VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

# MODUL PRO DOLOVÁNÍ Z DAT

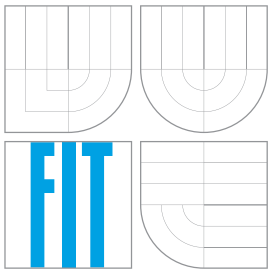
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

JIŘÍ PETRLÍK

BRNO 2009

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

# MODUL PRO DOLOVÁNÍ Z DAT

DATA MINING MODULE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — JIŘÍ PETRLÍK
AUTHOR

VEDOUCÍ PRÁCE — Doc. Ing. JAROSLAV ZENDULKA, CSc.
SUPERVISOR

BRNO 2009

# Abstrakt

Obsahem této bakálářské práce je seznámení s problematikou dolovaní z dat. Zaměřuji se především na problematiku klasifikace pomocí neuronových sítí. Proto zde popisuji některé základní algoritmy pro učení neuronových sítí. Hlavním cílem práce bylo vytvořit nový modul do systému pro dolování z dat, který je vyvíjen na FIT VUT v Brně. Tento systém zde stručně představuji a popisuji zde návrh jeho nového modulu. Výsledný modul jsem otestoval na cvičných datech.



# Abstract

The aim of the bachelor´s theses is to introduce the problematic of data mining. I have especially focused on the problematic of the classification with the help of neural networks. This is why I describe some basic algorithms for neural network teaching as well. The main goal of this work is to create a new module for the system of data mining. This system has been developed in the cooperation with other people from FIT VUT in Brno. I introduce this system here as well and I also describe the proposal for it´ s new module. I have already tested some training data with the final module.



# Klíčová slova

dolování z dat, získávání znalostí z databází,klasifikace, neuronové sítě, upstart algoritmus, pocket algoritmus, DMSL

# Keywords

data mining, knowledge discovery in databases,classification, neural network, upstart algorithm, pocket algorithm, DMSL


# Citace

Jiří Petrlík: MODUL PRO DOLOVÁNÍ Z DAT, bakalářská práce, Brno, FIT VUT v Brně, 2009

# MODUL PRO DOLOVÁNÍ Z DAT

## Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Doc. Ing. Jaroslava Zendulky, CSc. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Jiří Petrlík
20. května 2009


## Poděkování

Děkuji svému vedoucímu Doc. Ing. Jaroslavu Zendulkovi, CSc. za odborné vedení, cenné rady a podněty, které mi při řešení tohoto projektu poskytl.

# Obsah

# Kapitola 1

# Úvod

Tato práce se zabývá problematikou dolování dat z databází. Jedná se o poměrně širokou oblast, s jejímiž základy jsem se v průběhu psaní této práce seznámil. V této práci se snažím vysvětlit, co znamená „dolování z dat“, jaké jsou fáze tohoto procesu a stručně vysvětlit základní dolovací úlohy.

Podrobněji se zabývám úlohou klasifikace. Stručně vysvětluji zadání úlohy, co je jejím účelem a jaké má fáze. Dále také popisuji některé nejrozšířenější postupy, které se pro klasifikaci používají.

Hlavním úkolem této práce bylo rozšířit systém pro dolování z dat, který se vyvíjí na Fakultě informačních technologií o další modul pro dolování. Po dohodě s vedoucím práce jsem se rozhodl implementovat modul pro klasifikaci pomocí neuronových sítí.

Tato práce je rozčleněna do jedenácti kapitol. První kapitola obsahuje úvod a popisuje, co je možné v práci nalézt a co od ní očekávat. Ve druhé kapitole se zabývám problematikou dolování z dat. Ve třetí kapitole podrobněji popisuji úlohu klasifikace.

Následující čtvrtá kapitola je úvodem do problematiky umělých neuronových sítí. Píši zde o tom, jak lze simulovat umělý neuron a jak je možné tento neuron učit. Popisuji zde různé možnosti propojení těchto neuronů a v závěrečné kapitole se zaměřuji na to, jak lze využít neuronové sítě k řešení různých dolovacích úloh.

Pátá kapitola obsahuje popis systému pro dolování z dat, který je vyvíjen na FIT. Lze v ní nalézt informace o tom, jak tento systém používat a jak funguje uvnitř. Vysvětluji zde, co je to jazyk DMSL a jak je v tomto systému pro dolování dat z relačních databází využíván. Stěžejní části kapitoly je pak návod, jak lze rozšířit tento systém o nový dolovací modul.

V sedmé kapitole podrobně popisuji návrh mého nového modulu pro klasifikaci. Jsou zde uvedeny všechny potřebné algoritmy, které jsou nutné pro jeho implementaci. Následující osmá kapitola pak obsahuje popis samotné implementace modulu. Práce s modulem je popsána v deváté kapitole.

Předposlední desátá kapitola se zabývá testováním nově implementovaného modulu. Popisuje průběh klasifikace celkem pro tři pokusné soubory dat. V poslední jedenácté kapitole se pak snažím zhodnotit, jak se mi podařilo splnit úkoly ze zadání bakalářské práce.

# Kapitola 2

# Úvod do získávání znalostí z databází

Získávání znalostí z databázi je poměrně mladý obor, který se rozvíjí teprve několik posledních desetiletí. Jeho cílem je získávat nové znalosti z často velmi rozsáhlých databází. Znalostmi v tomto případě myslíme informace, které jsou nové, lze je rozumně využít praxi a k jejichž získání je nutné použít složitější postupy, než je například použití agregačních funkcí v databázi. Pro rozvoj tohoto oboru byl naprosto stěžejní bouřlivý vývoj v oblasti výpočetní techniky a především neustále se zvětšující objemy dat, které se ve výpočetních systémech uchovávají. Tato data jsou nejčastěji uchovávána ve velkých relačních databázích. Jedním z takových databázových systémů je právě také MySQL. Tento databázový systém využívá systém pro dolování z dat, který je vyvíjen na FIT. Dalším prostředkem pro uchovávání dat jsou datové sklady, které obsahují již integrovaná data a nabízejí velké množství nástrojů pro OLAP analýzu.

Proces získávání znalostí z databází má většinou několik fází. Ze všeho nejdříve je nutné získat data z kterých budeme dolovat. Tato data mohou pocházet z různých zdrojů, proto je nutné je nejprve integrovat. Často se také stává, že se musí přistoupit k tzv. „čištění dat“. Při tomto procesu se snažíme vypořádat například s chybějícími nebo odlehlými hodnotami. Chybějící hodnoty můžeme nahrazovat předem zvolenou konstantou, průměrnou hodnotou a podobně. Pro odstranění odlehlých hodnot musíme využít některou z vyhlazovacích technik. Například můžeme skutečné hodnoty nahradit hodnotami, které získáme proložením skutečných údajů regresní křivkou apod. Pro samotné dolování pak použijeme pouze ty hodnoty a vlastnosti, které mohou mít na výsledek zadané úlohy vliv. Může být také nutné data nějakým způsobem transformovat, aby na ně mohl být použit příslušný dolovací algoritmus. Například hodnoty kategorických atributů do kódu 1 z n a podobně.

Po těchto přípravných fázích již můžeme přistoupit k samotnému získávání znalostí. To v podstatě znamená, že využijeme některý z vhodných algoritmů pro řešení daného typu úlohy. Informace, které jsou výstupem příslušného algoritmu, je pak nutné umět správně interpretovat a v praxi využít. Bez této poslední fáze nemá celý proces získávání znalostí z databází téměř žádný význam. V následujících podkapitolách jsou stručně charakterizovány základní typy dolovacích úloh. Podrobnější informace je možné nalézt v literatuře, z které jsem čerpal [3] a [7].

## 2.1 Získávání asociačních pravidel

Pomocí této úlohy se snažíme získat nová pravidla pro některé vztahy v databázi. Tento typ pravidel lze výhodně využít pro sledování chování zákazníků. Například můžeme v databázi pojišťovací společnosti vysledovat, že pokud si zákazník sjedná povinné ručení pro své osobní auto a zároveň toto auto nebude starší, než určité datum, pak si s určitou pravděpodobností pořídí i havarijní pojištění. Asociační pravidlo může vypadat například takto:

*povinné ručení* $\wedge$ *nové auto* $\Rightarrow$ *havarijní pojištění (podpora=17%,spolehlivost=44%)*

U asociačního pravidla lze sledovat dva parametry a to podporu a spolehlivost. Podpora nám říká, jak často se toto asociační pravidlo vyskytuje ve vstupní množině dat. Tedy například pokud budeme zkoumat uzavřené pojišťovací transakce a každá pátá transakce bude splňovat podmínku levé i pravé strany pravidla, pak jeho podpora je 20%. Druhý parametr udává s jakou pravděpodobností bude pravdivý závěr v případě, že byly splněny podmínky. V našem případě to znamená s jakou pravděpodobností si zákazník objedná i havarijní pojištění v případě, že splnil podmínku levé strany pravidla.

## 2.2 Predikce

Předchozí úloha se snažila pouze popisovat data. Zadání této úlohy se od ní poněkud liší. Řekněme, že máme určitou množinu dat, například máme dlouhodobé záznamy o teplotě ze čtyř meteorologických stanic. Bohužel se stane, že jedna z těchto stanic přestane posílat data a my přesto potřebujeme alespoň přibližně určit, jaká je teplota v dané oblasti.

Nejprve musíme získat model, který je schopen tato data dopočítat. Tím může být například neuronová síť a podobně. Tento model vytvoříme na základě dat z dlouhodobých záznamů. Poté musíme ověřit na cvičných datech, s jakou přesností model pracuje. Pokud má model dostatečnou přesnost, tak ho můžeme začít používat.

## 2.3 Klasifikace

Tato úloha je v jistém smyslu podobná té předchozí. Celá tato práce se zabývá řešením právě této úlohy, proto je jí věnována celá další kapitola.

## 2.4 Shluková analýza

Cílem shlukové analýzy je rozdělit záznamy do skupin tzv. „shluků“, pro které platí, že záznamy v nich jsou si nějakým způsobem podobné. Zároveň přitom ale platí, že záznamy z různých shluků se od sebe nějakým výraznějším způsobem liší. Mezi algoritmy, které se pro shlukovou analýzu využívají, patří k-means, PAM, DBSCAN, DENCLUE a další.

# Kapitola 3

# Klasifikace

## 3.1 Popis úlohy

Při řešení této dolovací úlohy se snažíme roztřídit prvky vstupní množiny do dvou a více tříd. Pokud chceme tuto úlohu řešit, musíme mít nejprve k dispozici tzv. trénovací množinu, u jejíchž prvků předem známe, do které třídy patří. Tuto množinu použijeme pro naučení klasifikačního algoritmu. Čím kvalitnější a větší máme trénovací množinu, tím lze očekávat lepší výsledky při samotné klasifikaci. Poté, co systém pro klasifikaci zpracuje trénovací množinu, je třeba otestovat, zda-li byla dostatečná pro jeho naučení. K tomu potřebujeme další množinu vstupních dat a to tzv. testovací množinu. U této množiny také dopředu známe, do jakých tříd patří její prvky. Postup testování je takový, že si necháme od programu pro klasifikaci určit, do jaké třídy zadaný prvek patří, a poté porovnáme jeho výsledek se správnou hodnotou. Pokud je počet chybně určených prvků pro nás přijatelný a systém je dostatečně dobře naučen, můžeme ho začít používat. Při jeho používání mu předáváme prvky, u nichž neznáme, do které třídy patří.

## 3.2 Některé algoritmy pro klasifikaci

Existuje poměrně značné množství algoritmů pro klasifikaci. To, jaký použít, nejvíce závisí na typu vstupních dat, počtu tříd, do kterých chceme klasifikovat (především, zda chceme klasifikovat pouze do dvou, nebo do více) a množství zpracovávaných dat (zda je systém schopen tato data zpracovat v rozumně dlouhé době). Asi nejdůležitějším kritériem pro výběr metody je samozřejmě přesnost klasifikace.

Nejznámějším způsobem klasifikace je klasifikace pomocí rozhodovacího stromu. Tato metoda byla známa dávno před vynalezením počítačů. Uzly rozhodovacího stromu reprezentují jednotlivé atributy klasifikovaného prvku. Při klasifikaci začneme od kořene stromu. Vybereme větev, která odpovídá hodnotě atributu, který je v něm uveden. Na dalším uzlu provedeme další výběr a tak dále, dokud nenarazíme na list, který obsahuje odpověď na to, do které třídy máme prvek klasifikovat.

# Kapitola 4

# Neuronové sítě

Zajímavou možností, jak řešit klasifikační úlohu, je využití umělých neuronových sítí. Tyto sítě se snaží více, nebo méně přesně simulovat funkci biologických neuronů. Biologický neuron je tvořen z těla neuronu (soma) a dvou typů výběžků. Výběžky prvního typu jsou dostředivé a nazýváme je dendrity. Výběžky druhého typu jsou odstředivé a nazýváme je axony nebo též neurity. Rozlišujeme několik druhů biologických neuronů. Unipolární neurony mají pouze axon a vyskytují se u smyslů. Dalším typem jsou bipolární neurony, které mají jeden dendrit a jeden axon. Z našeho pohledu jsou nejdůležitější neurony multipolární, které mají velké množství dendritů a pouze jeden axon. Neuron přijímá vzruchy z ostatních neuronů pomocí dendritů, tyto vzruchy pak mění chemické prostředí uvnitř neuronu a ten pak buď reaguje vysláním vzruchu do axonu, nebo ne.

V této a další kapitole se pokusím nastínit některé principy, jak neuronové sítě pracují. Jedná se především o popis skutečností, které jsem využíval při návrhu nového modulu, nebo které s ním nějakým způsobem souvisí. Další informace o neuronových sítích je možné nalézt v [6].

## 4.1 Modely neuronů

Taková simulace, která by dokázala věrně napodobit skutečný neuron by byla příliš složitá, a proto se v praxi využívají některé zjednodušené modely. Jedním z takových modelů je perceptron.

### 4.1.1 Princip fungování umělého neuronu

Perceptron má libovolné množství vstupů a jeden výstup. Každému vstupu je přiřazena určitá váha. Právě změnami těchto vah se mění reakce perceptronu na vstupní podněty a učení perceptronu tedy znamená postupné měnění těchto vah až do žádoucího stavu.

Abychom si mohli něco říci o učení neuronu, musíme nejprve vědět, jak zjistit jeho výstup. Ten zjistíme podle vzorce:

$$y = f(\sum_{i=1}^{n} w_i x_i) \tag{4.1}$$

Kde $x_i$ je prvek ze vstupního vektoru a $w_i$ je příslušná váha. Konstanta $n$ pak udává počet prvků vstupního vektoru a $f$ je přenosová funkce. Tedy v podstatě vypočítáme všechny

součiny příslušných vstupů a vah. Ty sečteme a na výsledný součet aplikujeme přenosovou funkci.

Na začátku procesu učení nastavíme všechny váhy na náhodné hodnoty. Pak pro všechny prvky z trénovací množiny provádíme následující kroky. Vypočítáme výstup neuronu podle jeho současného nastavení. Podle rovnice 4.2. přepočítáme nastavení vah.

$$w_{i+1} = w_i + px_i(y - c) \tag{4.2}$$

Kde $w_i$ je opravovaná váha, $x_i$ je k ní příslušný prvek vstupního vektoru. Dále pak $y$ je výstup podle současného nastavení a $c$ je správný výstup. Tajemné $p$ je tzv. koeficient učení. Čím vyšší je tento koeficient, tím rychleji se síť učí. Ale zároveň také rychleji zapomíná, jak reagovat na předchozí vstupy.

Poté můžeme pokračovat v učení dalším prvkem, ale již s novým nastavením vah.

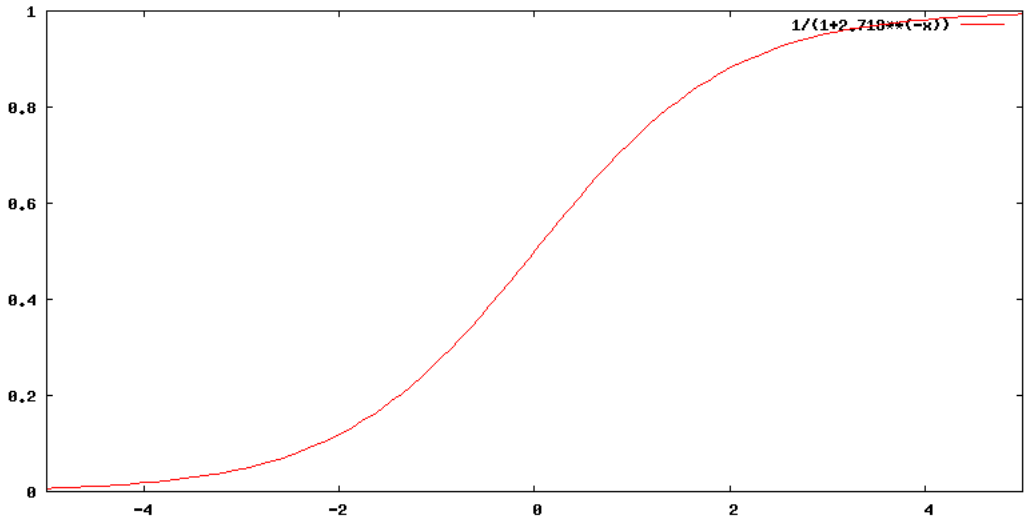
### 4.1.2 Přenosová funkce

Klíčovou záležitostí je volba přenosové funkce. Nejjednodušší funkcí, kterou můžeme zvolit je skoková funkce. Tato funkce nabývá pro hodnoty nižší než je zadaná konstanta $c$ hodnoty $a$ a pro hodnoty vyšší, nebo rovny této konstantě nabývá hodnoty $b$. Skoková funkce je tedy dána předpisem:

$$y = \begin{cases} a, & x < c \\ b, & x \geq c \end{cases} \tag{4.3}$$

Tato jednoduchá funkce nám nemusí vyhovovat a to především při učení neuronu. Další častou využívanou přenosovou funkcí je funkce sigmoidální. Pro tuto funkci platí, že je spojitá a její limita jdoucí k zápornému nekonečnu se rovná $a$ a limita jdoucí ke kladnému nekonečnu se rovná $b$. Příkladem sigmoidální funkce může být funkce na obrázku 4.1 zadaná následujícím předpisem:

$$y = \frac{1}{1 + e^{-x}} \tag{4.4}$$




Obrázek 4.1: Příklad průběhu sigmoidální funkce

Dalším příkladem může být funkce zadána takto:

$$y = \frac{tanh(x)}{2} + 0.5 \tag{4.5}$$

## 4.2 Pocket algoritmus

Běžný perceptron je schopen řešit pouze tzv. lineárně separovatelné problémy. Bohužel velmi často se vyskytují problémy podstatně složitější. V takovém případě je nutné používat algoritmy, které napomáhají tomu, aby se perceptron naučil co nejlépe i přesto, že není schopen sám zadaný problém vyřešit. Jedním z takových algoritmů, který se tímto zabývá je právě pocket algoritmus.

### 4.2.1 Princip činnosti

Princip činnosti algoritmu není příliš složitý:

1. Nastav nejdelší délku bezchybného běhu a délku aktuálního běhu na nulu.
2. Ulož do dočasné paměti aktuální nastavení perceptronu.
3. Postupně načítej náhodné vektory a prováděj jejich klasifikaci.
4. Pokud je vektor správně klasifikován inkrementuj délku aktuálního běhu a pokračuj v bodě 3.
5. Pokud byl vektor chybně klasifikován a pokud je aktuální délka běhu menší než délka nejdelšího běhu, pak přenastav perceptron podle parametrů v dočasné paměti.
6. Ulož do dočasné paměti aktuální nastavení perceptronu.
7. Proveď učení perceptronu aktuálním vektorem.
8. Nastav aktuální délku běhu na nulu a vrať se na bod 3.
9. Algoritmus končí ve chvíli, kdy jsme prošli podle našeho názoru dostatečné množství náhodných vektorů.

### 4.2.2 Zaostřovací pocket algoritmus (pocket algorithm with ratchet check)

Algoritmus popsaný výše má jednu nevýhodu a tou je jeho silná závislost na tom, jaké náhodné vektory budou vybírány. Rozhodujícím kriteriem pro to, jak dobře je perceptron naučený, je tzv. délka aktuálního běhu, která může být dobrým, nebo špatným náhodným výběrem vektorů značně ovlivněna.

Může se tak stát, že v hodnocení zvítězí horší nastavení perceptronu nad lepším. Toto se snaží řešit zaostřovací pocket algoritmus, který vypadá následovně:

1. Nastav počet správně klasifikovaných vektorů na nulu.
2. Postupně načítej náhodné vektory.
3. Ulož do dočasné paměti aktuální nastavení perceptronu.
4. Proveď učení perceptronu s náhodným vektorem.

5. Projdi celou trénovací množinu a spočítej správně klasifikované vektory.

6. Pokud je počet správně klasifikovaných vektorů menší než předchozí, přenastav perceptron na nastavení ve vyrovnávací paměti.

7. Pokud je počet správně klasifikovaných vektorů stejný, nebo vyšší než předchozí, ulož aktuální počet správně. klasifikovaných vektorů.

8. Dokud neproběhne dostatečný počet učení (určuje nejčastěji konstanta), vrať se na druhý bod.

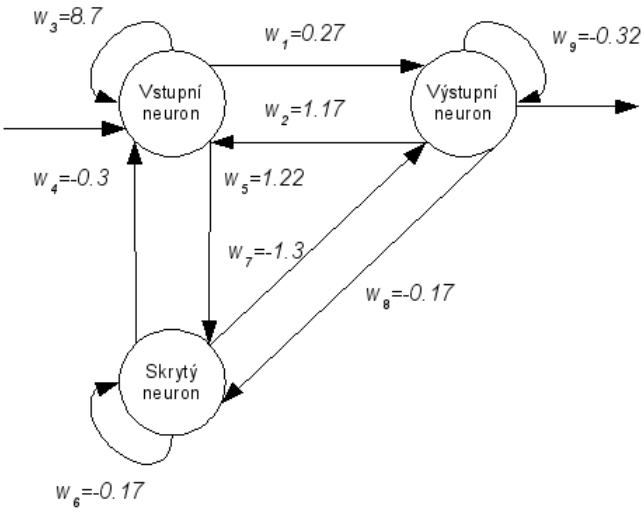
## 4.3 Topologie

Dalším klíčovým problémem při vytváření simulací neuronových sítí je po konstrukci neuronu způsob, jakým jsou tyto modely neuronů navzájem propojeny. Jako obvykle existuje více možností.

### 4.3.1 Plně propojené sítě

První možností, jak vytvořit neuronovou síť, je propojit každý neuron s každým a navíc vytvořit propojení mezi výstupem a vstupem u každého neuronu. V obecnější variantě mohou mít propojení v obou směrech různé váhy. U plně propojených symetrických sítí mají pak stejné váhy v obou směrech.

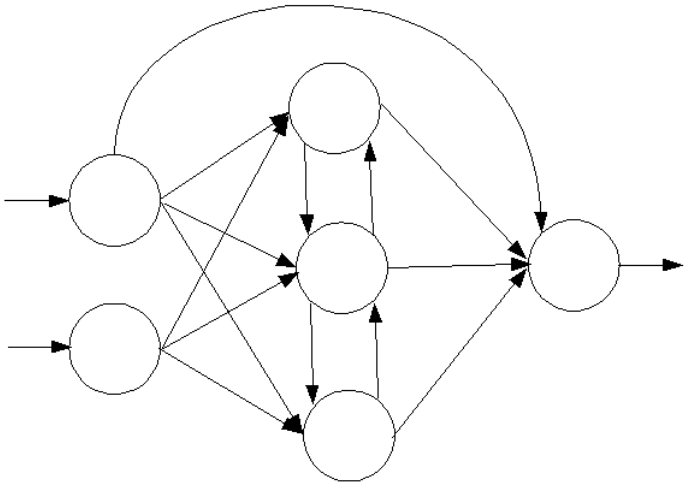
Plně propojené sítě nejsou příliš často používány. Jedním z důvodu je, že s rostoucím počtem neuronů velice rychle roste počet propojení a z toho plyne nutnost uchovávat příliš velké množství vah. Příkladem takovéto sítě může být síť na obrázku 4.2.




Obrázek 4.2: Ukázka plně propojené sítě

### 4.3.2 Sítě s vrstvami

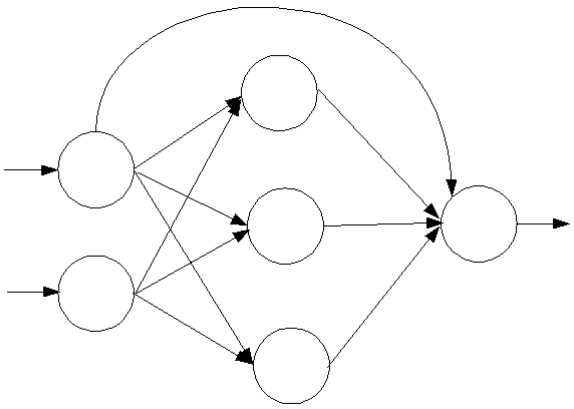
U těchto sítí jsou všechny neurony uspořádány do vrstev. Propojení mezi jednotlivými neurony se řídí pravidlem, které říká, že nelze vytvářet propojení k neuronům z předchozích vrstev. Lze ale vytvářet propojení mezi neurony v téže vrstvě. Ukázku takovéto sítě znázorňuje obrázek 4.3.



Obrázek 4.3: Ukázka sítě s vrstvami

### 4.3.3 Acyklické sítě

Doposud jsme se bavili o tzv. rekurentních sítích. U těchto sítí je výpočet výstupu komplikovaný. Acyklické sítě jsou sítě s vrstvami, pro které platí, že mezi neurony v téže vrstvě nesmí existovat žádná propojení. Tím se značně zjednodužší proces výpočtu. Příkladem této sítě může být síť na obrázku 4.4.



Obrázek 4.4: Ukázka acyklické sítě

### 4.3.4 Modulární sítě

Někdy může být zajímavé organizovat jednotlivé menší neuronové sítě do různých struktur. Takto lze řešit některé problémy, které by byly pomocí jedné sítě obtížně řešitelné, nebo příliš náročné na zdroje počítače. Moduly těchto sítí mohou být a často jsou, tvořeny pouze jednovrstevnými sítěmi.

## 4.4 Využití neuronových sítí

Neuronové sítě mají dosti široké množství využití, která se poměrně často kryjí s úlohami, které jsem vyjmenoval v předchozí kapitole. Úlohy řešitelné pomocí neuronových sítí mohou být v zásadě dvojího typu.

První typ úloh využívá tzv. učení s učitelem. To znamená, že můžeme v průběhu učení kontrolovat zda-li neuronová síť vyhodnotila danou situaci správně, nebo ne. Typickými příklady učení s učitelem jsou klasifikace a predikce, kde u trénovací množiny dopředu známe výsledky.

Druhý typ úloh využívá tzv. učení bez učitele. To znamená, že při učení dopředu nevíme nic o tom, jaký je jeho správný výsledek. Typickou úlohou, která využívá učení bez učitele, je shluková analýza.

### 4.4.1 Klasifikace

Úloha klasifikace byla podrobně popsána v předchozí kapitole. Základní myšlenkou, jak použít neuronové sítě ke klasifikaci, je na vstupní vrstvu sítě předat vstupní vektor s informacemi, které popisují klasifikovaný záznam. Výstupní vrstva pak obsahuje stejné množství neuronů, jako máme počet tříd, do kterých budeme klasifikovat.

### 4.4.2 Predikce

Při predikci se snažíme předvídat hodnotu spojité veličiny, například vlhkost vzduchu, cenu akcií a podobně. Obvykle máme v k dispozici záznam o vývoji těchto veličin z minulosti a chceme předvídat její vývoj z $n$ posledních hodnot. Při řešení této úlohy pomocí neuronových sítí postupujeme následovně. Opět budeme využívat učení s učitelem. Z naměřených hodnot vytvoříme vektory o $n+1$ prvcích, které obsahují bezprostředně za sebou následující hodnoty. Při učení pak na vstup neuronové sítě předáme $n$ prvních hodnot a výstup pak budeme srovnávat s poslední hodnotou vektoru.

Velmi výhodné je při predikci používat více veličin, které mohou s predikovanou veličinou nějakým způsobem souviset. Například při předpovídání vlhkosti vzduchu nás může zajímat teplota, tlak a podobně.

### 4.4.3 Shluková analýza

Toto je úloha, ve které na rozdíl od úloh předchozích používáme učení bez učitele. Známe totiž pouze polohu jednotlivých prvků, ze kterých budeme shluky utvářet, a nemusíme často znát ani počet těchto shluků.

Tuto úlohu lze řešit následujícím postupem. Vytvoříme několik náhodně nastavených neuronů. Postupně na jejich vstupy přivádíme hodnoty s polohou jednotlivých prvků. Pokud neuron reagoval na vstupní vektor vysokou hodnotou výstupu, nastavíme jeho parametry

tak, aby na tuto hodnotu reagoval ještě silněji. Pokud reagoval nízkým výstupem, přenastavíme ho tak, aby příště reagoval ještě nižším výstupem.

### 4.4.4 Další využití

Neuronové sítě mají mnoho dalších využití, která ani nemusí příliš souviset se získáváním znalostí z databází. Mohou například být výhodně použity pro aproximaci různých funkcí, rozpoznávání vzorů, optimalizaci a podobně.

# Kapitola 5

# Určení potřebného počtu neuronů

Častým problémem u neuronových sítí je určit, kolik neuronů je potřeba k řešení určitého problému. Může se totiž stát, že pokud zvolíme málo neuronů, tak výpočet bude sice rychlý, ale síť nebude možné naučit. Pokud naopak zvolíme příliš mnoho neuronů, výpočet může být zbytečně neefektivní a učení sítě také nebude optimální.

## 5.1 Sítě s proměnnou topologií

Bohužel zatím neexistuje a asi dlouho existovat nebude nějaký vzorec, nebo postup podle kterého by bylo možné spočítat, kolik neuronů budeme potřebovat. Naštěstí ale máme algoritmy, které umožňují přidávat neurony a vytvářet topologii sítě v průbehu učení.

### 5.1.1 Network pruning

Jednou z možností, jak se s tímto problémem vypořádat, je naučit velkou síť a pak postupně odebírat propojení a neurony do té doby, dokud je síť ještě schopna pracovat. Klíčovou otázkou pro tento algoritmus je která spojení, případně které neurony, je nejvhodnější odebrat.

Především můžeme odebrat spojení, která mají nízké váhy. Taktéž můžeme odebrat neurony, ke kterým jsou tato spojení s nízkými vahami připojena. Druhou možností je změnit váhu spojení, které chceme potenciálně odstranit, na nulu. Pokud se výrazně nezmění výstup sítě, pak můžeme toto spojení odstranit.

### 5.1.2 Algoritmus Marchand, Golea, Rujan

Druhým možným způsobem je vytvořit minimalistickou síť, do které poté můžeme postupně přidávat jednotlivé neurony a jejich propojení. Jedním ze zástupců algoritmů pro tvorbu takovýchto sítí je algoritmus Marchand, Golea, Rujan. Tento algoritmus vytváří třívrstvou síť, kde počet vstupních neuronů odpovídá počtu prvků ve vstupním vektoru. Výstupní vrstva pak obsahuje jeden neuron. Z toho tedy plyne, že tato síť je určena k rozdělování prvků do dvou tříd.

Pro učení sítě je důležitá skrytá vrstva neuronů. Tato vrstva je na počátku učení prázdná a neurony jsou do ní přidávány při učení sítě.

### 5.1.3 Neuronové stromy

S rozdílným přístupem k postupnému zvětšování sítě přicházejí tzv. neuronové stromy. Tyto sítě jsou zvláštní tím, že mezi neurony neexistují propojení jako u předchozích typů sítí. Jednotlivé neurony jsou totiž uspořádány do struktury, která je podobná rozhodovacímu stromu. Neuron zde funguje jako uzel rozhodovacího stromu, který pouze určuje do jaké větve se bude při zjišťování výsledku pokračovat.

### 5.1.4 Tiling algoritmus

Dalším algoritmem, který umožňuje postupný růst sítě je tiling algoritmus. Tento algoritmus vytváří acyklické dopředné sítě. V každé skryté vrstvě je jeden master neuron a potřebný počet dalších služebných neuronů.

## 5.2 Upstart algoritmus

Tomuto algoritmu bych se chtěl věnovat trochu podrobněji. Jedná se také o algoritmus, který řeší postupný růst neuronové sítě a pracuje následujícím způsobem:

1. Mějme množiny $T_0$ a $T_1$. Pro prvky množiny $T_0$ platí, že výstup pro ně by měl být $-1$ a pro prvky $T_1$ by měl být 1.

2. Nejprve natrénujeme neuron tak, aby od sebe dokázal rozlišit prvky množin $T_0$ a $T_1$.

3. Pokud i po naučení zařadí neuron některé prvky z množin $T_0$ a $T_1$ špatně, pak musíme rozdělit prvky do množin. $T_0^+$ a $T_0^-$, kde první z nich obsahuje správně určené prvky množiny $T_0$ a druhá pak prvky této množiny, které byly určeny špatně. Obdobně postupujeme pro množinu $T_1$.

4. Pomocí rekurzivního volání algoritmu upstart vytvoříme síť, která dokáže rozlišit mezi prvky množin $T_0$ a $T_1^-$ a druhou síť která dokáže rozlišit prvky $T_1$ a $T_0^-$.

5. Nyní potřebujeme vypočítat hodnotu váhy, se kterou budeme výstupy těchto dvou sítí připojovat na náš původní neuron. Tuto hodnou určíme tak, že pro každý prvek $T_0$ a $T_1$ vypočítáme hodnotu výrazu $\sum_{k=0}^{n-1} |w_k * x_k|$, kde $w$ je váha a $x$ je hodnota ze vstupního vektoru. Jako potřebnou hodnotu váhy zvolíme libovolnou hodnotu vyšší než byla nejvyšší hodnota předchozího výrazu. Takto získanou hodnotu váhy označme například $MAX$.

6. Připojíme náš neuron na výstup sítě, která odděluje prvky množin $T_1$ a $T_0^-$ s hodnotou váhy $-MAX$ a na výstup sítě oddělující $T_0$ od $T_1^-$ spojením s hodnotou váhy $MAX$.

### 5.2.1 Nevýhody algoritmu upstart

Přestože je tento algoritmus často používán, má některá omezení a nevýhody. Jeho hlavní nevýhodou je, že s jeho pomocí lze klasifikovat prvky pouze do dvou tříd.

Druhá nevýhoda plyne z jeho podstaty. Stejně jako u všech algoritmů, které postupně zvětšují neuronovou síť, tak i u tohoto může nastat situace, že problém, který řešíme, je příliš složitý. V takovém případě by mohla velikost sítě narážet na paměťové možnosti počítače.

# Kapitola 6

# Systém pro získávání znalostí z databází

Na naší fakultě je dlouhodobě vyvíjen systém pro získávání znalostí z databází. Tento systém je napsán v jazyce Java a pro práci s daty využívá relační databázi MySQL. Podstatnou výhodou tohoto systému je to, že má modulární architekturu. To umožňuje poměrně snadné přidávání nových modulů, které implementují nové dolovací algoritmy.

K popisu dolovacích úloh využívá tento systém jazyk DMSL, který je odvozen od jazyka XML. Jeho výhodou je také jeho snadná rozšiřitelnost o popis nových typů úloh a znalostí, které jsou jejich výstupem. Program je schopen ukládat projekty se získanými znalostmi v tomto jazyce do souboru a poté je zpětně načítat. Pro více informací o systému doporučuji následedující práce [5], [2] a [4].

## 6.1 Práce se systémem

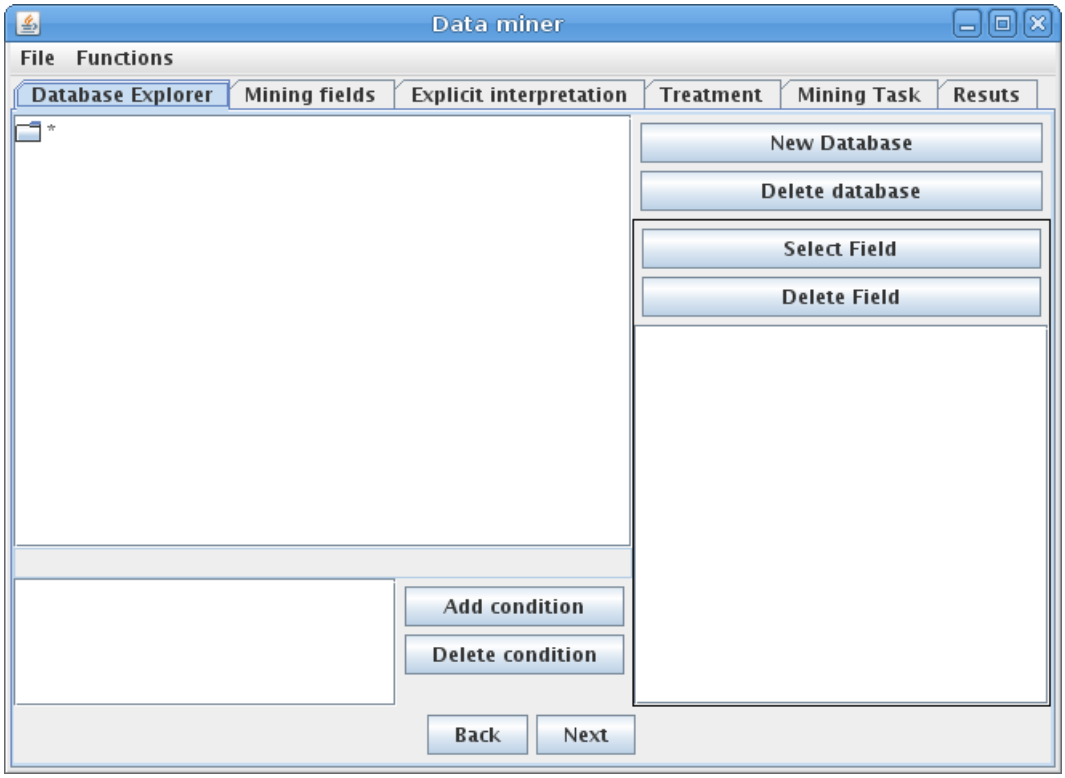
Vzhled programu si lze prohlédnout na obrázku 6.1. Celý program je rozdělen do záložek. V každé záložce je popsána určitá část dolovací úlohy. Uživatel tak postupně prochází od první záložky až do předposlední a tím zadá dolovací úlohu. Po stisku tlačítka „Mine result“ se dolovací úloha spustí. Uživatel je informován o průběhu úlohy a výsledky si může prohlédnou na poslední záložce programu.

### 6.1.1 Výběr dat pro dolování

Před celým tímto procesem je nutné správně nakonfigurovat připojení k databázi. To je možné udělat v dialogovém okně File → Options. Poté můžeme přejít na první záložku „Database Explorer“. Zde klepnutím na tlačítko „New database“ vybereme databázový server a můžeme pokračovat výběrem databázové tabulky a sloupců potřebných pro dolování.

### 6.1.2 Výběr dolovací úlohy

Na dalších záložkách můžeme nastavit parametry pro předzpracování dat. Samotný výběr a nastavení dolovací úlohy lze provést na záložce „Mining Task“. V menu „Task type“ vybereme typ úlohy. Každá úloha má svůj panel, který se zobrazí pod menu s výběrem úlohy. V tomto panelu je možné vyplnit doplňující parametry úlohy.




Obrázek 6.1: Program pro dolování z dat po spuštění

### 6.1.3 Prezentace získaných znalostí

Pokud dolovací algoritmus proběhne v pořádku, zobrazí se poslední záložka s výsledky. Získaný model se ukládá v rámci projektu v DMSL elementu Knowledge.

## 6.2 Hotové moduly

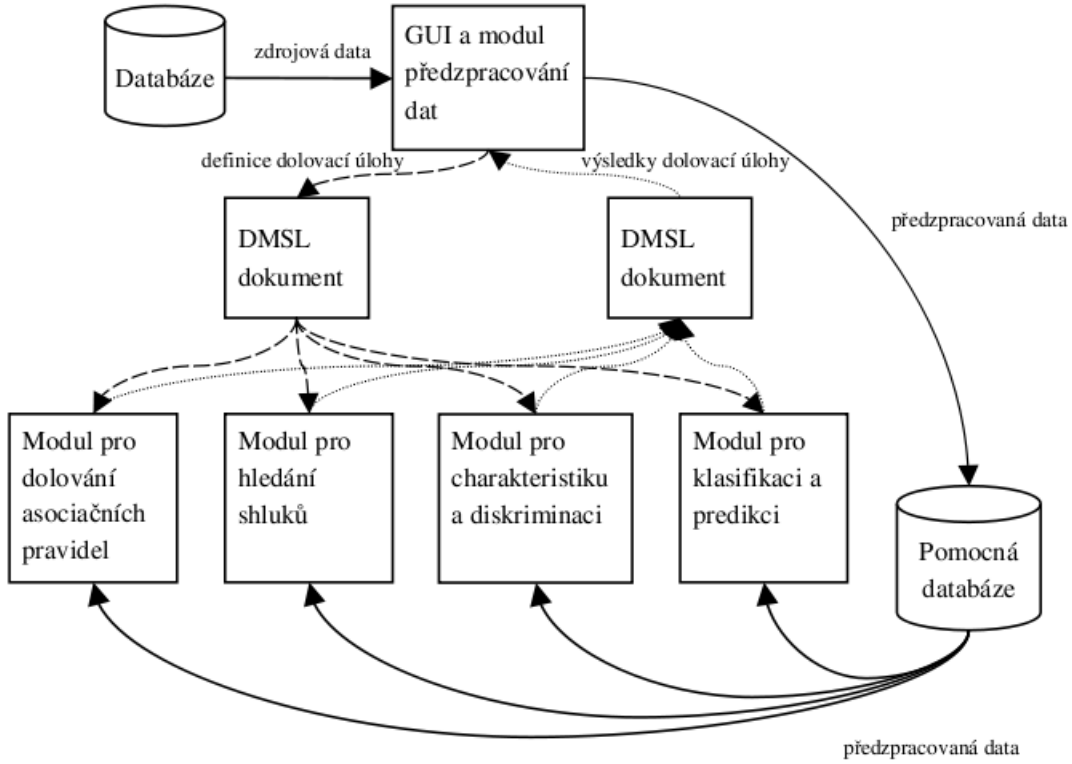
Do systému již bylo přidáno množství rozšiřujících modulů, které implementují rozličné algoritmy a slouží k řešení množství dolovacích úloh. Nyní jsou dostupné tyto moduly pro tyto úlohy:

1. Charakterizace a diskriminace
2. Asociační analýza
3. Shluková analýza
4. Klasifikace pomocí ID3 algoritmu
5. Klasifikace pomocí support vector machine

## 6.3 Popis systému z programátorského hlediska

### 6.3.1 Jak probíhá zpracování úlohy

Strukturu systému lze znázornit pomocí obrázku 6.2. Zde je vidět, že data z databáze jsou nejprve předzpracovány modulem pro předzpracování dat. Takto předzpracovaná data jsou uložena do pomocné tabulky v pomocné databázi.




Obrázek 6.2: Schéma práce systému - převzato z [5]

Do DMSL dokumentu s projektem se následně přidají informace o tom, jaká se má použít dolovací úloha a s jakými parametry. Takto získaný DMSL dokument se předá příslušnému modulu, který zadanou úlohu řeší.

Tento modul provede výpočet a vytvoří část DMSL dokumentu, která obsahuje získané znalosti. Tento DMSL dokument je nakonec zpracován a pomocí GUI jsou přehledně znázorněny jeho výsledky.

### 6.3.2 Jazyk DMSL

Jazyk DMSL slouží k popisu dolovací úlohy. Umožňuje uchovávat informace nejen o výběru a nastavení dolovacího algoritmu, ale i o tom, jak mají být data předzpracována. Základní struktura dokumentu v tomto jazyce je následující:

```
<!ELEMENT DMSL (Header?,(FunctionPool | DataModel | DataMiningModel
| DomainKnowledge | DataMiningTask | Knowledge)+)>
```

Element header lze využít k uchování obecných informací o úloze. To může být například název a podobně. Element FunctionPool obsahuje použité funkce. V elementu DataModel je uložen popis vstupních dat. DataMiningModel obsahuje informace o tom, jak mají vypadat data pro zpracování dolovacím algoritmem.

Element DataMiningTask pak obsahuje popis použitého dolovacího algoritmu a jeho parametrů. Získané znalosti se nakonec ukládají do elementu Knowledge.

## 6.4 Rozšíření systému o nový modul

Pokud chceme psát rozšiřující modul pro tento systém, musíme si uvědomit, že náš modul bude komunikovat s jádrem systému především pomocí jazyka DMSL. Je proto většinou nutné rozšířit tento jazyk o elementy, které umožní popis úlohy, kterou chceme řešit a znalostí, které bude náš modul získávat.

### 6.4.1 DMSL

Z hlediska rozšíření modulu nás budou zajímat elementy DataMiningTask a Knowledge. Do elementu DataMiningTask zapisujeme informace s podrobnostmi o dolovací úloze. Tuto značku lze libovolně rozšiřovat. Také značku Knowledge můžeme libovolně rozšiřovat. Tato značka obsahuje informace získané pomocí dolovací úlohy.

### 6.4.2 Třídy s implementací nového modulu

Pokud chceme implementovat nový modul, musíme napsat třídu, která implementuje rozhraní MiningTaskGui. V tomto rozhraní jsou z hlediska implementace nového modulu nejdůležitější metody *getDefinitionCard* a *getResultCard*. První z nich se stará o vykreslení panelu, kde je možné zadat doplňující informace pro dolovací algoritmus. Druhá pak vykresluje panel s výsledky dolovací úlohy, tedy se získanými znalostmi.

Další důležité metody tohoto rozhraní jsou *LoadDefinitionFromDmsl* a *SaveDefinitionToDmsl*, které mají za úkol načítat a vytvářet podstrom DataMiningTask. O podstrom Knowledge se pak starají metody *SaveResultToDmsl* a *LoadResultFromDmsl*.

Druhá třída, kterou musíme implementovat, je třída, která bude implementovat rozhraní MiningTask. U tohoto rozhraní je nejdůležitější metodou metoda *RunMiningTask*, která by měla implementovat samotný dolovací algoritmus.

# Kapitola 7

# Návrh nového modulu pro klasifikaci

Cílem mé bakalářské práce bylo především navrhnout a implementovat nový rozšiřující modul pro dolování z dat. Po dohodě s vedoucím bakalářské práce jsem se rozhodl řešit problém klasifikace do libovolného množství tříd pomocí neuronových sítí. Nejprve jsem chtěl implementovat neuronové sítě typu backpropagation.

Tyto sítě mají od začátku pevně zadanou topologii. To znamená, že nelze přidávat ani ubírat neurony za běhu a tedy je nutné dopředu odhadnout, jak velkou síť budeme k řešení daného problému potřebovat.

Po dohodě s vedoucím práce jsem se rozhodl od implementace těchto sítí ustoupit. Důvodem bylo, že by nebylo vhodné od běžného uživatele systému požadovat, aby navrhoval topologii sítě. Musel by totiž manuálně zadávat počty neuronů v jednotlivých skrytých vrstvách sítě.

Jako náhradní řešení, které nevyžaduje od uživatele jakékoliv znalosti o neuronových sítích, jsem zvolil algoritmus, který vytváří rozhodovací strom, jehož jednotlivé uzly jsou tvořeny jednovrstevnými neuronovými sítěmi.

U tohoto algoritmu musí uživatel zadávat pouze maximální hloubku stromu a jaké množství chyb je ochoten tolerovat v případě, že uzel nedokázal problém rozdělit zcela správně. První parametr je nutný zejména proto, aby nedošlo v případě složitého problému k zahlcení veškerých paměťových zdrojů počítače.

## 7.1 Implementace jednovrstevné neuronové sítě

### 7.1.1 Popis činnosti jednovrstevné sítě

Algoritmus, který používám, využívá pro rozhodování v jednotlivých uzlech jednovrstevné neuronové sítě. Pro tyto sítě platí, že jednotlivé neurony nejsou mezi sebou nijak propojeny a navzájem se tedy neovlivňují.

Tato neuronová síť obsahuje tolik neuronů, do kolika tříd chceme vstupní vektory klasifikovat. Na každý neuron je připojeno tolik vstupů, kolik obsahuje vstupní vektor prvků.

Pokud chceme klasifikovat vektor připojíme ho na vstupy všech neuronů a vypočítáme jejich výstup. Neuron, který odpovídá třídě, do které zadaný vektor patří, by měl aktivovat svůj výstup. Tím se určí do které třídy tento vektor patří.

Může se stát a často se i bohužel stává, že se najednou aktivuje více neuronů. To je samozřejmě problém, protože zadání úlohy předpokládá, že každý vstupní vektor patří do

právě jedné klasifikační třídy.

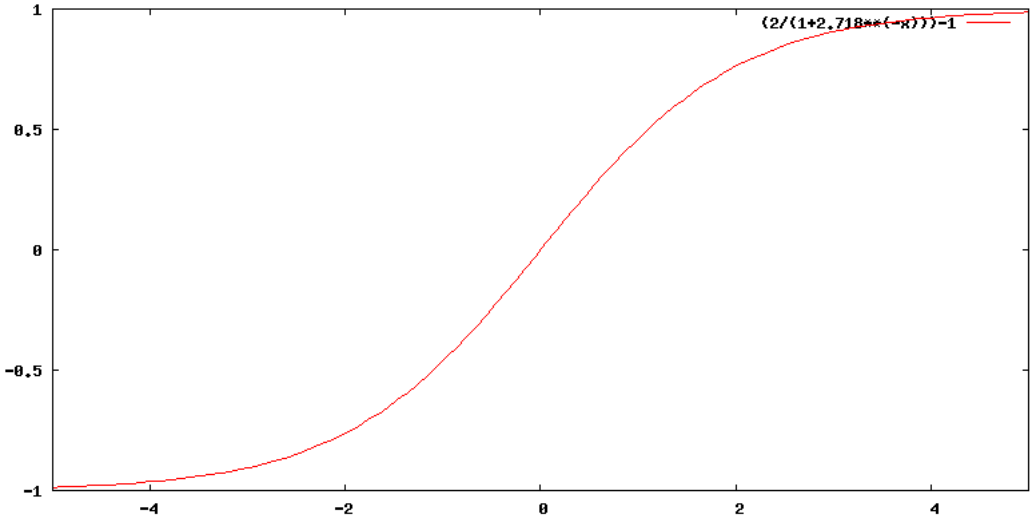
Tento problém lze řešit více způsoby. Jedním z přístupů je považovat za správnou tu třídu, u které byl výstup neuronu nejsilnější. Druhým možným způsobem je vybrat třídu posledního neuronu, který kladně reagoval. Tento přístup využívám i já ve svém modulu.

### 7.1.2 Učení jednovrstevné sítě

Jednovrstevná síť je složena z jednoduchých perceptronů. Při jejich samotném použití používám jako přenosovou funkci funkci skokovou, která byla popsána ve třetí kapitole. U této funkce jsem zvolil $c = 0$. Pro použití perceptronu je tato funkce naprosto dostačující.

Naopak při učení neposkytuje informaci o tom, jak moc byl neuron vybuzen, a posléze nelze bohužel ani zjistit, jak velké chyby se tento neuron dopustil. Mnohem vhodnější je tedy při učení neuronu použít sigmoidální funkci. Průběh této funkce je znázorněn na obrázku 7.1. a má následující tvar:

$$y = \frac{2}{1 + e^{-x}} - 1 \tag{7.1}$$




Obrázek 7.1: Přenosová funkce používaná při učení perceptronů v mém modulu

Před samotným učením je ještě vhodné provést předzpracování vstupních dat tak, aby se všechny prvky vektorů pohybovaly v intervalu $< -1, 1 >$.

Nyní je již možné popsat učící algoritmus této sítě. Podrobnosti o učení perceptronu jsem popsal v teoretické části v kapitole 3. Následující algoritmus je modifikací pocket algoritmu:

Pro všechny prvky trénovací množiny proveď následující kroky:

1. Ulož nastavení všech neuronů.
2. Zjisti u všech neuronů hodnotu hodnotící funkce a jejich hodnoty ulož.
3. Na daném prvku proveď učení všech neuronů v síti.

4. Po učení opět zjisti hodnotu hodnotící funkce u všech neuronů.

5. Pokud je u některých neuronů hodnota nově získané hodnotící funkce menší než předchozí, vrať jejich nastavení do stavu před učením z kroku 3.

V předchozím popisu se objevil termín hodnotící funkce. Tato funkce má za úkol ohodnotit, jak dobře je dotyčný neuron nastaven. Tuto funkci spočítáme následovně:

1. Spočítej pro každý neuron počet prvků, na které by měl pozitivně reagovat $P_n$ a počet prvků na které správně pozitivně reaguje $P_n^+$

2. Dále pro každý neuron spočítej počet prvků, na které by měl negativně $N_n$ reagovat a počet prvků na které správně negativně reaguje $N_n^-$.

3. Výslednou hodnotu funkce pak získáme jako hodnotu výrazu:

$$y = \frac{P_n^+}{P_n} . \frac{N_n^+}{N_n} \tag{7.2}$$

## 7.2 Budování rozhodovacího stromu

### 7.2.1 Omezení jednovrstevné sítě

Hlavní nevýhodou jednovrstevných sítí je to, že se dají použít pouze na řešení velmi jednoduchých problémů. Pokud chceme vyřešit problém pomocí jednoho perceptronu, musí být tento problém tzv. lineárně separovatelný. Takové problémy se však v praxi nevyskytují příliš často.

### 7.2.2 Rozhodovací strom

Řešením problému může být vytváření struktury velmi blízké rozhodovacímu stromu, kde každý uzel obsahuje jednovrstevnou neuronovou síť, která řeší pouze část problému a říká kterou větví má hledání výsledné třídy pokračovat. Následující postupy jsou inspirovány tímto článkem [1].

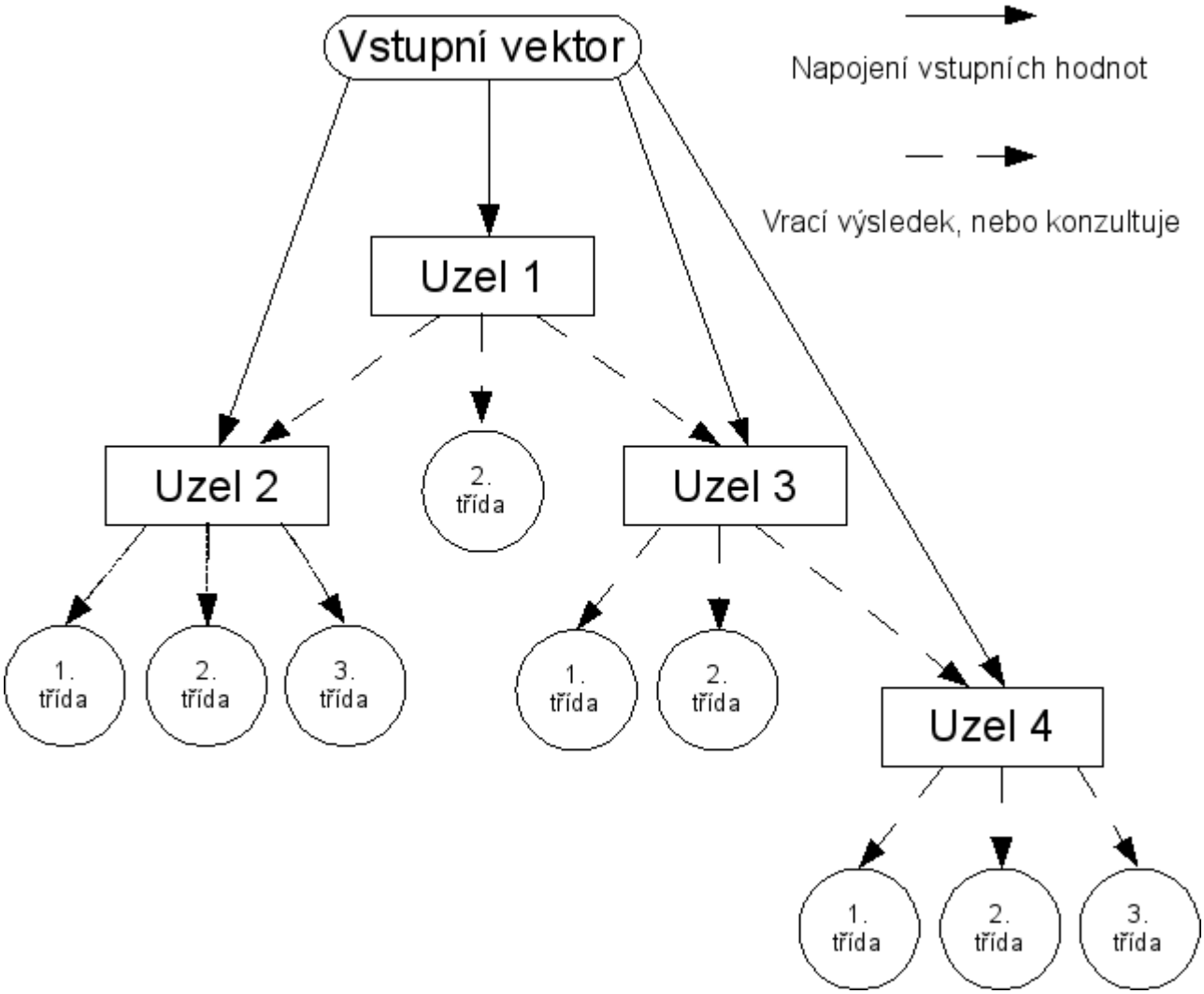
Příklad takového stromu je znázorněn na obrázku 7.2. Zde je vidět, že první uzel dokáže správně rozeznat pouze prvky, které vyhodnotí tak, že patří do druhé třídy. Pokud vyhodnotí, že prvek patří do první třídy, pak se musí na výsledek přeptat dalšího uzlu. Tento vztah je na obrázku znázorněn přerušovanou čarou.

V našem případě se jedná o uzel číslo 2. Tento uzel můžeme označit jako „konzultanta“ pro uzel číslo 1. Uzel číslo dva také obsahuje jednovrstevnou síť a tak pro vyhodnocení výsledku musíme na jeho vstup předat vstupní vektor (což je znázorněno plnou šipkou). Z obrázku je vidět, že tento uzel již nepotřebuje k vyhodnocení správného výsledku dalšího konzultanta.

### 7.2.3 Výstavba a použití stromu

Postup pro výstavbu tohoto stromu je dán následujícím algoritmem:

1. Vytvoř jednovrstevnou síť a podle postupu uvedeného dříve v této kapitole se jí pokus naučit na trénovací množině.




Obrázek 7.2: Ukázka vybudovaného stromu

2. Proveď kontrolu správnosti naučení celé sítě a pokud existuje neuron, který do své třídy klasifikuje větší množství chybných prvků než je nastavený povolený práh, pokračuj dalším bodem, jinak skonči.

3. Spusť nad těmito neurony rekurzivně tento algoritmus a jako trénovací množinu jim předej prvky, které byly klasifikovány do jejich třídy.

4. Získané podstromy připoj jako konzultanty pro dané neurony.

Tímto způsobem je možné získat rozhodovací strom, který již bude schopen řešit i poměrně rozsáhlé problémy. Jeho použití je už velice snadné a na rozdíl od jeho získání i nepoměrně rychlejší. Postupujeme následovně:

1. Vezmeme kořen rozhodovacího stromu a spočítáme pomocí jeho sítě, do které třídy patří klasifikovaný prvek.

2. Podíváme se, zda-li neuron, odpovídající třídě, do které jsme klasifikovali prvek, má konzultanta. Pokud ne, vrátíme třídu jako třídu pro zadaný prvek

3. Pokud má odpovídající neuron konzultanta, zavoláme pro tohoto konzultanta rekurzivně tento algoritmus a vrátíme jeho výstup.

# Kapitola 8

# Implementace modulu

V této kapitole bych chtěl stručně popsat hlavní třídy modulu, které provádějí výše popsaný algoritmus. Tyto třídy se také snažím znázornit na digramu 8.1. Dále pak popíši rozšíření jazyka DMSL, které jsem vytvořil za účelem reprezentace získaných znalostí.

## 8.1 Popis základních tříd modulu

### 8.1.1 Třída UpstartNet

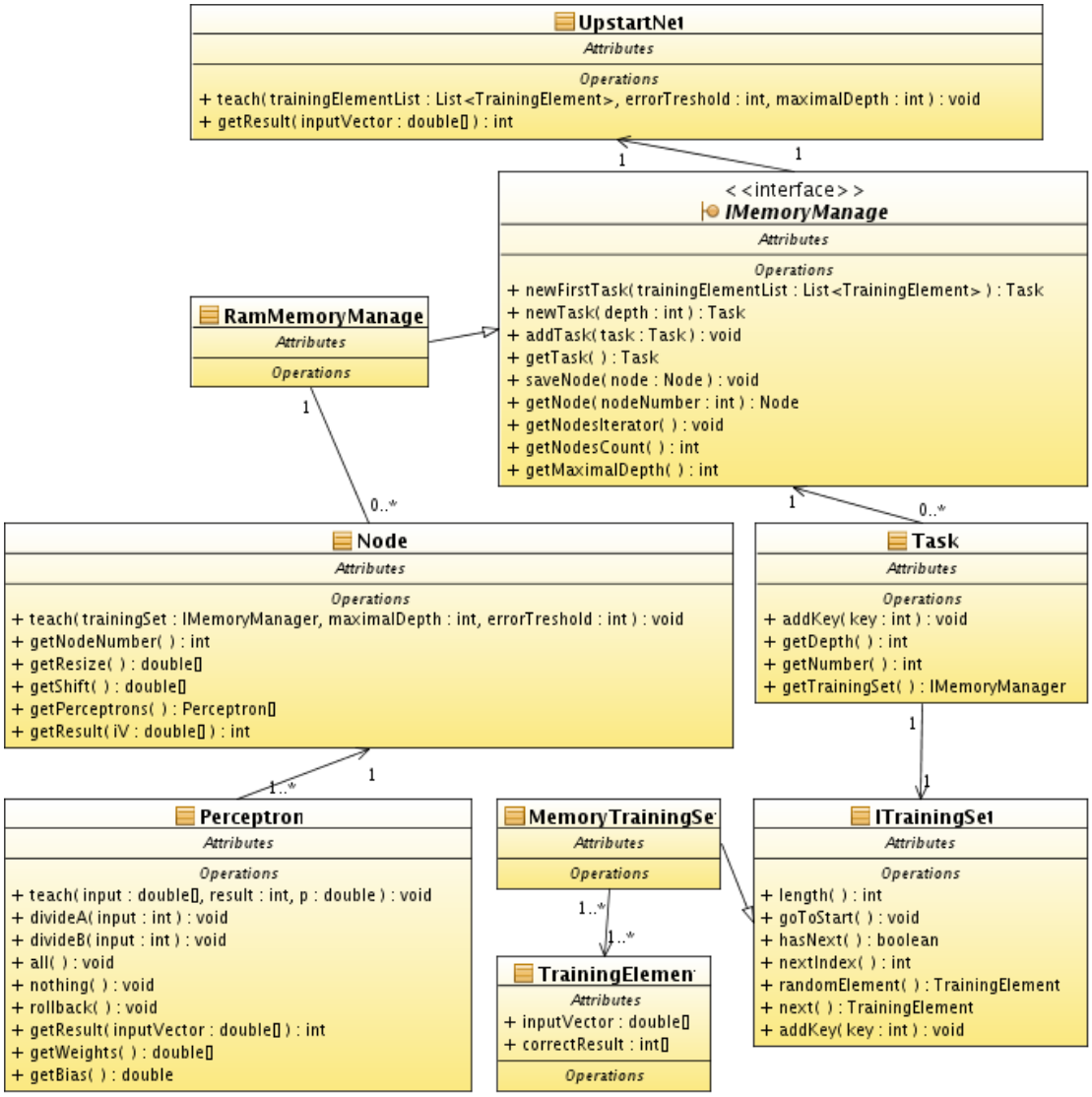
Tato třída je hlavním obalem reprezentujícím celý strom neuronových sítí a umožňuje práci s ním jako s celkem. Umožňuje vytvářet a učit síť. Zároveň také samozřejmě umožňuje její použití. Její veřejné metody jsou následující:

- **Konstruktor UpstartNet** - pro něho je nutné zadat počet vstupů sítě a počet tříd pro klasifikaci.
- **Metoda teach** - slouží k učení, jako parametr se jí předává seznam trénovacích elementů a hodnoty maximální hloubky stromu a množství tolerovatelných chyb.
- **Metoda getResult** - získá výsledek u naučené sítě, jako vstup se použije vstupní vektor prvku a výstupem je číslo od 0 do $n - 1$, kde $n$ je počet tříd, do kterých je možné klasifikovat.

### 8.1.2 Třída Node

Reprezentuje uzel rozhodovacího stromu. V každém uzlu je obsažena jednovrstevná neuronová síť. Z toho plyne, že tato třída vždy obsahuje pole s referencemi na objekty třídy perceptron. Její veřejné metody jsou následující:

- **Metoda teach** - Tato metoda se postará o nejlepší možné naučení jednovrstevné neuronové sítě pro daný uzel. Jako vstup použijeme třídu implementující rozhraní ITrainingSet, která obsahuje prvky z trénovací množiny, které byly klasifikovány předchozími uzly tak, že mají být vyhodnoceny tímto uzlem.
- **Metoda getResult** - Nechá síť daného uzlu vyhodnotit, do které třídy patří zadaný vstupní vektor. Výsledek může být dvojího typu. Buď je třeba pro zjištění třídy vyhodnotit další uzel, pak tato metoda vrací číslo tohoto uzlu. Pokud je již tento uzel schopen vrátit číslo třídy, pak ho vrátí vynásobené hodnotou $-1$.




Obrázek 8.1: Diagram tříd

Z toho plyne, že pokud je výsledkem kladné číslo, je třeba dále konzultovat s uzlem s tímto číslem. Pokud je výsledkem nekladné číslo, pak se jedná o výsledek klasifikace vynásobený hodnotou $-1$.

- **Metoda getNodeNumber** - Vrací číslo uzlu.
- **Metody vracející nastavení uzlu** - Metody getPerceptrons, getConsultants, getResize a getShift vrací údaje, které charakterizují uzel. Tyto údaje potřebujeme například pokud chceme uložit naučný uzel do XML.

### 8.1.3 Třída Perceptron

Její objekty vytváří modely jednotlivých neuronů. Poskytuje tyto metody:

- **Metoda teach** - Provede učení perceptronu na zadaném prvku trénovací množiny.

- **Metoda rollback** - Tato metoda vrátí nastavení perceptronu do stavu před jeho posledním učením.
- **Metoda divideA** - Tato metoda provede umělé nastavení neuronu. Jejím parametrem je číslo vstupu neuronu, který bude po provedení této metody rozhodujícím pro výstup perceptronu. Pokud bude hodnota tohoto na tomto vstupu kladná, bude perceptron reagovat pozitivně a naopak.
- **Metoda divideB** - Tato metoda funguje podobně jako předchozí, jenom s tím rozdílem, že perceptron bude reagovat kladně v případě, že se na příslušném vstupu perceptronu objeví záporné číslo.
- **Metoda all** - Provede umělé nastavení neuronu a to tak, že tento neuron reaguje vždy pozitivně.
- **Metoda nothing** - Provede umělé nastavení neuronu a to tak, že tento neuron reaguje vždy negativně.
- **Metoda getResult** - Vrátí hodnotu $-1$, nebo 1 podle naučení neuronu.

### 8.1.4 Třída MemoryTrainingSet (implementuje ITrainingSet)

Slouží k uchovávání jednotlivých prvků trénovací množiny. Nejdůležitější vlastností objektů této třídy je to, že mohou sdílet jeden seznam s prvky původní trénovací množiny a samy obsahují pouze čísla prvků, které jsou prvky jejich trénovací množiny. Umožňuje tak vytvářet jakýsi výběr prvků z trénovací množiny, aniž by docházelo k jejich duplikování.

Díky tomu dochází k značné úspoře místa v paměti, protože vektory trénovací množiny se uchovávají pouze jednou. Nejdůležitější metody jsou tyto:

- **Konstruktor**, který přijímá jako parametr referenci na seznam s prvky celé trénovací množiny
- **Metoda length** - Vrací počet prvků trénovací množiny.
- **Metoda addKey** - Přidá vektor se zadaným číslem do výběru. Po vytvoření instance konstruktorem jsou ve výběru všechny prvky. Pro vyprázdnění takto inicializovaného výběru použijeme tuto metodu s parametrem $-1$.
- **Metoda goToStart** - Nastaví iterování na začátek trénovací množiny.
- **Metoda hasNext** - Při iterování přes trénovací množinu informuje o tom, zda-li je dostupný další prvek, nebo množina již skončila.
- **Metoda next** - Vrací další prvek trénovací množiny při iterování.
- **Metoda nextIndex** - Vrací index prvku následujícího při iterování v seznamu s původní trénovací množinou.

### 8.1.5 Třída Task

Pokud se zamyslíme nad principem budování rozhodovacího stromu podle výše popsaného algoritmu zjistíme, že musíme uchovávat informace o tom, které podstromy je ještě potřebné dobudovat a na jakých prvcích trénovací množiny začneme trénovat jejich kořenové uzly. Pro uchovávání těchto informací o podstromu, který je nutné dotvořit, používám tuto třídu.

- **getNumber** - Vrací číslo úlohy. Zpracování úlohy vlastně znamená vytvoření nového uzlu podle vektorů z trénovací množiny. Nově vytvořený uzel pak bude mít stejné číslo jako měla jeho úloha.
- **getDepth** - Vrátí hloubku, na které se bude nacházet uzel vytvořený touto úlohu. Tuto informaci je nutné uchovávat, aby bylo možné v případě přílišné hloubky stromu úlohu přerušit.
- **addKey** - Přidá číslo prvku z hlavní trénovací množiny do trénovací množiny pro tuto úlohu.
- **getTrainingSet** - Vrací množinu trénovacích prvků pro tuto úlohu.

### 8.1.6 Třída RamMemoryManager (implementuje IMemoryManager)

Při postupném sestavování rozhodovacího stromu je nutné neustále udržovat velké množství informací o tom, jaké vektory byly klasifikovány do které třídy, abychom mohli vytvořit uzel, který tyto prvky bude dále dělit.

Zpočátku jsem se domníval, že udržované množství takovýchto informací by mohlo narážet na paměťové možnosti počítače. Chtěl jsem umožnit implementaci různých možností uchování těchto informací. Z tohoto důvodu jsem vytvořil rozhraní IMemeryManager, které umožní s těmito informacemi pracovat, aniž by bylo dopředu známé jakým způsobem budou uchovávány. Třídy implementující toto rozhraní musí podporovat především následující metody:

- **Metoda newTask** - Vrací nově vytvořený objekt třídy Task, ten zatím neobsahuje informaci o trénovací množině.
- **Metoda addTask** - Uloží objekt třídy Task do memory manageru, který ho zařadí do fronty k budoucímu zpracování.
- **Metoda getTask** - Získá první objekt třídy Task z fronty pro zpracování.
- **Metoda getNode** - Vytvoří uzel zadaného čísla.
- **Metoda saveNode** - Uloží naučený uzel.
- **Metoda getNodesIterator** - Vrátí iterátor, který prochází postupně všechny uzly.
- **Metoda getNodesCount** - Vrací počet uložených uzlů.
- **Metoda getMaximalDepth** - Zjistí maximální hloubku stromu.

## 8.2 Rozšíření jazyka DMSL

Pro uchovávání zadání a výsledků dolovací úlohy pro můj modul jsem musel rozšířit jazyk DMSL o některé nové elementy. Je nutné udržovat informace o zadání úlohy (tyto informace jsou uchovávány v DMSL elementu *DataMiningTask*) a o výsledcích dolování (v DMSL elementu *Knowledge*).

### 8.2.1 Rozšíření elementu DataMiningTask

V elementu *DataMiningTask* jsou uchovávána data, která určují použitý klasifikační algoritmus a jeho nastavení. Tento element má následující tvar:

```
<!ELEMENT DataMiningTask ANY>
<!ATTLIST DataMiningTask
    name                    CDATA                    #REQUIRED
    type                    CDATA                    #IMPLIED
    dataMiningModelRef      CDATA                    #IMPLIED>
```

Rozhraní pro práci s klasifikační úlohou již bylo dříve v systému vytvořeno. Element pro popis klasifikační dolovací úlohy se nazývá *MineClassification* a je definován takto:

```
<!ELEMENT MineClassification>
<!ATTLIST MineClassification
    dataMiningMatrixRef     CDATA                    #REQUIRED
    modelName               CDATA                    #REQUIRED
    classifAttribute        CDATA                    #REQUIRED>
```

V atributu *dataMiningMatrixRef* je odkaz na dolovací matici, v atributu *modelName* je identifikován typ modelu, který chceme vytvořit a v *classAttribute* se uchovává informace o atributu, podle kterého se bude klasifikovat. Tento element bylo nutné rozšířit, a to následujícím způsobem:

```
<!ELEMENT NeuralNetworkTask>
<!ATTLIST NeuralNetworkTask
    maximalDepth            CDATA                    #REQUIRED
    errorTreshold           CDATA                    #REQUIRED>
```

Atribut *maximalDepth* udává maximální povolenou hloubku stromu a atribut *errorTreshold* udává počet tolerovatelných chyb při klasifikaci v jednom uzlu do jedné třídy.

### 8.2.2 Rozšíření elementu Knowledge

K uchovávání získaných znalostí se v DMSL používá element *Knowledge*. Ten je definován následovně:

```
<!ELEMENT Knowledge ANY>
<!ATTLIST Knowledge
    name                    CDATA                    #REQUIRED
    type                    CDATA                    #IMPLIED
    dataMiningTaskRef       CDATA                    #IMPLIED>
```

V atributu *name* se uchovává název znalosti. V elementu *type* typ znalosti a v *dataMiningTaskRef* název použité dolovací úlohy. Výsledná síť bude reprezentována v jazyce DMSL následujícím způsobem. Kořenový element je popsán takto:

```
<!ELEMENT NeuralNetwork (Node+)>
<!ATTLIST NeuralNetwork
    inputLength                 CDATA                           #REQUIRED
    outputLength                CDATA                           #REQUIRED
    depth                       CDATA                           #REQUIRED
    nodes                       CDATA                           #REQUIRED>
```

Atributy *inputLength* a *outputLength* uchovávají informace o počtu vstupů a počtu tříd, do kterých je možné klasifikovat. Do atributu *depth* se pak ukládá hloubka stromu a do atributu *nodes* počet uzlů stromu. Pro uchovávání informací o jednotlivých uzlech se používá následující XML element:

```
<!ELEMENT Node (Dimension+,Neuron+)>
<!ATTLIST Node
    number                      CDATA                           #REQUIRED
    depth                       CDATA                           #REQUIRED>
```

Číslo uzlu se uchovává v atributu *number* a hloubka jeho zanoření ve stromu v atributu *depth*. Informace pro předzpracování dat do požadovaného intervalu má na starosti tato značka:

```
<!ELEMENT Dimension>
<!ATTLIST Dimension
    number                      CDATA                           #REQUIRED
    resize                      CDATA                           #REQUIRED
    shift                       CDATA                           #REQUIRED>
```

Atribut *number* udává číslo prvku ve vstupním vektoru, pro který parametry platí. Atributy *resize* a *shift* pak obsahují potřebné parametry tohoto předzpracování. Další element slouží k uložení samotného neuronu:

```
<!ELEMENT Neuron (Con+)>
<!ATTLIST Neuron
    number                      CDATA                           #REQUIRED
    bias                        CDATA                           #REQUIRED
    consultant                  CDATA                           #REQUIRED>
```

Pořadí neuronu v jednovrstevné síti se uchovává v atributu *number*. Nastavení biasu pak v atributu *bias* a číslo případného konzultanta pro třídu odpovídající příslušnému neuronu v *consultant*. Jednotlivé váhy neuronu se zapisují v XML značce:

```
<!ELEMENT Con>
<!ATTLIST Con
    from                        CDATA                           #REQUIRED
    weight                      CDATA                           #REQUIRED>
```

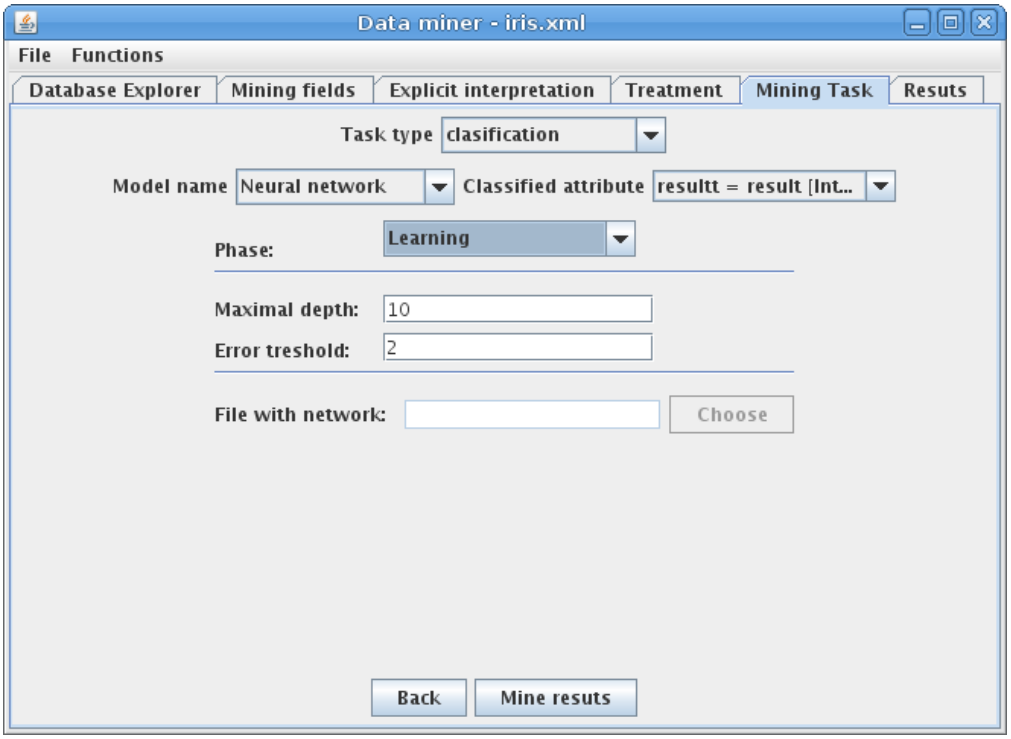
V tomto elementu atribut *from* udává, pro který prvek vstupního vektoru tato váha platí. V atributu *weight* se pak uchovává hodnota váhy.

# Kapitola 9

# Uživatelské rozhraní modulu

Nejprve je nutné zadat informace o vstupních datech. Poté musí uživatel vyplnit záložku *MiningTask* (obrázek 9.1). V této záložce se nejprve musí z nabídky *Task type* vybrat typ úlohy. V našem případě zvolíme *clasification*. Klasifikační modul obsahuje výběr z několika typů modelů, které můžeme pro klasifikaci použít. Typ modelu zvolíme v nabídce *Model name*. Pro použití mého modelu je nutné vybrat *Neural network*.

Nyní je možné přistoupit k nastavení parametrů pro klasifikaci. Nejprve je třeba v menu *Classified attribute* vyznačit, který atribut obsahuje informaci kam příslušný vektor patří. Tento atribut musí nabývat celočíselných hodnot od 0 do $n - 1$ pro klasifikaci do $n$ tříd.



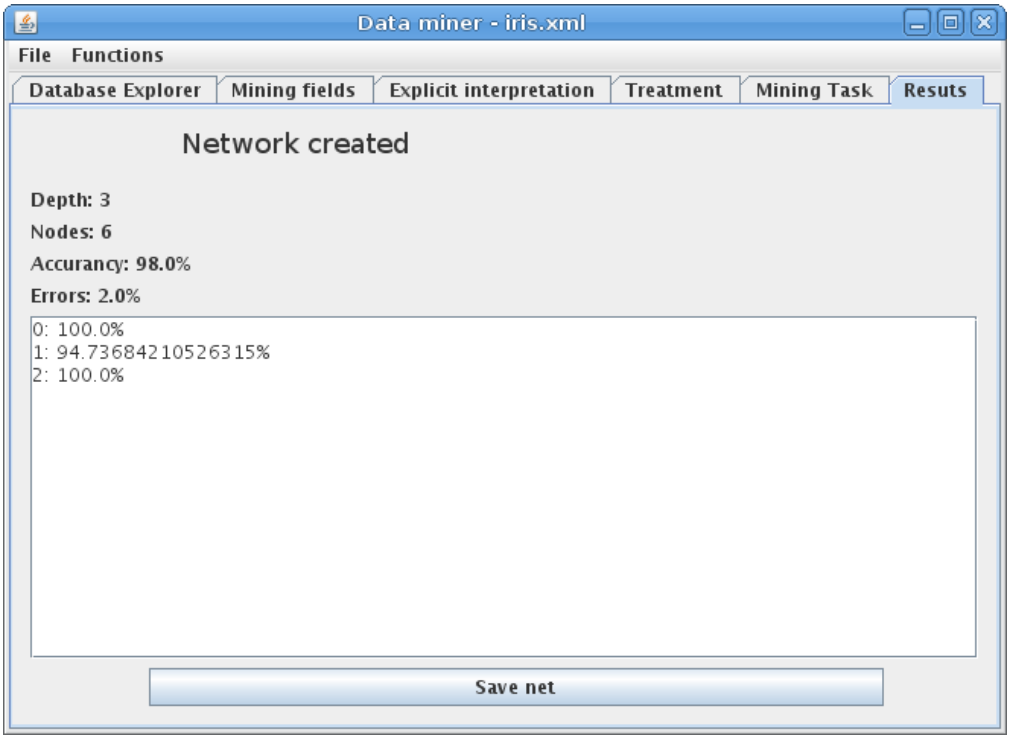

Obrázek 9.1: Příklad vyplněné záložky MiningTask

Nakonec nastavíme parametry pro budování stromu. Těmito atributy jsou *Maximal depth*. Tento parametr udává, do jaké maximální hloubky bude dolovací algoritmus budovat

výsledný strom. Druhým parametrem je *errorTreshold*, ten udává počet přípustných chyb v jednom uzlu na jednu třídu. Po zadání všech těchto nastavení spustíme dolovací proces klepnutím na tlačítko *Mine results*.

## 9.1 Interpretace výsledků

Vstupní množina dat pro naučení modelu se rozdělí na třetiny. První dvě třetiny prvků se použijí na učení. Zbývající třetina se použije pro otestování kvality získaného modelu. Informace o výsledcích dolovacího procesu se zobrazí na kartě *Results* (obrázek 9.2).




Obrázek 9.2: Příklad výsledné záložky Results

Tato karta obsahuje údaje o počtu uzlů a hloubce stromu (parametry *Depth* a *Nodes*). Dále pak o správnosti klasifikace vektorů, které byly určeny pro testování (parametry *Accurancy* a *Errors*). Důležité jsou i údaje o správnosti klasifikace prvků patřících do jednotlivých tříd (v bílém okně dole).

## 9.2 Použití naučeného modelu

Naučený model lze použít ke klasifikaci neznámých dat. Tato data musí mít stejné atributy jako data, na kterých byla síť naučena. Nejprve musíme model uložit pomocí tlačítka *Save net*. Poté vytvoříme novou úlohu pro klasifikovaná data. Na záložce *Mining Task* vybereme v menu *Phase* položku *Using*. Dolovací úlohu spustíme. Výsledky klasifikace budou uloženy v dočasné tabulce, kterou jsme zvolili v dialogu *File* → *Options* → *Temporary Database Table*.

# Kapitola 10

# Testování

Pro ověření správné činnosti modulu je dobré zkusit ho použít na klasifikaci testovacích dat. Pro otestování jsem použil tři testovací množiny vektorů.

První dvě množiny mají poměrně málo prvků a modul je tedy zpracuje poměrně rychle. Třetí množina už je nepoměrně rozsáhlejší a její zpracování trvá několik minut.

## 10.1 Průběh testování

U každé testovací úlohy je třeba nejprve pochopit, charakter vstupních dat. Proto se snažím zadání každé úlohy nejprve slovně popsat. Dále pak jsem pro každou úlohu vytvořil tabulku se souhrnnými daty o jejích prvcích. Těmito údaji jsou celkový počet prvků, počet prvků vstupního vektoru, počet tříd, do kterých je třeba klasifikovat a počty prvků v nich.

Dále pak považuji za důležité popsat, jak dlouho vytvoření modelu trvalo a jak je model rozsáhlý. Za tímto účelem poskytuji informace o počtu uzlů a největší hloubce stromu. Jako údaj, který charakterizuje náročnost vytvoření modelu uvádím délku jeho výpočtu na mém notebooku. Samozřejmě se tento údaj liší stroj od stroje, a proto zde uvádím některé jeho parametry:

- Procesor: Intel Celeron M 1.5 GHz
- Operační paměť: 512 MB DDR
- Java Virtual Machine: 1.6.0_11-b03

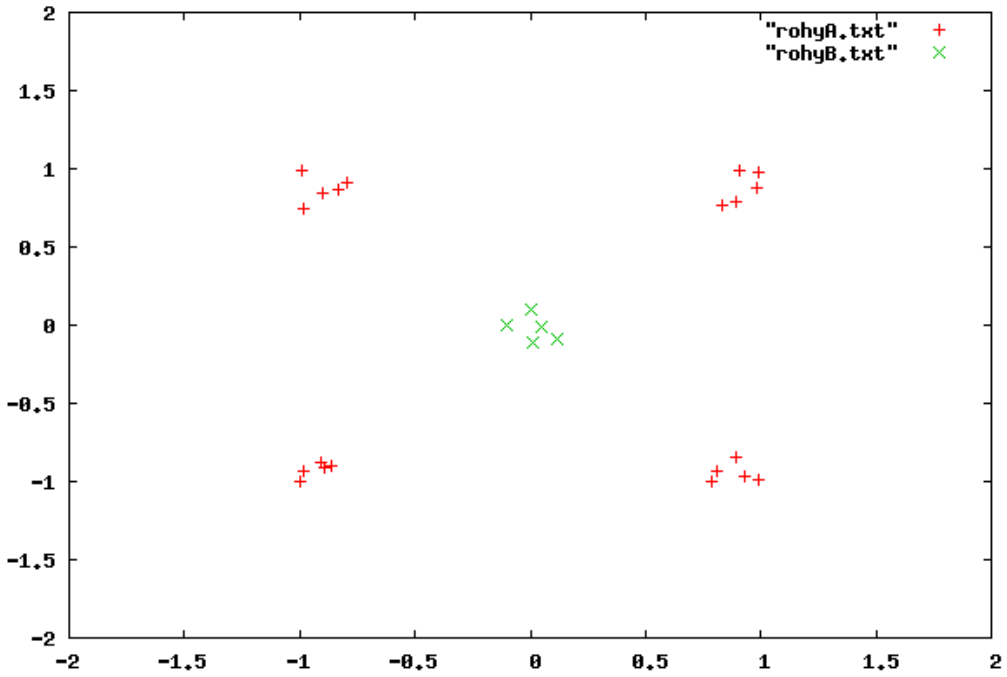
Nejdůležitější informací o testu je, jak kvalitní model pro klasifikaci se nám podařilo vytvořit. Za tímto účelem uvádím procento správně klasifikovaných vektorů a dále pak procento úspěšnosti klasifikace pro jednotlivé třídy.

Protože generování počátečních vah neuronů je náhodné, vytvoří se při každém spuštění úlohy jiný model. Z tohoto důvodu jsem vždy provedl učení a testování na zadané množině třikrát.

## 10.2 Problém izolovaných rohů

### 10.2.1 Popis úlohy

První testovací množina reprezentuje klasický problém tzv. izolovaných rohů. Ten je zajímavý tím, že není lineárně separovatelný. V mém případě je rozložení prvků z trénovací množiny naznačeno na obrázku 10.1.




Obrázek 10.1: Problém izolovaných rohů

Jedná se tedy o dvoudimenzionální problém, kde se vektory klasifikují do dvou tříd. Další podrobnosti o trénovací množině se lze dočíst v tabulce 10.1.

| **Vlastnosti trénovací množiny** | |
|---|---|
| **Počet prvků vstupního vektoru** | 2 |
| **Počet klasifikačních tříd** | 2 |
| Počet prvků 1. třídy | 20 |
| Počet prvků 2. třídy | 5 |
| **Celkový počet prvků** | 25 |

Tabulka 10.1: Vlastnosti dat pro problém izolovaných rohů

### 10.2.2 Výsledky

Výsledky třech pokusů o vytvoření modelu jsou vypsány v tabulce 10.2.

Závěrem lze tedy říci, že se modul s tímto problémem vypořádal dobře. Všechny testovací vektory byly totiž klasifikovány správně. čas potřebný k naučení je také bezproblémový.

| | 1. Měření | 2. Měření | 3. Měření |
|---|---|---|---|
| **Hloubka stromu** | 4 | 9 | 2 |
| **Počet uzlů** | 7 | 12 | 5 |
| **Celková přesnost** | 100% | 100% | 100% |
| **Chybovost** | 0% | 0% | 0% |
| Přesnost klasifikace 1. třídy | 100% | 100% | 100% |
| Přesnost klasifikace 2. třídy | 100% | 100% | 100% |
| **Délka výpočtu** | 2 sekundy | 2 sekundy | 2 sekundy |

Tabulka 10.2: Výsledky klasifikace pro problém izolovaných rohů

## 10.3 Data o kosatcích

### 10.3.1 Popis úlohy

Druhá množina testovacích vektorů obsahuje informace o kosatcích. Vstupní prvky obsahují informace o délce a šířce okvětních a kališních lístků. Cílem úlohy je pak poznat podle těchto údajů o jaký druh kosatce se jedná. Jednotlivé třídy reprezentují následující druhy - Setosa, Versicolor a Virginia.

Tabulka 10.3 obsahuje stručnou charakteristiku dat.

| **Vlastnosti trénovací množiny** | |
|---|---|
| **Počet prvků vstupního vektoru** | 4 |
| **Počet klasifikačních tříd** | 3 |
| Počet prvků 1. třídy | 50 |
| Počet prvků 2. třídy | 50 |
| Počet prvků 3. třídy | 50 |
| **Celkový počet prvků** | 150 |

Tabulka 10.3: Vlastnosti dat pro problém rozpoznávání kosatců

### 10.3.2 Výsledky

Výsledky pokusů jsou vypsány v následující tabulce 10.4.

Bohužel u této pokusné množiny se už systému nepodařilo správně zařadit všechny testovací vektory. Nejvíce chyb se objevilo při třetím pokusu a to 8% chybně klasifikovaných testovacích vektorů.

## 10.4 Vyhodnocování žádostí o přijetí

Poslední a nejrozsáhlejší testovací množinu jsem získal ze stránek předmětu získávání znalostí z databází. Jedná se o část databáze žádostí o přijetí dítěte do mateřské školky. Tato databáze vznikla v osmdesátých letech ve Slovinsku. V té době převyšoval počet žádostí možnosti přijetí.

Pro hodnocení se využívá několika kritérií a to:

- rodiče (běžní, úspěšní, velmi úspěšní)

| | 1. Měření | 2. Měření | 3. Měření |
|---|---|---|---|
| **Hloubka stromu** | 3 | 3 | 3 |
| **Počet uzlů** | 4 | 5 | 4 |
| **Celková přesnost** | 96% | 98% | 92% |
| **Chybovost** | 4% | 2% | 8% |
| Přesnost klasifikace 1. třídy | 100% | 100% | 94,118% |
| Přesnost klasifikace 2. třídy | 94,44% | 94,74% | 100% |
| Přesnost klasifikace 3. třídy | 94,44% | 100% | 83,333% |
| **Délka výpočtu** | 3 sekundy | 3 sekundy | 3 sekundy |

Tabulka 10.4: Výsledky klasifikace pro problém rozpoznávání kosatců

- péče (vhodná, méně vhodná, nevyhovující, kritická, velmi kritická)
- typ rodiny (kompletní, doplněná, neúplná, pěstounská)
- počet dětí (1, 2, 3, více)
- bydlení (vhodné, méně vhodné, kritické)
- peníze (vhodné, méně vhodné)

Každý tento kategorický atribut jsem zakódoval v kódování 1 z $n$. Žádosti o přijetí jsou pak rozdělovány do pěti kategorií s rostoucí prioritou:

1. Nedoporučuje se
2. Doporučuje se
3. Velmi se doporučuje
4. Prioritní
5. Velmi prioritní

| **Vlastnosti trénovací množiny** | |
|---|---|
| **Počet prvků vstupního vektoru** | 27 |
| **Počet klasifikačních tříd** | 5 |
| Počet prvků 1. třídy | 4092 |
| Počet prvků 2. třídy | 2 |
| Počet prvků 3. třídy | 316 |
| Počet prvků 4. třídy | 4066 |
| Počet prvků 4. třídy | 3838 |
| **Celkový počet prvků** | 12314 |

Tabulka 10.5: Vlastnosti dat pro problém klasifikace žádostí o přijetí

### 10.4.1 Výsledky

Výsledky třech spuštění dolovacího algoritmu jsou v tabulce 10.6.

| | **1. Měření** | **2. Měření** | **3. Měření** |
|---|---|---|---|
| **Hloubka stromu** | 11 | 11 | 11 |
| **Počet uzlů** | 169 | 132 | 128 |
| **Celková přesnost** | 93,71% | 94,67% | 94,91% |
| **Chybovost** | 6,29% | 5,33% | 5,09% |
| Přesnost klasifikace 1. třídy | 99,93% | 99,93% | 100% |
| Přesnost klasifikace 2. třídy | 0% | 0% | 0% |
| Přesnost klasifikace 3. třídy | 78,89% | 82,35% | 84,146% |
| Přesnost klasifikace 4. třídy | 92,42% | 92,97% | 93,78% |
| Přesnost klasifikace 5. třídy | 89,75% | 91,86% | 91,47% |
| **Délka výpočtu** | 310 sekundy | 301 sekundy | 288 sekundy |

Tabulka 10.6: Výsledky klasifikace pro problému žádosti o přijetí

# Kapitola 11

# Závěr

V této kapitole bych chtěl stručně zhodnotit výsledky své práce. Využiji proto zadání, které jsem obdržel po výběru tématu bakalářské práce. Toto zadání obsahuje několik bodů, které jsem měl splnit. Pokusím se je tedy zrekapitulovat.

## 11.1 Problematika získávání znalostí z databází

V průběhu práce jsem se seznámil s tím, co se rozumí pod pojmem „znalost“ a se základy procesu, který k získávání znalostí vede. Dozvěděl jsem se, jaké rozlišujeme druhy úloh při dolování z dat. Principy těchto úloh jsem pochopil a snažil jsem se je stručně popsat v první kapitole této práce.

Podrobněji jsem se věnoval úloze klasifikace. Naučil jsem se, jaké algoritmy se k řešení této úlohy používají a jaké mají výhody a nevýhody.

## 11.2 Systém pro získávání znalostí vyvíjený na FIT

Poznal jsem, k čemu je možné použít a jak pracuje systém pro získávání znalostí vyvíjený na naší fakultě. Seznámil jsem se také s jeho rozsáhlými možnostmi pro předzpracování dat.

Dále mě zajímalo, jak tento systém pracuje a jakou má architekturu. Nastudoval jsem zejména, jak je možné tento systém rozšířit o nový modul, který bude řešit další dolovací úlohu. Tyto znalosti jsem později využil pro vytvoření nového modulu pro klasifikaci.

## 11.3 Sestavení návrhu nového modulu

Po dohodě s vedoucím práce jsem se rozhodl implementovat modul pro klasifikaci, který by využíval neuronových sítí. Abych mohl tento modul implementovat, musel jsem se nejprve seznámit s problematikou práce neuronových sítí. O této problematice pojednává třetí kapitola mé práce.

Naučil jsem se, jak pracují některé modely, které se používají k simulaci neuronových sítí. Jak se tyto neurony učí a jak mohou být navzájem uspořádány. Také jsem poznal, jak je možné těchto umělých neuronových sítí využít k řešení některých dolovacích úloh.

## 11.4 Implementace modulu

Jakmile jsem věděl, jaký bude účel modulu a jaké algoritmy k jeho práci využiji, bylo nutné tento modul naprogramovat. Pro implementaci jsem musel navrhnout třídy, které použiji a postupně je naprogramovat.

Takto vzniklý program jsem následně zabudoval do systém pro získávání znalostí a vytvořil potřebná rozšíření jazyka DMSL. Program se stal součástí již vytvořeného modulu pro klasifikaci. Ten nyní umožňuje vytvořit rozhodovací strom, SVM klasifikátor a nově také neuronové sítě.

## 11.5 Zhodnocení výsledků

Vytvořený modul jsem otestoval na třech množinách vstupních dat s různým rozsahem a složitostí. Výsledky testů se vždy pohybovaly nad 90% úspěšností.

Za nevýhodu modulu považuji především dobu zpracování rozsáhlejších dat. Ta je způsobena především použitím pocket algoritmu při učení perceptronu. Druhým problémem by mohla být velká paměťová náročnost, která je zapříčiněna nutností uchovávat informace o tom, které vektory jsou součástí trénovací množiny, odpovídající určitému uzlu.

Naopak velkou výhodou je rychlost klasifikace již naučeného systému. Ta je daná tím, že při tomto procesu není nutné vyhodnocovat všechny uzly naučeného stromu. Počet neuronů, u kterých je nutné spočítat výstup, je často podstatně nižší než u běžných neuronových sítí.

# Literatura


[1] Fanguy, R.; Kubat, M.: *Modifying Upstart for Use in Multiclass Numerical Domains* [online]. [cit. 2009-05-18].
URL http://www.aaai.org/Papers/FLAIRS/2002/FLAIRS02-067.pdf

[2] Forgáč, M.: *Modul shlukové analýzy systému pro získávání znalostí z databází.* Diplomová práce, FIT VUT v Brně, 2006.

[3] Han, J.; Kamber, M.: *Data Mining: Concepts and Techniques.* Morgan Kaufmann Publishers, 2001, ISBN 1-55860-489-8.

[4] Hlosta, M.: *Modul pro dolování z dat.* Bakalářská práce, FIT VUT v Brně, 2008.

[5] Hromčík, P.: *Systém pro získávání znalostí z databází.* Diplomová práce, FIT VUT v Brně, 2003.

[6] Mehrotra, K.; Mohan, C. K.; Ranka, S.: *Elements of Artificial Neural Networks.* The MIT Press, 2000, ISBN 0-262-13328-8.

[7] Zendulka, J.; kol.: *Získávání znalostí z databází - studijní opora.* FIT VUT v Brně, 2006.