z tréningových dát. Breiman ukázal, že priemerné predikcie z viacerých modelov vedú k stabilnejším a presnejším výsledkom, ako predikcie z jednotlivého modelu.

- **Vplyv a prijatie:**

Od svojho uvedenia sa bagging stal základom pre mnohé ďalšie techniky strojového učenia, vrátane náhodných lesov, ktoré sú dnes jednou z najpopulárnejších a najúčinnějších techník v oblasti prediktívneho modelovania a strojového učenia. Tento prístup má široké uplatnenie nielen v akademických kruhoch, ale aj v praxi, kde sa používa na riešenie rôznych problémov od kreditného skórovania po medicínsku diagnostiku.

- **Technický vývoj a inovácie:**

Breimanov príspevok do strojového učenia a štatistiky cez metódu baggingu pomohol objasniť, ako môže redukcia variance v predikciách zlepšiť celkovú presnosť modelu v nepredvídateľných alebo vysoke variabilných dátových prostrediach.

Metóda zároveň ilustruje, ako môže diverzifikácia v tréningových dátach viesť k robustnejším rozhodnutiam.

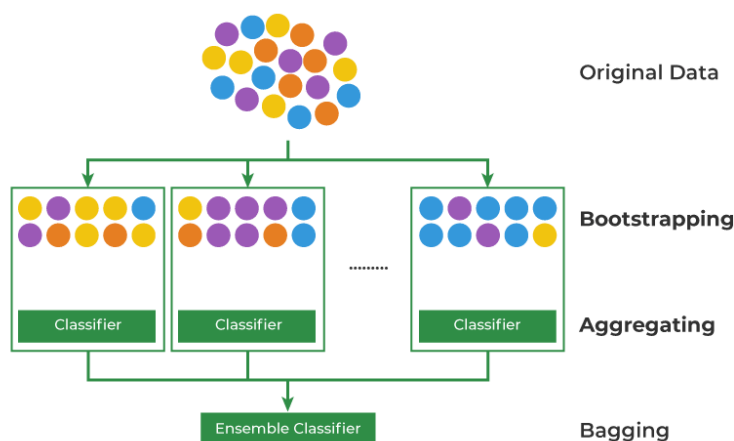


Obrázok 1 – 1 Leo Breiman

### 1.3 Základné princípy baggingu

Bagging je technika strojového učenia, ktorá využíva súborové metódy pre zlepšenie výsledkov a stabilnosti modelov. Začína sa generovaním mnohých nezávislých tréningových súborov z pôvodného datasetu pomocou metódy bootstrap, čo znamená náhodné vzorkovanie s opakovaným začlenením. Tento proces vytvára "bootstrap vzorky", ktoré môžu obsahovať opakované inštancie pôvodných dát. Na každej z týchto bootstrap vzoriek sa nezávisle trénuje model. Tieto modely môžu byť rovnakého typu, ako sú napríklad rozhodovacie stromy, ale každý model sa trénuje na mierne odlišnej vzorke údajov, čo zaisťuje diverzifikáciu v modelovaní. Po trénovaní sa výstupy z každého modelu kombinujú do jedného výsledného modelu. V prípade regresie sa výstupy modelov zvyčajne priemerujú, zatiaľ čo pri klasifikácii sa využíva hlasovanie väčšiny alebo vážené hlasovanie. Keďže jednotlivé modely sú trénované na rôznych vzorkách, výsledné predikcie sú menej náchylné na nadmerné prispôbenie

sa konkrétnym vlastnostiam tréningových údajov, čo vedie k nižšej variabilite a potenciálne vyššej presnosti predikcií ako u jednotlivého modelu. Bagging môže byť aplikovaný na širokú škálu modelov strojového učenia a je obzvlášť účinný pri modeloch, ktoré sú náchylné k veľkej variabilite vo svojich predpovediach. Táto metóda je tiež cenná v prípadoch, kde modely vykazujú značné riziko preučenia. Bagging poskytuje robustný spôsob, ako zvýšiť spoľahlivosť a efektívnosť predikčných modelov prostredníctvom systematického vzorkovania a integrácie výsledkov z mnohých modelov, čím sa znižuje riziko chýb vplyvom nadmerného prispôsobenia alebo anomálií v dátach.



**Obrázok 1 – 2** Princíp delenia údajov v bagginge

Bagging znamená vytvorenie mnohých rôznych tréningových sád pomocou náhodného výberu s návratom z pôvodnej sady údajov. Napríklad, ak máme tréningovú sadu s 1000 príkladmi, bagging môže vytvoriť stovky nových tréningových sád, každá s 1000 príkladmi, kde niektoré príklady sa opakujú a iné možno chýbajú.

Každá z týchto tréningových sád sa používa na tréning jedného modelu. Ak je základným modelom rozchodovacie stromy, každý strom sa naučí na mierne odlišnej sade údajov, čo vedie k rozdielom vo výsledkoch, ktoré tieto stromy generujú.

Keďže každý model (napríklad rozchodovacie stromy) v ansámble bol vyškolený na mierne odlišnej podmnožine údajov, ich individuálne predpovede môžu byť rôzne.

Agregovaním (napríklad priemerovaním alebo hlasovaním) týchto predpovedí získame konečnú predpoveď, ktorá je typicky presnejšia a robustnejšia než predpovede získané od individuálnych modelov.

## 1.4 Druhy vzorkovania pri baggingu

Bagging využíva rôzne druhy vzorkovania na generovanie viacerých trénovacích podmnožín z pôvodnej dátovej sady, čím umožňuje vytvoriť viacero nezávislých a odlišných modelov. V tejto kapitole sa pozrieme na rôzne prístupy k vzorkovaniu používané v baggingu, ako sú vzorkovanie s vrátením, vzorkovanie bez vrátenia, náhodné podpriestory a náhodné záplaty.

- **Vzorkovanie s vrátením (Bootstrap sampling)** je najčastejšie používaná metóda pri baggingu, kde každý vzorok z pôvodnej dátovej sady môže byť vybraný viackrát pri tvorbe podmnožiny. Po vytvorení týchto podmnožín sa jednotlivé modely učia na každej z nich. Predikcie modelov sú následne agregované, napríklad hlasovaním pre väčšinu pri klasifikácii alebo priemerovaním pre regresiu.
- **Vzorkovanie bez vrátenia (Pasting)** sa používa menej často, pretože môže viesť k vytváraniu menších a menej rozmanitých podmnožín, čo môže znižovať efektívnosť metód v porovnaní s bootstrapom. Táto metóda je vhodnejšia v situáciách, kde je veľká dátová sada a potrebujeme rýchlejšie výsledky, pretože každý vzorok je vybraný iba raz.
- **Náhodné podpriestory (Random Subspaces) a Náhodné záplaty (Random Patches)** sú menej tradičné, ale veľmi účinné pri znižovaní korelácie medzi

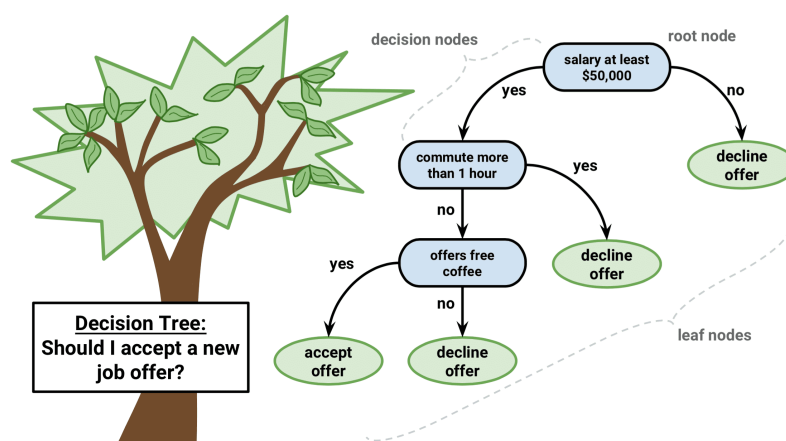
jednotlivými modelmi v ensemble. Náhodné podpriestory sa zameriavajú na vzorkovanie prvkov a sú vhodné, keď máme dataset s veľkým počtom prvkov. Náhodné záplaty kombinujú myšlienky náhodných podpriestorov a baggingu/pastingu a sú ideálne pri veľkých a rozmanitých datasetoch, kde je potrebné znižovať zaujatosť aj varianciu.

Medzi všetkými týmito technikami, vzorkovanie s vrátením ostáva najpopulárnejšou a najčastejšie používanou technikou vďaka svojej schopnosti efektívne znižovať varianciu bez zvyšovania zaujatosti, čo je kľúčové pre stabilné a robustné predikčné modely.

## 1.5 Rozchodovacie stromy C4.5

### 1.5.1 Základné princípy a fungovanie

Rozchodovacie stromy C4.5 sú rozšírením algoritmu ID3 a boli navrhnuté Rossom Quinlanom v 90-tych rokoch. Algoritmus C4.5 je obľúbený pre svoju schopnosť generovať klasifikačné modely, ktoré sú ľahko interpretačné a účinné pri riešení rôznych typov úloh predikcie. V tejto kapitole sa podrobne pozrieme na princípy, podľa ktorých algoritmus C4.5 pracuje, a na jeho využitie v rámci baggingu.



Obrázok 1 – 3 Příklad stromu

- **1. Základné princípy a fungovanie**

Konštrukcia stromu: Algoritmus C4.5 začína so všetkými trénovacími dátami v koreňovom uzle a rozdeľuje údaje rekurzívne na základe entropickej miery zisku (gain ratio). Každé rozdelenie (alebo "rozvetvenie") je vykonané tak, aby maximalizovalo informačný zisk z vybraného atribútu medzi dostupnými atribútmi.

Prerezávanie stromu: Na rozdiel od ID3, C4.5 implementuje postrezávanie stromu, čo pomáha predchádzať nadmernému prispôbeniu sa dátam. Prerezávanie znižuje veľkosť stromu a zlepšuje jeho generalizačnú schopnosť odstraňovaním sekcií stromu, ktoré poskytujú malý prínos k prediktívnemu výkonu modelu.

- **2. Výber atribútu**

Gain Ratio: C4.5 vylepšuje kritérium výberu atribútu používané v ID3 použitím pomery zisku, ktorý normalizuje informačný zisk podľa inherentnej informačnej hodnoty rozdelenia atribútu. To pomáha predchádzať predsudkom voči atribútom s veľkým počtom hodnôt.

- **3. Práca s rôznymi typmi dát**

C4.5 dokáže efektívne spracovávať ako numerické, tak kategorické atribúty. Pri numerických atribútoch algoritmus hľadá optimálny prahový bod pre rozdelenie dát na dve skupiny, zatiaľ čo kategorické atribúty sú rozdelené na základe jednotlivých kategórií.

- **4. Chýbajúce hodnoty**

Zvládanie chýbajúcich hodnôt: C4.5 dokáže zaobchádzať s chýbajúcimi hodnotami tým, že pridelí pravdepodobnosti rôznym rozdeleniam na základe dostupných dát. To umožňuje stromu rásť, aj keď nie všetky údaje sú úplné.

- **5. Využitie v baggingu**

Keď sa C4.5 používa v rámci baggingu, každý strom je vyškolený na odlišnej podvzorke dát vytvorenej bootstrappingom. Kombinácia viacerých stromov

C4.5 v ansámble zvyšuje robustnosť modelu tým, že redukuje varianciu bez zvyšovania zaujatosti, čo výrazne zlepšuje výkonnosť v predikcii.

### 1.5.2 Delenie priestoru príznakov v algoritme C4.5

Pri rozhodovacích stromoch, ako je C4.5, delenie priestoru príznakov je kľúčovým krokom na vytvorenie modelu, ktorý môže efektívne klasifikovať dáta. C4.5 používa špecifické metódy na určenie, ako najlepšie rozdeliť dáta na uzly v strome. Tu je podrobnejší opis, ako sa to deje, vrátane výpočtu, ktorý sa používa na optimalizáciu tohto procesu:

- **Výber príznaku pre rozdelenie:**

Pri každom uzle stromu C4.5 vyhodnotí, ktorý príznak (atribút) z dostupných dát je najlepší na rozdelenie údajov na ďalšie uzly. Výber sa robí na základe 'pomery zisku', ktorý je vylepšenou metódou informačného zisku. Pomer zisku pomáha predchádzať predsudkom voči príznakom s veľkým počtom hodnôt.

- **Výpočet pomery zisku:**

Pomer zisku sa počíta ako informačný zisk delený intrinzičnou informáciou príznaku. Informačný zisk meria, o koľko sa zlepši predikcia (zníži entropia) využitím daného príznaku na rozdelenie údajov. Intrinsicá informácia príznaku hodnotí, ako je príznak sám o sebe rozdelený, nezávisle od cieľovej premennej, čo pomáha normalizovať zisk.

$$GR(S, A) = \frac{IG(S, A)}{IV(A)} \quad (1.1)$$

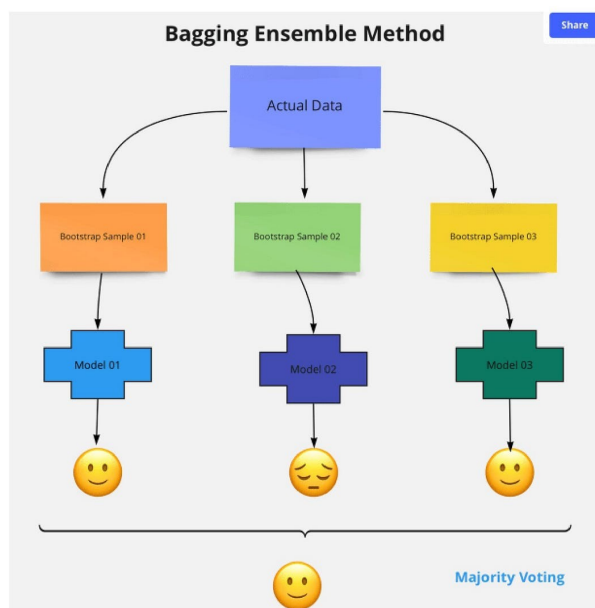
## 1.6 Metaforické vysvetlenie baggingu

Predstavte si, že máte skupinu expertov, ktorí sa nezávisle pokúšajú vyriešiť problém. Každý expert má trochu iný pohľad na situáciu v dôsledku rôznych skúseností a informácií, ktoré majú k dispozícii. Ak by každý z nich predložil svoj návrh riešenia a výsledné riešenie by bolo založené na hlasovaní všetkých expertov, je pravdepodobné,

že konečné riešenie by bolo lepšie ako návrhy poskytnuté individuálne. Bagging funguje na podobnom princípe – kombinuje predikcie z viacerých modelov na zlepšenie celkovej predikcie a stability systému.

## 1.7 Teoreticky zaver

Bagging je všestranný nástroj, ktorý nachádza uplatnenie v rôznych oblastiach dátových vied. Od bankovníctva a finančných služieb, cez zdravotníctvo, až po bezpečnostné systémy, bagging pomáha zvyšovať presnosť prediktívnych modelov tým, že efektívne znižuje riziko preučenia. Najčastejšie sa používa s rozhodovacími stromami, ale môže byť aplikovaný aj s inými typmi modelov.



Obrázok 1 – 4 Hlasovanie



## 2 Popis postupu a jednotlivých funkcí

### 2.1 Knižnice

V mojej práci boli použité tieto knižnice: `pandas`, `math`, `copy`, `random` a `tqdm`. Jedným z bodov úlohy bolo nepoužívať knižnice tretích strán, ktoré už implementujú niektoré funkcie algoritmu. `Pandas` potrebujem na čítanie csv súboru, balík `math` implementuje funkciu `log`, balík `copy` slúži len na hlboké kopírovanie obsahu sekvencií (väčšinou slovníky), tiež balík `random` a `tqdm` na zobrazenie indikátorov. Za zmienku tiež stojí, že moje riešenie nepoužívalo `numpy` knižnicu, ktorej použitie poskytuje výrazné zvýšenie rýchlosti vykonávania programu vďaka efektívnej implementácii vstavaných funkcií a použitiu typizovaných údajov, čo vedie k optimalizovanému využitiu pamäte a skrátený výpočtový čas. Chcel som sa pokúsiť napísať algoritmus pomocou štandardných dátových typov Python, v mojom prípade sa dátové typy - zoznam a `dict`, používajú hlavne na ukladanie hodnôt.

```
1 import pandas as pd
2 import math as mth
3 import copy
4 import random
5 from tqdm import tqdm
```

### 2.2 ReadCSV

Prvou úlohou, ktorú je potrebné vyriešiť, je čítanie súboru. Ak to chcete urobiť, vytvorte triedu, ktorá bude zodpovedná za čítanie súboru **ReadCSV**.

Trieda musí implementovať konštruktor, v ktorom sa súbor číta, potom potrebujeme vlastnosti, aby sme mohli získať vstupné a výstupné dáta. Bolo by tiež pekné, keby naša trieda implementovala metódu filtrovania údajov (bohužiaľ ju nebudeme potrebovať). Metóda spracuje dáta a vyčistí ich od duplikátov a zbaví dáta šumu (body na mape príznačov, ktoré sa nachádzajú na rovnakom mieste, ale majú odlišné štítky výstupných dát).

```
1 class ReadCsv():
2     def __init__(self, file_path, className, label):
3         pass
4
5     def get_x(self):
6         pass
7
8     def get_y(self):
9         pass
10
11     def _prepare_data(self, x, y):
12         pass
```

Pozrime sa bližšie na implementáciu metód.

### 2.2.1 `__init__`

Konštruktor plne implementuje čítanie údajov zo súboru. Čítanie prebieha spolu s randomizáciou údajov, od samého začiatku (vďaka pandas). Ďalej údaje transformujeme do slovníka pomocou pandas, ale bohužiaľ potrebujeme ďalšiu transformáciu, keďže momentálne ide o slovník, v ktorom sú kľúče polohou riadku. Rozhodneme sa urobiť slovník, v ktorom budú kľúčmi názvy atribútov a ich hodnotou bude mapa funkcií pre tieto atribúty tiež potrebujeme rozdeliť dáta na prichádzajúce a odchádzajúce. Výstupné dáta jednoducho uložíme ako zoznam. + a - sa začali používať preto, lebo pri testovaní programu a prezeraní výstupu cez konzolu sa príjemnejšie pozeralo na + a - ako na 0/1, dobre/zle a podobne. Táto hodnota je špecifikovaná spolu s kľúčom (-l, -label) pri spustení programu.

Konštruktor berie ako parametre umiestnenie dátového súboru, názov stĺpca s výstupnými dátami a názov atribútu, ktorý bude považovaný za pozitívny príklad (+).

```
1 def __init__(self, file_path, className, label):
2     print("ReadCsv")
3     data = pd.read_csv(file_path).sample(frac=1)
4     data_dict = data.to_dict()
5     self._dictData = dict()
6     for i in data_dict:
7         self._dictData[i] = [j[1] for j in data_dict[i].items()]
8     data_dict = copy.deepcopy(self._dictData)
9     self._y = []
10    for i in data_dict.pop(className):
11        if str(i) == label:
12            self._y.append('+')
13        else:
14            self._y.append('-')
15    self._x = data_dict
```

### 2.2.2 \_\_prepare\_data

Ďalej máme metódu na filtrovanie údajov. Funkcia jednoducho prejde všetkými znakmi a potom potrebujeme ďalší cyklus na nájdenie duplicitných alebo zašumených údajov. Za týmto účelom jednoducho zhromaždíme slovník s hodnotami pre každý atribút a potom sa porovná od suseda až po koniec, označené indexy sa pridajú do zoznamu na následné odstránenie. Vyzerá to ako rad v nemocnici, ak príde pacient s unikátnou históriou, každého, kto je v rade s rovnakou históriou po ňom, odtiaľ jednoducho vyhodia. Ako parametre berie vstupnu a vystupnu množinu.

```
1 def _prepare_data(self, x, y):
2     pbar = tqdm(total=len(y)+1, desc="Prepare data", unit="iter")
3     similarity = []
4     for idx, y_item in enumerate(y):
5         main_item = {i: x[i][idx] for i in x}
6         for idx_2 in range(idx+1, len(y)):
7             some_dict = {i: x[i][idx_2] for i in x}
8             if some_dict == main_item:
9                 similarity.append(idx)
10        pbar.update(1)
11    similarity = list(set(similarity))
12    new_x, new_y = {key: [] for key in x}, []
13    for idx in range(len(y)):
14        if idx in similarity:
15            continue
16        for key in x:
17            new_x[key].append(x[key][idx])
18            new_y.append(y[idx])
19    pbar.update(1)
20    pbar.close()
21    return new_x, new_y
```

Metóda je uzavretá, pretože v určitom štádiu vývoja, keď bola táto metóda požadovaná, sa myslelo, že táto metóda sa bude používať vždy a bude jednoduchšie ju volať z konštruktora. Momentálne sa táto funkcia v konštruktore nepoužíva a nepotrebujeme ju, takže ju môžeme nastaviť ako voliteľnú, napríklad pomocou metódy, ktorá po zavolaní bude dáta filtrovať.

```
1 def prepare_data(self):
2     self._x, self._y = self._prepare_data(self._x, self._y)
```

## 2.3 DataSplitter

Táto trieda, **DataSplitter**, je navrhnutá na pohodlné rozdelenie súboru údajov na tréningové a testovacie vzorky, ako aj na rozdelenie údajov na časti. Tieto funkcie

zohrávajú kľúčovú úlohu v procese tréningovania a hodnotenia modelov strojového učenia. Rozdelenie dát na tréningové a testovacie vzorky nám umožňuje vyhodnotiť výkonnosť modelu na nezávislom dátovom súbore, a metóda delenie na porcie, využijeme priamo pri bagging.

```
1 class DataSplitter():
2     def __init__(self, x, y):
3         pass
4
5     def split_data(self, train_per):
6         pass
7
8     def make_portion(self, portions_count):
9         pass
```

Pozrime sa bližšie na to, ako tieto metódy fungujú vo vnútri triedy. V konštruktoze nie je nič zaujímavé, iba údaje sú priradené k súkromným premenným v triede. Pozrime sa bližšie na ďalšie dve

### 2.3.1 split\_data

Pri tejto metóde chceme dáta rozdeliť na tréningové a testovacie na základe percentuálnej hodnoty pre tréningový set. Výpočet prahu pre množiny sa počíta celkom jednoducho, vypočítame poslednú hodnotu v tréningovej množine ako jej index a potom použijeme slicing na rozdelenie. Vrátime štyri hodnoty - vstupy a výstupy pre tréningové a testovacie množiny.

```
1 def split_data(self, train_per):
2     train_length = int(len(self._y) / 100 * train_per)
3     x = {i for i in self._x}
4     train_x = dict()
5     test_x = dict()
6     train_y = self._y[:train_length]
7     test_y = self._y[train_length:]
8     for i in self._x:
9         train_x[i] = self._x[i][:train_length]
10        test_x[i] = self._x[i][train_length:]
11    return [train_x, train_y], [test_x, test_y]
```

### 2.3.2 make\_portion

Táto metóda je zodpovedná za oddelenie údajov pre samotné odpočúvanie. Parametre nám takpovediac určujú počet porcií, do ktorých musíme dáta rozdeliť. Implementácia je trochu nezvyčajná. Tu najprv vypočítame, aká celočíselná veľkosť môže rozdeliť dáta na  $n$  častí, pomocou celočíselného delenia nájdeme to, ale sú tu nejaké priznaky, ktoré ostanú po hlavnom delení, keďže na začiatku bolo použité celočíselné delenie. O týchto piznakoch sa dá povedať, že sú šťastné, pridávajú sa do každej porcie. Naše časti teda obsahujú absolútne všetky priznaky z údajov.

```
1 def make_portion(self, portions_count):
2     portions = []
3     portion_size = len(self._y) // portions_count
4     start = 0
5     end = portion_size
6     for i in range(portions_count):
7         new_x = dict()
8         for i in self._x:
9             new_x[i] = self._x[i][start:end]
10        portions.append([new_x, self._y[start:end]])
11        start = end
12        end += portion_size
13    for i in self._y[start:]:
14        for x, y in portions:
15            for key in self._x:
16                x[key].append(self._x[key][start])
17            y.append(i)
18        start += 1
19    return portions
```

## 2.4 BasicTreeMethod

Ďalej potrebujeme triedu, ktorá bude obsahovať všetky základné metódy stromu, teda entropiu, informačný rast, ako aj metódu, pomocou ktorej vieme spočítať počet pozitívnych a negatívnych príkladov pre rôzne situácie. Z tejto triedy robíme základnú triedu pre dedenie stromov, ktoré sú už špecifické pre implementáciu. Implementácia rozdelenia stromu do podpriestorov a tiež prechádzanie stromom na vyhľadávanie bude implementované v špecifickej implementácii stromu, takže rôzne implementácie môžu mať odlišný algoritmus pre tieto akcie. Z dôvodu zložitosti implementácie, tak aj z dôvodu, že sme chceli vytvoriť modulárne riešenie, ktoré dokáže pracovať s rôznymi typmi dát, výsledkom je pomerne komplikované riešenie, ktoré môže mať rôznu implementáciu napríklad uzlov rozhodovacieho stromu.

Všetky metódy sú privatné, bolo to urobené s cieľom, aby tieto metódy mohli používať iba zdedené stromové triedy v nasledujúcich zložitejších častiach algoritmu.

```
1 class BasicTreeMethod():
2
3     def _entropy(self, positive, negative, allCount):
4         pass
5
6     def _conditional_entropy(self, probability, entropy):
7         pass
8
9     def _info_gain(self, entropy, conditional_entropy):
10        pass
11
12    def _shannon_entropy(self, probability):
13        pass
14
15    def _normalize_information_gain(self, info_gain,
16    shannon_entropy):
17        pass
18
19    def _find_count_pos_and_neg(self, x, y, key, attr):
20        pass
21
22    def _find_count_for_number(self, x, y, key, number, larger):
23        pass
24
25    def _find_pos_neg(self, y):
26        pass
```

### 2.4.1 \_\_entropy

"Entropia je koncept, ktorý sa využíva v rôznych odvetviach, vrátane teoretickej informatiky, pravdepodobnosti a strojového učenia. Vo strojovom učení sa entropia často používa ako miera neistoty alebo neporiadku v dátach. Čím vyššia je entropia,



tým viac zmätených alebo neporovnateľných informácií je v datasete. V praxi sa entropia často používa pri rozhodovacích stromoch a klasifikačných modeloch na hodnotenie kvality rôznych rozhodnutí a rozdelení dát. Cieľom je minimalizovať entropiu, čo znamená, že sa snažíme redukovať neistotu v dátach a vytvoriť čo najlepší model pre predikciu alebo klasifikáciu.

Implementácia metódy zodpovedá vzorcu entropie. Ak je počet záporných alebo pozitívnych nula, potom je v údajoch poriadok a entropia je nula, ak sú všetky hodnoty na mieste, vypočítame entropiu.

```
1 def _entropy(self, positive, negative, allCount):
2     if positive == 0 or negative == 0:
3         return 0
4     entropy = (positive/allCount * math.log2(positive/allCount) +
5               negative/allCount * math.log2(negative/allCount)) * -1
6     return entropy
```

$$S = - \sum_i p_i \log p_i \quad (2.1)$$

#### 2.4.2 \_\_conditional\_entropy

Podmienená entropia je miera neistoty alebo neporiadku v dátach, ktorá sa vypočíta za predpokladu, že poznáme určitú dodatočnú informáciu. V podstate to znamená, že ak máme závislé udalosti, podmienená entropia nám dáva mieru neistoty o jednej udalosti, keď vieme o výsledku druhej udalosti.

V našej metóde najskôr skontrolujeme formát parametrov a ich význam, ak nespĺňajú očakávania, vypíšeme chybu. Ak je všetko v poriadku s danými parametrami, vypočítame podmienenú entropiu pomocou jej vzorca.

```
1 def _conditional_entropy(self, probability, entropy):
2     if len(probability) == 0 or len(probability) != len(entropy) or
3       len(probability[0]) != 2:
4         raise ValueError("[BasicTreeMethod] condition_entropy have
5                             wrong value")
6     result = 0
```

```
5     for i in range(len(entropy)):
6         result += probability[i][0]/probability[i][1] * entropy[i]
7     return result
```

$$H(Y|X) = \sum_{i=1}^n \left( \frac{p_i}{q} \times E_i \right) \quad (2.2)$$

### 2.4.3 \_\_info\_gain

Informačný prírast je miera, ktorá posudzuje, do akej miery pridanie nových informácií znižuje neistotu v dátach. Informačný prírast umožňuje určiť, aké rozdelenie dát na základe určitého príznaku alebo podmienky bude najinformatívnejšie pre klasifikáciu alebo predpovedanie. Metóda je tiež jednoduchá a presne sa riadi vzorcom. Preto nie je potrebné vysvetľovať

```
1 def __info_gain(self, entropy, conditional_entropy):
2     return entropy - conditional_entropy
```

$$IG(Y|X) = H(Y) - H(Y|X) \quad (2.3)$$

### 2.4.4 \_\_shannon\_entropy

Shannonova entropia je miera neistoty alebo nepredvídateľnosti v systéme. Vyjadruje, ako veľmi sme neistí o tom, akú hodnotu môže nadobudnúť určitá náhodná premenná. Čím vyššia je entropia, tým väčšia je neistota.

Táto metóda je tiež implementovaná podľa vzorca. Najprv sa skontroluje správnosť parametrov, potom sa vypočíta Shannonova entropia. Tu spracujeme chybu, pretože pravdepodobnosť môže byť nulová, čo nemôže byť v logaritme, čo znamená, že ak sa spustí chyba, potom je entropia nulová.

```

1 def _shannon_entropy(self, probability):
2     if len(probability) == 0 or len(probability[0]) != 2:
3         raise ValueError("[CN45] shannon_entropy have wrong value")
4     result = 0
5     for i in range(len(probability)):
6         prob = probability[i][0]/probability[i][1]
7         try:
8             result += prob * math.log2(prob)
9         except ValueError:
10             return 0
11     return result * -1

```

$$H(Y) = - \sum_{i=1}^n p_i \log_2(p_i) \quad (2.4)$$

#### 2.4.5 \_\_normalize\_information\_gain

Normalizácia informačného prírastku je postup, ktorý sa používa na porovnanie významnosti rôznych atribútov pri tvorbe stromov. Keďže informačný prírastok závisí na počte možných hodnôt atribútu, môže byť zavádzajúce pri porovnávaní atribútov s rôznym počtom hodnôt. Normalizácia informačného prírastku umožňuje vyvážiť tento efekt a zabezpečiť, že hodnoty informačného prírastku sú v porovnateľnom rozsahu, čo uľahčuje rozhodovanie o tom, ktorý atribút je najinformatívnejší. Implementácia sa tiež nelíši od vzorca. Ak je Shannonova entropia nulová, potom je informačný zisk nulový.

```

1 def __normalize_information_gain(self, info_gain, shannon_entropy):
2     try:
3         return info_gain / shannon_entropy
4     except ZeroDivisionError:
5         return 0

```

$$(IG_{\text{norm}}) = \frac{IG(Y|X)}{H(X)} \quad (2.5)$$

#### 2.4.6 `__find_count_pos_and_neg`

Ďalej sú uvedené tri podobné metódy na nájdenie pozitívnych a negatívnych vlastností v podmnožine. Táto metóda je zodpovedná za nájdenie týchto hodnôt, ak to chceme urobiť pomocou atribútu. Všetko je tu jednoduché, najprv sa pozrieme, či sa dĺžky polí zhodujú, pre prípad nepríjemných prekvapení. Ďalej jednoducho v slučke nájdeme hodnoty, ktoré spĺňajú naše požiadavky, pomocou kľúča a názvu atribútu.

```
1 def __find_count_pos_and_neg(self, x, y, key, attr):
2     if len(x[key]) != len(y): raise ValueError("
    find_count_pos_and_neg")
3     pos = neg = allCount = 0
4     for idx, item in enumerate(y):
5         if x[key][idx] == attr:
6             allCount += 1
7             if item == "+":
8                 pos += 1
9             else:
10                neg += 1
11     return pos, neg, allCount
```

#### 2.4.7 `__find_count_for_number`

Táto metóda zistí počet kladných a záporných známk, už z číselného atribútu môže metóda nájsť hodnoty pre stav väčšie alebo menšie.

```
1 def _find_count_for_number(self, x, y, key, number, larger):
2     pos = neg = allCount = 0
3     if larger:
4         new_arr = [(idx, item) for idx, item in enumerate(x[key])
5                     if float(item) > number]
6     else:
7         new_arr = [(idx, item) for idx, item in enumerate(x[key])
8                     if float(item) < number]
9     for idx, item in new_arr:
10        if y[idx] == '+':
11            pos += 1
12        else:
13            neg += 1
14    return pos, neg, len(new_arr)
```

#### 2.4.8 \_\_find\_pos\_neg

Táto metóda je zodpovedná za zistenie počtu pozitívnych a negatívnych príkladov jednoducho pre nejaký súbor výstupných údajov.

```
1 def _find_pos_neg(self, y):
2     pos = 0
3     neg = 0
4     for i in y:
5         if i == "+":
6             pos += 1
7         else:
8             neg += 1
9     return pos, neg, len(y)
```

## 2.5 Node

Uzly v rozhodovacích stromoch predstavujú bod, ktorý rozdeľuje dáta na menšie podskupiny. Každý uzol testuje hodnotu určitého atribútu a na základe výsledku

testu smeruje dáta do príslušnej vetvy stromu. Cieľom každého uzlu je čo najviac zredukovať neistotu alebo nečistotu v dátach, čo umožňuje efektívnejšie klasifikovanie príkladov.

Uzol musí predstavovať uzol, ktorý oddeľuje údaje podľa číselného atribútu, ako aj podľa kategorického atribútu. Preto je implementované nasledovné, táto implementácia nie je najefektívnejšia. Rozhodli sme sa, že pre kategorické dáta budeme ukladať hodnotu atribútu spolu s jeho potomkom, ktorému zodpovedá (keďže rozdelenie môže byť do viacerých kategórií). Pri číselných údajoch, kde vieme, že len my máme dva poduzly (menej a viac), použijeme toto číslo v hodnote uzla (keďže existuje len jedna hodnota). attrName - názov atribútu.

Implementujú sa aj setre a getre.

```
1 class Node():
2     def __init__(self, parent, attr_value=None, child_attr_value=
    None):
3         self._parent = parent
4         if self._parent is not None:
5             self._parent.add_child(self, child_attr_value)
6         self._childs = []
7         self._attr_name = None
8         self._attr_value = None
9
10    def set_attr_name(self, attr_name):
11        self._attr_name = attr_name
12
13    def set_attr_value(self, attr_value):
14        self._attr_value = attr_value
15
16    def add_child(self, child, attr_value=None):
17        self._childs.append([child, attr_value])
18
19    @property
20    def attr_value(self):
```

```
21         return self._attr_value
22
23     @property
24     def attr_name(self):
25         return self._attr_name
26
27     def get_childs(self):
28         return self._childs
```

## 2.6 Leaf

Listy rozhodovacích stromov sú koncové uzly, ktoré nevedú k ďalšiemu rozdeleniu. Každý list obsahuje príklady, ktoré majú podobné vlastnosti alebo patria do tej istej kategórie. Výsledná klasifikácia príkladu je určená väčšinou triedou príkladov v liste alebo najpravdepodobnejším výsledkom na základe trénovacích dát.

List nie je zvlášť zaujímavý. Pri implementácii sme sa rozhodli, že to nebude uzol, ktorý bude pridávať listy, ale bude to hlásiť jeho dieťa. Rovnaké pravidlo platí pre situáciu uzol-uzol, kde podriadený uzol hlási, že je podriadený, a nie nadradený uzol, ktorý pridáva deti.

```
1 class Leaf():
2     def __init__(self, className, parent, parent_attr_value=None):
3         self._className = className
4         self._parent = parent
5         self._parent.add_child(self, parent_attr_value)
6
7     @property
8     def ClassName(self):
9         return self._className
```

## 2.7 CN45

Ďalej musíme implementovať samotný strom CN4.5. Trieda zdedí triedu so základnými vzorcami na prístup k nim a my implementujeme zostávajúce metódy. Strom dokáže spracovať kategorické aj číselné údaje. Boli pokusy čo najviac kombinovať a minimalizovať opakovanie v kóde, ale bohužiaľ nebolo možné urobiť všetko tak, ako bolo zamýšľané. Chcel som čo najviac minimalizovať rozdiely medzi výpočtami pre kategorické a číselné údaje.

Trieda implementuje 7 metód. `maketree` implementuje hlavnú funkciu stromu, a to jeho výstavbu, táto metóda je podobná šablonej metóde, ktorá kombinuje iné metódy z tejto triedy. Chcel by som hneď povedať, že rozhodovací strom je algoritmus rozdelenia a panovania, ktorý sa ideálne vykonáva pomocou rekurzcie. Chcel som však nájsť v strome čo najšpecifické riešenie, jeden z výberov obsahoval 4 tisíc riadkov údajov, čo znamená, že to bude okamžite StackOverflow. Preto sa riešenie ukázalo ako dosť ťažké a nie krásne v porovnaní s klasickou rekurziou. Rozdelili a panovali sme v mnohých cykloch.

Vysvetlenie nasledujúcich funkcií si necháme na kapitoly im venované. V konštruktoze inicializujeme hlavný uzol a premenné na uloženie našich priestorov.



```
1 class CN45(BasicTreeMethod):
2     def __init__(self, x, y):
3         self._x = x
4         self._y = y
5         self._root = Node(None)
6         self._total_raws = len(self._y)
7
8     def _get_root_node(self):
9         return self._root
10    def make_tree(self):
11        pass
12    def _separate_space(self, x, y, result, keys, parent):
13        pass
14    def make_prediction(self, x):
15        pass
16    def _give_new_space_numeric(self, x, y, best_key, attr_value,
17                                parent):
18        pass
19    def _give_new_space_attr(self, x, y, best_key, attr_value,
20                              parent):
21        pass
22    def _calculate_gain_numeric(self, x, y, total_entropy, attr):
23        pass
24    def _calculate_gain(self, x, y, total_entropy, attr):
25        pass
```

### 2.7.1 make\_tree

Toto je naša základná metoda, ktorá je zodpovedná za stavbu stromu. Na zobrazenie priebehu stavby stromu používame indikátor. Najprv inicializujeme priestor funkcií(celkova trenovacia množina) a uzol(hlavný uzol stromu). Potom používame cyklus, až kým nedosiahneme bod, v ktorom sme identifikovali všetky znaky zo sady do niekoľkých listov, keď je náš strom úplne vytvorený a už nie sú žiadne znaky na rozdelenie, cyklus sa končí. Ďalej prichádza slučka, ktorá iteračne prechádza cez všetky zostávajúce

priestory, ktoré neboli definované ako listy. Vypočíta sa im entropia a potom iteratívne vypočítame informačný zisk pri delení buď číslami alebo kategóriami. Ďalej sú výsledky odovzdané metóde, ktorá rozdelí náš priestor na podpriestor. Ďalej pomocou slučky skontrolujeme výsledný priestor, či sme dostali list alebo uzol. Ak je naším objektom list, potom je podpriestor definitívny a tento podpriestor do ďalších iterácií nezahŕňame. Ak je naším objektom uzol, tak ho musíme ďalej deliť, a preto ho prenášame ďalej do nových podpriestorov. Po dokončení delenia podpriestorov v aktuálnej iterácii aktualizujeme ich hodnotu pomocou objektu, ktorý ukladá ďalšie podpriestory s uzlami.

```

1 def make_tree(self):
2     itter = 0
3     pbar = tqdm(total=self._total_raws, desc="Building tree", unit=
4         "iter")
5     space_and_nodes = [[self._x, self._y, self._root]]
6     while len(space_and_nodes) != 0:
7         new_space_and_nodes = []
8         for x, y, node in space_and_nodes:
9             total_entropy = self._entropy(*self._find_pos_neg(y))
10            result = list()
11            keys = list()
12            for key in x:
13                keys.append(key)
14                try:
15                    int(float(x[key][0]))
16                    result.append(self._calculate_gain_numeric(x, y,
17                        total_entropy, key))
18                except ValueError:
19                    result.append(self._calculate_gain(x, y,
20                        total_entropy, key))
21            for i in self._separate_space(x, y, result, keys, node):
22                if type(i[2]) != Leaf:
23                    new_space_and_nodes.append(i)
24                    continue
25            pbar.update(len(i[1]))
26            space_and_nodes = new_space_and_nodes
27    pbar.close()

```

### 2.7.2 \_\_separate\_space

Ďalej prichádza metóda na rozdelenie priestoru na podpriestor. Najprv skontrolujeme najlepši informačný zisk (maximum), ak sa rovná nule, potom nemáme priestor čím rozdeliť, čo znamená, že musíme okamžite vytvoriť list a nepokračovať. Ak náš zisk nie je nulový, ideme ďalej, aby sme našli atribút, ktorý zodpovedá maximálnemu

zisku. Ak je takýchto hodnôt viacero, vyberieme ich náhodne. Ďalej, v závislosti od toho, že sme to vypočítali pre číselný alebo kategorický atribút, zavoláme zodpovedajúcu metódu, ktorá rozdelí priestor. Tieto metódy budú ukázané a vysvetlené neskôr.

```

1 def _separate_space(self, x, y, result, keys, parent):
2     best_gain = max(list(map(lambda x: x[1], result)))
3     if best_gain == 0:
4         plus, minus, all_count = self._find_pos_neg(y)
5         return [[None, y, Leaf('+' if plus >= minus else '-', parent
6     )]]
7     best = {}
8     for idx, item in enumerate(result):
9         if item[1] == best_gain:
10            best[keys[idx]] = item
11    if len(best) > 1:
12        best_key, [attr_value, best_score] = random.choice(list(
13    best.items()))
14    else:
15        best_key, [attr_value, best_score] = list(best.items())[0]
16    if attr_value is not None:
17        return self._give_new_space_numeric(x, y, best_key,
18    attr_value, parent)
19    else:
20        return self._give_new_space_attr(x, y, best_key, attr_value
21    , parent)

```

### 2.7.3 make\_prediction

Táto metóda je zodpovedná za predikciu. Metóda musí prejsť celým stromom v závislosti od jeho delení podľa atribútov príkladu, ktorého triedu chceme predpovedať. Začneme hľadať, kým nenájdeme prvý list, ktorého triedu použijeme ako výsledok metódy. Najprv určíme názov a hodnotu atribútu, potom ako predtým prejdeme do nového uzla v závislosti od typu atribútu. Potom nájdeme potomka, ktorý zodpovedá našim údajom, a tak ďalej, kým nenájdeme určitý list.

```
1 def make_prediction(self, x):
2     current_pos = self._root
3     finish = False
4     while not finish:
5         attrName = current_pos.attr_name
6         attrValue = current_pos.attr_value
7         new_pos = None
8         if current_pos.attr_value is not None:
9             if float(x[current_pos.attr_name]) > float(current_pos.
attr_value):
10                 new_pos = current_pos.get_childs()[0][0]
11             else:
12                 new_pos = current_pos.get_childs()[1][0]
13         else:
14             for i in current_pos.get_childs():
15                 try:
16                     if x[attrName] == i[1]:
17                         new_pos = i[0]
18                 except KeyError:
19                     return i[0].ClassName
20         current_pos = new_pos
21         if type(current_pos) is Leaf:
22             finish = True
23         elif current_pos is None:
24             print("random")
25             return random.choice(['+', '-'])
26     return current_pos.ClassName
```

#### 2.7.4 \_\_give\_new\_space\_numeric

Táto metóda je zodpovedná za vytvorenie nového podpriestoru, ak rozdelíme pomocou hodnoty na dva uzly, ktoré zodpovedajú hodnote väčšej a menšej ako. Najprv vytvoríme premenné na uloženie zoradeného súboru. Zoradíme množinu do týchto polí podľa väčšej alebo menšej podmienky. Ďalej pripravíme návratové pole. Hodnotu

nastavíme na náš nadradený uzol. A prejdeme do fázy vytvárania uzlov alebo listov v strome. Máme niekoľko podmienok, ak je entropia nulová, potom naše údaje majú rovnakú hodnotu a vytvoríme list. Ak nie, najskôr skontrolujeme priestor atribútov, aby sa nerovnal 0, ak sa rovná nule, potom nie je potrebné deliť, prípadne skontrolujeme počet atribútov v uzle, táto hodnota sa môže meniť v závislosti od preferencií a údajov. Dá sa povedať, že to sú naše podmienky na orezanie stromu. Ak nie je splnená žiadna z podmienok, existujú všetky podmienky na vytvorenie uzla a pokračovanie v rozdelení.

```

1 def _give_new_space_numeric(self, x, y, best_key, attr_value,
    parent):
2     result = [{i:[] for i in x} for _ in range(2)]
3     y_new = [[] for _ in range(2)]
4     for idx, item in enumerate(x[best_key]):
5         if float(item) > attr_value:
6             y_new[0].append(y[idx])
7             for i in x:
8                 result[0][i].append(x[i][idx])
9         else:
10            y_new[1].append(y[idx])
11            for i in x:
12                result[1][i].append(x[i][idx])
13    for_return = []
14    parent.set_attr_name(best_key)
15    parent.set_attr_value(attr_value)
16    for idx, item in enumerate(y_new):
17        for key in copy.deepcopy(result[idx]):
18            if len(set(result[idx][key])) <= 1:
19                result[idx].pop(key)
20        if self._entropy(*self._find_pos_neg(item)) == 0:
21            for_return.append([None, item, Leaf(item[0], parent)])
22        elif len(result[idx]) == 0 or len(item) < 3:
23            plus, minus, all_count = self._find_pos_neg(item)

```

```
24         for_return.append([None, item, Leaf('+ ' if plus >=
    minus else '- ', parent)])
25     else:
26         for_return.append([result[idx], item, Node(parent)])
27     return for_return
```

### 2.7.5 \_\_give\_new\_space\_attr

Táto metóda je v mnohom podobná predchádzajúcej, len je zodpovedná za rozdelenie priestoru pre kategorické atribúty. Nemohol som urobiť tieto dve metódy o nič krajšie, pretože keď prišiel moment na prepracovanie kódu, jednoducho som nedokázal tieto dve metódy navzájom krásne skombinovať. Najprv nájdeme pole všetkých hodnôt atribútu best, rovnako ako naposledy inicializujeme polia pre nové podpriestory. Ďalej ich oddelíme podľa hodnoty ich atribútu. Ďalej prichádzajú rovnaké podmienky ako v predchádzajúcej metóde. Jediná pozitívna vec, podľa môjho názoru, že som tieto dve metódy neskombinoval do jednej je, že správanie stromu v podmienkach s typom atribútov môžeme riadiť rôznymi spôsobmi, pričom jediná implementácia by mohla zúžiť rozsah toho, čo je povolené.

```

1 def _give_new_space_attr(self, x, y, best_key, attr_value, parent):
2     set_attributes = set([i for i in x[best_key]])
3     result = {_: {i: [] for i in x if i != best_key or len(set(x[i]))
4 >1} for _ in list(set_attributes)}
5     y_new = {_: [] for _ in range(2) for _ in list(set_attributes)}
6     for idx, item in enumerate(x[best_key]):
7         y_new[item].append(y[idx])
8         for i in x:
9             if i != best_key or len(set(x[i])) > 1:
10                 result[item][i].append(x[i][idx])
11
12     for_return = []
13     parent.set_attr_name(best_key)
14     parent.set_attr_value(attr_value)
15     for i in result:
16         for key in copy.deepcopy(result[i]):
17             if len(set(result[i][key])) <= 1:
18                 result[i].pop(key)
19             if self._entropy(*self._find_pos_neg(y_new[i])) == 0:
20                 for_return.append([None, y_new[i], Leaf(y_new[i][0],
21 parent, i)])
22             elif len(result[i]) == 0 or len(y_new[i]) <= 3:
23                 plus, minus, all_count = self._find_pos_neg(y_new[i])
24                 for_return.append([None, y_new[i], Leaf('+' if plus >=
25 minus else '-', parent, i)])
26             else:
27                 for_return.append([result[i], y_new[i], Node(parent,
28 child_attr_value=i)])
29     return for_return

```

### 2.7.6 \_\_calculate\_gain\_numeric

Ďalej prichádza metóda na výpočet informačného zisku pre číselné údaje. Všetko sa robí podľa klasického príkladu. Postupne nájdeme hodnoty pre každú hodnotu a



potom nájdeme tú najlepšiu a vrátime ju.

```
1 def _calculate_gain_numeric(self, x, y, total_entropy, attr):
2     numb_set = sorted([i for i in set(x[attr])])
3     numb_set = [(float(numb_set[idx])+float(numb_set[idx+1]))/2 for
4                 idx in range(len(numb_set)-1)]
5     best = [0, 0]
6     for key in numb_set:
7         vals = []
8         vals.append(self._find_count_for_number(x, y, attr, key,
9         True))
10        vals.append(self._find_count_for_number(x, y, attr, key,
11        False))
12        entropies = []
13        probabilities = []
14        for val in vals:
15            entropies.append(self._entropy(*val))
16            probabilities.append((val[2], len(y)))
17            condition_entr = self._conditional_entropy(probabilities,
18            entropies)
19            info_gain = self._info_gain(total_entropy, condition_entr)
20            shannon = self._shannon_entropy(probabilities)
21            norm_gain = self._normalize_information_gain(info_gain,
22            shannon)
23            if norm_gain > best[1]:
24                best[1] = norm_gain
25                best[0] = key
26        return best
```

### 2.7.7 \_\_calculate\_gain

Táto metóda už vypočítava informačný zisk pre kategorické hodnoty. Tiež všetko samotné sa robí podľa klasickej metódy. Jediný rozdiel je v tom, že teraz nemusíme hľadať najlepšiu hodnotu, ako to bolo v prípade číselných údajov, keďže tie počítajú niekoľko potenciálnych hodnôt na delenie, z ktorých si musíme jednu vybrať. V

tejto metóde musíme vrátiť iba hodnotu prírastku pre tento atribút.

```

1 def _calculate_gain(self, x, y, total_entropy, attr):
2     attr_entropy = []
3     probabilities = []
4     for key in set(x[attr]):
5         pos, neg, allCount = self._find_count_pos_and_neg(x, y,
6 attr, key)
7         attr_entropy.append(self._entropy(pos, neg, allCount))
8         probabilities.append((allCount, len(y)))
9     condition_entr = self._conditional_entropy(probabilities,
10 attr_entropy)
11     info_gain = self._info_gain(total_entropy, condition_entr)
12     shannon = self._shannon_entropy(probabilities)
13     norm_gain = self._normalize_information_gain(info_gain, shannon
14 )
15     return [None, norm_gain];

```

## 2.8 BaggingCN45

Ďalej sa pozrime na našu základnú metódu delenia údajov na časti.

Naša trieda má konštruktor, v ktorom inicializujeme všetky potrebné údaje a vytvárame kúsky.

```

1 class BaggingCN45():
2     def __init__(self, x, y, tree_count, portion_count, boot_strap)
3     :
4         pass
5     def make_prediction(self, test_x, test_y, print_accuracy=False)
6     :
7         pass

```

V konštrukte berieme ako parametre vstupné a výstupné dáta, počet stromov, počet častí, do ktorých musíme rozdeliť naše tréningové dáta a počet častí, ktoré by sme mali použiť pri boot\_strap. Pomocou DataSplitter rozdeľujeme naše tréningové

dáta na určitý počet častí. Použitá separačná metóda bola Bootstrap sampling. V tom istom cykle vytvárame pole so stromami. Stromy trénujeme aj v konštruktérovi.

```
1 def __init__(self, x, y, tree_count, portion_count: int, boot_strap
  : int):
2     self._dataSplitter = DataSplitter(x, y)
3     self._portions = self._dataSplitter.make_portion(portion_count)
4     self._trees = []
5     for i in range(tree_count):
6         portion = [{key: [] for key in x}, []]
7         for _ in range(boot_strap):
8             some = random.choice(self._portions)
9             for key in some[0]:
10                 portion[0][key] += some[0][key]
11             portion[1] += some[1]
12         self._trees.append(CN45(portion[0], portion[1]))
13     for i in self._trees:
14         i.make_tree()
```

### 2.8.1 make\_prediction

Naša ďalšia metóda je zodpovedná za predikciu v stromoch, mnohé testované vlastnosti sú odovzdávané ako parametre, ktorých triedy musíme uhádnuť. V každej iterácii vygenerujeme jeden testovací prípad, ktorý potom prechádza každým stromom a výsledok sa zaznamená do poľa. Ďalej si volíme, ktorú triedu predpovedáme hlasovaním. Potom môžete tiež odvodiť presnosť pre testovaciu sadu.

```

1 def make_prediction(self, test_x, test_y, print_accuracy=False):
2     randomKey = random.choice([i for i in test_x])
3     prediction = []
4     for idx, item in enumerate(test_x[randomKey]):
5         x_1 = dict()
6         for _ in test_x:
7             x_1[_] = test_x[_][idx]
8         y_predicts = []
9         for tree in self._trees:
10             y_predicts.append(tree.make_prediction(x_1))
11         if y_predicts.count('++') > y_predicts.count('--'):
12             prediction.append('+' == test_y[idx])
13         else:
14             prediction.append('--' == test_y[idx])
15
16     if print_accuracy:
17         print(len(test_y))
18         accuracy = metrics.accuracy(sum(prediction), len(test_y))
19         print(accuracy)

```

## 2.9 mlMetrics

Ďalšou triedou, ktorú sme implementovali, boli metriky. V triede sú metriky implementované podľa ich vzorcov.

### 2.9.1 Accuracy

Presnosť. Platnosť analytického postupu vyjadruje stupeň zhody medzi hodnotou akceptovanou ako zistená skutočná hodnota alebo akceptovanou referenčnou hodnotou a zistenou hodnotou. Niekedy sa tomu hovorí pravda.

```

1 def accuracy(correct_pred, total_number):
2     return correct_pred/total_number

```

$$acc = \frac{\text{Correct prediction}}{\text{TM length}} \quad (2.6)$$

### 2.9.2 Confusion matrix

Aby bolo možné porovnávať predpovede a realitu, Data Science používa zmätokovú maticu – tabuľku so 4 rôznymi kombináciami predpovedaných a skutočných hodnôt. Predpokladané hodnoty sú opísané ako pozitívne a negatívne, zatiaľ čo skutočné hodnoty sú opísané ako pravdivé a nepravdivé.

```
1 def confusionMatrix(tp, tn, fp, fn):  
2     mat = pd.DataFrame([[tp, fp], [fn, tn]], index=['Positive', 'Negative'], columns=['Positive', 'Negative'])  
3     print(mat)
```

### 2.9.3 Precision

Presnosť je chápaná ako miera, do akej výsledky získané v procese výskumu, meraní a experimentov zodpovedajú skutočným hodnotám.

```
1 def precision(tp, fp):  
2     return tp/(tp+fp)
```

$$Precision = \frac{TP}{TP + FP} \quad (2.7)$$

### 2.9.4 F1 Score

F1 je metrika, ktorá sa pokúša zohľadňovať presnosť, keď sú triedy nevyvážené, pričom sa zameriava na presnosť pozitívnych predpovedí a skutočne pozitívnych záznamov.

```
1 def f1Scores(precision, recall):  
2     return 2*((precision*recall)/(precision+recall))
```

$$F1score = 2 * \frac{precision * recall}{precision + recall} \quad (2.8)$$

### 2.9.5 Reccal

Reccal (tiež známe ako citlivosť) je podiel relevantných prípadov, ktoré boli nájdené.

```
1 def recall(tp, fn):  
2     return tp/(tp+fn)
```

$$Recall = \frac{TP}{TP + FN} \quad (2.9)$$

## 3 Popis dát

Na testovanie výkonu algoritmu som použil tri rôzne súbory údajov.

### 3.1 Employee.csv

Prvý súbor údajov obsahuje údaje o pracovníkoch odchádzajúcich zo zamestnania a snažíme sa ich predpovedať. Súbor obsahuje 4654 záznamov. Nižšie si môžete pozrieť popis tohto súboru údajov od Kaggle. Tento súbor údajov nájdete na tomto odkaze.

(<https://www.kaggle.com/datasets/tawfikelmetwally/employee-dataset>)(text odkazu, ak nie je zobrazený v dokumente)

O množine údajov(z Kaggle)

**Kontext:** Tento súbor údajov obsahuje informácie o zamestnancoch v spoločnosti vrátane ich vzdelania, pracovnej histórie, demografie a faktorov súvisiacich so zamestnaním. Bola anonymizovaná, aby chránila súkromie a zároveň poskytovala cenné informácie o pracovnej sile.

**Stĺpce:**

- **Vzdelanie:**

Vzdelanostná kvalifikácia zamestnancov vrátane stupňa, inštitúcie a študijného odboru.

- **Rok nástupu:**

Rok, kedy každý zamestnanec nastúpil do spoločnosti, s uvedením ich dĺžky služby.

- **Mesto:**

Miesto alebo mesto, kde každý zamestnanec sídli alebo pracuje.

- **Platová úroveň:**

Kategorizácia zamestnancov do rôznych platových úrovní.

- **Vek:**

Vek každého zamestnanca, ktorý poskytuje demografické informácie.

- **Rod:**

Rodová identita zamestnancov, podpora analýzy diverzity.

- **Ever Benched:**

Označuje, či bol zamestnanec niekedy dočasne bez pridelenej práce.

- **Skúsenosti v súčasnej doméne:**

Počet rokov skúseností zamestnancov vo svojom súčasnom odbore.

- **Nechať alebo nie:**

cieľový stĺpec

**Použitie:**(od Kaggle)

Tento súbor údajov možno použiť na rôzne analýzy týkajúce sa ľudských zdrojov a pracovnej sily vrátane udržania zamestnancov, hodnotenia mzdovej štruktúry, štúdií diverzity a inklúzie a analýz vzorov odchodov. Výskumníci, analytici údajov a odborníci na ľudské zdroje môžu z tohto súboru údajov získať cenné poznatky.

### 3.2 apple\_quality.csv

Ďalším súborom údajov, na ktorom bol algoritmus testovaný, bola klasifikácia dobrých a zlých jabĺk. Súbor obsahuje 4000 záznamov o jablkách. Nasledujúce informácie sú prevzaté z webovej stránky Kaggle, rovnako ako samotný súbor údajov. Nájdete ho tu na odkaze.

(<https://www.kaggle.com/datasets/nelgiryewithana/apple-quality>)(text odkazu, ak nie je zobrazený v dokumente)

**Kaggle:**

Tento súbor údajov obsahuje informácie o rôznych atribútoch množiny ovocia a poskytuje prehľad o ich vlastnostiach. Súbor údajov obsahuje podrobnosti, ako je



ID ovocia, veľkosť, hmotnosť, sladkosť, chrumkavosť, šťavnatosť, zrelosť, kyslosť a kvalita.

**Stĺpce:**

- **Veľkosť:**  
Veľkosť plodu
- **Hmotnosť:**  
Hmotnosť ovocia
- **Sladkosť:**  
Stupeň sladkosti ovocia
- **Chrumkavosť:**  
Textúra označujúca chrumkavosť ovocia
- **Šťavnatosť:**  
Úroveň šťavnatosti ovocia
- **Zrelosť:**  
Štádium zrelosti ovocia
- **Acidita:**  
Úroveň kyslosti ovocia
- **Kvalita:**  
Celková kvalita ovocia (cieľový stĺpec)

### 3.3 startupdata.csv

Ďalším údajom, na ktorom bol algoritmus testovaný, boli údaje o úspechu alebo neúspechu startupu, ktoré sú pre investorov veľmi dôležité). Údaje majú iba 924 záznamov, čo je v porovnaní s údajmi, ktoré prišli predtým, dosť malé číslo. Pre nás to hralo rolu v množstve dát, ktoré sa mohli zúčastniť experimentu, čo znamenalo

malý tréningový a testovací set. Taktiež sme v týchto dátach nepoužili všetky atribúty, ale tie, ktoré sme považovali za potrebné a pri ktorých sme si boli istí, že ich algoritmus spracuje bez zbytočných následkov pre nás (napríklad zmena kódu)). Údaje tiež prevzaté z Kaggle. Nájdete ho tu na odkaze.

(<https://www.kaggle.com/datasets/manishkc06/startup-success-prediction/data>)(text odkazu, ak nie je zobrazený v dokumente)

**Kaggle:**

Cieľom je predpovedať, či sa startup, ktorý práve funguje, zmení na úspech alebo neúspech. Úspech spoločnosti je definovaný ako udalosť, ktorá dáva zakladateľom spoločnosti veľkú sumu peňazí prostredníctvom procesu M&A (fúzie a akvizície) alebo IPO (Initial Public Offering). Spoločnosť by sa považovala za zlyhanú, ak by musela byť zatvorená.

**Stĺpce:**

- city
- relationships
- milestones
- category
- has VC
- has angel
- has roundA
- has roundB
- has roundC
- has roundD
- average participants

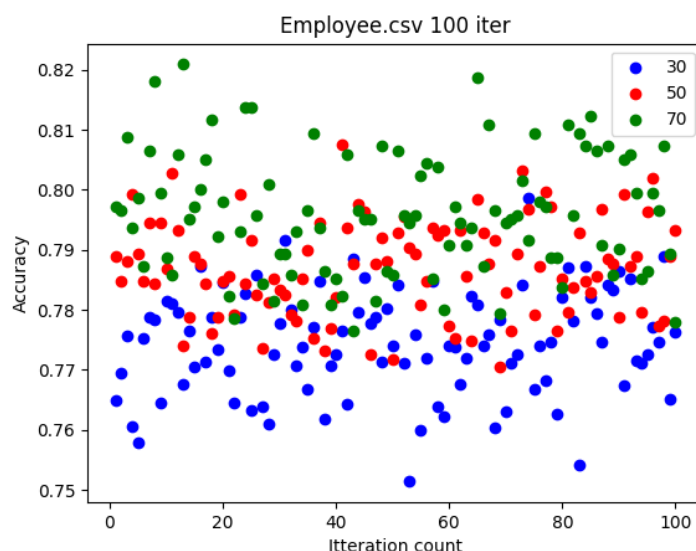
- **status**

## 4 Vyhodnotenie

Prejdime k hodnoteniu našej práce. Najprv sa pozrieme na to, ako dobre funguje jednoduchý rozhodovací strom s našimi tromi rôznymi množinami údajov, a potom sa pozrieme na metriky nášho algoritmu na odstraňovanie chýb. Ďalej prejdime k rozsiahlejšiemu prehľadu nášho algoritmu a toho, ako jednotlivé parametre ovplyvňujú výsledky modelu.

### 4.1 Testovanie CN4.5

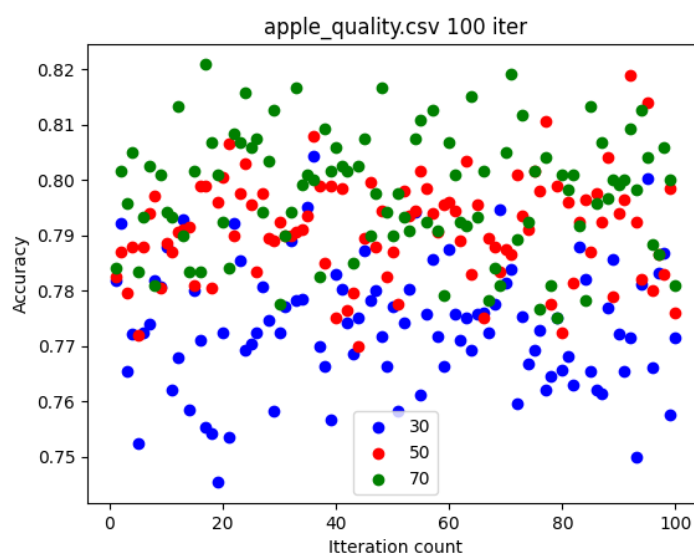
Teraz chceme vidieť, ako na našich údajoch funguje iba rozhodovací strom. Na vizuálnu prezentáciu výsledkov a toho, čo chceme ukázať, použijeme grafy. Chceme otestovať, ako separácia dát ovplyvňuje výsledok a ako je náš model trénovaný na určitej množine dát. Experiment sme uskutočnili spolu s 3 súbormi údajov. Ako hodnoty na rozdelenie tréningových podmnožín použijeme hodnoty 30/50/70. Pre množinu tréningových údajov použijeme 100 iterácií pre každú hodnotu. Aj týmto chceme ukázať nielen to, ako náš model funguje, ale aj kvalitu dát.



**Obrázok 4 – 1** Vplyv veľkosti testovacej vzorky na presnosť CN4.5 nad 100 iterácií Employee.csv

Aby sme otestovali naše riešenie s jablkami, musíme zmeniť jeden riadok kódu na zaokrúhlenie reálnych čísel. Bolo to urobené, pretože používajú dlhé čísla, ktoré sú takmer v každom stĺpci iné, a my sme potrebovali ušetriť čas na spracovanie dreva. Na to nám stačí zmeniť prvý riadok našej metódy `_calculate_gain_numeric` na výpočet informačného zisku pre číselné atribúty.

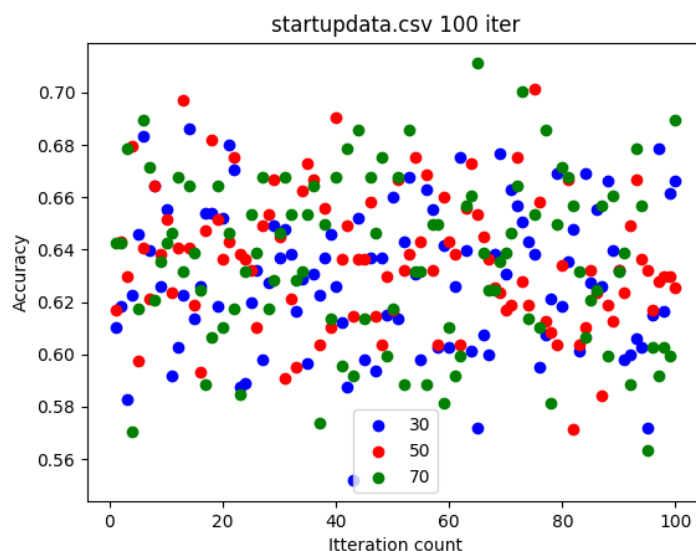
```
1 numb_set = sorted([i for i in set([round(item) for item in x[attr  
    ]))])
```



**Obrázok 4–2** Vplyv veľkosti testovacej vzorky na presnosť CN4.5 nad 100 iterácií `apple_quality.csv`

Ako môžeme vidieť z údajov, množiny údajov pre pracovníkov a pre jablká fungovali celkom dobre, plus dobrým indikátorom je, že s rastom trénovacej množiny sa zvyšuje aj priemerná miera presnosti, rovnako ako vidíme z grafov týchto dvoch množín údajov, rozptyl v presnosti nie je taký veľký. Dospeli sme k záveru, že pre tieto dva sme dosiahli pomerne dobré výsledky. Jediná vec, s ktorou nie sme veľmi spokojní, je skutočnosť, že pri použití pomerne špecifického riešenia (o to sa autor snažil) dostaneme pomerne veľký rozptyl s rovnakou nastavenou veľkosťou, to znamená, že sme v tomto prípade veľmi závislí o kvalite údajov. Ale doplnkový

tréning modelky môže byť podľa mňa liečivá tabletká. Čo sa týka tretieho súboru



**Obrázok 4 – 3** Vplyv veľkosti testovacej vzorky na presnosť CN4.5 nad 100 iterácií startupdata.csv

údajov, tu je všetko smutné. Po prvé, získame pomerne nízku presnosť, ktorá sa prakticky nezvyšuje v závislosti od zmeny percenta tréningovej sady. Po druhé, ako môžeme vidieť počas iterácií, pri rovnakých tréningových sériách dostaneme aj dosť veľký rozptyl bodov. Z toho môžeme usúdiť, že algoritmus na týchto dátach nefunguje veľmi dobre, jediné čo autora teší je, že minimálna presnosť je lepšia ako náhodný výber a nad tým všetkým sa týčila jedna zelená bodka. S čím autor nie je spokojný, je výsledok ako celok, preto tento set ďalej nevyužijeme. Problémom tohto výsledku môžu byť aktuálne atribúty, keďže sme nezobrali všetky stĺpce. V závere z toho možno v skutočnosti vyjadriť, ak algoritmus funguje správne, že medzi atribútmi, ktoré sme použili, neexistuje žiadna korelácia, takže pre zvýšenie presnosti stojí za zváženie použitie týchto údajov s inými súbormi atribútov.

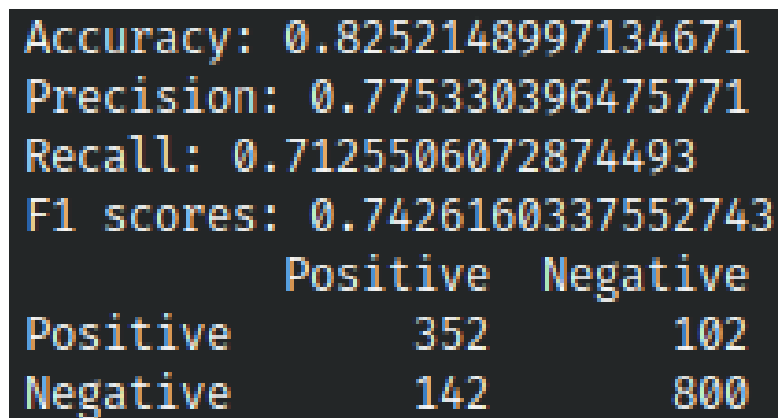
## 4.2 Testovanie Bagging

Na prezentovanie hlavných metrík a testovanie algoritmu priamo pomocou Bagging použijeme všetky hlavné metriky, ktoré sme definovali v zodpovedajúcej triede. Na

testovanie vyberieme údaje o odchode pracovníkov, ktoré otestujeme s nasledujúcimi parametrami, tréningové dáta budú rozdelené na 12 chunkov, na jeden strom použijeme 7 chunkov určených bootstrapom a použijeme 5 stromov . Dáta budú rozdelené 70/30. Vytvorili sme riešenie, ktoré funguje v režime konzoly, preto sme použili nasledujúce príkazy.

```
1 python test.py -f Employee.csv -y LeaveOrNot -l 1 -p 12 -b 7
2   --treeCount 5
3 python test.py -f Employee.csv -y LeaveOrNot -l 1 -p 1 -b 1
4   --treeCount 1
5 python test.py -f Employee.csv -y LeaveOrNot -l 1 -p 12 -b 7
6   --treeCount 1
```

Získame nasledujúce výsledky.



```
Accuracy: 0.8252148997134671
Precision: 0.775330396475771
Recall: 0.7125506072874493
F1 scores: 0.7426160337552743
          Positive  Negative
Positive   352      102
Negative   142      800
```

Obrázok 4–4 Metriky pomocou Bagging spolu s piatimi stromy Employee.csv

Pre porovnanie ukážeme výsledky jedného stromu bez rozdelenia údajov na kúsky, teda jeho bežnú prevádzku.

```
Accuracy: 0.7779369627507163
Precision: 0.6495901639344263
Recall: 0.6951754385964912
F1 scores: 0.6716101694915255
           Positive  Negative
Positive    317      171
Negative    139      769
```

Obrázok 4–5 Metriky pomocou jedného stromu Employee.csv

Teraz sa skúsme pozrieť, ako si jeden strom poradí s rovnakými parametrami bagging a bootstarp co mali v prvom rade.

```
Accuracy: 0.7521489971346705
Precision: 0.625250501002004
Recall: 0.6624203821656051
F1 scores: 0.6432989690721649
           Positive  Negative
Positive    312      187
Negative    159      738
```

Obrázok 4–6 Metriky pomocou Bagging spolu s jedným stromom Employee.csv

Ako vidíme, výsledky sa od seba líšia, v prvom prípade sme dostali najvyššiu presnosť z 3, jeden strom sa umiestnil na druhom mieste a vrecovanie s jedným stromom ukázalo najnižšiu presnosť, ale to nie je prekvapujúce, pretože sme už

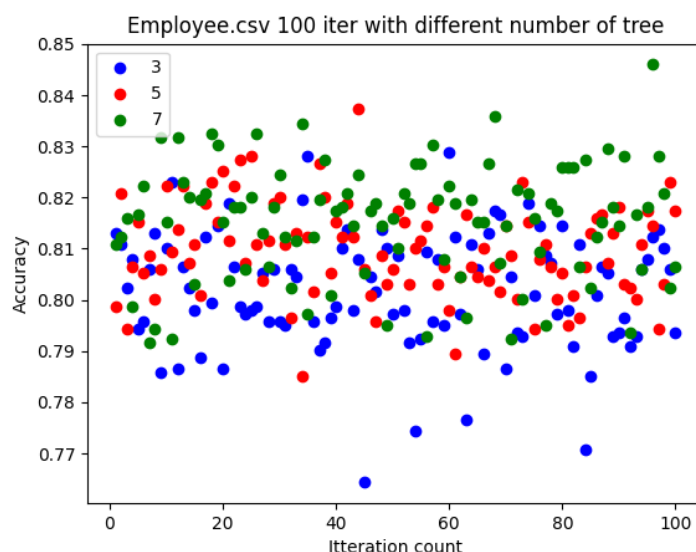


znížili jemu priestor.

### 4.3 Zmena parametrov bagging

V tejto časti sa pozrieme na to, ako zmena niektorých parametrov ovplyvňuje algoritmus. Bude sa testovať na súbore údajov s pracovníkmi. Budeme experimentovať s rôznym počtom stromov a počtom bootstrapov. Použijeme rovnaké proporcie tréningovej množiny ako 70/30. Nebudeme experimentovať s počtom kúskov, keďže dôležitý je aj počet kúskov, ktoré v každom strome použijeme, pre ktoré autor nevedel nájsť vhodné vysvetlenie svojho konania.

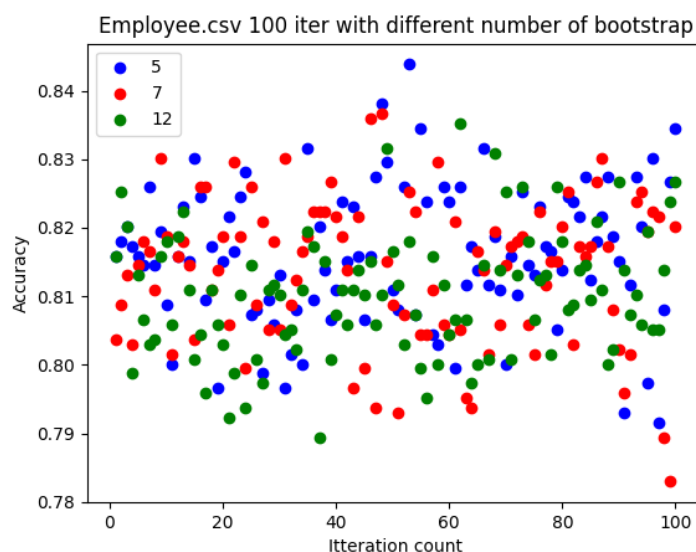
Náš experiment začneme zmenou počtu stromov, použijeme nasledujúce parametre, rozdelíme naše údaje na 12 častí a potom každému stromu pridáme 7 kusov.



**Obrázok 4–7** Použitie rôzneho počtu stromov v algoritme Employee.csv

Na základe výsledkov môžeme zdôrazniť nasledujúce body. Po prvé, s nárastom stromov, aj keď nie veľmi, sa rozptyl vo výsledkoch znižuje, čím viac stromov, tým sa body približujú k určitej priemernej hodnote; Po druhé, vidíme, že aj keď nie s veľkým krokom, priemerný výsledok sa zvyšuje. Z toho môžeme usúdiť, že zvýšenie

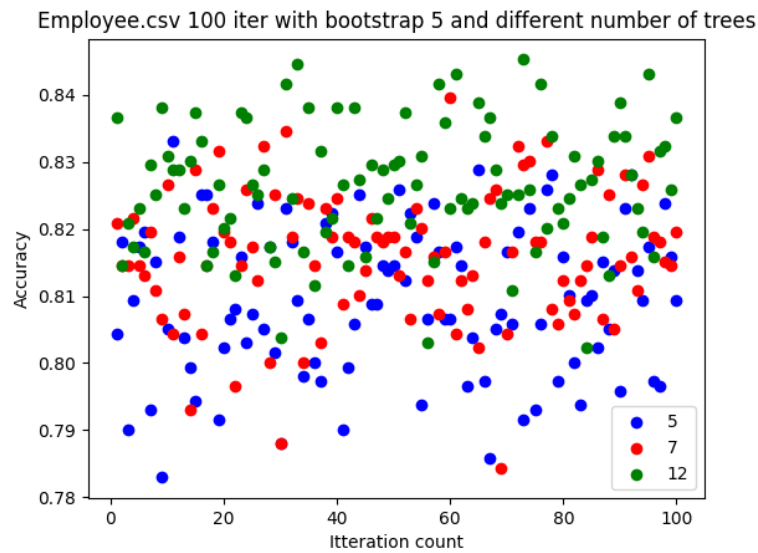
počtu stromov, aj keď nie príliš, pomáha. Preto ponecháme hodnotu 7 stromov a potom otestujeme, ako to ovplyvní zmena veľkosti bootstrapu. Dáta sme rozdelili na 12 častí a testovali sme, ako sa zmení výsledok, ak do stromu pridáme 5/7/12 častí. Z výsledkov môžeme vidieť nasledovné, výsledok, ktorý v podstate využíval plnú



**Obrázok 4–8** Použitie rôzneho množstva bootstrapu Employee.csv

veľkosť údajov (existuje možnosť, že sa vyskytnú duplikáty a nevyužijeme všetky údaje), dopadol najhoršie zo všetkých. Keďže chceme vytvárať odborníkov, ktorí spoločným hlasovaním rozhodnú, aká trieda bude, nemôžeme použiť odborníkov, ktorí v podstate vedia to isté ako ostatní. Preto môžeme konštatovať, že potrebujeme odborníkov, ktorí poznajú svoju dátovú oblasť čo najkonkrétnejšie. 5 a 7 dosahovali v porovnaní so sebou približne rovnakú úroveň. Ale keďže vidíme, že bližšie k vrcholu máme stále modré bodky, vezmeme ich. Ale aj 7 má mierny rozdiel oproti 5, to je rozptyl bodov, keďže podľa mňa experti síce poznajú v podstate iné oblasti, ale s najväčšou pravdepodobnosťou majú takpovediac spoločnú bázu znalostí. Ale v nasledujúcich experimentoch vezmeme hodnotu 5.

Keďže sme už našli hodnotu 5, otestujme ju na inom počte stromov, len tentoraz bude číslo väčšie ako minule.

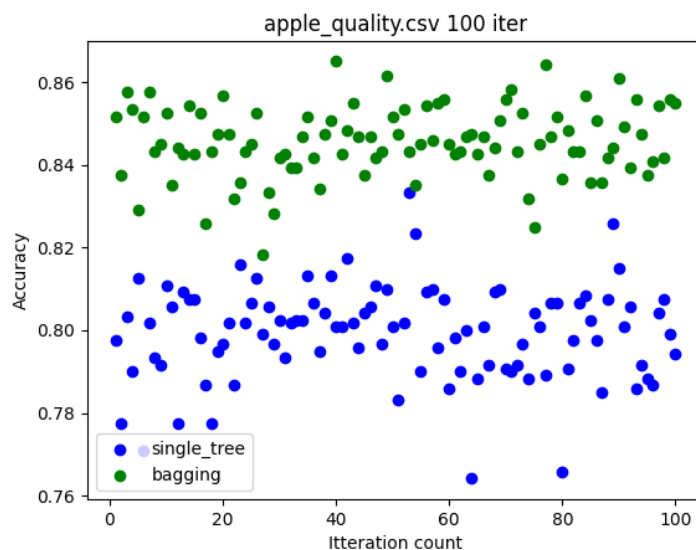


**Obrázok 4–9** Použitie bootstrapu s hodnotou 5 pre rôzny počet stromov Employee.csv

Z výsledku môžeme povedať nasledovné: koniec koncov, ak sa dodržia a upraví určité parametre, je možné zvýšiť presnosť modelu. Ako môžeme vidieť na grafe oproti prvému grafu v tejto kapitole, nárast počtu stromov má tentokrát dostatočnú tendenciu navýšiť (aj keď mierne) výsledok.

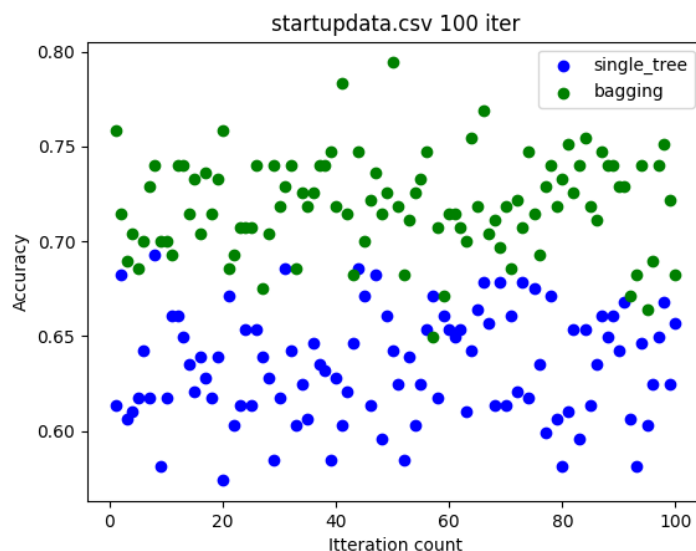
## 4.4 Zaver

Najprv sa pozrime, ako sa zmenili naše ukazovatele v súbore údajov o jablkách pomocou vrecovania alebo jediného rozhodovacieho stromu.



**Obrázok 4 – 10** Rozdiel medzi používaním Bagging a jedným rozhodovacím stromom na množine údajov apple\_quality.csv

Z grafu je jasne vidieť, že metóda, aj keď mierne, zvyšuje predikciu modelu, čo je pre nás veľmi dobré. Pozrime sa potom na údaje o startupoch, aby sme mohli posúdiť, či sa presnosť s týmito údajmi zvýši. Vzhľadom na skutočnosť, že náš osamelý rozhodovací strom nevykazoval obzvlášť dobrý výsledok.



**Obrázok 4 – 11** Rozdiel medzi používaním Bagging a jedným rozhodovacím stromom na množine údajov startupdata.csv

Skvelé! Ako vidíme z výsledku, v prípade týchto údajov model ukázal zvýšenie presnosti predpovedí, čo je pre nás tiež dobrá správa.

A keďže sme mohli vidieť v experimentoch, algoritmus nám skutočne pomáha zvýšiť presnosť nášho modelu a ako sme ukázali, pracuje s rôznymi údajmi. Pozreli sme sa na históriu tvorby algoritmu, pozreli sme sa na to, ako funguje z teoretického hľadiska a pozreli sme sa na štruktúru rozhodovacích stromov. Ďalej sme boli schopní napísať úplne nízkoúrovňovú implementáciu algoritmu s minimálnym využitím knižníc tretích strán. Implementácia nie je najlepšia, ale snažili sme sa. Potom sme sa pozreli na údaje, ktoré sa zúčastnili experimentov. Nakoniec sme sa pozreli na výsledky a na to, ako jednotlivé parametre ovplyvňujú fungovanie algoritmu. Teraz môžeme s istotou povedať, že súborové učenie je cool.

## Zoznam použitej literatúry

Bagging Wikipedia(Rus.) odkaz

Bagging (Rus.) odkaz

Bagging Wiki(Rus.) odkaz

Bagging(bootstrap aggregating)(Rus.) odkaz

C4.5 Wikipedia(Rus.) odkaz

Ensemble learning(Kristina Machova) odkaz

C4.5(Kristina Machova) odkaz