VŠB TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Symbolické metody strojového učení a rozhodovací stromy

Symbolic Methods of Machine Learning and Decision Trees

Vojtěch Habrnal

Bakalářská práce

Vedoucí práce: Ing. Adam Albert

Ostrava, 2023

VŠB TECHNICKÁ UNIVERZITA OSTRAVA | FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Č.j.VSB/22/078997
vsboee220260D2

Zadání bakalářské práce

Student: **Vojtěch Habrnal**

Studijní program: B0613A140014 Informatika

Téma: Symbolické metody strojového učení a rozhodovací stromy
Symbolic Methods of Machine Learning and Decision Trees

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem práce je shrnou teorii strojového učení v multiagentních systémech za pomocí symbolické reprezentace poznatků. Student v případové studii naprogramuje agenta, který bude na základě znalostí specifikovaných v jazyce predikátové logiky 1. řádu budovat reprezentaci pro klasifikaci příkladů v podobě rozhodovacího stromu.

Cíle práce:
1. Shrnutí teorie strojového učení na základě symbolické reprezentace znalostí.
2. Analýza, návrh a implementace agenta schopného budovat hypotézu z pozitivních a negativních příkladů metodou rozhodovacích stromů.

Seznam doporučené odborné literatury:

[1] DUŽÍ, Marie. Logika pro informatiky (a příbuzné obory): učební text. Ostrava: VŠB-TU Ostrava, 2012. ISBN 978-80-248-2662-2.
[2] LUGER, George F. Artificial intelligence: structures and strategies for complex problem solving. 6th ed. Boston: Pearson Addison-Wesley, c2009. ISBN 978-0-321-54589-3.
[3] MITCHELL, Tom M.Machine Learning. New York: McGraw-Hill, c1997. ISBN 00-704-2807-7.
[4] KUBÍK, Aleš. Inteligentní agenty. Brno: Computer Press, 2004. ISBN 80-251-0323-4.
[5] POOLE, David L., MACKWORTH Alan K. Artificial intelligence: foundations of computational agents. 2nd pub. Cambridge: Cambridge University Press, 2010. ISBN 978-0-521-51900-7.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Adam Albert**

Datum zadání: 01.09.2021

Datum odevzdání: 30.04.2023

Garant studijního programu: doc. Mgr. Miloš Kudělka, Ph.D.

V IS EDISON zadáno: 07.11.2022 12:18:12

Abstrakt

Cílem práce je shrnout teorii strojového učení v multiagentních systémech za pomocí symbolické reprezentace poznatků. Student v případové studii naprogramuje agenta, který bude na základě znalostí specifikovaných v jazyce predikátové logiky 1. řádu budovat reprezentaci pro klasifikaci příkladů v podobě rozhodovacího stromu.

Klíčová slova

typografie; LaTeX; bakalářská práce; multiagentní systém; strojové učení; rozhodovací stromy

Abstract

The aim of the thesis is to summarize the theory of machine learning in multi-agent systems using the symbolic representation of knowledge. In the case study, the student will program an agent that will build a representation for classifying examples in the form of a decision tree based on knowledge specified in the language of predicate logic of the 1st order.

Keywords

typography; LaTeX; bachelors thesis, multi-agent system; machine learning; decision trees

Poděkování

Rád bych poděkoval vedoucímu práce Adamu Albertovi za dostatečnou trpělivost

Obsah

Seznam použitých symbolů a zkratek **6**

Seznam obrázků **7**

Seznam tabulek **8**

1 Úvod **9**

2 Pojem Agent **10**
2.1 Kategorizace Architektury Agentů 10
2.2 Multiagentní Systémy 11
2.3 Užitá Komunikační Norma 11

3 Strojové Učení **12**
3.1 Symbolická Reprezentace Znalostí 13
3.2 Indukce Rozhodovacích Stromů 14

4 Případová Studie **21**
4.1 Analýza 21
4.2 Návrh Implementace 22
4.3 Užité Programátorské Techniky 27
4.4 Implementace 31

5 Závěr **35**

Přílohy **35**

A Zdrojové Kódy **36**
A.1 Soubor functions.py 36
A.2 Soubor main.py 40

Seznam použitých zkratek a symbolů

IT	–	Informační technologie
ML	–	Machine Learning
AI	–	Artificial Intelligence
DT	–	Decision Tree
MAS	–	Multiagentní systém
RT	–	Reálný čas
UML	–	Unified Modeling Language

Seznam obrázků

3.1 Grafické vysvětlení pojmů 13
3.2 Rozhodovací strom pro dataset DS rozdělen podle X a Y 18

4.1 Příklad výstupu programu 22
4.2 UML: Třídní diagram 25
4.3 UML: Sekvenční diagram 26
4.4 Ilustrační graf G 30
4.5 Kostra grafu G 30

Seznam tabulek

2.1 Parametry zprávy normy FIPA-ACL 11

3.1 Datový set *DS* pro vzorové výpočty 16
3.2 DS rozdělen dle sloupce X 17
3.3 DS rozdělen dle dloupce Y 19

Kapitola 1

Úvod

V rámci této práce se budu věnovat dvěma věcem. Prvním je shrnutí teorie ML a MAS, kde rozeberu a vysvětlím důležité pojmy z AI a DT tak, aby práce byla dostupná i pro širší veřejnost. Druhou částí bude implementace MAS s cílem vytvoření DT užitím ID3 algoritmu. Tento DT bude následně otestován na *testovacích datasetech*. Výstupem bude, krom samotného programu, shrnutí procentuální chybovosti na přiložených datasetech.

V úvodu práce se nacházejí seznamy, jmenovitě seznam symbolů a zkratek, seznam obrázků a seznam tabulek. Následuje pět kapitol, kde první je tento Úvod (1), následuje kapitola Pojem Agent 2, která zavádí pojem (*agent*), shrnuje některé pojmy z MAS a obsahuje přehled parametrů komunikační normy FIPA. Kapitola 3 pojednává o teorii ML, zmiňuje základní dělení, uvádí příklady a jejich následné zařazení. Kapitola končí vysvětlením DT (z čeho se strom skládá, co obsahuje apod.) a zavedením pojmů *entropie* a *informační zisk*. Před závěrem je kapitola 4, ve které provádím návrh implementace, analýzu a konkrétní popis postupu implementace programu. Také zde vysvětluji některé programátorské techniky, konkrétně *rekurzi* a *hladové algoritmy*. Práci ukončuje kapitola 5, ve které shrnuji výsledky práce a procentuální chybovost ID3 algoritmu na užitých datasetech.

Kapitola 2

Pojem Agent

Kubík v [1] shrnuje nároky na *agenta* do této definice:

„Agent je entita zkonstruovaná za účelem kontinuálně a do jisté míry autonomně plnit své cíle v adekvátním prostředí na základě vnímání prostřednictvím senzorů a prováděním akcí prostřednictvím aktuátorů. Agent přitom ovlivňuje podmínky v prostředí tak, aby se přibližoval k plnění cílů.“ ([1], s. 12)

Z definice agenta plyne, že by měl být zástupcem pro vykonání určitého úkolu, či cíle. Zároveň by měl být schopen těchto cílů dosáhnout s minimálním zásahem ze strany správce systému. Agenti obecně mohou interagovat s prostředím, ve kterém se nacházejí tak, aby se neustále přibližovali k dokončení svých cílů.

2.1 Kategorizace Architektury Agentů

Dle svého účelu, vnitřní architektury a kapacity zdrojů, lze agenty rozdělit do následujících kategorií:

- **Reaktivní**: Agenti v této kategorii se vyznačují architekturou, která obsahuje několik procesů různé náročnosti. Tyto procesy jsou spouštěny v reakci na vnější podněty z prostředí, či od jiného *agenta*. Vyjma reakčních procesů mají k dispozici záznam o svém stavu a vhodně zvolené množství vyrovnávací paměti pro výpočetní potřeby.
- **Deliberativní**: Tato architektura má, oproti architektuře *reaktivní*, k dispozici paměťovou reprezentaci okolního prostředí. Tyto informace jsou nadále používány k tvorbě, a následné realizaci, plánů pro vykonání přednastavených cílů. Agent má možnost sestavení vícera plánů a následného upřednostnění, dle plánové priority.
- **Sociální**: *Sociálním* je agent ve chvíli, kdy v rámci svého plánování a chování komunikuje s jinými agenty pomocí vyšších komunikačních jazyků.
- **Hybridní**: Agenti této architektury kombinují několik prvků z výše zmíněných architektur.

V rámci *případové studie* budou použiti dva agenti architektury *reaktivní agent*.

2.2 Multiagentní Systémy

Označení pro systém nebo prostředí, ve kterém se nachází alespoň dva *agenti*, kteří aktivně plní určené cíle. Důvodem vzniku MAS je rostoucí náročnost a složitost požadavků z různých oborů. Tyto požadavky je možné rozdělit, zjednodušit a spravovat pomocí *agentů*.

Užití MAS je možné například v těchto oblastech: automatizace průmyslových procesů (přemisťování zboží v logistických centrech), transport (autonomní řízení dopravy v RT), finance (modelování průběhu, či predikce cen), hry (umělá inteligence protivníků) a robotika (koordinace robotů v rámci stavby).

2.3 Užitá Komunikační Norma

FIPA-ACL (Foundation for Intelligent Physical Agents - Agent Communications Language, Nadace pro Inteligentní Fyzické Agenty - Jazyk Agentní Komunikace) je standardem pro komunikaci *agentů*, počínaje komunikací jednoho *agenta* s prostředím a konče multiagentním systémem o N sociálních *agentech*. Zdrojem informací je oficiálního webu nadace [2].

2.3.1 FIPA-ACL Struktura Zprávy

Zpráva může využít libovolný počet z parametrů uvedených v Tabulce 2.1, avšak FIPA doporučuje použít minimálně tyto parametry: *performative*, *sender*, *receiver* a *content*. V rámci práce jsem využil výše zmíněné parametry.

Parametr	Kategorie	Užití
performative	Typ komunikačního aktu	Slouží k definování typu komunikace (povinný parametr)
sender	Účastník komunikace	Identifikace odesílajícího agenta
receiver	Účastník komunikace	Identifikace příjemce
reply-to	Účastník komunikace	Adresát odpovědi na zprávu
content	Obsah zprávy	Uchovává data pro příjemce
language	Popis obsahu	Specifikuje jazyk zprávy
encoding	Popis obsahu	Specifikuje kódování zprávy
ontology	Popis obsahu	Specifikuje symboliku zprávy
protocol	Kontrola konverzace	Identifikátor protokolu konverzace
conversation-id	Kontrola konverzace	Identifikační číslo konverzace
reply-with	Kontrola konverzace	Výraz pro odpověď na zprávu
in-reply-to	Kontrola konverzace	Identifikuje zprávu jako odpověď na *reply-with*
reply-by	Kontrola konverzace	Lhůta odpovědi

Tabulka 2.1: Parametry zprávy normy FIPA-ACL

Kapitola 3

Strojové Učení

V rámci následujících stránek budu užívat tyto pojmy (grafické vysvětlení je v Obrázku 3.1):

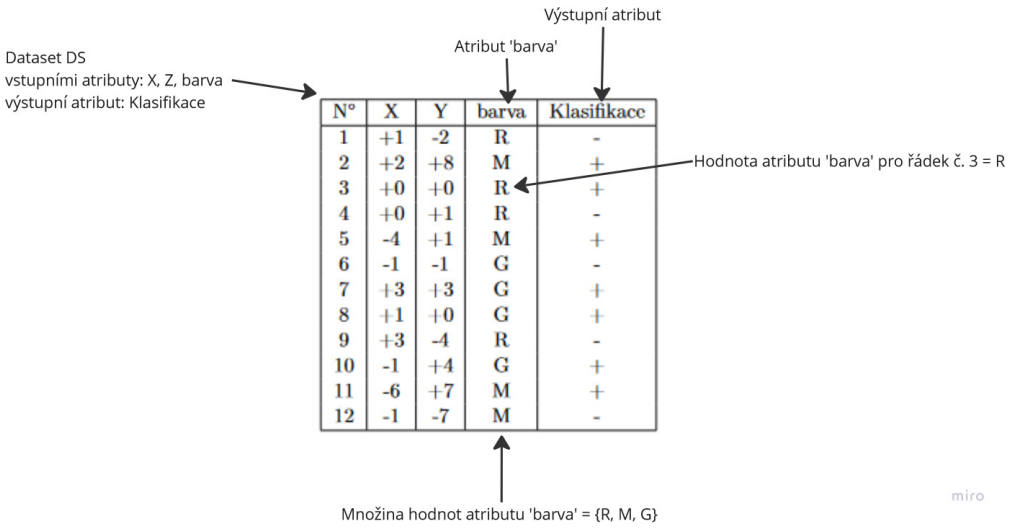
Strojové učení je podoblast *umělé inteligence* (AI), která se zabývá technikami (algoritmy) pro zefektivnění průběhu budoucích úkolů v daných systémech (např. multiagentní systémy, predikce kurzu akcií, těžení dat apod.). Jinými slovy umožňují strojům proces učení.

Existuje celá řada paradigmat (přístupů), mezi nejčastěji používané metody patří: *statistické metody učení*, *metody genetického učení*, *umělé neuronové sítě* a *symbolická reprezentace znalostí*, na které je tato práce zaměřená. Tyto paradigmata se následně dělí do těchto tří kategorií:

- **Učení s učitelem** ('supervised learning'): *Studentovi* (*student* zde označuje neživou entitu, např. *agenta*) jsou příklady předávány ve dvojicích, vždy vstupní atributy a hodnota výstupního atributu (v případě Obrázku 3.1 např.: $\{[X = +1, Y = -2, Barva = R], -\}$). Hodnoty *výstupního atributu* jsou generovány modelu nespecifikovanou matematickou funkcí $y = f(x)$. Úkolem programu je nalézt funkci $h(x)$ (hypotézu), která je schopna aproximovat originální funkční hodnoty $f(x)$. Pokud je množina *výstupního atributu* diskrétní množinou, taková množina, která může nabývat pouze jednotlivých hodnot (např. druh počasí = {slunečno, přeháňky, bouřka}, či celá čísla N), pak hovoříme o *klasifikaci*. V případě, kde je množina *výstupního atributu* spojitá, tedy může nabývat libovolných hodnot ohraničených intervalem (např. zítřejší teplota, nebo reálná čísla R), se jedná o *regresi*. [3]

- **Učení bez učitele** ('unsupervised learning'): Na rozdíl od učení s učitelem zde nejsou příklady podávány ve dvojicích (*vstupní atributy* a korespondující *výstupní atribut*). Tato forma vyžaduje od *studenta* samoorganizaci na základě různých kritérií. Ideálním příkladem je *shlukování*, kde vstupem jsou neorganizovaná data (nejčastěji body v grafu) a způsob měření jejich podobnosti. Cílem je zorganizovat data do skupin (shluků), které splňují určitá kritéria (např. body jsou součástí jedné skupiny, pokud je jejich vzdálenost menší než 6). [4]

- **Učení zpětnou vazbou** ('reinforced learning'): Tato kategorie by se dala zařadit jako podkategorie *učení bez učitele* s důležitou změnou přístupu k učení. V předchozí kategorii *student*



N°	X	Y	barva	Klasifikace
1	+1	-2	R	-
2	+2	+8	M	+
3	+0	+0	R	+
4	+0	+1	R	-
5	-4	+1	M	+
6	-1	-1	G	-
7	+3	+3	G	+
8	+1	+0	G	+
9	+3	-4	R	-
10	-1	+4	G	+
11	-6	+7	M	+
12	-1	-7	M	-

Obrázek 3.1: Grafické vysvětlení pojmů

nemá žádnou zpětnou vazbu od *učitele* a problémy řeší na základě podobnosti dat v datasetu. V této kategorii se postup *studenta* odměňuje na základě jeho výsledků. Pokud jsou výsledky správné (např. výhra v šachu), pak student dostane odměnu (třeba formou přípisu několika bodů). V opačném případě *student* dostane trest (odebrání několika bodů). Nadále je na studentovi, aby zjistil, který ze *vstupních atributů* nebo aplikovaných postupů měl největší dopad na výsledek. [3, 4]

3.1 Symbolická Reprezentace Znalostí

V této kapitole přiblížím jednu z výše zmíněných metod (*paradigmat*), konkrétně *symbolickou reprezentaci*. Jedná se o jeden z původních přístupu k ML, který spoléhá na existenci gramatiky formovanou *symboly* (příkladem je *predikátová logika 1. řádu*), jejichž řetězením lze vytvořit, respektive popsat, objekty reálného světa. Luger ve své knize [4] popisuje objekty tímto abstraktem: $atribut(objekt, hodnota) \wedge barva(objekt, hodnota) \wedge tvar(objekt, hodnota)$. Abstrakt se dá použít pro příklad, kde chceme popsat míč s názvem *obj*, který je malý, má červenou barvu a je kulatý: $velikost(obj, malý) \wedge barva(obj, červená) \wedge tvar(obj, kulatý)$.

Výše zmíněný abstrakt je použit pro reprezentaci dat, která bývají dvojího charakteru: *pozitivní* a *negativní* příklady. Příklady jsou předkládány *studentovi* tak, aby byl schopen odvodit či usoudit (třeba pomocí DT) obecný přístup k datům podobného rázu. Jinými slovy jsou předkládána data, na jejichž základě má *student* přijít na obecný postup pro data následující. Dalším příkladem může být výše zmíněné *shlukování*, kde vstupní data jsou nekategorizovaný set a úkolem *studenta* je najít kategorizaci či smysl v takovém datasetu.

Symbolická reprezentace se dělí na další skupiny, které následně krátce popíši:

- **Heuristické hledání**: Jde o využití dostupných dat a *heuristik* (pravidel, znalostí, intuice či praktik) k vytvoření konceptuálního modelu, respektive zjednodušeného a obecného *konceptu* pro daný problém. Z výše popsaného příkladu pro míč lze obecně odvodit následující: $velikost(x, malý) \wedge barva(x, červená) \wedge tvar(x, kulatý)$. Postupným přidáváním objektů s různou velikostí a barvou lze odvozený *koncept* vylepšit třeba takto: $velikost(x, y) \wedge barva(x, z) \wedge tvar(x, kulatý)$. [4]

- **Hledání verze prostoru**: Stejně jako *heuristické hledání* i toto *paradigma* spadá do *učení s učitelem*, kde vstupními daty jsou pozitivní i negativní příklady a také používá *zobecňování* (nahrazení konstant, odstranění, či přidání podmínky a nahrazení atributu obecnější hodnotou) a *specializace*. Algoritmy užívající tyto dva přístupy vytvářejí množinu hypotéz (platných pro všechny vstupní data), na kterou následně uplatňují jak zobecnění, tak specializaci. Postupným zpracováním dat vznikne jenom jedna hypotéza, platná a dostatečně obecná, pro všechny data v *trénovacím datasetu*. [4]

- **Indukce rozhodovacích stromů**: Oproti předchozím dvěma paradigmatům se toto liší svojí reprezentací naučených znalostí. Kde *hledání verze prostoru* užívá obecnou hypotézu, která je platná pro všechny data z *testovacího datasetu*, tam *indukce rozhodovacích stromů* užívá DT. Což jsou datové struktury, reprezentovatelné grafem, který lze užít pro rychlou klasifikaci dat nových. [4]

3.2 Indukce Rozhodovacích Stromů

Strojové učení založené na rozhodovacích stromech je odnoží *učení s učitelem* a *učení pomocí příkladů*. Užívá *heuristického hledání* k vytvoření datové struktury, tedy DT. V prvotní fázi si z datasetu vybereme atribut (sloupec), pro který chceme vytvořit algoritmus klasifikace. Vstupem tohoto algoritmu (v rámci této práce budu programovat ID3 algoritmus) budou zbylé atributy datasetu, tedy vstupní vektor o velikosti n. Tyto vektory jsou následně předány klasifikačnímu algoritmu pro trénink, nebo pro klasifikaci v případě již hotového DT.

Každopádně před popisem algoritmu ID3 je potřeba vysvětlit následující termíny: *Rozhodovací strom Entropie*, *Informační zisk* a *Rekurze*, které bude popsaná později.

3.2.1 Rozhodovací Strom

Ve Strojovém učení se pod pojmem *rozhodovací strom* rozumí hierarchický graf, či model stromovité struktury, který pro dané vstupní data (vektor vstupních dat) vrátí *rozhodnutí* či *výsledek* (není to ovšem pravidlem, například Russell pohlíží na DT jako na matematické funkce $F(vstupní\ data)$). Tyto grafy se skládají z uzlů, které se dělí následovně:

- **Kořen**: Počátek grafu, jedná se o první uzel, ze kterého vzniká hierarchické větvení

- **List**: Konečný uzel, v rozhodovacích stromech označuje výsledek rozhodnutí (např. barva bodu v grafu, důvěryhodnost klienta banky, apod.)
- **Vnitřní**: Tyto uzly spojují kořen s listy, v rozhodovacích stromech slouží pro dělení vstupních dat na základě hodnoty *atributu*

Kromě uzlů se v rozhodovacích stromech (a ve stromech obecně) vyskytují ještě tyto pojmy:

- **Větev**: Též *cesta* spojuje dva uzly mezi sebou, v rámci rozhodovacích stromů děli původní data na části, dle podmínky v uzlu
- **Hloubka stromu**: Počet větví, které vedou od kořenu k nejvzdálenějšímu listu. Některé algoritmy využívají tuto hodnotu jakožto omezení pro velikost rozhodovacího stromu [5]
- **Podstrom**: Nový strom vzniklý zvolením nového *kořene* v původním stromě

Kořen a vnitřní uzly aplikují podmínky a dělí vstupní data na pod-sety (počet pod-setů je určen velikostí *množiny hodnot* daného atributu), toto se děje pořád dokola, dokud všechny možné pod-sety neobsahují právě jednu hodnotu pro zvolený *výstupní atribut*, tato hodnota se následně stává *klasifikací* či *rozhodnutím*. Po průchodu rozhodovacím stromem je sloupec *výstupního atributu* vyplněn na základě *klasifikace* v listech stromu (např. klient banky je nebo není důvěryhodný; objekt je nebo není míčem). Pokud vybraný *výstupní atribut* obsahuje pouze hodnoty *ano* a *ne* (*true* a *false*), pak mluvíme o *binární* klasifikaci, jejíž výstupem je *binární* DT.

Ilustrační příklad trénovacího datového setu lze nalézt v Tabulce 3.1 a příklad neoptimalizovaného rozhodovacího stromu v Obrázku 3.2.

3.2.2 Entropie Datového Setu

Jak Luger v [4], tak i Russell v [3] popisují entropii jako měrnou hodnotu daného datového setu, zprávy, či informace obecně, míry nejistoty výsledku. Čím nižší *entropie* je, tím jistější si jsme výsledkem (výsledek se dá lépe předvídat). Oba zmiňují Shanonův bitový výpočet entropie podle vzorce:

$$E(S) = -\sum_{i=1}^{n} p(m_i) log_2(p(m_i)) \tag{3.1}$$

Kde:

$E(P) =$ Entropie datasetu S
$p(m_i) =$ Podíl četnosti prvku m_i ku celkovému počtu prvků v rámci datasetu S

Binární logaritmus zajišťuje, že výsledná *entropie* bude vycházet v bitech. Hodnota 0 reprezentuje čistý dataset, kde se jedna hodnota (např. *pravda*, či *nepravda*) vyskytuje s pravděpodobností 100%. Pomocí rovnice můžeme vypočítat příklad dvou typů mincí.

V prvním příkladu se bude jednat o standardní minci s pravděpodobností 50% pro každou stranu. Ve druhém případě bude mince upravena pro 80% upřednostnění jedné strany. Rovnice pro první příklad tedy bude:

$$\begin{aligned} E(Standardní) &= -(\frac{1}{2}log_2\frac{1}{2} + \frac{1}{2}log_2\frac{1}{2}) && (3.2)\\ &= -(-0,5 - 0,5) && (3.3)\\ &= 1\ bit && (3.4) \end{aligned}$$

Obdobně pro druhý příklad s upravenou mincí:

$$\begin{aligned} E(Upravená) &= -(\frac{4}{5}log_2\frac{4}{5} + \frac{1}{5}log_2\frac{1}{5}) && (3.5)\\ &= -(-0,25 - 0,46) && (3.6)\\ &= 0,72\ bitů && (3.7) \end{aligned}$$

Z výsledků jasně vyplývá, že standardní mince je méně předvídatelná a tudíž informace o výsledku hodu je důležitější než v případě mince upravené, která se dá lépe předvídat.

Příklad výpočtu *entropie* pro datový set *DS* (Tabulka 3.1) s *množinou hodnot výstupního atributu*: pozitivní (+) a negativní (-):

N°	X	Y	Barva	Klasifikace
1	+1	-2	R	-
2	+2	+8	M	+
3	+0	+0	R	+
4	+0	+1	R	-
5	-4	+1	M	+
6	-1	-1	G	-
7	+3	+3	G	+
8	+1	+0	G	+
9	+3	-4	R	-
10	-1	+4	G	+
11	-6	+7	M	+
12	-1	-7	M	-

Tabulka 3.1: Datový set *DS* pro vzorové výpočty

$$\begin{aligned} E(DS_{Klasifikace}) &= -\frac{7}{12}\log_2\frac{7}{12} - \frac{5}{12}\log_2\frac{5}{12} && (3.8)\\ &\approx 0.526 + 0.453 && (3.9) \end{aligned}$$

N°	X	Y	Barva	Klasifikace
5	-4	+1	M	+
6	-1	-1	G	-
10	-1	+4	G	+
11	-6	+7	M	+
12	-1	-7	M	-

(a) Tabulka pro X < 0 ($DS_{X<0}$)

N°	X	Y	Barva	Klasifikace
3	+0	+0	R	+
4	+0	+1	R	-

(b) Tabulka pro X $= 0$ ($DS_{X=0}$)

N°	X	Y	Barva	Klasifikace
1	+1	-2	R	-
2	+2	+8	M	+
7	+3	+3	G	+
8	+1	+0	G	+
9	+3	-4	R	-

(c) Tabulka pro X > 0 ($DS_{X>0}$)

Tabulka 3.2: DS rozdělen dle sloupce X

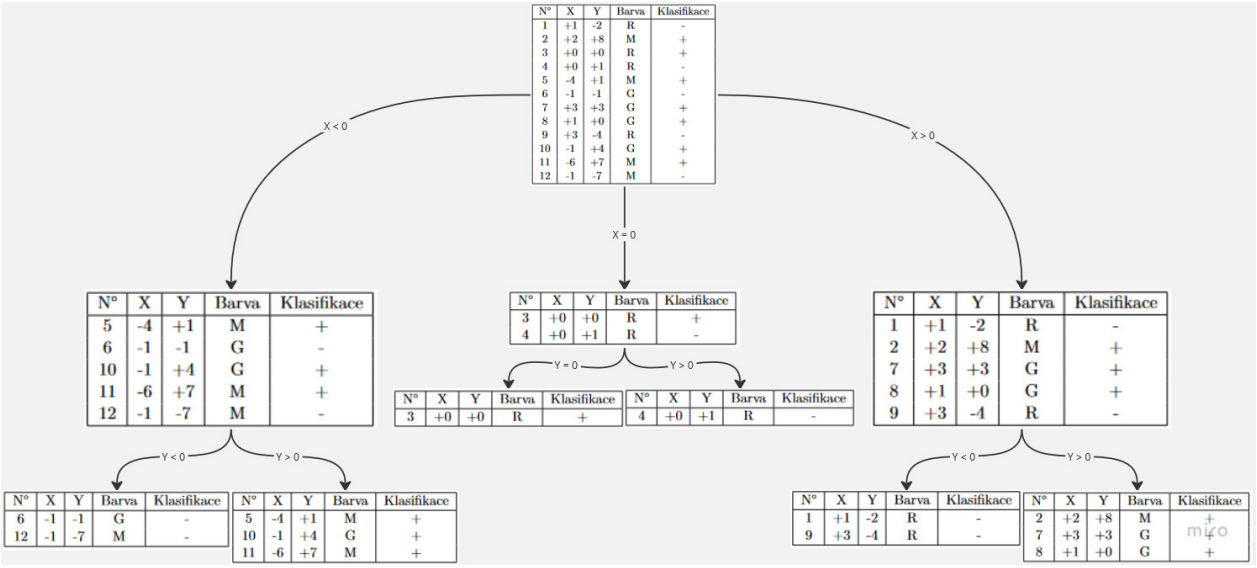
$$\approx \quad 0.979 \tag{3.10}$$

3.2.3 Informační Zisk Atributu Datasetu

Informační zisk je definován jako rozdíl mezi *entropií* datasetu před rozdělením dat a součinu *entropie* měřeného *atributu* s podílem četností hodnot proměnné. Jinými slovy jde o získané snížení entropie po rozdělení dle *atributu* A, tedy množství informací získaných o *výstupním atributu* na základě pozorování *atributu* jiného.

Pokračování předešlého příkladu (s datasetem DS) s rozdělením dat dle sloupce X a výpočet informačního zisku $I(DS_X)$ získaného rozdělením:

$$\begin{aligned}
E(DS_{X<0}) &= -\frac{3}{5}\log_2\frac{3}{5} - \frac{2}{5}\log_2\frac{2}{5} && (3.11)\\
&\approx 0.442 + 0.528 && (3.12)\\
&\approx 0.970 && (3.13)\\
E(DS_{X=0}) &= -\frac{1}{2}\log_2\frac{1}{2} - \frac{1}{2}\log_2\frac{1}{2} && (3.14)\\
&= 0.5 + 0.5 && (3.15)\\
&= 1 && (3.16)\\
E(DS_{X>0}) &= -\frac{3}{5}\log_2\frac{3}{5} - \frac{2}{5}\log_2\frac{2}{5} && (3.17)\\
&\approx 0.442 + 0.528 && (3.18)\\
&\approx 0.970 && (3.19)\\
I(DS_X) &= E(DS) - p(DS_{X<0})E(DS_{X<0}) - p(DS_{X=0})E(DS_{X=0}) - p(DS_{X>0})E(DS_{X>0}) && (3.20)
\end{aligned}$$



Obrázek 3.2: Rozhodovací strom pro dataset *DS* rozdělen podle X a Y

$$= \quad 0.979 - 5/12 * 0.970 - 2/12 * 1 - 5/12 * 0.970 \tag{3.21}$$

$$\approx \quad 0.004 \tag{3.22}$$

Z výsledku $I(DS_X)$ lze poznat, že rozdělení datasetu dle X přineslo minimální *informační zisk*. Avšak pro demonstraci výpočtů a rozdílu oproti DT to bohatě stačí, z tabulek v 3.2 lze vypozorovat, že pro konečnou *klasifikaci* stačí rozdělit graf podle Y a RS bude hotov. Výsledek by vypadal podobně jako Obrázek 3.2.

3.2.4 Shrnutí Teorie na Datasetu DS

DT v Obrázku 3.2 je neefektivní, rozdělení dat na základě *atributu* X přináší minimální *informační zisk*. Pro správné sestavení stromu na základě *entropie* a *informačního zisku* je nutné provést tyto výpočty pro každý atribut v datasetu *DS*. Pro zjednodušení zápisu použiji pouze výpočty *I(P)*.

$$I(DS_X) \quad = \quad E(DS) - 5/12 * 0.970 - 2/12 * 1 - 5/12 * 0.970 \tag{3.23}$$

$$= \quad 0.979 - 5/12 * 0.970 - 2/12 * 1 - 5/12 * 0.970 \tag{3.24}$$

$$\approx \quad 0.004 \tag{3.25}$$

$$I(DS_Y) \quad = \quad E(S) - 4/12 * 0 - 2/12 * 0 - 6/12 * 0,650 \tag{3.26}$$

$$= \quad 0,979 - 0 - 0 - 0,325 \tag{3.27}$$

$$\approx \quad 0,654 \tag{3.28}$$

$$I(DS_{Barva}) \quad = \quad E(S) - 4/12 * 0,811 - 4/12 * 0,811 - 4/12 * 0,811 \tag{3.29}$$

N°	X	Y	Barva	Klasifikace
1	+1	-2	R	-
6	-1	-1	G	-
9	+3	-4	R	-
12	-1	-7	M	-

(a) Tabulka pro Y < 0 ($DS_{Y<0}$)

N°	X	Y	Barva	Klasifikace
3	+0	+0	R	+
8	+1	+0	G	+

(b) Tabulka pro Y $= 0$ ($DS_{Y=0}$)

N°	X	Y	Barva	Klasifikace
2	+2	+8	M	+
4	+0	+1	R	-
5	-4	+1	M	+
7	+3	+3	G	+
10	-1	+4	G	+
11	-6	+7	M	+

(c) Tabulka pro Y > 0 ($DS_{Y>0}$)

Tabulka 3.3: DS rozdělen dle dloupce Y

$$= \quad 0,979 - 0,270 - 0,270 - 0,270 \tag{3.30}$$

$$\approx \quad 0,169 \tag{3.31}$$

Výše vypočítané *informační zisky* říkají, že nejvhodnější *atribut*, na jehož základě by se měl dataset rozdělit, je souřadnice Y. Z hodnot v Tabulce 3.3 je zjevné proč je toto rozdělení nejlepší. Vznikají tím dvě tabulky, které jsou rovnou plně *klasifikovány* a pouze jeden záznam ve třetí tabulce, který nemá stejnou *hodnotu výstupního atributu* jako zbytek tabulky. Nyní by mělo být relativně irelevantní na základě jakého *atributu* data rozdělíme, pro jistotu však přidávám výpočet:

$$E(DS_{Y>0}) = -5/6 * log2(5/6) - 1/6 * log2(1/6) \tag{3.32}$$

$$\approx \quad 0,650 \tag{3.33}$$

$$I(DS_{Y>0_X}) = E(DS_Y) - 3/6 * 0 - 1/6 * 0 - 2/6 * 0 \tag{3.34}$$

$$= \quad 0,654 - 0 - 0 - 0 \tag{3.35}$$

$$\approx \quad 0,650 \tag{3.36}$$

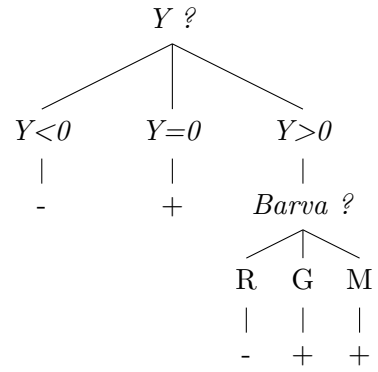
$$I(DS_{Y>0_{Barva}}) = E(DS_Y) - 3/6 * 0 - 1/6 * 0 - 2/6 * 0 \tag{3.37}$$

$$= \quad 0,654 - 0 - 0 - 0 \tag{3.38}$$

$$\approx \quad 0,650 \tag{3.39}$$

Hodnoty *informačního zisku* potvrzují původní hypotézu (o irelevanci výběru atributu), je tedy jedno, který *atribut* zvolíme, my zvolíme sloupec *Barva*. V případě, že by jeden *atribut* (např. *Barva*) rozdělil strom pouze na dva listy (G a M), tak by bylo lepší zvolit tuto proměnnou, v našem případě

oba sloupce rozdělují data na tři listy. Výsledný rozhodovací strom tedy vypadá takto:



Kapitola 4

Případová Studie

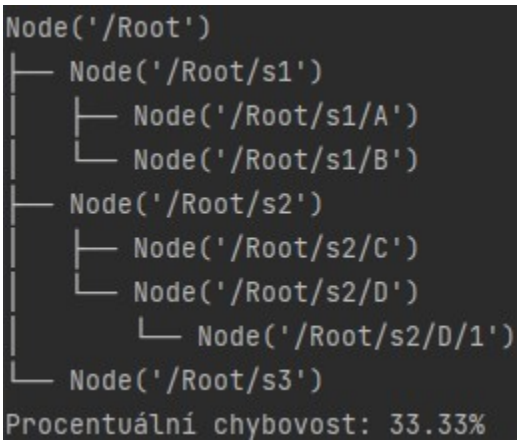
Cílem této případové studie je analýza, návrh a implementace samostatného agenta, nebo MAS, schopného budovat hypotézu na základě pozitivních a negativních příkladů užitím metody *indukce rozhodovacích stromů*.

V následujících částech se budu věnovat detailněji různým částem zadání, tedy *analýze*, *návrhu* a *implementaci*. V rámci analýzy popíši problém z pohledu uživatele, tedy jak by měl vypadat vstup a výstup programu, analýza funkčních požadavků a nároků atd.. V návrhu budu využívat postupů softwarového inženýrství a UML diagramů, vysvětlím problematiku ID3 algoritmu, také zde zvolím vhodný programovací jazyk a patřičné knihovny. Následně, před konečným popisem implementace, přidám kapitolu s popisem programovacích technik (pouze důležitých a v programu užitých), jako je například *rekurze*, *hladové algoritmy* aj.

4.1 Analýza

Cílem analýzy je připravit si podklady a návrh pro tvorbu MAS, který bude využívat *ID3 algoritmus* pro generování DT. Tento systém bude schopen načíst data (forma *csv*, nebo *excel* souboru) a rozdělit je formou 80-20, tedy 80% trénovací a 20% testovací dataset. Využije testovací data pro tvorbu DT a následně ho otestuje na druhém datasetu. Výstup bude přehledně formátovaný a čitelný.

Implementace bude ve formě dvou agentů, kde agent, reprezentující prostředí či zdroj bude odesílat data ve formě datasetů. Druhý agent příjme dva datasety, *trénovací* a *testovací*. Z trénovacího datasetu si vytvoří hypotézu ve formě DT, kterou následně otestuje na datech v *testovacím* datasetu. Výstup je standardní výpis stromu podporovaný python knihovnou anytree, konkrétně funkce *RenderTree*, příklad užití této funkce lze nalézt v Příloze A.2. Kromě výpisu stromu bude výstup obsahovat i procentuální chybovost oproti správné klasifikaci, jedná se o porovnání dvou atributů zprava (například atributy: *klasifikace_originál* a *klasifikace_dt*). Výstup programu je zaznamenán v Obrázku 4.1.



Obrázek 4.1: Příklad výstupu programu

4.2 Návrh Implementace

Cílem návrhu je objasnění fungování *ID3 algoritmu* a vytvoření konkrétního plánu pro architekturu programu, který má být výstupem této *bakalářské práce*.

Pro implementaci jsem se rozhodl použít programovací jazyk python, protože je k tomu nejlépe uzpůsoben. Respektive má velké množství podpůrných knihoven pro různé druhy problému, jako příklad zmíním knihovny *math*, *SPADE* a *padnas*. Z důvodu kompatibility s knihovnou *SPADE* používám python verzi 3.9.0. Jakožto IDE, ve kterém budu program vyvíjet, jsem zvolil studentskou licenci programu *PyCharm* dostupnou z této stránky [6].

4.2.1 ID3 Algoritmus

Algoritmus ID3 je předchůdcem algoritmu *C4.5* a užívá zmíněných pojmů *rekurze*, *entropie* a *informační zisk*. Vyjma těchto pojmů také užívá *indukci shora dolů*. Kombinací indukce s rekurzí se jedná o rekurzivní tvorbu stromu shora dolů metodou dělení na základě zvolené proměnné. Taky využívá metodu *hladových algoritmů*, kde při každém dělení zvolí nejvhodnější proměnnou (proměnnou s nejvyšším informačním ziskem). Toto vede k největší přednosti ID3 algoritmu, tedy že dokáže efektivně zpracovat datasety s velkým množstvím vstupních proměnných. Nevýhodou je, že negarantuje nejlepší možný strom.

Honzík [5] shrnuje postup rozhodování algoritmu do těchto čtyř bodů (pseudokód pro algoritmus je v Algorithm 3):

1. Mějme jeden uzel obsahující celý dataset D se všemi proměnnými, které budou použity k modelování stromů.

2. Pokud uzel splňuje ukončovací podmínky (všechny hodnoty cílové proměnné jsou v jedné kategorii - v případu Tabulky 3.1 se jedná o + nebo -), pak se uzel stane listem a nastaví se jeho *rozhodnutí*. Další rekurzivní prohledávání tohoto uzlu se zastaví.

3. Pokud není uspokojen bod 2 tohoto seznamu, pak se děje následující: určí se další nejvhodnější *atribut*, na jehož základě se data rozdělí do několika uzlů.

4. Na takto vzniklé uzly se pohlíží jako na nové kořeny dle bodu 1.

[!h]

Algorithm 1 Pseudokód výpočtu entropie

Require: Dataset df
  $entropie \leftarrow 0$
  for all $četnost$ **in** $Frekvence(df)$ **do**
    $entropie \leftarrow četnost * log2(četnost)$
  end for
  return $-1 * entropie$

Algorithm 2 Pseudokód výpočtu informačního zisku

Require: Dataset df, $sloupec$, $entropie\_setu$
Ensure: $četnost \leftarrow Velikost(subset)/Velikost(df)$
  $entropie\_subsetů \leftarrow 0$
  for all $subset$ **in** $Subsety(df, sloupec)$ **do**
    $entropie\_subsetů \leftarrow četnost * Entropie(subset)$
  end for
  return $entropie\_setu - entropie\_subsetů$

4.2.2 UML Diagramy

V rámci návrhu jsem si připravil pár poměrně důležitých diagramů. Jedná se o diagram tříd a o sekvenční diagram komunikace mezi *agenty*. Diagramy lze nalézt, ve zmíněném pořadí, v Obrázcích 4.2 a 4.3. Implementace obou diagramů je v souboru main.py v Příloze A.2.

4.2.3 Diagram Tříd

V materiálech VŠB-TU [7] je třídní diagram definován takto: *diagram popisující typy objektů a vazby mezi nimi*. Nejčastějším objektem je třída, která může například dědit z jiné třídy nebo může obsahovat proměnné typu jiné třídy aj. Diagram zobrazuje třídu v obdélníku o třech částech. První část obsahuje celý název, druhá obsahuje atributy třídy včetně jejich viditelnosti, tedy *veřejných* (označovaných +) pro atributy dostupné odkudkoli a *soukromých* (označovaných -) dostupných

Algorithm 3 Pseudokód implementace ID3 algoritmu

Require: Dataset df, $rodič$
  if $Prázdný(df)$ **then**
    return Prázdný uzel stromu
  end if
  if $Sloupce(df) = 1$ **then**
    return Uzel s četností: $Nejčetnější(df[Výstupníatribut])$
  end if
  if $Entropie(df) = 0$ **then**
    return Uzel s hodnotou *Výstupního atributu* z df a rodičem: $rodič$
  end if
  $node \leftarrow$ Uzel se jménem $Nejlepší\_sloupec(df)$ a rodičem: $rodič$
  if $Prázdný(rodič)$ **then**
    $node.hodnota = Nejčetnější(df[Výstupníatribut])$
  end if
  for all $subset$ **in** $Subsety(df, Nejlepší\_sloupec(df))$ **do**
    $id3(df, node)$
  end for

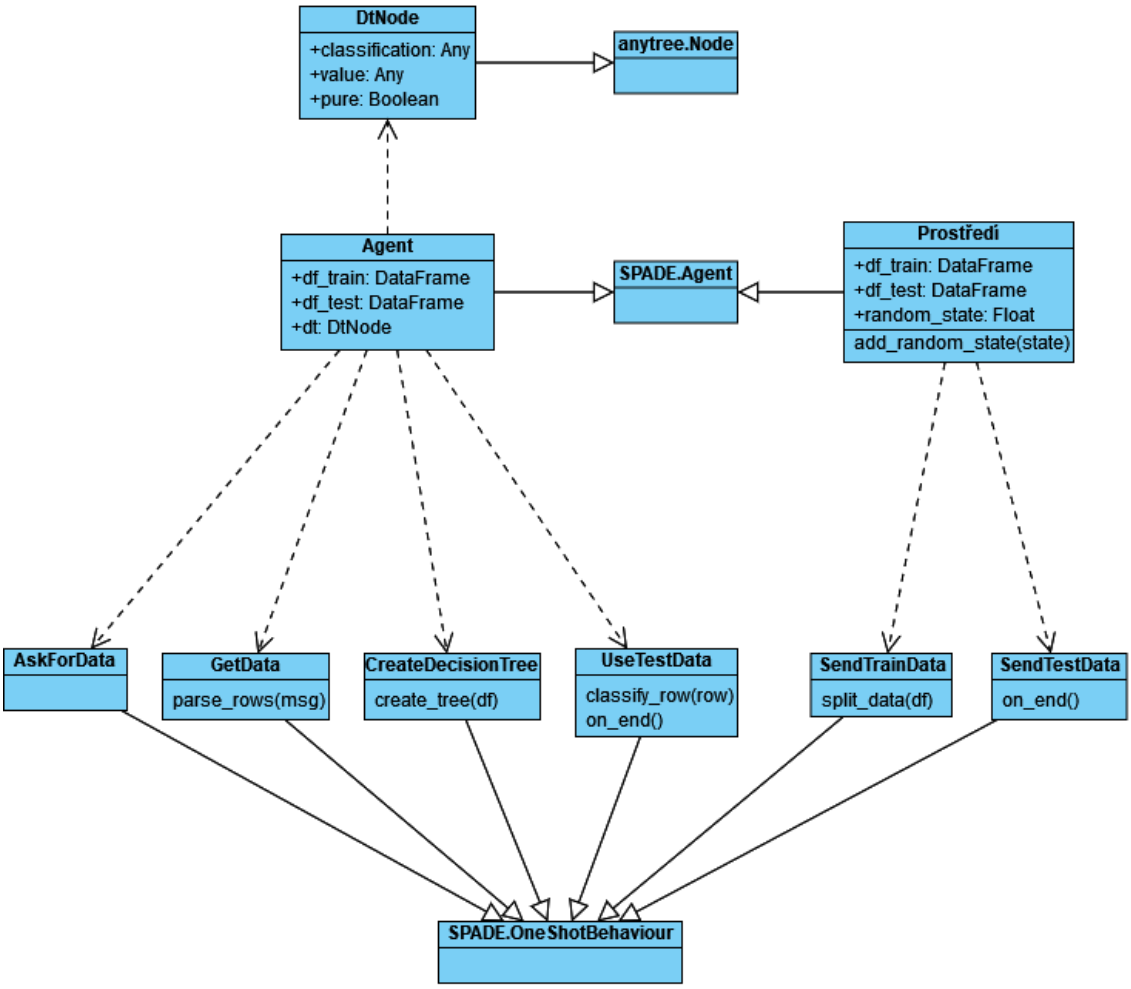
pouze třídě samotné. Python naneštěstí nemá možnost *soukromých* atributů, tedy všechny jsou označeny symbolem *+*.

Hlavní částí modelu jsou dvě třídy pro agenty pojmenované *Agent* a *Prostředí*, obě třídy dědí ze základní třídy *Agent* knihovny *SPADE*. Obě třídy následně obsahují atributy *df_train* a *df_test*, třída *Agent* navíc obsahuje atribut *dt* pro uložení samotného DT a třída *Prostředí* obsahuje atribut *náhodného stavu* pro výše zmíněné rozdělení dat 80-20. Za tímto účelem existuje také metoda, která dodává tento *náhodný stav*. Třída *Prostředí* je tedy jediná, která obsahuje všechny zmíněné části.

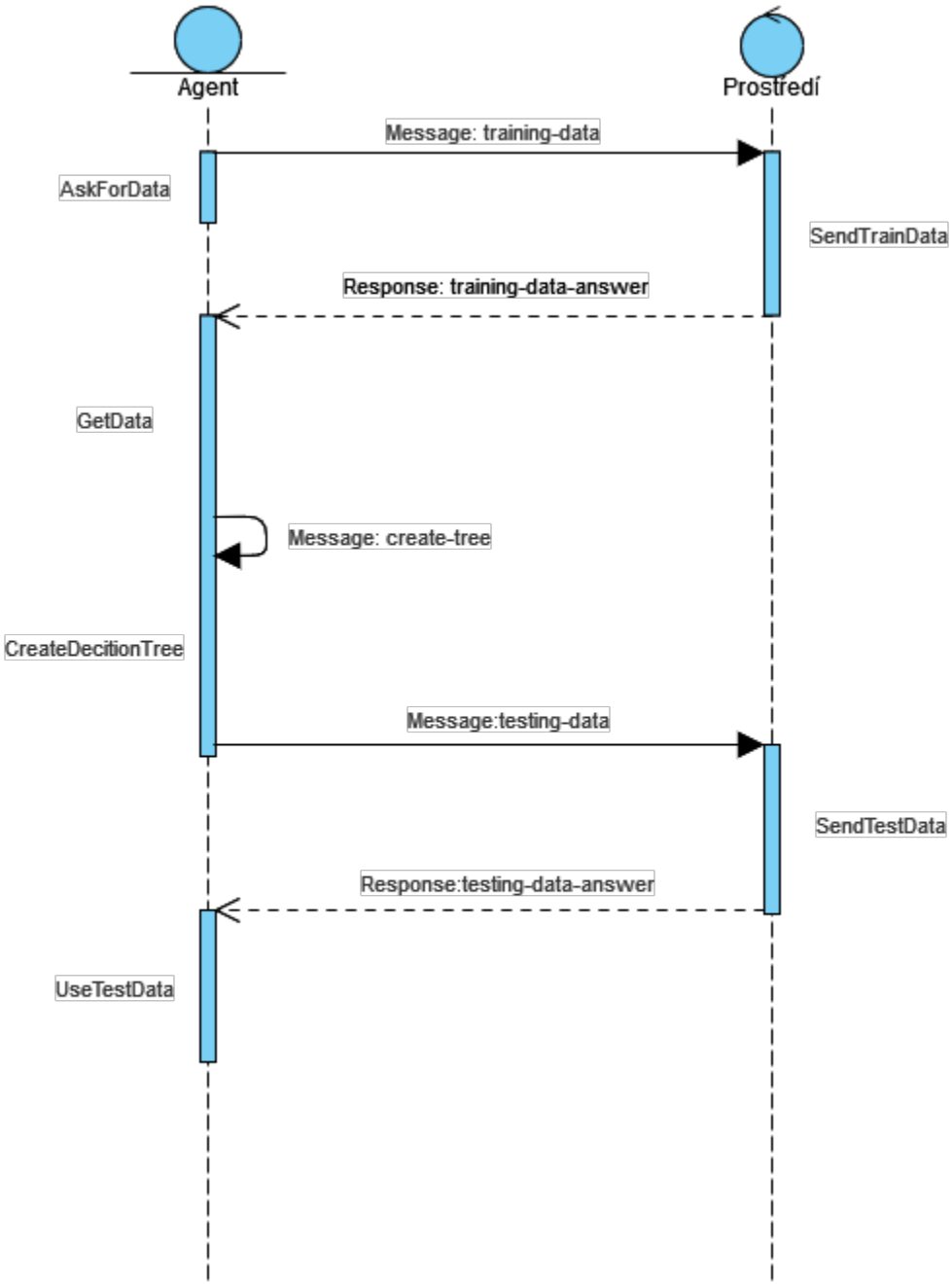
4.2.4 Sekvenční Diagram Komunikace

Sekvenční diagram [8] oproti třídnímu zobrazuje interakci zvolených objektů v čase. Konkrétně je reprezentován svislými čarami u každého objektu, v mém případě se jedná o oba agenty a jejich vzájemnou komunikaci, tedy *Agent* a *Prostředí*. Tyto svislé čáry reprezentují čas a obdélníky zobrazují chování, či akce, které objekty k interakci využívají. V mém případě se jedná o chování (znázorněné ve třídním diagramu) dědící z abstraktních tříd knihovny *SPADE*. Šipky znázorňují směr komunikace a jeho typ, plná čára znamená zprávu z místa A do místa B (v mém případě z chování do chování). Čárkované značí odpověď na tyto zprávy.

V mém případě modré obdélníky znázorňují chování (pojmenovány stejně jako ve třídním diagramu) a šipky značí standardní komunikaci. *Prostředí* vždy reaguje na aktivitu *Agenta*, tedy na požádání (s informací o úspěchu) odesílá potřebná data. *Agent* naopak provádí hlavní cíl práce, tedy tvorbu DT (*Message: create-tree*).



Obrázek 4.2: UML: Třídní diagram



Obrázek 4.3: UML: Sekvenční diagram

4.2.5 Testování a Ověření

Na doporučení pana Ing. Radka Svobody jsem k ostrému testování využil volně dostupné datasety, dataset v tabulce 3.1 je pouze ilustrační. Datasety budou reprezentovány jakožto mix *csv* (data jsou rozdělená pomocí čárky a odřádkování) a *excel* (data mají tabulkovou formu) souborů pro ověření multifunkčnosti. Další testovací proměnnou je *náhodný stav*, tedy pro dvě iterace programu se stejným *náhodným stavem* musí vycházet stejný DT a stejná procentuální odchylka.

4.2.6 Plánované Užití

V rámci práce *agent* odešle zprávu s žádostí o testovací data, na kterou obdrží odpověď obsahující *trénovací dataset* o velikosti 80% celku. Agent přijme data a zpracuje je do DT, po jeho dokončení požádá o zbylých 20%, které užije jako *testovací dataset*. *Agent* reprezentující prostředí se v tuto chvíli ukončí. Druhý *agent* dokončí *klasifikaci* a také se ukončí. Mezi těmito kroky se odehrává komunikace pomocí FIPA-ACL zpráv.

4.3 Užité Programátorské Techniky

Před samostatnou implementací je potřeba přidat vysvětlení několika programovacích technik, v kapitole 3.2 jsem sliboval vysvětlení *rekurze*, ke které se připojí technika *hladových algoritmů*. *ID3 algoritmus* používá obě tyto techniky pro tvorbu DT.

4.3.1 Rekurze

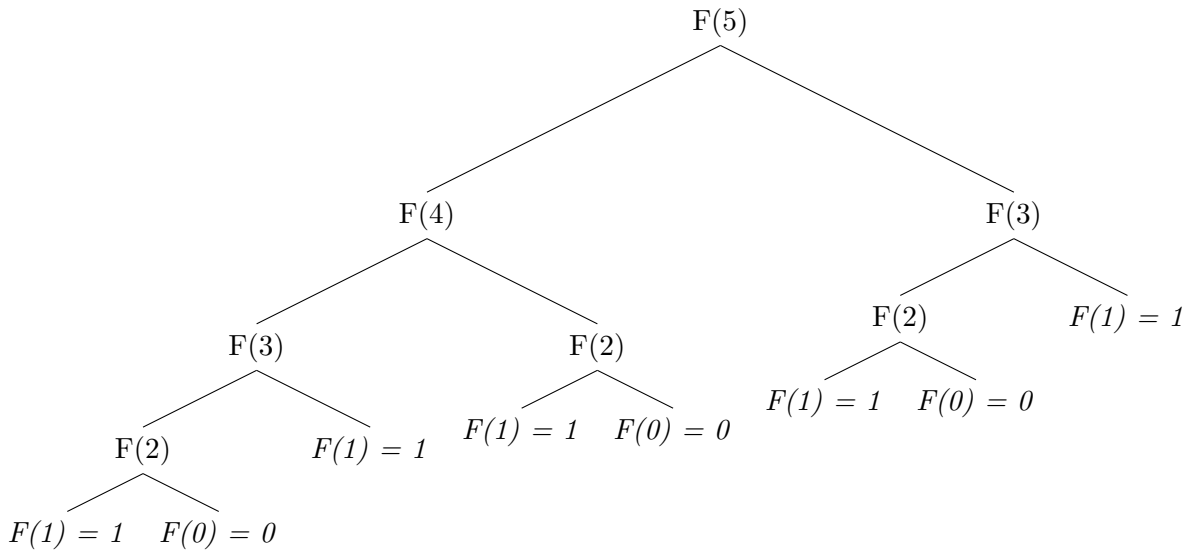
Rekurze je programovací technika opakovaného volání funkce, před ukončením jejího původního volání. Tímto se problém dá rozložit na menší podproblémy stejného nebo podobného rázu, které lze vyřešit pomocí konvenčních metod.

Výborným příkladem je rekurzivní řešení Fibonacciho posloupnosti (posloupnosti čísel, kde se sčítají předchozí dva členy, s předpisem: 0, 1, $F(n) = F(n-1) + F(n-2); n \in [2, \infty)$). Listing 4.3.1 obsahuje vzorovou implementaci Fibonacciho posloupnosti v programovacím jazyce *python*.

```python
# Definice funkce Fibonacci; F(n)
def fibonacci(number):
    # Ukončovací podmínka; vrací první 2 členy posloupnosti
    if number in [0, 1]:
        return number

    # Rekurzivní volání definice pro: n > 2; F(n - 1) + F(n - 2)
    return fibonacci(number - 1) + fibonacci(number - 2)
```

Listing 4.1: Implementace Fibonacciho posloupnosti v Pythonu



Výše zobrazený strom slouží jako vizuální reprezentace rekurze Fibonacciho posloupnosti se vstupem n = 5.

4.3.2 Hladové Algoritmy

Hladové algoritmy v [9] (taky nazývány *hltavé* v [10]) odkazují na algoritmy, které hledají globálně optimální řešení pomocí opakované volby lokálně optimálních řešení. Tento postup je vhodný pro problémy, které vyžadují větší množství dílčích rozhodnutí během průběhu (většinou se jedná o problémy užívající *rekurze*). Toto může vést k problému, kde lokální posloupnost optimálních řešení nevede k optimálnímu konečnému řešení. Tyto algoritmy jsou nejčastěji používány pro rychlé, jednoduché a levné řešení, které jsou například potřebné v oblastí řízení dopravy, či financí. Dobrým konkrétním příkladem je nalezení minimální kostry grafu, či nalezení nejkratší cesty mezi body v grafu. Pro lepší pochopení popíši výše zmíněné nalezení minimální kostry stromu.

Možným ilustračním příkladem, problému nalezení minimální kostry grafu, je problém stavitele železnic. Tento stavitel má k dispozici graf, kde každý vrchol symbolizuje město v regionu a každá hrana symbolizuje cestu, na které lze stavět železnici. Tyto hrany jsou označeny číslem, které reprezentuje částku potřebnou k realizaci takové cesty. Úkolem je spojit všechna města železnicí za co nejmenší obnos.

Vzhledem k tomu, že hledáme minimální kostru grafu, bez cyklů, pak platí, že výsledek bude stromem. První vrchol se stane *kořenem* a následně se budou přidávat další vrcholy jakožto *vnitřní* uzly. Obecný postup tedy lze shrnout do několika bodů:

1. Vyber první uzel, může být libovolný z celého grafu

2. K aktuálnímu grafu připoj nejlevnější hranu (úsek železnice)

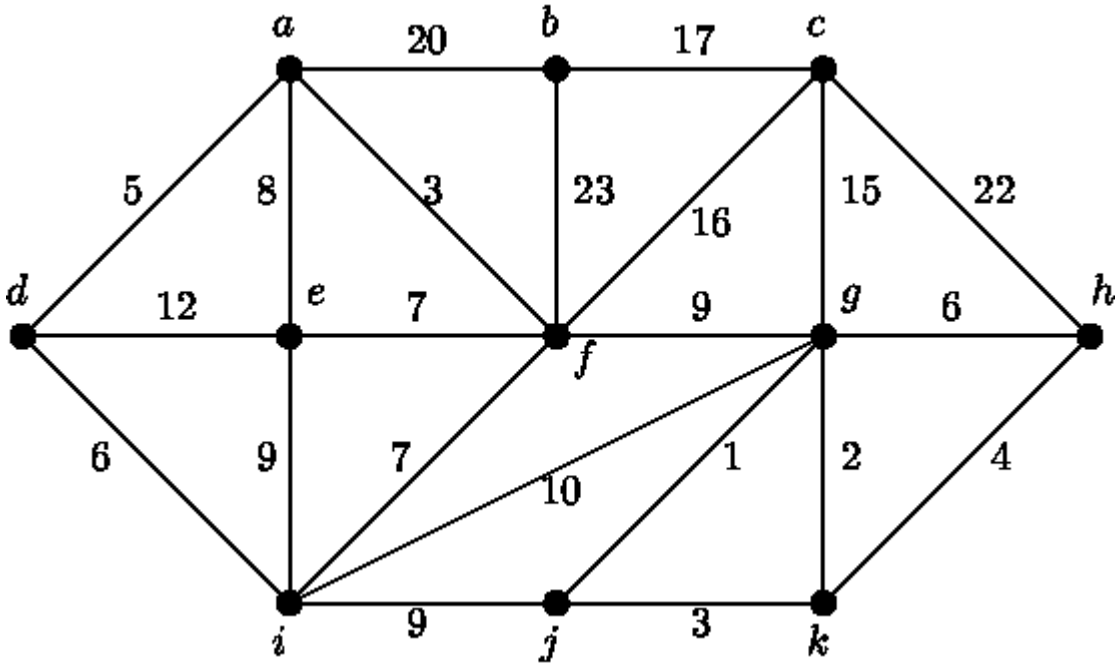
3. Opakuj bod č. 2 dokud nebude kostra hotová

Výše zmíněné body se dají shrnout do relativně jednoduchého pseudokódu, kde vstupem je graf G daný množinou vrcholů V_G a množinou hran H_G. Dále existuje funkce f, která vrací pro danou hranu její ohodnocení, tedy cenu. Výsledek je graf H se svými množinami vrcholů a hran V_H, H_H. Pseudokód lze nalézt v Algorithm 4.

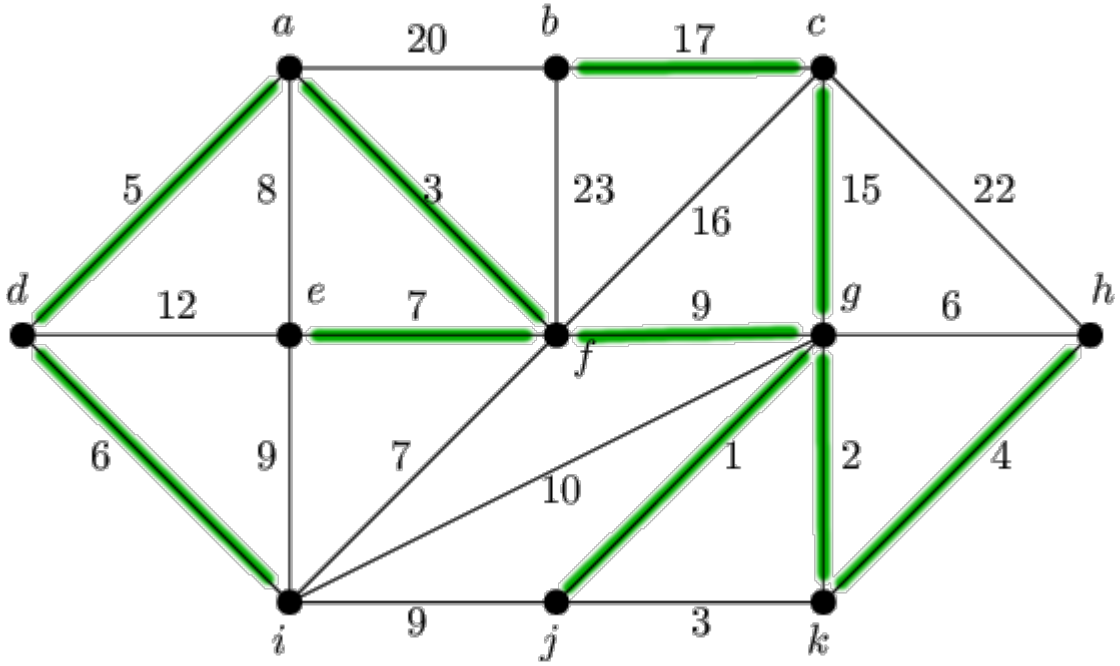
Algorithm 4 Pseudokód pro hledání nejmenší kostry grafu

```
Require: V_G, H_G, f
  uzel ← V_G[0]
  V_H ← uzel
  H_H ← ∅
  M ← V_G − uzel
  for all vrchol in M do
    if (uzel, vrchol) in H_G then
      otec(vrchol) ← uzel
      hodnota(vrchol) ← f(uzel, vrchol)
    else
      hodnota(vrchol) ← ∞
    end if
  end for
  while not Prázdný(M) do
    v ← Minimální_hodnota(M)
    V_H ← V_H ∪ v
    H_H ← H_H ∪ (v, otec(v))
    M ← M − vrchol
    for all w in M and f(v, w) < hodnota(w) do
      if (v, w) in H_G then
        otec(w) ← v
        hodnota(w) ← f(v, w)
      end if
    end for
  end while
  return H(V_H, H_H)
```

Výše popsaný algoritmus můžeme aplikovat na graf G z Obrázku 4.4. V prvním kroku zvolíme libovolný vrchol, tedy vrchol *g*, jakožto výchozí. Odtud postupujeme postupně po hranách s nejmenší cenou, tedy do vrcholu *j*, postupně přidáváme další hrany a vrcholy v tomto pořadí: *k, h, f, a, d, i, e, c, b*. Při změně výchozího vrcholu se změní pouze pořadí, v jakém se vrcholy přidávají. Výsledná minimální kostra grafu je zobrazena na Obrázku 4.5.



Obrázek 4.4: Ilustrační graf G



Obrázek 4.5: Kostra grafu G

4.4 Implementace

V této části se budu věnovat popisu implementace, konkrétně popíši jaké funkce či třídy jsem použil a jak jsem je zakomponoval do projektu. Samostatně se budu věnovat knihovnám SPADE a pandas, protože dohromady tvoří většinu užité infrastruktury. Vyjma knihoven *SPADE* a *pandas* jsem využil tyto funkce: *log2* z knihovny *math* potřebnou pro výpočet *entropie* datasetu. Třídu *Node* z knihovny *anytree*, která posloužila jakožto předek ke třídě *DtNode*, ze které se skládá výsledný DT. Jako poslední jsem využil funkce $sleep(ms : int)$ z knihovny *time*, pro uspání vláken agentů a funkci *randomint* z knihovny *random* pro náhodné zamíchání dat. Kompletní seznam knihoven a funkcí lze nalézt na prvních řádcích Příloh A.1 a A.2.

4.4.1 Knihovna SPADE

Jak jsem zmiňoval v úvodu této kapitoly pro implementaci jsem zvolil jazyk *python* a volně dostupné knihovny a jejich dokumentace. MAS, tedy agenti a jejich vnitřní chování, je realizováno pomocí knihovny *SPADE* (Smart Python Agent Development Environment) s dostupností dokumentace zde [11]. Výhodou, a tedy i důvodem proč jsem si ji zvolil je, že v základu podporuje FIPA strukturu zpráv. Pomocí třídy *Message* a metody $set\_metadata(key : str, value : str) \rightarrow None$ je možné dodat libovolné množství parametrů FIPA (v mém případě postačil základní parametr *performative*, parametry *sender* a *receiver* jsou zakomponovány přímo ve třídě *Message*). Vyjma tohoto zjednodušení má knihovna SPADE implementovány třídy pro agenta (*Agent*) a pro chování agentů (např. *OneShotBehaviour*, *PeriodicBehaviour* a *CyclicBehaviour*) v mém případě postačilo chování *OneShotBehaviour*. Toto chování, jak název vypovídá, se po vytvoření a zařazení k agentovi provede pouze jednou. Jako příklad uvedu poslední chování agenta *agent_dt_bp*, které má za úkol projít DT a klasifikovat příchozí dataset, jakožto poslední chování v agentovi má za úkol agenta ukončit, po dokončení své práce. Chování se nazývá *UseTestData* a nachází se zhruba uprostřed Přílohy A.2.

SPADE realizuje agentní komunikaci (v případě potřeby i komunikaci mezi agenty a lidmi) pomocí XMPP serveru. Z doporučeného seznamu v dokumentaci [11] jsem si vybral server Openfire a Gajim klienta. Po standardní instalaci jsem přidal dva účty: *agent_dt_bp* a *data_dt_bp*, kde *agent* má za úkol požádat o data, vytvořit DT a zpracovat *testovací* dataset. Agent *data* pouze načte *csv* soubory, provede rozdělení 80-20 a sloučení datasetu dle formátu: $jméno\_sloupce(hodnota\_ve\_sloupci)$. Výsledkem je FIPA zpráva od *data* pro *agenta* s performative nastavením *answer-training-data*, nebo *answer-testing-data* obsahující dataset v těle zprávy. $X(1)\&Y(-2)\&Barva(R)\&Klasifikace(-)$ je příkladem komponování prvního řádku testovacího datasetu z Tabulky 3.1. Agent *agent* po převzetí zprávy provede separování daného datasetu s podporou těchto datových typů: *int*, *bool* a *string* (*char* není třeba, protože python reprezentuje *char* jakožto *string*), testování hodnot probíhá ve

stejném pořadí. Funkce pro sloučení a separování se jmenují *compose_rows* a *parse_rows*, lze je nalézt na druhé straně Přílohy A.1.

4.4.2 Knihovna Pandas a Užité Datasety

Způsob načtení *csv* souborů a práce s datasety je realizována pomocí knihovny *pandas*, která značně zjednodušuje práci s datasety. Konkrétně se jedná o funkci $read\_csv(path : str, header : int)$ a třídu $DataFrame()$. Výstup separovací funkce, stejně jako $read\_csv()$ je DataFrame obsahující korektní data.

Všechny použité datasety jsou volně dostupné zde [12]. Konkrétně jsem vybral datasety: Iris, Nemoci srdce v Cleavelandu a Piškvorky. Přesnější odkazy na zmíněné datasety: [13, 14, 15]. Agenta *data* rozdělí dataset na dva (*trénovací* a *testovací* části) s náhodným zamícháním dat a poměrem 80% trénovací a 20% testovací (80-20). Následně agent odebere *testovací* data z *trénovacích* a do *trénovacího* datasetu přidá sloupec pro klasifikaci, oba datasety uloží a připraví k odeslání druhému *agentovi*.

Zmíněné datasety se liší ve velikosti, kde nejmenší je dataset Iris [13] o zhruba 150 řádcích a pěti atributech, kde čtyři z nich jsou číselné a poslední je textová klasifikace. Tento dataset by měl teoreticky mít největší chybovost, protože zmíněné rozdělení 80-20 bude mít málo vhodných dat pro tvorbu DT. Taky DT bude pouze pro konkrétní číselné hodnoty, tedy pokud není dostatek stejných hodnot, pak bude DT málo rozvětvený.

O něco lépe by se mělo dařit druhému datasetu, tedy Srdečním nemocem [14], kde se nachází 303 řádků a 14 výhradně číselných atributů s velkým množstvím opakování. Tento dataset by měl mít mírnou chybovost. DT by měl mít malou hloubku (malý počet *Vnitřních* uzlů) a měl by být poměrně široký (větší množství *listů* na každý *vnitřní* uzel).

V posledním datasetu s Piškvorky [15] je 958 řádků a 10 atributů. Jedná se o *binární* (výsledkem této klasifikace jsou pouze dvě hodnoty, nejčastěji *true* a *false*, tedy *pravda* a *nepravda*) klasifikaci, kde většina atributů je textových a obsahují tyto hodnoty: x, o, b. Chybovost tohoto datasetu by měla být minimální a DT by měl být značné rozsáhlý, jak do hloubky, tak do šířky.

4.4.3 Hlubší Popis Implementace Funkcí a Algoritmu

Původním plánem byl soketový server, kde server by sloužil jakožto zprostředkovatel dat a klient by dával žádosti o potřebná data (*trénovací* a *testovací* datasety). Tento postup jsem následně zavrhl z důvodu, že při použití soketů není zaručeno, že zpráva dojde vcelku a bez chyb. Toto vyústilo v hledání lepšího řešení, tedy užití knihovny SPADE (z výše zmíněných důvodů) a XMPP serveru.

Drobnou nevýhodou je nutnost programování chování agentů jakožto třídy (dokážu si představit, že ve větších MAS je toto výhodou z důvodu znovuvyužití, každopádně v mém případě by bohatě postačily metody tříd). Toto vyústilo k vytvoření dvou souborů, kde *main* a třídy pro agenty budou v souboru *main.py* a pomocné funkce se třídami ve *functions.py*.

Nejprve začnu popisem souboru *functions.py*. Na jeho začátku se nachází definice třídy *DtNode*, která dědí ze třídy *Node* z knihovny *anytree*. Jedná se o rozšíření přidáním atributů: *classification*, *value* a *pure*, kde *pure* jednoznačně definuje *list* a jeho přidružené *rozhodnutí* (*klasifikaci*). Třídní atribut *value* slouží při klasifikaci k jasnému postupu stromem směrem ke správnému *listu*. Tento soubor taky obsahuje čistě stylistické funkce pro tvorbu zpráv a jejich šablon (třídy *Message* a *Template* z knihovny SPADE). Nachází se zde i výše zmíněná funkce *Fibonacci* a funkce pro práci s datasety. Soubor je přiřazen k této práci jakožto Příloha A.1.

Dále následuje implementace výpočtu *entropie*, *informačního zisku* a aktuálního nejlepšího sloupce. Jedná se o čistou implementaci pseudokódu z Algorithm 1, 2 a 3. Pseudokód je poměrně sebe vysvětlující, tedy nebudu užívat komentářů pro vysvětlení funkcionality. Výsledné kódy lze nalézt zhruba v půlce souboru *functions.py* (A.1).

V neposlední řadě je, po vytvoření stromu, potřeba stromem projít, *klasifikovat* daný řádek a následně vypočítat chybovost. Výpočet chybovosti je poměrně jednoduchý, jde o porovnání posledních dvou sloupců v *testovacím* datasetu a následné vydělení velikostí celého datasetu. Tato operace vrací číslo v tomto rozmezí: $x \in< 0, 1 >$, kde vynásobení stem dává procentní výsledek, tedy chybovost DT.

Vzhledem k tomu, že nepotřebuji projít celý strom, jako například při použití třídícího algoritmu *TreeSort*, tak jsem implementoval vlastní funkci pro průchod stromu. Jde o rekurzivní funkci, která na základě shodnosti atributu *value* prochází stromem shora dolů, tedy od *kořene* po správný *list*. Pokud nenajde správný list (hodnota *pure* je nastavena na *true*), pak vrací výchozí hodnotu stromu, tedy atribut *classification* v *kořenu*. Jedná se o poslední dvě funkce v Příloze A.1.

Výše zmíněné funkce musí být někde volány a jejich výsledky zpracovány, toto se děje v tělech *agentů* v souboru *main.py*. Každý agent knihovny SPADE musí být vytvořen s minimálně dvěma parametry, konkrétně *jid* a *password*, jedná se o jméno učtu, doménu serveru a heslo od XMPP účtu. Například *agent_dt_bp@HAB0065*, kde *agent_dt_bp* je jméno účtu a *@HAB0065* je doména mého lokálního serveru.

Následuje spuštění agenta, které vrací *příslib* (koncept paralelního programování, který neblokuje původní vlákno), na který je možné počkat pomocí metody *result* (zablokování vlákna a počkání na výsledek z vlákna jiného). Vyjma této metody ještě existuje metoda *wait*, která zajišťuje počkání na další *agenty* pro komunikaci (lze nalézt v sekci *Agent communications* v [11]). Dokumentace také zmiňuje užití funkce *quit_spade*, která má za úkol ukončit všechny agenty, jejich chování a uvolnění všech používaných zdrojů. Ukázka funkce *main* s užitím zmíněných metod je na konci souboru *main.py* přiloženého jako Příloha A.2.

Po spuštění agenta se provede jeho metoda *setup*, kde je doporučeno tvořit chování a jejich nastavení, toto lze udělat i mimo tuto funkci při použití agentovy *add_behaviour* metody, avšak dokumentace tento způsob práce nedoporučuje. V této metodě je také vhodné přidávat atributy dostupné pouze agentovi a jeho prostřednictvím i jeho chováním.

Při tvorbě chování je možné přidat i šablonu zprávy (třída *Template*). Pokud chování očekává zprávu od nějakého agenta, tak se přidáním šablony zajišťuje, že zprávu, která odpovídá šabloně, dostane správné chování (v případě, že na zprávu čeká vícero chování).

Kapitola 5

Závěr

V rámci *úvodu* jsem vytyčil dva hlavní cíle pro tuto práci, tedy shrnutí teorie ML a MAS, společně s následnou tvorbou MAS o dvou agentech s cílem implementovat ID3 algoritmus a vyzkoušet jeho chybovost.

Shrnutí teorie a vysvětlení pojmů jsem se věnoval ve druhé kapitole nazvané *teoretická část*. Druhé části, tedy MAS a implementaci ID3 algoritmu, jsem se věnoval v kapitole *případová studie*.

Výsledkem jsou dva soubory se zdrojovým kódem (*functions.py* a *main.py*), které jsou rozebrány v textu této práce a vloženy jakožto přílohy. Tento kód jsem následně otestoval na třech, výše popsaných, volně dostupných datasetech s těmito výsledky:

Dataset *Iris*, po jeho rozložení na 80-20 za použití čísla 1000, má chybovost $6,67\%$, šířka stromu je středně velká a hloubka je 2. Tyto hodnoty jsou v rozporu s úvahou uvedenou v kapitole 3.

Druhý dataset (*Srdeční nemoci*) má, se stejnými podmínkami, chybovost $38,33\%$, značnou šířku stromu a hloubka je stejná jako u prvního datasetu. Kromě chybovosti se má úvaha z kapitoly 3 potvrdila.

Poslední dataset nazvaný *Piškvorky* má chybovost $14,66\%$, šířka stromu je velmi rozsáhlá a hloubka dosahuje sedmé úrovně. Výsledek je tedy stejný jako u *Srdečních nemocí*, tedy rozdíl oprati úvaze je ve vyšší chybovosti.

Tato chybovost vznikla z důvodu volby čísla, které může být optimální pro menší dataset a být poměrně nevhodné pro datasety větší, či střední velikosti. Objektivnějšího výsledku je možné dosáhnout puštěním stokrát pro každé číslo z intervalu $n \in < 1, 1000 >$. Po následném průměrování a vybrání nejlepšího čísla bych mohl výslednou chybovost snížit řádově o menší desítky procent.

Je možnost rozšíření této práce, dokáži si představit vícero agentů, každý s implementací jiného algoritmu pro generování DT, a jejich vzájemnou soutěž. Alternativně porovnání DT s *neuronovými sítěmi*, případně jinou odvětví ML. Dalším rozšířením může být zvolení optimálního čísla, které bylo popsáno výše. Všechny tyto rozšíření jsou možnostmi při navazujícím studiu, tedy v semestrálním projektu, či přímo u *diplomové práce*.

Příloha A

Zdrojové Kódy

A.1 Soubor functions.py

```
from pandas import DataFrame
from spade.message import Message
from spade.template import Template
from math import log2
from anytree import Node


# slightly modification for class Node
class DtNode(Node):
    def __init__(self, name, classification, parent=None, children=None):
        super().__init__(name, parent=parent, children=children)
        self.classification = classification
        self.value = None

        if classification is None:
            self.pure = False
        else:
            self.pure = True


# Helper function for Message creation
def create_msg(to, sender, body):
    msg = Message()
    msg.to = to
```

```
    msg.sender = sender
    msg.body = body

    return msg


# Helper function for Template creation
def create_template(to, sender, body):
    template = Template()
    template.to = to
    template.sender = sender
    template.body = body

    return template


# Fibonacci algorithm implemented
def fibonacci(number):
    if number in [0, 1]:
        return number

    return fibonacci(number - 1) + fibonacci(number - 2)


# Create sendable body from dataset
def compose_rows(row):
    return '&'.join(f"{col}({row[col]})" for col in row.index)


# Parse incoming response which contains dataset in body
def parse_rows(response):
    split_df = [x.split("&") for x in response.body.split("\n")]
    df = DataFrame(columns=split_df[0])
    for row in split_df[1::]:
        vals = [x[x.find("(") + 1:-1] for x in row]

        for i in range(len(vals)):
            try:
```

```
                vals[i] = int(vals[i])
            except ValueError:
                if vals[i] == "True":
                    vals[i] = True
                elif vals[i] == "False":
                    vals[i] = False

        df.loc[len(df.index)] = vals

    return df


# Calculate Entropy based on last column -> Classification
def calculate_set_entropy(df: DataFrame, col):
    entropy = 0.0
    frequencies = df[col].value_counts()
    total = len(df[col].values)

    for val in frequencies:
        p_val = val/total
        entropy += p_val*log2(p_val)

    return -1*entropy


# Based on values in last column, calculate entropy for each col
def calculate_information_gain_for_col(df: DataFrame, col, set_entropy: float):
    subset_entropies = 0.0
    values = df[col].value_counts()

    for val, count in values.items():
        subset = df[df[col] == val]
        subset_entropies += count/len(df[col].values) * calculate_set_entropy(
            subset, subset.columns[-1])

    return set_entropy - subset_entropies
```

```
# Calculate and return column with highest information gain for current dataset
def calculate_best_column(df: DataFrame):
    set_entropy = calculate_set_entropy(df, df.columns[-1])
    best = ("", 0)

    for col in df.columns[:-1]:
        tmp = (col, calculate_information_gain_for_col(df, col, set_entropy))

        if tmp[1] > best[1]:
            best = tmp

    return best[0]


# Main logic of id3 algorithm, with default value of df[df.columns[-1]].mode()[0]
def create_tree(df: DataFrame, parent):
    if df.empty:
        return DtNode(None, None)

    if len(df.columns) == 1:
        return DtNode("list", df[df.columns[-1]].mode()[0], parent)

    if calculate_set_entropy(df, df.columns[-1]) == 0:
        return DtNode("list", df[df.columns[-1]].mode()[0], parent)

    best_col = calculate_best_column(df)
    node = DtNode(best_col, None, parent)

    if parent is None:
        node.classification = df[df.columns[-1]].mode()[0]

    values = df[best_col].value_counts()
    for val, count in values.items():
        subset = df[df[best_col] == val]
        subtree = create_tree(subset, node)
        subtree.value = val

    return node
```

```
# Main logic for traversing of created tree
def classify_row(node: DtNode, data_row):
    if node.pure:
        return node.classification

    col_to_split = node.name
    row_val = data_row[col_to_split]
    child = None

    for chld in node.children:
        if chld.value == row_val:
            return classify_row(chld, data_row)

    if child is None:
        return node.root.classification


# Compare last 2 columns and get error rate between them (since the last column of
    test dataset is empty)
def get_error_rate(df: DataFrame):
    cols = df.columns[-2:]

    mask = df[cols[0]] != df[cols[1]]

    return sum(mask) / len(df) * 100
```

Listing A.1: Pomocné funkce a třídy užité v *případové studii*

A.2 Soubor main.py

```
from time import sleep
from spade.agent import Agent
from spade.behaviour import OneShotBehaviour
from spade import quit_spade
from pandas import read_csv, read_excel
from anytree import RenderTree
```

```
from random import randint
from functions import compose_rows, parse_rows, create_msg, create_template,
    create_tree, classify_row, get_error_rate


class BpAgent(Agent):
    class AskForData(OneShotBehaviour):
        async def run(self):
            msg = create_msg("data_dt_bp@habro", str(self.agent.jid), "I am ready
                for Decision Tree creation using "
                                                                "ID3 algorithm.
                                                                    Please send
                                                                    training data."
                                                                    )
            msg.set_metadata("performative", "training-data")
            # send msg
            await self.send(msg)

            # stop agent from behaviour
            self.kill(exit_code="I asked for data")
            return

    class GetData(OneShotBehaviour):
        async def run(self):
            response = await self.receive(timeout=100)

            # Create message for CreateDecisionTree
            msg = create_msg(str(self.agent.jid), str(self.agent.jid), None)
            msg.set_metadata("performative", "create-tree")

            if response:
                # Parse data:
                self.agent.df_train = parse_rows(response)

                msg.body = "Data parsed successfully."
            else:
                print("No response from Data to Agent.")
                msg.body = "Problem with message or with data parsing."
```

```
        await self.send(msg)

        self.kill(exit_code="response received")
        return

class CreateDecisionTree(OneShotBehaviour):
    async def run(self):
        response = await self.receive(timeout=100)

        msg = create_msg("data_dt_bp@habro", str(self.agent.jid), None)

        if response.body.__contains__("successfully"):
            try:
                self.agent.tree = create_tree(self.agent.df_train, None) #
                    Create tree using ID3 Algorithm
                print(RenderTree(self.agent.tree))

                msg.body = "Decision tree made successfully. Please send testing
                     data"
                msg.set_metadata("performative", "testing-data")
            except Exception as E:
                msg.body = "There was a problem with decision tree creation"
                msg.set_metadata("performative", "problem")

        await self.send(msg)

        self.kill(exit_code="Tree successfully built")
        return

class UseTestData(OneShotBehaviour):
    async def run(self):
        response = await self.receive(timeout=100)

        if response:
            self.agent.df_test = parse_rows(response)
```

```
            # print("Training data: \n {}".format(self.agent.df_test.to_string
                ()))
            self.agent.df_test[self.agent.df_test.columns[-1]] = self.agent.
                df_test.apply(lambda x: classify_row(self.agent.tree, x), axis
                =1)
            # print("Classified data: \n {}".format(self.agent.df_test.
                to_string()))

            print("Procentuální chybovost: {:.2f}%".format(get_error_rate(self.
                agent.df_test)))

        self.kill(exit_code="Success")
        return

    async def on_end(self):
        await self.agent.stop()

async def setup(self):
    # Init variables
    self.df_train = None
    self.df_test = None
    self.dt = None

    # Start data transfer
    b = self.AskForData()
    self.add_behaviour(b)

    # Message template for data parsing
    c = self.GetData()
    template = create_template(str(self.jid), "data_dt_bp@habro", None)
    template.set_metadata("performative", "answer-training-data")
    self.add_behaviour(c, template)

    a = self.CreateDecisionTree()
    template2 = create_template(str(self.jid), str(self.jid), None)
    template2.set_metadata("performative", "create-tree")
    self.add_behaviour(a, template2)
```

```
        d = self.UseTestData()
        template3 = create_template(str(self.jid), "data_dt_bp@habro", None)
        template3.set_metadata("performative", "answer-testing-data")
        self.add_behaviour(d, template3)

# ------------------------------------------------------------------------


class BpEnvir(Agent):
    class SendTrainingData(OneShotBehaviour):
        async def run(self):
            msg = await self.receive(timeout=100) # wait for a message for 10
                seconds
            code = "response not send"

            if msg and msg.body.__contains__("Please send training data."):
                self.agent.df_train = read_csv(r".\data\iris.data", header=0)
                # self.agent.df_train = read_csv(r".\data\processed.cleveland.data
                    ", header=0)
                # self.agent.df_train = read_csv(r".\data\tic-tac-toe.data", header
                    =0)
                # self.agent.df_train = read_excel(r".\data\points_train.xlsx",
                    sheet_name="List1", header=0)
                self.agent.df_train.reset_index()

                # 80-20 split
                self.agent.df_train = self.agent.df_train.sample(frac=1,
                    random_state=self.agent.random_state)
                # self.agent.df_train = self.agent.df_train.sample(frac=1,
                    random_state=1000)
                self.agent.df_train = self.agent.df_train.copy()
                self.agent.df_test = self.agent.df_train.iloc[:int(len(self.agent.
                    df_train)*0.2)]
                self.agent.df_test = self.agent.df_test.copy()
                self.agent.df_train.drop(self.agent.df_test.index, inplace=True)
                self.agent.df_train.reset_index()
                self.agent.df_test.reset_index()
```

```
            # Add empty column
            col = self.agent.df_test.columns[-1]
            self.agent.df_test.rename(columns={col: "{}_correct".format(col)},
                inplace=True)
            self.agent.df_test[col] = None

            cols = self.agent.df_train.columns.tolist()

            # Convert all rows to sendable objects / send entire dataframe at
                once
            exportable_df = ['&'.join(cols)]
            exportable_df.extend(self.agent.df_train.apply(compose_rows, axis
                =1).tolist())

            # Msg preparation
            response = create_msg("agent_dt_bp@habro", str(self.agent.jid), '\n
                '.join(exportable_df))
            response.set_metadata("performative", "answer-training-data")

            await self.send(response)

            code = "response send"
        else:
            print("Did not received any message after 10 seconds")

        # stop agent from behaviour
        self.kill(exit_code=code)
        return

class SendTestData(OneShotBehaviour):
    async def run(self):
        msg = await self.receive(timeout=100) # wait for a message for 10
            seconds
        code = "response not send"

        response = create_msg("agent_dt_bp@habro", str(self.agent.jid), None)
        response.set_metadata("performative", "answer-testing-data")
```

```
        if msg and msg.body.__contains__("Please send testing data"):
            cols = self.agent.df_test.columns.tolist()

            # Convert all rows to sendable objects / send entire dataframe at
                once
            exportable_df = ['&'.join(cols)]
            exportable_df.extend(self.agent.df_test.apply(compose_rows, axis=1)
                .tolist())

            # Msg preparation
            response = create_msg("agent_dt_bp@habro", str(self.agent.jid), '\n
                '.join(exportable_df))
            response.set_metadata("performative", "answer-testing-data")

            await self.send(response)

            code = "response send"
        else:
            print("Did not received any message after 10 seconds (Data->
                SendTestData)")

        # stop agent from behaviour
        self.kill(exit_code=code)
        return

    async def on_end(self):
        await self.agent.stop()

async def setup(self):
    self.df_train = None
    self.df_test = None
    self.random_state = None

    template = create_template(str(self.jid), "agent_dt_bp@habro", None)
    template.set_metadata("performative", "training-data")
    b = self.SendTrainingData()
    self.add_behaviour(b, template)
```

```
        template2 = create_template(str(self.jid), "agent_dt_bp@habro", None)
        template2.set_metadata("performative", "testing-data")
        a = self.SendTestData()
        self.add_behaviour(a, template2)

    def add_random_state(self, state):
        self.random_state = state


if __name__ == "__main__":
    environment = BpEnvir("data_dt_bp@hab0065", "Data")
    environment.add_random_state(randint(1, 1000))

    future = environment.start()
    future.wait() # wait for receiver agent to be prepared.

    agent = BpAgent("agent_dt_bp@hab0065", "Agent")
    agent.start()

    while environment.is_alive():
        try:
            sleep(1)
        except KeyboardInterrupt:
            agent.stop()
            environment.stop()
            break
    print("Agents finished")

    quit_spade()
```

Listing A.2: Soubor *main.py* s definicí tříd *agentů* zmiňován v *případové studii*

Bibliografie

1. KUBÍK, Aleš. *Inteligentní agenty*. 1. vydání. Brno: Computer Press, 2004. ISBN 978-802-5103-234.
2. *FIPA ACL Official Website* [online] [cit. 2023-02-13]. Dostupné z: `http://www.fipa.org/`.
3. RUSSELL, Stuart J.; NORVIG, Peter. *Artificial intelligence: a modern approach*. 3rd ed. Upper Saddle River: Prentice Hall, 2010. ISBN 978-0-13-604259-4.
4. LUGER, Georg F. *Artificial intelligence: structures and strategies for complex problem solving*. 6th ed. Boston: Pearson Education, 2008. ISBN 978-0-321-54589-3.
5. HONZÍK, Petr. *Honzik, P., 2006. Strojové učení, Brno* [online] [cit. 2023-03-07]. Dostupné z: `http://vision.uamt.feec.vutbr.cz/STU/others/Honzik%5C%20-%5C%20Strojove_uceni_S.pdf`.
6. *PyCharm: The Python IDE for Professional Developers* [online]. Online: JetBrains s.r.o., c2000-2023 [cit. 2023-04-28]. Dostupné z: `https://www.jetbrains.com/pycharm/`.
7. SZTURCOVÁ, Daniela. *OOT Objektově orientované technologie: Logická struktura systému (Diagram tříd)* [online]. Online: Institut geoinformatiky, HGF, 2015 [cit. 2023-04-28]. Dostupné z: `http://gisak.vsb.cz/wikivyuka/images/e/e3/OotxCD.pdf`.
8. SZTURCOVÁ, Daniela. *Objektově orientované technologie: Dynamický náhled Sekvenční diagram (Realizace UC)* [online]. Online: Institut geoinformatiky, HGF, 2015 [cit. 2023-04-28]. Dostupné z: `http://gisak.vsb.cz/wikivyuka/images/9/98/OotxSeD.pdf`.
9. SAWA, Zdeněk. *Seminář z Programování* [online]. Online, 2015 [cit. 2023-04-28]. Dostupné z: `https://www.cs.vsb.cz/sawa/spr/spr.pdf`.
10. JANČAR, Petr. *Teoretická informatika: učební text* [online]. Online, 2015 [cit. 2023-04-28]. Dostupné z: `https://www.cs.vsb.cz/sawa/ti/materialy/ti.pdf`.
11. *SPADE's Documentation: Welcome to SPADE's documentation!* [Online]. 2020 [cit. 2023-04-10]. Dostupné z: `https://spade-mas.readthedocs.io/en/latest/index.html`.

12. *Archiv ML datasetů: UC Irvine Machine Learning Repository* [online]. University of Massachusetts Amherst: University of Massachusetts Amherst, 2007 [cit. 2023-04-18]. Dostupné z: https://archive-beta.ics.uci.edu/.

13. FISHER, R. A.; FISHER, R.A. *Iris* [UCI Machine Learning Repository]. 1988. DOI: 10.24432/C56C76.

14. JANOSI, Andras; STEINBRUNN, William; PFISTERER, Matthias; DETRANO, Robert; M.D., M.D. *Heart Disease* [UCI Machine Learning Repository]. 1988. DOI: 10.24432/C52P4X.

15. AHA, David. *Tic-Tac-Toe Endgame* [UCI Machine Learning Repository]. 1991. DOI: 10.24432/C5688J.