Univerzita Karlova v Praze, Filozofická fakulta
Katedra logiky

Jakub Dolejšek

ZÍSKÁVÁNÍ ZNALOSTÍ Z DATABÁZÍ
KNOWLEDGE DATABASE DISCOVERY

Bakalářská práce

Vedoucí práce: PhDr. Michal Peliš, Ph.D.

2013

Děkuji především vedoucímu práce PhDr. Michalu Pelišovi, Ph.D. za cenné rady, připomínky, nadhled, přátelský přístup a veškerý čas, který mi věnoval. Děkuji PhDr. Šárce Honsové, Ph.D. za konzultace a poskytnutí dat. Také děkuji Bc. Tereze Liepoldové za motivaci, připomínky a opravy. A díky patří mým rodičům za mnohé.

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a že jsem uvedl všechny použité prameny a literaturu.

V Praze 19. srpna 2013

Jakub Dolejšek

Abstrakt

Práce se zabývá problematikou dobývání znalostí z databází se zaměřením na metody rozhodovacích stromů a neuronových sítí spolu s ukázkami jejich použití na konkrétních příkladech.

Abstract

This paper describes problematic of the knowleadge database discovery with focus on methods of decision trees and neural networks with examples of their application on concrete examples.

Obsah

1 Úvod

1.1 Cíl práce

Práce se zabývá problematikou dobývání znalostí z databází. Snaží se podat celkový vhled do problematiky a zároveň klíčové části ilustrovat na praktických ukázkách. Podrobněji jsou rozpracovány vybrané analytické metody - algoritmus *ID3* pro vytvoření *rozhodovacích stromů* (kapitola 4.3) a algoritmus *Backpropagation* jako reprezentant *neuronových sítí* (kapitola 4.4). Tyto algoritmy v řadě ohledů stojí na opačných pólech a jejich srovnání tak nabízí zajímavý přehled problematiky.

Pro praktické ukázky jsou využita data získaná z motorických testů provedených na FTVS[1] a v rámci práce byla provedena jejich analýza z pohledu data miningu.

Zdrojové kódy algoritmů, jejichž implementace byla zpracována v rámci práce, a také funkce generující vizualizace použité v textu jsou dostupné v přílohách A.

1.1.1 Použité technologie

Přiložené implementace algoritmů jsou zpracovány v programovacím jazyce *Python* [sw1]. Ilustrační grafy a diagramy jsou vytvořeny s pomocí knihovny funkcí *Matplotlib* [sw5], jejich generování je zpravidla přímo navázané na výstupy nebo průběhy jednotlivých algoritmů.

Pro sazbu textu byl použit systém LaTeX. Za zmínku také stojí rozšíření LaTeXu *Minted* [sw3], které výrazně zjednodušuje prezentaci zdrojového kódu programů v textu práce.

[1] Fakulta tělesné výchovy a sportu Univerzity Karlovy.

2 Přehled problematiky

Empirické tvrzení známé jako *Kryderův zákon* [w4] říká, že kapacita zařízení k ukládání dat se každých 18 měsíců zdvojnásobí při zachování stejné ceny.[2] S tím přirozeně dochází i k využití této kapacity.[3] IBM odhaduje, že 90% dat vzniklo během posledních dvou let [w1].

Množství dat, ke kterým mají lidé přístup, je nesrovnatelně větší, než jaké je lidská mysl schopná pojmout. Uživatelé „blogují“, „tweetují“, fotí nebo nahrávají videa. Jak mezi nimi najít informace, co stojí za pozornost? Roste objem hudební a filmové tvorby, vzniká řada zpravodajských nebo úzce specializovaný serverů. Které stojí za to sledovat? Komerční společnosti mají k dispozici množství informací o svých zákaznících. Jak je vhodně využít? Umělé družice sledující Zemi, sluneční soustavu nebo namířené do „hlubokého vesmíru“ produkují nepřeberné množství dat, která jsou často volně dostupná ke stažení a zpracování každému, kdo má zájem. Co neobjeveného se v nich skrývá? Toto jsou některé z otázek, na které je možné hledat odpovědi za pomoci metod *dobývání znalostí z databází* nebo *data miningu*.[4]

Stručná charakteristiku této problematiky je uvedena například v publikaci *Principles of Data Mining* [5]:

> **Data mining** je analýza (často objemných) množin dat s cílem nalézt předem neznámé vztahy a shrnout data novými způsoby, které jsou srozumitelné a užitečné.

Praktické aplikace můžeme nalézt v řadě oblastí. Banky, pojišťovny nebo telekomunikační společnosti analyzují údaje o svých zákaznících tak, aby mohly lépe cílit své produkty. Investiční společnosti se za pomoci dataminingových metod snaží předvídat vývoj na akciovém trhu. Obchodní řetězce provádí *analýzu nákupního košíku* a mohou tak například lépe uspořádávat zboží. V medicíně, obchodu či strojírenství se používají *expertní systémy*, které s využitím vhodné *báze dat* mohou dávat odpovědi na otázky v té které problematice.

Jako běžní uživatelé se můžeme na internetu setkat s řadou systémů na doporučování relevantního obsahu[5]: elektronické obchody nabízejí „související zboží“, filmová databáze imdb.com registrovaným uživatelům navrhuje filmy dle dříve za-

[2] *Kryderův zákon* je obdobou známějšího *Mooreova zákona*, který pozoruje, že *počet transistorů v integrovaných obvodech* se zdvojnásobí každých 18 měsíců.

[3] S nadsázkou můžeme toto tvrzení podpořit jedním z Murphyho počítačových zákonů, který říká: „Disky jsou vždy plné. Je marné pokoušet se získat větší diskový prostor. Data expandují tak, aby zaplnila veškerý volný prostor.“

[4] Termíny *dobývání znalostí z databází* (Knowleadge Discovery in Databases, KDD) a *data mining* (DM) jsou často používány jako synonyma, někteří autoři ale chápou pojem KDD v širším smyslu a termínem DM pak označují jen samotné využití analytických algoritmů.

[5] Metodám na doporučování obsahu je věnovaná kniha (aktuálně nedokončená, dostupná online) *A Programmer's Guide to Data Mining* [1].

daných hodnocení ostatních filmů, vyhledávač google.com se u registrovaných uživatelů „učí“ znát jejich oblasti zájmu a zpřesňuje tak výsledky vyhledávání.

2.1 Klasifikace úloh

Úlohy data miningu můžeme rozlišovat například podle toho, s jakým cílem jsou data analyzována. Jednu možnou klasifikaci nabízí *Principles of Data Mining* [5]:

Explorační analýza dat (Exploratory Data Analysis, EDA)

Jak název naznačuje, jedná se o úlohy, při kterých jsou data zkoumána, aniž by předem bylo jasné, co je konkrétním cílem. Typicky se zde používají vizualizace, které při vhodném použití mohou odhalit různé vztahy ve zkoumaných datech nebo vytipovat zajímavé podmnožiny k dalšímu zkoumání.

Deskriptivní modelování (Descriptive Modeling)

Deskriptivní modely se tvoří s cílem sestavení uceleného popisu všech dostupných dat. Můžeme mezi ně zařadit například statistické metody shlukové analýzy, kde jsou jednotlivé objekty tříděny do skupin (shluků) na základě vzájemné podobnosti. Jiným typem deskriptivních modelů jsou „modely závislostí“, které popisují vztahy mezi jednotlivými atributy zkoumaných objektů.

Prediktivní modelování (Predictive Modeling)

Zde je cílem vytvořit modely, které budou schopné předvídat hodnoty některého zvoleného atributu na základě znalosti ostatních. Pokud je zvolený atribut kategoriálního typu (nabývá hodnoty z předem dané, omezené množiny), pak mluvíme o **klasifikaci**. Pokud se jedná o atribut číselný (spojitý), mluvíme o **regresi**.

Hledání vzorů a pravidel (Patterns and Rules)

Cílem úloh tohoto typu je objevení zajímavých vztahů v analyzovaných datech. Na rozdíl od předchozích úloh není výstupem kompletní popis všech dat. Jedná se naopak o nalezení podmožin, které se od zbytku nějakým způsobem odlišují.

Vyhledávání dle obsahu (Retrieval by Content)

Typickým příkladem těchto úloh je vyhledávání webových stránek na základě textového dotazu tak, jak jej implementuje například www.google.com. Můžeme sem zařadit také systémy na doporučování obsahu, které návštěvníkům webových stránek nabízejí obsah na základě pozorování jejich chování.

Je-li jasný cíl úlohy, je praktické zvolit standardizovaný postup pro zpracování celého projektu. Pravděpodobně nejznámější a nejčastěji užívanou metodikou je CRISP-DM, kterou představuje následující kapitola.

2.2 Metodika CRISP-DM

CRoss **I**ndustry **S**tandard **P**rocess for **D**ata **M**ining je metodika standardizující proces *dobývání znalostí z databází*.[6] Tato metodika vznikla v rámci evropské výzkumné iniciativy ESPRIT jako společný projekt několika velkých společností s rozsáhlými zkušenostmi v oblasti data miningu s cílem navrhnout univerzální postup, který by byl použitelný v nejrůznějších komerčních aplikacích. Dobývání znalostí probíhá dle metodiky CRISP-DM v šesti krocích, jak ilustruje obrázek 1.[7]

Obrázek 1: Diagram metodiky CRISP-DM.

1. **Porozumění problematice** (Bussiness undertanding). V první fázi se nejdříve formulují cíle projektu z obchodního hlediska a stanovují kritéria jeho úspěšnosti. Dalším úkolem je revize dostupných zdrojů (personálních, datových, výpočetní techniky, softwaru, ...), odhad nákladů, rizik a přínosu projektu. Nakonec se stanovuje předběžný plán průběhu projektu.

2. **Porozumění datům** (Data undertanding). Cílem této fáze je získání přehledu o dostupných datech. Zjišťují se jejich základní charakteris-

[6] Pro kompletní popis metodiky CRISP-DM viz [6]. Mezi další známé metodiky patří např. 5A firmy SPSS nebo SEMMA firmy SAS. Pro jejich přehled viz [3].

[7] Zdroj obrázku: Wikipedia [w3].

tiky (struktura, počty záznamů a atributů v jednotlivých tabulkách), dále dochází k prvotnímu zkoumání dat, formulaci předběžných hypotéz či vytipování zajímavých podmonžin. Důležitým krokem je posouzení kvality dat: zkoumá se jejich chybovost, zjišťují se chybějící hodnoty a zda jsou dostupná data vhodným reprezentativním vzorkem pro zkoumanou situaci.

3. **Příprava dat** (Data preparation). Výstupem této fáze jsou data ve formátu odpovídajícím zvolenému analytickému algoritmu. Zahrnuje výběr vhodných atributů (sloupců) a záznamů (řádků), dále čištění dat. V případě chybějícíh atributů nastavení vhodných implicitních hodnot (případně použití sofistikovanějších postupů pro doplnění). Součástí této fáze může být transformace dat, vytvoření nových atributů, případně i nových záznamů na základě ostatních dostupných hodnot. Nakonec se provádí integrace a převedení do vhodné struktury a formátu, který bude již přímým vstupem pro analytický algoritmus.

4. **Modelování** (Modeling). Ve fázi modelování jsou nasazeny vlastní analytické algoritmy. Důležitým krokem je vytoření testovacího mechanismu, který umožní snadné ověřování výstupu algoritmu. Obvyklým postupem je rozdělení dat na dvě nezávisle množiny - *trénovací data*, která jsou přímým vstupem analytického algoritmu, a *validační data*, pomocí kterých je výstup ověřován. Tento mechanismus často pomáhá řešit problém přeučení (viz 4.1.1). Nejčastěji používaným analytickým metodám se věnuje kapitola 4.

5. **Vyhodnocení výsledků** (Data evaluation). Po ukončení fáze modelování, jejíž výstupy jsou v pořádku z pohledu analytických metod, dochází k vyhodnocení výsledků z hlediska zadavatele. Zhodnotí se celkový průběh projektu a navrhují se další kroky. Jak naznačuje výše uvedený diagram, často dochází k další iteraci celého procesu s využitím nově získaných informací.

6. **Využití výsledků** (Deployment). Finálním výstupem celého procesu je plán popisující, jakým způsobem budou nově získané informace nasazené v praxi, a postup pro monitorování a údržbu tohoto nasazení.

Pro účely této práce jsou relevantní fáze porozumění a přípravy dat, kterým je věnována následující kapitola 3 a dále fáze modelování, kterou rozebírá kapitola 4.

3 Příprava dat a předběžná analýza

V této kapitole jsou ilustrovány kroky porozumění datům a jejich předzpracování podobně, jak je navrhuje metodika CRISP-DM v bodech 2 a 3.

3.1 Import a čištění dat

Datový soubor, který je dostupný pro analýzu, zpravidla není připravený ve formátu, který by byl vhodný pro přímé strojové zpracování.

Vzorek dat motorických testů *UNIFITTEST* [4] získaný z FTVS obsahoval přibližně 130 různých výsledků, kde se každý záznam skládal ze základních údajů testované osoby (pohlaví, výška, váha, datum narození, naměřený podkožní tuk), a měření jednotlivých testů (skok daleký, leh-sed, ...). Základní údaje byly dostupné pro všechny vzorky (s výjimkou naměřeného podkožního tuku, který byl dostupný přibližně u poloviny z nich). Z naměřených testů byly u všech vzorků vyplněné dva testy: skok daleký a počet leh-sedů. Ostatní testy byly dostupné jen u některých měřených vzorků, částečně také proto, že dle metodiky UNIFITTEST se některé testy provádějí jen pro určité skupiny (na základě věku nebo pohlaví). V analýze se zaměříme pouze na dva kompletně vyplněné testy; u ostatních by byly počty natolik nízké, že by metody data miningu velice pravděpodobně selhávaly.

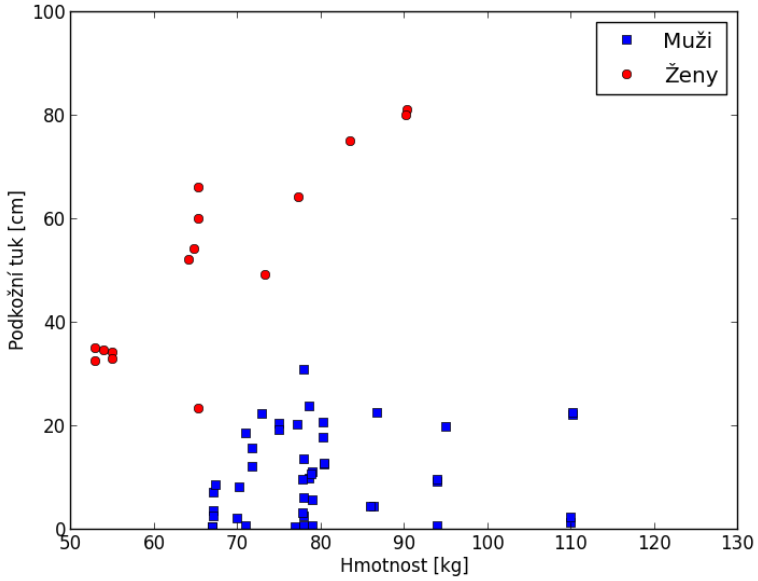
Data byla dodána v souboru formátu .xlsx (Microsoft Excel 2007/2010). Prvním potřebným krokem tedy bylo provést *import* do datové struktury programovacího jazyka. Zároveň bylo nutné provést první *čištění dat* (např. odstranit nadbytečné řetězce „kg“ nebo „cm“) a převedení textových řetězců obsahujících číslice na reálná čísla. Pro import z formátu .xlsx byla využita knihovna *xlrd* [sw2]. Ukázka funkce, která zajišťovala import a začištění dat je v příloze A.1.

Struktura dat byla pro účely dalšího zpracování vyhovující. Jednalo se o jedinou tabulku, která obsahovala všechny potřebné údaje. Krok *transformace dat* tedy v podstatě nebyl nutný.

3.2 Předběžná pozorování

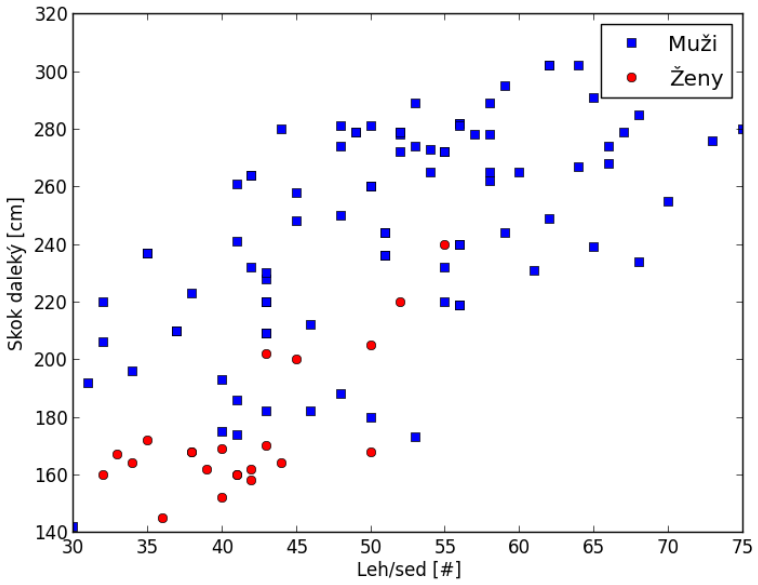
V další fázi byla provedena pozorovnání za účelem získání předběžných výsledků a orientace v datech. V příloze A.2.2 je přehled provedených vizualizací, které se snaží odhalit závislosti mezi jednotlivými atributy.

Například graf 2 ukazuje vztah hmotnosti testované osoby k naměřenému podkožnímu tuku s barevným oddělením pro muže a ženy. Můžeme z něj například vyčíst, že množství podkožního tuku u žen je zpravidla vyšší, což je ovšem známý fakt (viz tabulky UNIFITTEST [4]).



Obrázek 2: Vztah váhy a podkožního tuku.

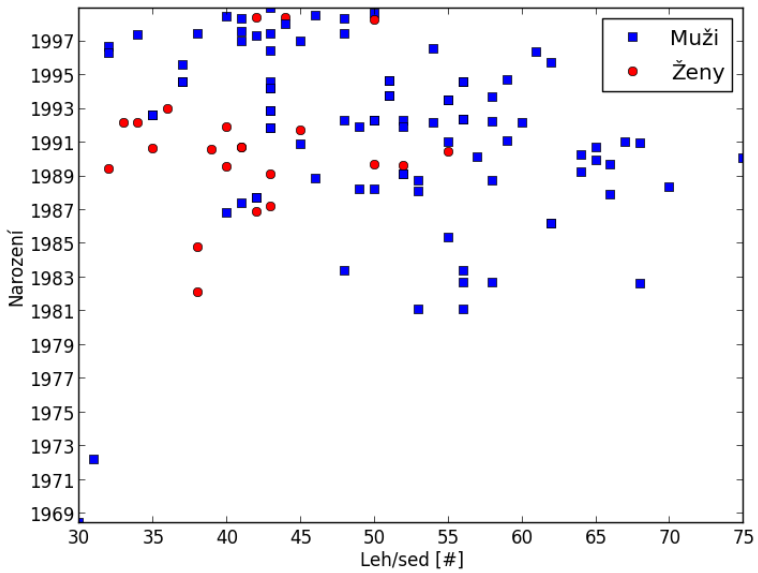
Graf 3 dává do vztahu výsledky jednotlivých testů. I zde je výsledek pozorování předvídatelný, totiž že vyšší výkon v jednom testu koreluje s vyšším výsledkem v testu druhém.



Obrázek 3: Vztah výkonnostních testů.

V některých grafech v příloze A.2.2 můžeme odhalit další vztahy. Například že vyšší výška naznačuje lepší výsledky ve skoku do dálky nebo že

existuje vztah mezi výškou a váhou. Jiné grafy (například vztah mezi věkem a počtem leh-sedů - viz graf 4) ukazují, že mezi těmito atributy žádná snadno pozorovatelná závislost (alespoň pro tento vzorek dat) neexistuje.



Obrázek 4: Vztah věku a testu leh-sed.

Celkově můžeme shrnout, že předběžná analýza žádné překvapivé výsledky neodhalila. Pomohla nám ale lépe se seznámit s dostupnými daty. Také ukázala na dobrou kvalitu dat - dá se předpokládat, že pokud by data obsahovala některé hodnoty zcela chybné, tento vhled do dat by je pomohl odhalit.

Nicméně mohou existovat i závislosti, které není možné odhalit „pouhým okem“. K tomu, aby bylo možné „rozšifrovat“ složitější vztahy v analyzovaných datech, je nutné použít sofistikovanějších analytických metod.

Následující kapitola představuje základní komponenty analytických algoritmů a představuje ty nejčastěji používané.

4 Analytické metody

4.1 Komponenty analytických algoritmů

Přestože podstata jednotlivých analytických algoritmů se často diametrálně liší, některé základní „komponenty“ můžeme nalézt u většiny používaných metod. Mezi ně patří především modely, pravidla a hodnotící funkce. Dále je pak důležitá optimalizační a vyhledávací metoda a způsob, jakým jsou data ukládána. Stručně je představen problém *přeučení*, kterým „trpí“ většina běžných algoritmů.

Modely a pravidla

Základem každého analytického algoritmu je vhodná datová struktura schopná reprezentovat nalezené znalosti.

Pokud je taková struktura „globální“ (reprezentuje znalosti o všech prvcích univerza), nazývame ji **model**. V případě struktury „lokální“ (reprezentující znalosti pouze pro vybrané prvky) pak mluvíme o **pravidlech** (*pattern*).

Jednoduchým příkladem může být předpis $Y = aX + b$, kde X a Y jsou proměnné (atributy popisovaných či zkoumaných instancí), a a b jsou konstanty získané v průběhu učící fáze algoritmu. Předpis spolu s konstantami pak je jednoduchým příkladem *modelu*. Můžeme jej chápat ve smyslu deskripce, tedy jako konstatování, že pro všechny prvky univerza je hodnota jejich atributu Y v takovém a takovém vztahu k atributu X. Nebo jej můžeme použít pro predikci - jako funkci, která u instance s hodnotou atributu X „předvídá“ hodnotu atributu Y.

Příkladem *pravidla* by pak mohlo být tvrzení: $if\ (X > x_1)\ then\ prob(Y > y_1) = p_1$. Tedy pokud je hodnota atributu X větší než konstanta x_1, pak s pravděpodobností p_1 je hodnota atributu Y větší než konstanta y_1. V tomto případě mohou existovat prvky univerza (splňující $X > x_1$), pro které toto pravidlo nedává žádnou odpověď, v čemž spočívá podstata rozdílu mezi *modely* a *pravidly*.

V praxi používané modely a pravidla jsou samozřejmě složitější, ale základní princip zůstává podobný.

Zpravidla se tedy jedná o nějakou strukturu a k ní náležející hodnoty naučených konstant. Struktury nejčastěji používaných metod budou ještě zmíněny v sekci 4.2.

Hodnotící funkce

Hodnotící funkce (*score function*) kvantifikuje, jak „dobře“ vytvořený model odpovídá dané množině dat. Typickým příkladem může být

metoda nejmenších čtverců a její chybová funkce:

$$\sum_{i=1}^{n}(y_i - y_i')^2,$$

kde y_i označuje hodnoty *očekávané* a y_i' hodnoty *vypočítané* (predikované) za pomoci učené struktury pro n výstupních hodnot.

Optimalizační a vyhledávací metoda

Úkolem optimalizační či vyhledávací metody je nalézt takový model či sadu pravidel (tedy navrhnout vhodnou strukturu a nastavit její konstanty) tak, aby se maximalizoval (či minimalizoval - záleží na kontextu) výstup hodnotící funkce pro dostupná *trénovací* data. Často se zde využívají heuristické algoritmy prohledávání v prostoru možných kombinací. V případě regresních metod učící algoritmus na základě vrácené chyby nějakým způsobem upravuje strukturu a celý postup opakuje.

Způsob uložení dat

Poslední „komponentou“ analytických algoritmů je způsob uložení dat - jak jsou uložena, indexována a jak je možné k nim přistupovat. Většina používaných algoritmů předpokládá, že data jsou uložená v paměti s přímým přístupem (RAM[8]). S narůstajícím objemem dat ale mohou nastat situace, kde operační paměť počítače není postačující, dochází k odkládání dat na pevný disk (který má řádově nižší přístupové rychlosti) a výkon algoritmu tak dramaticky klesá. V takových případech je vhodné uvažovat o algoritmech, které nevyžadují „náhodný“ přístup k datům, ale jsou schopné data zpracovávat například sériově či po menších částech.

4.1.1 Problém přeučení

Přeučení (*overfitting*) je problém, který se v různých obměnách týká většiny metod. Nastává v situaci, kdy algoritmus učící se na omezené trénovací množině neukončí svoji činnost ve vhodný okamžik a pokračuje v učení. Přeučený systém popisuje testovací množinu příliš konkrétně a přestává tak popisovat obecné principy, které jsou pro zkoumanou oblast charakteristické. Naučený model dává dobré výsledky na trénovací množině, ale na nových, během učení neznámých, datech selhává. Algoritmy mají k přeučení tendenci právě proto, že přeučený systém testovací množinu popisuje přesněji a hodnota chybové funkce je tak nižší. Tomuto problému je tedy nutné nějakým způsobem předcházet.

[8]RAM je zkratka pro Random-Access Memory, paměť umožňující přistupovat k uloženým informacím v konstantním čase nezávisle na pozici jejich uložení.

Jednou z možností je zvolit model takové velikosti, aby jeho vyjadřovací síla (*expressive power*) odpovídala obecným pravidlům, ale nedostačovala pro popsání příliš konkrétních vztahů. Takový model není přeučení vlastně „schopen“ nebo jen v omezené míře. Je ovšem nutné vyvarovat se situace, kdy by zvolený model měl vyjadřovací sílu naopak příliš malou, a nemohl tak popsat všechny potřebné vztahy v datech. Volba správné velikosti modelu pro daný problém je tedy klíčovým, ovšem často netriviálním, úkolem.

Jinou možností, jak snížit riziko přeučení, je průběžné porovnávání s nezávislou *validační množinou* dat. Tato množina není přímým vstupem pro učící algoritmus, ale hodnota chybové funkce na této množině se používá při rozhodování o ukončení učení. Kritériem pro ukončení učení bývá stav, kdy hodnota chybové funkce na validační množině začíná mít vzestupnou tendenci, byť chyba na trénovací množině stále klesá. Zde ale naopak hrozí riziko, že učení bude ukončeno předčasně - chybová funkce na validační množině nemusí mít monotonní průběh a může se jednat pouze o minimum lokální, a je tedy vhodné algoritmus neukončovat ihned po nalezení minima.

4.2 Stručný přehled metod

Pro představu, jaké metody jsou pro dobývání znalosti k dispozici, si zde velice stručně představíme ty nejčastěji užívané.

Rozhodovací stromy

Reprezentace pomocí rozhodovacích stromů je dobře známá z řady oblastí - příkladem mohou být „klíče k určování“ živočichů či rostlin. Při vyhodnocování postupujeme od kořene stromu, přičemž v každém uzlu je „otázka“ a každé hraně odpovídá jedna z možných „odpovědí“. Pokračujeme dle odpovědí, které platí pro daný případ, dokud nedojdeme do listu stromu, kterému odpovídá zařazení do jedné z možných kategorií. Rozhodovacím stromům se podrobněji věnuje kapitola 4.3

Asociační pravidla

Asociačním pravidlům odpovídají znalosti reprezentované jako množina pravidel tvaru IF-THEN.

S konceptem asociačních pravidel přišla koncem 60. let skupina českých vědců kolem P. Hájka. Jejich metoda GUHA (General Unary Hypotheses Automation) se snažila nalézt v datech všechny zajímavé souvislosti a nabídnout je uživateli.

Pravděpodobně nejznámější současnou metodou pro konstrukci asociačních pravidel je *algoritmus Apriori* navržený R. Agrawalem v roce 1996. Jádrem algoritmu je hledání často se opakujících množin položek. Postupně se konstruují konjunkce kategorií (tvrzení), vždy v n-tém kroku se konstruuje konjunkce obsahující $n + 1$ prvků s využitím znalosti všech kratších konjunkcí.

Pro další informace o asociačních pravidlech viz *Dobývání znalostí z databází* [3, kapitola 5.2]. Metodu GUHA ve své práci používá také Walter [2].

Neuronové sítě

Neuronové sítě pro reprezentaci znalostí využívají strukturu obsahující *neurony* uspořádané do několika vrstev, obvykle tak, že výstupy neuronů v n-té vrstvě směřují do vstupů $n + 1$-vé vrstvy. Každému spojení je přiřazena jistá váha. Neuron je pak „aktivován“ (vysílá impuls do další vrstvy), pokud součet všech vstupů přesáhne určitou mez. Výstup poslední vrstvy reprezentuje hledanou odpověď.

Algoritmům neuronových sítí se dále věnuje kapitola 4.4.

Evoluční metody

Mezi další metody, které jsou inspirovány biologickými principy patří genetické algoritmy, evoluční programování, evoluční strategie a genetické programování. Všechny tyto postupy mají společné následující základní pojmy:

- *Selekce*, kdy se z aktuální populace[9] vybírají nejkvalitnější jedinci na základě tzv. *fitness funkce*.
- *Reprodukce* (křížení), které kombinací vlastností dvou jedinců (vybraných pomocí selekce) vytvoří nového jedince.
- *Mutace*, která nějakým způsobem modifikuje (nového) jedince, například náhodně. Jinou strategií mutace je, že míra mutace je úměrná podobnosti „rodičů“. Evoluční metody totiž často tíhnou k „ustrnutí“ v lokálním minimu a toto je jedna z možných strategií, jak zvýšit šanci k „opuštění lokální pasti“.

Více o evolučních metodách se píše např. v *Machine Learning* [7, kapitola 9] nebo *Dobývání znalostí z databází* [3, kapitola 5.5].

Bayesovské metody

Bayesovské klasifikátory vycházejí z Bayesovy věty o podmíněných pravděpodobnostech. Jedná se tedy o pravděpodobnostní metody, které se často uplatňují v souvislosti se strojovým učením. Je možné je využívat jako základ pro algoritmy které přímo pracují s pravděpodobnostmi. Tyto metody můžeme ale také využít jako rozšíření (vylepšení) jiných metod, jež ve své původní podobě s pravděpodobností nepracují.

Podrobněji se Bayesovským metodám věnuje *Machine Learning* [7, kapitola 6] nebo *Dobývání znalostí z databází* [3, kapitola 5.6].

[9] Je dobré si uvědomit, že pod jedincem v populaci si můžeme představit také model, pravidlo či funkci.

V následující kapitole budou podrobněji představeny rozhodovací stromy jako reprezentant algoritmů pracujících s diskrétními hodnotami.

4.3 Rozhodovací stromy

Rozhodovací stromy jsou metoda umožňující klasifikovat množinu instancí na základě hodnot jejich atributů. Každému uzlu zkonstruovaného stromu odpovídá test některého vstupního atributu, každé hraně odpovídá možná hodnota tohoto atributu a každý list stromu reprezentuje hodnotu jednoho (předem zvoleného) výstupního atributu. Vytvořený strom je možné využít ke klasifikaci či predikci nové instance tak, že se strom prochází od kořene k listu a vždy se volí taková hrana, která odpovídá hodnotě vstupního atributu u zkoumané instance, a dosažený list určuje hodnotu výstupního atributu. Rozhodovací strom můžeme také chápat jako funkci, která vstupním hodnotám atributů přiřazuje hodnotu atributu výstupního.

Rozhodovací stromy je vhodné použít v situacích, kdy instance jsou reprezentovány jako množina dvojic atribut-hodnota. Předpokládá se fixní počet atributů, jejich hodnoty diskrétní, které nabývají malého počtu různých hodnot (tedy atributy jsou „výčtového typu“).[10] Výhodou rozhodovacích stromů je také přehlednost a snadná vizualizovatelnost získaných znalostí.

Základním algoritmem pro tvorbu rozhodovacích stromů, který bude aplikován na data UNIFITTEST, je algoritmus ID3. Ten bude blíže představen v následující kapitole.

4.3.1 Algoritmus ID3

Tento algoritmus, který předstvil Quinlan (1986), je příkladem „hladového“ algortimu, který prohledává prostor možných stromů shora dolů. Ukážeme si zjednodušenou verzi tohoto algoritmu, která odpovídá funkcím s binárními výstupními hodnotami.

Algortimus postupuje rekurzivně a v každém kroku umisťuje do kořene (pod)stromu atribut, který v jistém smyslu „nejlépe“ klasifikuje dostupné příklady. Jak přesně vybrat „nejlepší“ atribut bude popsáno dále.

Definujme si nejdříve pojem **entropie**.[11]

$$H(S) = \sum_{i=1}^{c} -p_i log_2 p_i, \tag{1}$$

kde S je množina příkladů, c počet hodnot, které může nabývat výstupní

[10] Existují i rozšíření této metody, které umožňují reálné výstupní hodnoty, ale aplikace rozhodovacích stromů na takové problémy není příliš obvyklá.

[11] Entropie (řečeno neformálně) vyjadřuje míru neuspořádanosti systému S.

atribut, a p_i je relativní četnost příkladů v množině S s odpovádající hodnotu výstupního atributu i.[12]

Speciálně pro binární verzi algoritmu můžeme entropii vyjádřit následovně:

$$H(S) = -p_{\oplus} log_2 p_{\oplus} - p_{\ominus} log_2 p_{\ominus}, \tag{2}$$

kde S je množina příkladů, $p_{\oplus}$ a $p_{\ominus}$ jsou relativní četnosti pozitivních respektive negativních příkladů v množině S. Pozitivní, respektive negativní příklad je takový, který nabývá pozitivních, respektive negativních hodnot výstupního atributu.

Nyní definujeme **informační zisk**[13] atributu a na množině S:

$$Gain(S, a) = H(S) - \sum_{v \in vals(a)} \frac{|S_{a_v}| \cdot H(S_{a_v})}{|S|}, \tag{3}$$

kde $vals(a)$ označuje hodnoty, kterých může nabývat atribut a, a S_{a_v} je podmnožina S prvků, jejichž hodnota atributu a má hodnotu v (tedy: $S_{a_v} = \{x \in S; x_a = v\}$). Informační zisk tedy určuje snížení entropie množiny S při rozkladu dle atributu a.

Hlavní část algoritmu ukazuje obrázek 5, kompletní zdrojový kód je pak v příloze A.3 Algoritmus ID3 postupuje, jak již bylo řečeno, rekurzivně. Vstupem každého kroku je seznam trénovacích příkladů, které odpovídají této části stromu, a seznam dostupných (zatím nevyužitých) atributů.

Nejdříve jsou ošetřeny situace, kdy jsou všechny příklady pozitivní, respektive negativní. V tomto případě je výstupem tohoto volání list s příslušnou hodnotou. Další možností je, že již všechny atributy byly využity. V tomto případě algoritmus volí nejlepší „odhad“, tedy nejčastější výstupní hodnotu z množiny příkladů.

Pokud nenastala žádná z předchozích situací, jsou postupně probírány všechny atributy a je zvolen ten, který dává největší *informační zisk*. Výstupem tohoto volání je (pod)strom jehož kořen reprezentuje zvolený atribut a hrany vedoucí z kořenu reprezentují možné hodnoty tohoto atributu. Koncovým uzlem každé hrany jsou pak podstromy vytvořené rekurzivním voláním algoritmu s tím, že z množiny atributů je odebrán právě použitý atribut a množina příkladů je omezena vždy podle příslušné hodnoty tohoto atributu. V krajním případě, kdy množina s touto hodnotou je prázdná, je ihned vrácen list s nejčastější hodnotou.

[12] Definujme speciálně pro naše účely hodnotu výrazu $0 \cdot log_2 0 = 0$.

[13] Termín informační zisk je známý také jako Kullback–Leiblerova divergence.

```python
def id3(examples, attributes):
    # rozděl příklady na pozitivní a negativní
    examples_pos, examples_neg = split_examples_by_out(examples)

    # zvol nejčastější hodnotu v testovací sadě
    most_common = int(len(examples_pos) > len(examples_neg))

    # pokud jsou všechny příklady pozitivní, vrať 1
    if examples_pos and not examples_neg: return 1

    # pokud jsou všechny příklady negativní, vrať 0
    elif examples_neg and not examples_pos: return 0

    # pokud jsou atributy prázdné, vrať nejčastější hodnotu
    elif not attributes: return most_common

    # jinak konstruuj podstrom
    else:
        # Najdi atribut s největším informačním ziskem
        max_information_gain = None

        # pro každý atribut
        for idx, attr in enumerate(attributes):
            # spočítej informační zisk
            attribute_information_gain = information_gain(attr, examples_pos, examples_neg)

            if attribute_information_gain > max_information_gain:
                # zkoumaný atribut má zatím nejlepší informační zisk
                best_attr = attr
                max_information_gain = attribute_information_gain
                attributes_without_best = attributes[:idx] + attributes[idx+1:]

        # rozděl příklady dle hodnoty zvoleného atributu
        examples_by_best = split_examples_by_attr(examples, best_attr)

        tree = [best_attr[0], {}]  # nový podstrom

        # pro každou možnou hodnotu zvoleného atributu vytvoř hranu
        for possible_value, examples_with_value in examples_by_best.iteritems():
            # pokud pro tuto hodnotu existují příklady
            if examples_with_value:
                # pod novou hranou je strom vytvořený rekurzivním voláním
                tree[1][possible_value] = id3(examples_with_value, attributes_without_best)
            else:
                # pod novou hranou je list
                tree[1][possible_value] = most_common

        return tree
```

Obrázek 5: Algoritmus ID3.

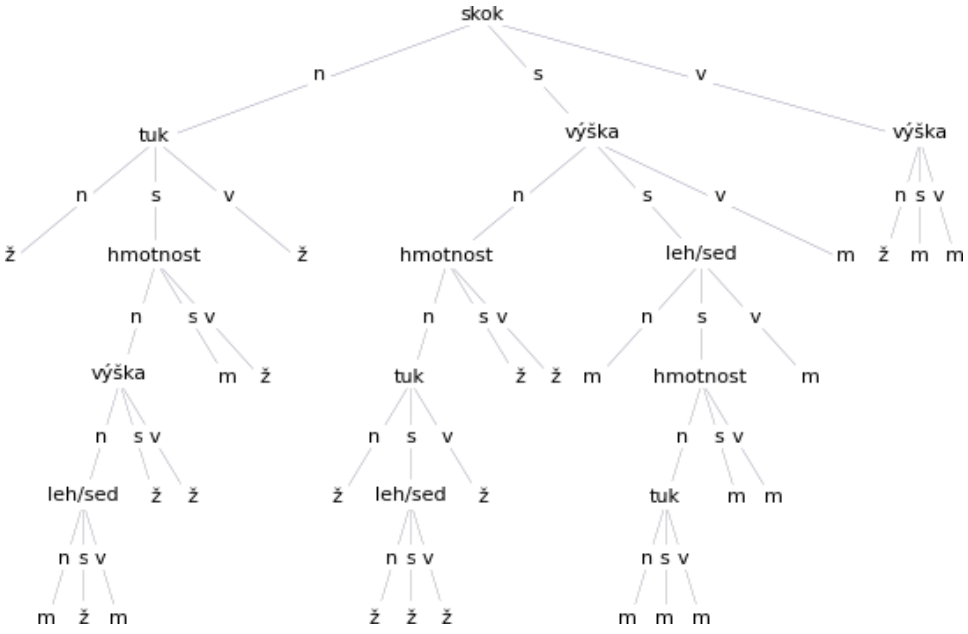
4.3.2 Použití algoritmu ID3

Následující diagram ilustruje použití algoritmu na datech UNIFITTEST. Vstupem v tomto případě byly následující atributy: hmotnost, váha, podkožní tuk a výsledky testů skoku do dálky a leh-sedů. Požadovaným výstupem pak bylo pohlaví testované osoby.

Vzhledem k tomu, že všechny vstupní atributy jsou reálná čísla, bylo nutné je nejdříve rozdělit do kategorií tak, jak je vyžaduje algoritmus ID3. Bylo použito „jednoduchého“ dělení na třetiny, ale lze si zde představit i sofistikovanější metody, např dělení s ohledem na normální rozdělení či rozdělení dle tabulek UNIFITTEST.

Další nutnou úpravou bylo doplnění chybějících hodnot (v našem případě měření podkožního tuku), chybějící atributy byly nahrazeny průměrnou hodnotou.

Diagram 6 ukazuje strom nalezený algoritmem ID3 na zkoumaných datech.

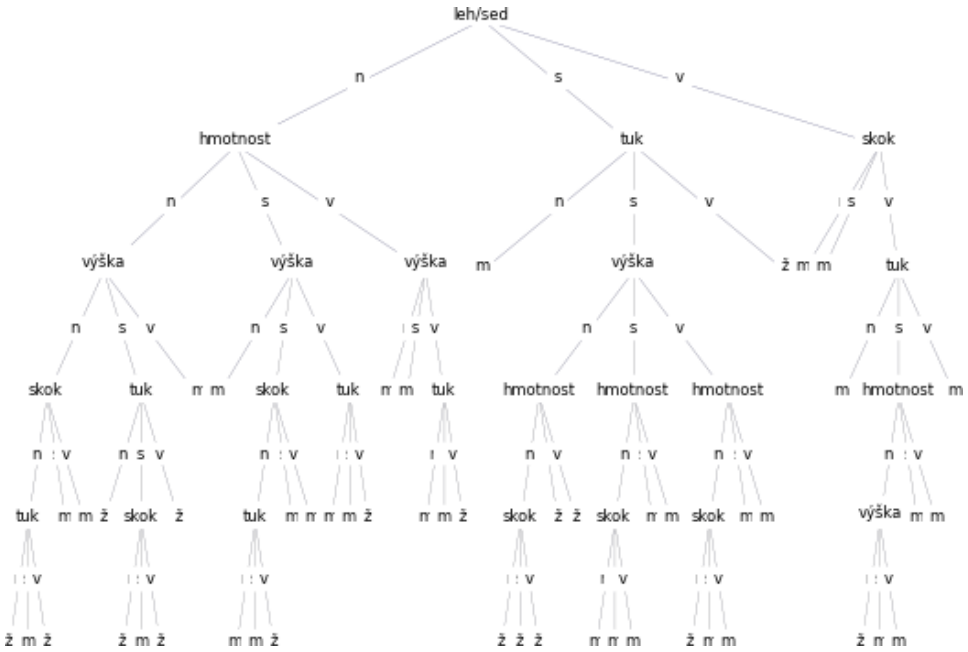


Obrázek 6: Rozhodovací strom (n - nízká, s - střední, v - vysoká).

Je vhodné upozornit na jednu zdánlivou „nesrovnalost“: v některých listech, například po posloupnosti (s-s-s-n), nalezneme podstrom, kde mají všechny tři podstromy stejnou hodnotou v listu. Nejedná se o chybu, je to důsledek situace, kdy na předposlední úrovni existují příklady obou skupin (muži i ženy), ale v poslední úrovni nejsou již dostupné další atributy podle kterých by se mohlo rozhodovat a v některé z podřízených větví dojde k „odhadu“ - použití nejčastějšího atributu, který je shodou okolností stejný jako

v sousedních větvích. Takové situace by samozřejmě bylo možné dodatečně „napravit“.

Zajímavé také může být ukázat, jaký důsledek má použití „informačního zisku“. Pro srovnání diagram 7 ukazuje strom, jak by ho algoritmus nalezl, když by funkce informačního zisku odpovídala obrácenou hodnotou (tedy „zcela chybně“). Můžeme pozorovat, že použití správné funkce hraje roli pro volbu pořadí atributů a v důsledku pak pro velikost nalezeného stromu, nicméně získané výsledky přímo chybné nejsou.



Obrázek 7: Rozhodovací strom - chybný informační zisk.

Algoritmus je možné na testovaných datech aplikovat v různých obměnách - vynechat některé vstupní atributy či zvolit jiný atribut jako výstupní.

Celkově analýza dostupných dat za pomoci rozhodovacích stromů odpovídala výsledkům, které byly zřejmé z grafů předběžného zkoumání. Žádné překvapivé závislosti v datech tato analýza neodhalila a tam kde výsledek nebyl očekávaný, jednalo se zpravidla o vztah podložený jediným či několika málo případy. Tyto situace by pak bylo zajímavé zkoumat na řádově větších vzorcích dat, které bohužel nemáme k dispozici.

Jinou, diametrálně odlišnou, analytickou metodou jsou neuronové sítě, které představuje následující kapitola.

4.4 Neuronové síťě

Neuronové sítě jsou metodou inspirovanou pozorováním fungování lidského mozku.

Pro srovnání - největší neuronové sítě v současnosti obsahují řádově 10^{10} synapsí [w2], zatímco lidský obsahuje mozek přibližně $5 \cdot 10^{14}$. Také „algoritmus“, který umělé neuronové sítě používají, je pouze hrubým zjednodušením chování skutečného neuronu (jehož přesné chování není stále přesně známé). Neuronové sítě je tedy nutné chápat jako metodu *inspirovanou* biologickou předlohou, která ovšem ani zdaleka nedosahuje její komplexnosti.

Vstupem neuronové sítě je vektor reálných čísel, která reprezentují předem určené atributy. Jejím výstupem je (v obecném případě) opět vektor číselných hodnot, které mohou být celočíselného nebo reálného typu.

Jak již bylo předesláno v předcházející sekci 4.2, základní jednotkou neuronové sítě jsou umělé *neurony* uspořádané do (obvykle) několikavrstvé sítě.

Jistou nevýhodou neuronových sítí je, že naučená pravidla lze jen těžko jednoduše a srozumitelně interpretovat. Na rozdíl od asociačních pravidel nebo rozhodovacích stromů, které jsou snadno „lidsky“ popsatelné, je neuronová síť spíše „černou skříňkou“ s komplikovanou, pro člověka jen těžko uchopitelnou, vnitřní strukturou. Trénování neuronové sítě obvykle vyžaduje delší čas, ale následné použití naučené sítě na vyhodnocení nové instance je zpravidla velice rychlé.

4.4.1 Perceptron

Perceptron je jednoduchým typem neuronové sítě obsahující jediný „neuron“. Perceptron je vhodným příkladem pro vysvětlení základních principů, které se u složitějších vícevrstvých sítí také využívají.

Vstupem perceptronu je vektor reálných čísel $\vec{x}$. Vektor $\vec{w}$ je konstanta, která přiřazuje váhu jednotlivým vstupům.

Definujme funkci *sgn* takto[14]:

$$sgn(x) = \begin{cases} 1 & \text{pro x } > 0 \\ 0 & \text{jinak} \end{cases}$$

a nechť zápis $\vec{x} \cdot \vec{y}$ pro vektory stejné dimenze n značí:

$$\vec{x} \cdot \vec{y} = \sum_{i=0}^{n} x_i \cdot y_i.$$

[14]Tato definice je odlišná od obvyklé matematické definice funkce signum, která pro $x<0 = -1$.

S pomocí výše uvedených zápisů můžeme perceptron reprezentovaný vektorem $\vec{w}$ vyjádřit jako funkci vektoru $\vec{x}$ do množiny $\{0, 1\}$ takto:

$$o(\vec{x}) = sgn(\vec{x} \cdot \vec{w}) \tag{4}$$

Učení perceptronu je postup, po jehož ukončení vektor $\vec{w}$ obsahuje takové hodnoty, že funkce o vrací požadovaný výstup pro všechny prvky množiny testovacích příkladů. Každý testovací příklad je tvořen vstupním vektorem $\vec{x}$ a požadovanou výstupní hodnotou t. Učící algoritmus prochází testovací příklady, dokud perceptron nevrací správnou hodnotu pro všechny z nich. Kdykoliv je vrácena špatná hodnota, jsou váhy $\vec{w}$ opraveny podle pravidla:

$$w_i = w_i + \eta(t - o)x_i, \tag{5}$$

kde η je konstanta ovlivňující rychlost učení.

Hlavní část algoritmu ukazuje obrázek 8. V příloze A.4 je pak dostupný kompletní zdrojový kód. Přiložená je také ukázka použití, ve které se perceptron učí rozpoznávat booleovské funkce.

```
1  def train_perceptron(n, examples, learning_rate=0.1, max_loops=1000):
2      w = [random() for i in range(n)]  # inicializace vah náhodným číslem
3
4      E = True  # inicializace chyby E
5      loop = 0  # čítač cyklů
6      while E:  # dokud je chyba alespoň na jednom vstupu
7          loop += 1
8          if loop > max_loops:  # při překročení maximálního počtu cyklů
9              return None  # ukonči s prázdným výstupem
10
11         E = False
12         for x, t in examples:  # vstupní vektor x, očekávaný výstup t
13             e = t - o(x, w)  # hodnota chyby pro tento příklad
14             E = E or (e != 0)  # pokud chyba, nastav i globální chybu
15
16             if E:
17                 for i in range(n):  # pro každý vstup/váhu
18                     w[i] += e * x[i] * learning_rate  # upravit váhy w
19     return w
```

Obrázek 8: Algoritmus učení perceptronu.

Lze dokázat, že algoritmus je finitní za předpokladu, že trénovací příklady jsou lineárně oddělitelné[15] a konstanta η má dostatečně malou hodnotu [7, kap. 4.4.2]. Předpoklad lineární oddělitelnosti je ovšem značně

[15] Dvě množiny jsou lineárně oddělitelné v n-dimensionálním prostoru, pokud je můžeme oddělit nadrovinou. Speciálně v dvourozměrném prostoru, tedy pokud jsou oddělitelné přímkou.

omezující (například booleovské funkce XOR a XNOR lineárně oddělitelné nejsou.) Algoritmus se v případě lineárně neoddělitelných trénovacích dat zacyklí[16].

4.4.2 Vícevrstvé sítě a algoritmus Backpropagation

Vícevrstvé neuronové sítě nejsou na rozdíl od jednoduchého perceptronu omezené jen na lineárně oddělitelné funkce.

Základní jednotkou vícevrstvé sítě je **sigmoid** - jednotka podobná perceptronu, od něhož se však liší výstupní funkcí. V místě, kde perceptron používá funkci *sgn*, která je kvůli své nespojitosti nevhodná, aplikuje sigmoid funkci:

$$\sigma(x) = \frac{1}{1 + e^{-y}}. \tag{6}$$

Dále definujme chybovou funkci E takto:

$$E(\vec{w}) = \frac{1}{2} \sum_{d \in D} \sum_{k \in out} (t_{kd} - o_{kd})^2, \tag{7}$$

kde D je množina trénovacích příkladů, out je množina výstupních jednotek sítě, t_{kd} a o_{kd} jsou očekávaný, respektive vypočítaný výstup jednotky k při zpracování trénovacího příkladu d.

Algoritmus *Backpropagation* v prvním kroku vytvoří neuronovou síť obsahující předem daný počet vrstev, každou s předem daným počtem jednotek (sigmoidů). Výstup každé jednotky (s výjimkou výstupní vrstvy) pak směřuje do každé jednotky vrstvy následující. Tomuto propojení (synapsi) algoritmus v prvním kroku přiřadí náhodně váhu (obvykle malé číslo např. mezi -0.5 a 0.5). Dokud není splněna ukončující podmínka, algoritmus opakovaně používá instance trénovací množiny k „učení“ tak, že každou instanci vyhodnotí a poté zpětně propaguje chybu a příslušně opraví váhy synapsí:

1. Aplikuj neuronovou síť na vstup $\vec{x}$ a spočítej výstup o_u pro každou jednotku v síti.

2. Pro každou výstupní jednotku k spočítej chybu δ_k:

$$\delta_k = o_k(1 - k)(t_k - o_k). \tag{8}$$

[16] V ukázce v příloze A.4 k takovému zacyklení dochází.
Existuje variace tohoto algoritmu - „Gradient Descent“, která je finitní i v případě lineární neoddělitelnosti, po jeho ukončení ale perceptron nevrací správné hodnoty na všech testovacích vzorcích. Pro podrobnosti viz [7, kap. 4.4.3]

3. Pro každou jednotku h ve skrytých vrstvách spočítej chybu δ_h

$$\delta_h = o_h(1 - o_h) \sum_{k \in outputs} w_{kh}\delta_k. \tag{9}$$

4. Pro každou synapsi v síti uprav její váhu w_{ji}

$$w_{ji} = w_{ji} + \eta\delta_j x_{ji}, \tag{10}$$

kde x_{ji} označuje vstup z jednotky i do jednotky j, w_{ji} odpovídající váhu a η je učící konstanta (*learning rate*).

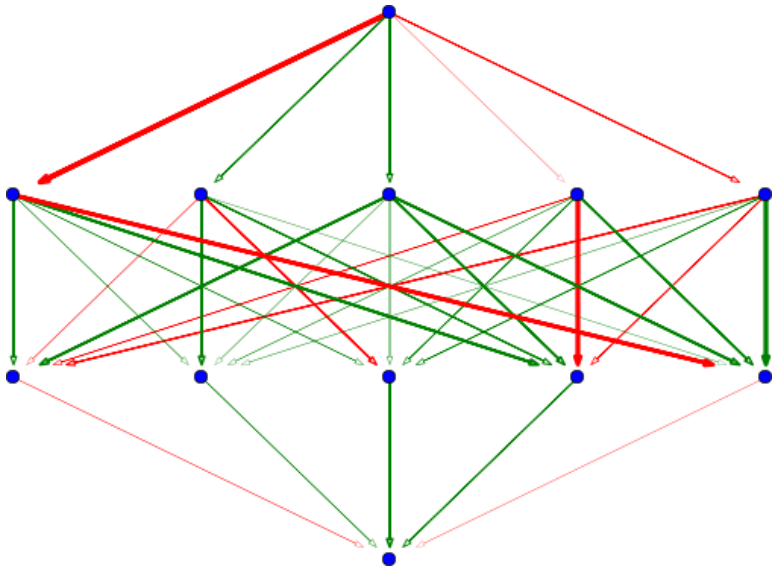
Ukončující podmínka může být implementována triviálně jako pevný počet cyklů, ale je možné použít i sofistikovanějších metod, např. sledováním hodnoty chybové funkce na validační množině.

4.4.3 Využití neuronových sítí na datech UNIFITTEST

V rámci práce byla nejdříve implementována neuronová síť tak, jak ji popisuje algoritmus *Backpropagation* (zdrojový kód je v příloze A.5.1). Nepodařilo se ji ale na dostupných datech vytrénovat tak, aby dávala smysluplné výsledky. Není zcela jisté, zda byla síť implementována chybně či zda byl problém v množství dostupných dat. Na některých jednoduchých úkolech (např. trénování booleovských funkcí) dávala výsledky správné.

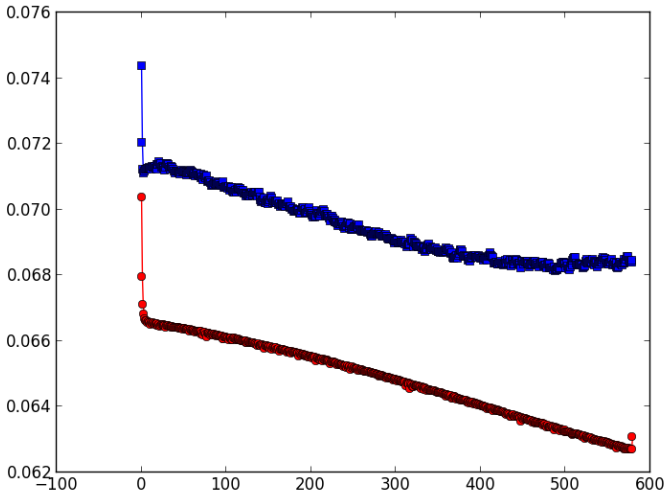
V rámci druhého pokusu byla využita knihovna PyBrain [sw4] implementující řadu funkcionalit neuronových sítí včetně algoritmu *Backpropagation*. Bohužel ani s využitím této knihovny se nepodařilo dosáhnout dobrých výsledků. Naučená síť dokázala správně odpovídat na data z trénovací množiny, ale na jiných datech rozumné výsledky nedávala. Pravděpodobně v průběhu algoritmu docházelo velice rychle k přeučení. Nejspíše proto, že testovací množina je (z pohledu data miningu) velice malá.

Diagram 9 ilustruje neuronovou síť zkonstruovanou na datech UNIFITTEST za pomoci knihovny PyBrain. Barevné šipky reprezentují váhy přiřazené jednotlivým synapsím. Červená odpovídá kladné (v naší ukázce přispívající „ženské“ složce), zelená záporné („mužské“) váze. Tloušťka šipky odpovídá její absolutní hodnotě. Jednotka v prvním řádku je „předsudek“ (*bias*), který modifikuje vstupní data o konstantu. Druhý řádek odpovídá vstupním datům v pořadí: podkožní tuk, hmotnost, výška, skok daleký, leh-sedy. Poslední je pak výstupní jednotka, která se snaží rozhodnout pohlaví subjektu.

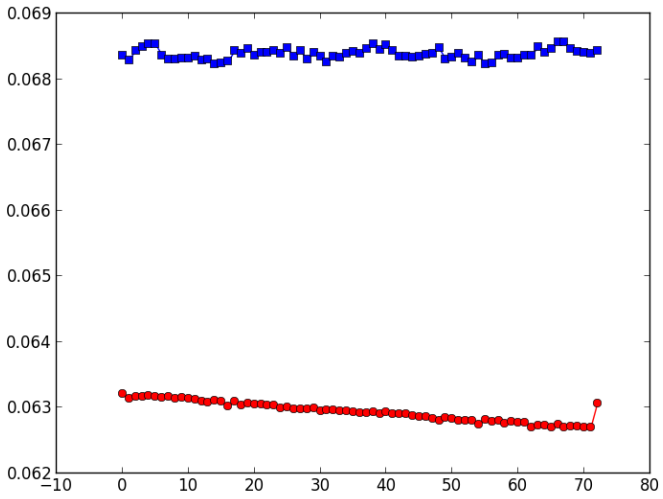


Obrázek 9: Neuronová síť.

Graf 10 ukazuje průběh chybové funkce v průběhu algoritmu, nižší hodnoty odpovídají trénovacímu vzorku dat, vyšší pak testovací množině. Následující graf 11 poskytuje detailnější pohled na posledních 10 % učících cyklů. Zde můžeme nahlédnout, jak algoritmus rozhoduje o ukončení učení - v situaci kdy chyba na testovací množině přestává klesat.

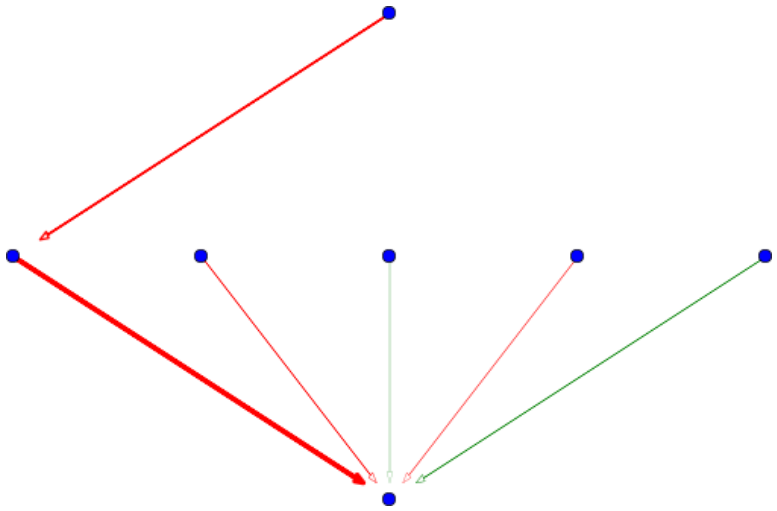


Obrázek 10: Neuronová síť - chyba - celkový průběh.



Obrázek 11: Neuronová síť - chyba - posledních 10 %.

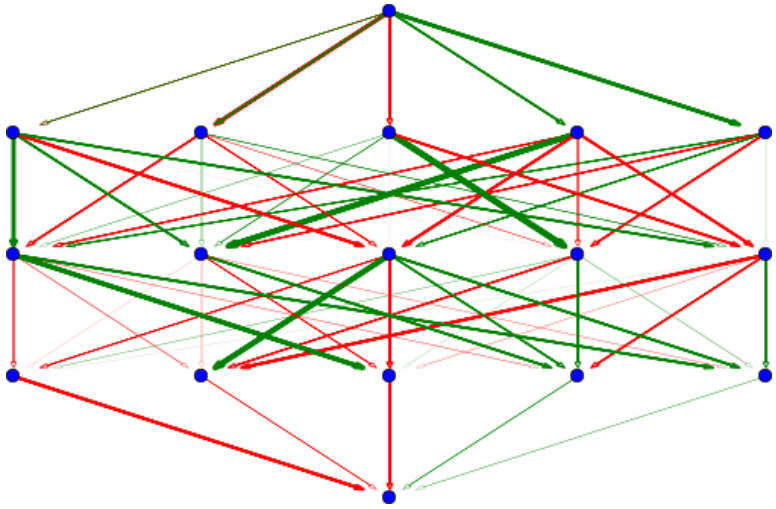
Diagram 12 ukazuje nalezené váhy v případě použití jednovrstvé sítě. Prvnímu vstupu (druhý řádek vlevo), kterému je pak zřejmě přiřazena nejvyšší váha, odpovídá množství podkožního tuku, což je výsledek známý již z předběžného pozorování a zde „znovuobjevený“ algoritmem „Backpropagation“. Víme ovšem, že jednovrstvá síť (tedy v podstatě perceptron) nedokáže zpracovat lineárně neoddělitelné množiny a je tedy ve složitějších situacích nepoužitelný.



Obrázek 12: Neuronová síť - jednovrstvá.

Poslední diagram 13 ilustruje třívrstvou neuronovu síť. Její složitost

naznačuje, že pokoušet se popisovat a nějakým způsobem „ručně“ verifikovat správnost nalezené sítě je téměř nemožné.



Obrázek 13: Neuronová síť - třívrstvá.

Přestože použité algoritmy dokončily svůj běh zdánlivě úspěšně, naučené sítě nedokázaly vyhodnocovat nové instance a celkově tak musíme použití neuronových sítí na těchto datech hodnotit jako neúspěšné. Pravděpodobnou příčinou je příliš malý vzorek dat.

5 Závěr

V rámci této práce byla představena problematika dobývání znalostí z databází se zaměřením na vybrané analytické algoritmy.

Za úspěšné můžeme považovat především podrobné představení a vlastní implementaci algoritmů pro konstrukci rozhodovacích stromů a perceptronu. Představen a použit byl také algoritmus využívající neuronové sítě.

Velice cenné jsou praktické zkušenosti získané srovnáním dvou odlišných přístupů:

Rozhodovací stromy jsou metodou, která je mnohem snáze „uchopitelná“. Zpracování algoritmu, trasování jeho postupu či odhalování chyb bylo poměrně neproblematické. Stejně tak i výstup algoritmu je snadno čitelnou strukturou.
Velkou roli zde hraje skutečnost, že rozhodovací stromy ve své základní podobě pracují s diskrétními hodnotami, což je dobré pro „přehledost“, zároveň je to ovšem jejich významné omezení.

Neuronové sítě mají relativně náročnější koncept. Povahou jsou to „černé skříňky“ a jejich interní reprezentace znalostí je příliš komplikovaná pro přímé „čtení“. To výrazně komplikuje sledování průběhu algoritmu a tedy i případné odhalování chyb. Stejně tak nalezené znalosti není snadné přehledně reprezentovat.
To, že neuronové sítě pracují ve spojitém prostoru a jejich schopnost samostatného nalezení vhodné reprezentace znalostí, nabízí ovšem široké uplatnění a má řadu praktických aplikací, kde by rozhodovací stromy či asociační pravidla použitelné nebyly.

Velice praktickými se také ukázaly automatizované vizualizace zpracovaných algoritmů, které značně zpřehlednily vyhodnocování výsledků na různých vstupech nebo různých konfiguracích algoritmů.

Je nutné konstatovat, že analýza dostupných dat z výsledků UNIFITTEST nové zajímavé znalosti neodhalila. Ukazuje se, že pro získání zajímavých výsledků metodami data miningu by bylo nutné mít k dispozici řádově větší vzorek dat.
Pro analýzu dat podobného typu jsou zřejmě praktičtější rozhodovací stromy (či asociační pravidla) a použití neuronových sítí na problém tohoto typu se ukázalo jako nevhodné.

Plánovanou návazností na tuto práci je praktické vytvoření webové aplikace pro sběr dat UNIFITTEST, čímž by se zásadně zjednodušily fáze čiš-

tění a transformace dat, a analytické algoritmy by pak bylo možné spouštět poloautomatizovaně či zcela automaticky.

Lze si také představit využití připraveného systému pro analýzu zcela odlišných dat, implementované algoritmy jsou v tomto ohledu univerzální. Nicméně by ale bylo vhodné zvážit i použití jiných implementací, kterých je dostupná řada a často mají implementované různé „nadstavby“ či optimalizace nad rámec základního algoritmu. V tomto ohledu jsou vytvořené implementace poměrně „naivní“ a jejich přínos spočívá spíše v pochopení problematiky, než přímé praktické využitelnosti.

Řada oblastí, kterých se tato práce více či méně dotkla, nabízí množství podnětů k dalšímu zkoumání. Velice zajímavé by bylo podrobnější seznámení s možnostmi neuronových sítí a evolučních algoritmů, jejichž inspirace biologickými procesy je pozoruhodná.

Literatura

[1] Ron Zacharski. *A Programmer's Guide to Data Mining*. http://guidetodatamining.com/. 2013.

[2] Jan Walter. "Selected Data Mining Methods and Their Applicability to the Television Audience Monitoring Data in the Czech Republic". MA thesis. Department of Logic, Faculty of Philosophy and Arts, Charles University, Prague, 2006.

[3] Petr Berka. *Dobývání znalostí z databází*. Academia, 2003.

[4] Jitka Chytráčková et al. *UNIFITTEST (6-60), Příručka pro manuální a počítačové hodnocení základní motorické výkonnosti a vybraných charakteristik tělesné stavby mládeže a dospělých v České republice*. 2002.

[5] D. J. Hand, H. Mannila, and P. Smyth. *Principles of Data Mining*. A Bradford book. Mit Press, 2001.

[6] Pete Chapman et al. *CRISP-DM 1.0 Step-by-step data mining guide*. Tech. rep. The CRISP-DM consortium, 2000. URL: http://www.crisp-dm.org/CRISPWP-0800.pdf.

[7] Tom M. Mitchell. *Machine Learning*. The McGraw-Hill Companies, Inc., 1997.

Softwarové nástroje

[sw1] Python Software Foundation. *Python Language Reference, version 2.7*. http://www.python.org. 2013.

[sw2] John Machin. *Python package xlrd*. http://www.python-excel.org/. 2013.

[sw3] Konrad Rudolph and Geoffrey Poore. *LaTeX package minted*. https://github.com/gpoore/minted. 2013.

[sw4] Tom Schaul et al. "PyBrain". In: *Journal of Machine Learning Research* (2010).

[sw5] J. D. Hunter. "Matplotlib: A 2D graphics environment". In: *Computing In Science & Engineering* 9.3 (2007), pp. 90–95. URL: http://matplotlib.org/.

Webové zdroje

[w1] IBM. *IBM - What is big data?* http://www.ibm.com/software/data/bigdata/. 2013.

[w2] NVIDIA. *NVIDIA news website: Researchers Deploy GPUs To Build World's Largest Artificial Neural Network*. http://nvidianews.nvidia.com/Releases/Researchers-Deploy-GPUs-to-Build-World-s-Largest-Artificial-Neural-Network-9c7.aspx. 2013.

[w3] Wikipedia. *Cross Industry Standard Process for Data Mining*. http://en.wikipedia.org/wiki/Cross_Industry_Standard_Process_for_Data_Mining. 2013.

[w4] Wikipedia. *Mark Kryder*. http://en.wikipedia.org/wiki/Mark_Kryder. 2013.

A Přílohy

A.1 Transformace dat

Zdrojový kód použitý pro import dat z formátu .xlsx do interní datové struktury a související základní čištění.

```
# -*- coding: utf-8 -*-
from datetime import date, datetime
from xlrd import open_workbook, xldate_as_tuple

structure_columns = [
    'timestamp',
    'author_first_name',
    'author_last_name',
    'author_email',
    'subject_sex',
    'subject_first_name',
    'subject_last_name',
    'subject_birth',
    'subject_location',
    'subject_sport_activity',
    None,
    'test_1',  # skok daleký z místa [a]
    'test_2',  # leh-sed [a]
    'test_3',  # běh 12 min [s]
    'test_4',  # člunkový běh [s]
    'test_5',  # chůze 2 km [s]
    None,
    'test_6',  # člunkový běh do 14 let [n]
    'test_7',  # shyby [s]
    'test_8',  # výdrž ve shybu [n]
    'test_9',  # předklon [n]
    'subject_height',
    'subject_weight',
    'subject_body_fat',
    'author_origin',
]

structure = {}
for idx, key in enumerate(structure_columns):
    structure[key] = idx

class Result(object):
    pass

# parse z .xlsx
def parse_file(filename):
    wb = open_workbook(filename=filename)
    sheet = wb.sheet_by_index(0)

    results = []
    for row_idx in range(1, sheet.nrows):
        row = sheet.row(row_idx)

        result = Result()
        result.subject_sex = {u'muž': 0, u'žena': 1}.get(row[structure['subject_sex']].value)
        result.subject_birth = date(*xldate_as_tuple(row[structure['subject_birth']] \
            .value, wb.datemode)[:3])

```

```
        result.subject_weight = row[structure['subject_weight']].value
        if isinstance(result.subject_weight, unicode):
            result.subject_weight = float(result.subject_weight.replace('kg', ''))

        result.subject_height = row[structure['subject_height']].value
        if isinstance(result.subject_height, unicode):
            result.subject_height = float(result.subject_height.replace('cm', ''))

        result.subject_body_fat = row[structure['subject_body_fat']].value
        if result.subject_body_fat:
            if isinstance(result.subject_body_fat, str) or \
               isinstance(result.subject_body_fat, unicode):
                result.subject_body_fat = float(result.subject_body_fat.replace('mm', '') \
                    .replace(',', '.'))
        else:
            result.subject_body_fat = None

        result.test_1 = row[structure['test_1']].value
        if isinstance(result.test_1, unicode):
            result.test_1 = float(result.test_1.replace('cm', ''))

        result.test_2 = row[structure['test_2']].value

        results += [result]
    return results
```

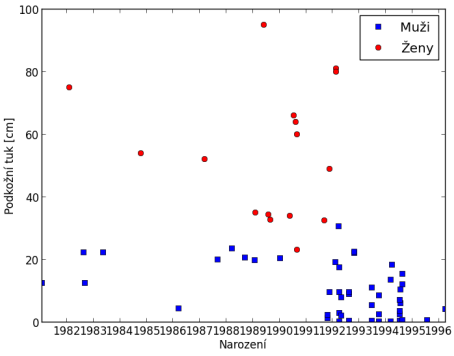
A.2 Předběžná analýza

Funkce generující vizualizace dostupných atributů po dvojicích „každý s každým“ a její výstup.

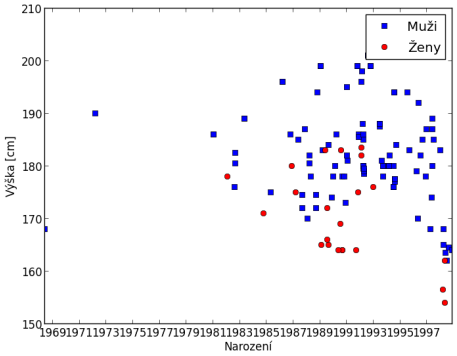
A.2.1 Předběžná analýza - zdroj

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import matplotlib.pyplot as plt
from matplotlib.legend import Legend
import xlsx_import

def plot_2d_categorical(x, y, out, data, attrs, attrs_categorical):
    fig = plt.figure()
    ax = fig.add_subplot(111)
    ax.set_xlabel(attrs[x][0])
    ax.set_ylabel(attrs[y][0])
    plot_repr = dict((attr, None) for attr in attrs_categorical[out][1].keys())
    for result in data:
        plot_repr[getattr(result, out)], =  plt.plot(
            getattr(result, x),
            getattr(result, y),
            attrs_categorical[out][1][getattr(result, out)][1]
        )
    ax.legend(plot_repr.values(), [v[0] for v in attrs_categorical[out][1].values()], numpoints=1)
    plt.savefig('../graphs/prep/ternary/%s_%s_%s.png' % (x, y, out), bbox_inches='tight')
    plt.clf()

def plot(data):
    attrs = {
        'test_1': (u'Skok daleký [cm]',),
        'test_2': (u'Leh/sed [#]',),
        'subject_weight': (u'Hmotnost [kg]',),
        'subject_height': (u'Výška [cm]',),
        'subject_body_fat': (u'Podkožní tuk [cm]',),
        'subject_birth': (u'Narození',),
    }

    attrs_categorical = {
        'subject_sex': (u'Pohlaví', {0: (u'Muži', 'bs'), 1: (u'Ženy', 'ro')}),
    }

    attr_keys = attrs.keys()
    for ii, x in enumerate(attr_keys):
        for y in attr_keys[ii+1:]:
            for out in attrs_categorical.keys():
                plot_2d_categorical(x, y, out, data, attrs, attrs_categorical)

if __name__ == '__main__':
    data = xlsx_import.parse_file(r'../data/data2.xlsx')
    plot(data)
```

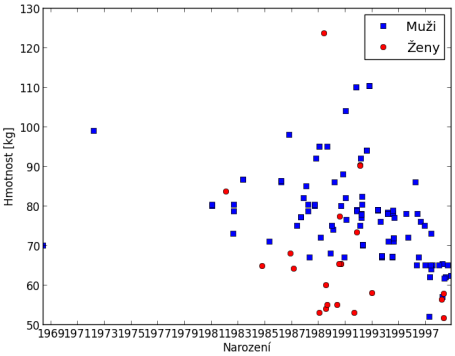
A.2.2 Předběžná analýza - výstup



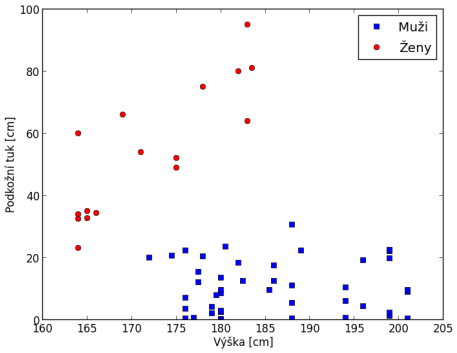
Obrázek A1



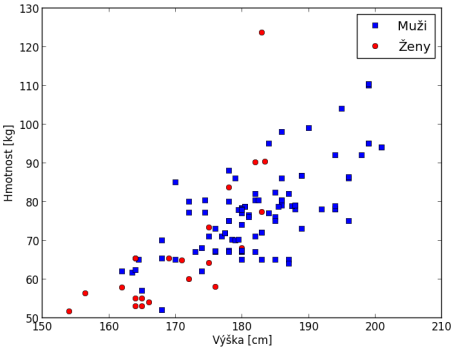
Obrázek A2



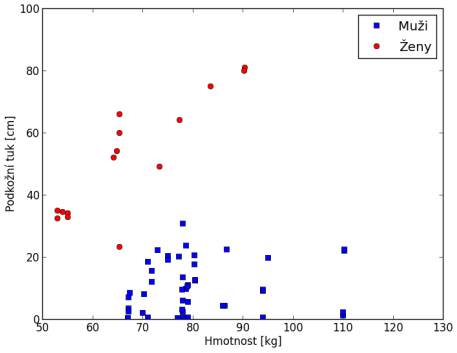
Obrázek A3



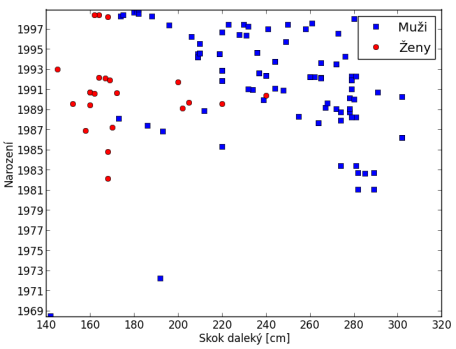
Obrázek A4



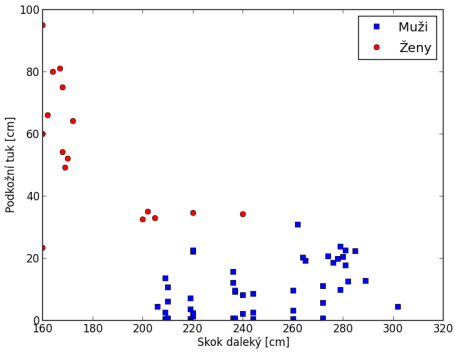
Obrázek A5



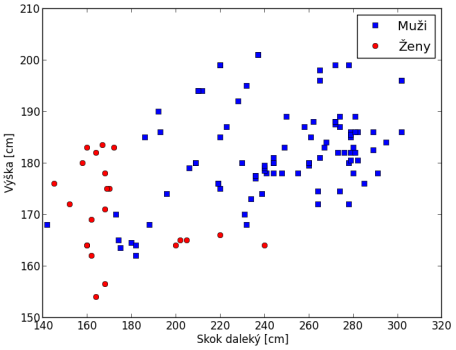
Obrázek A6



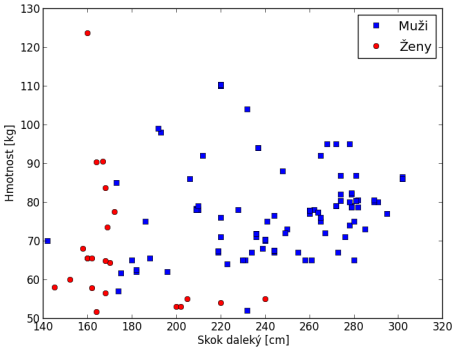
Obrázek A7



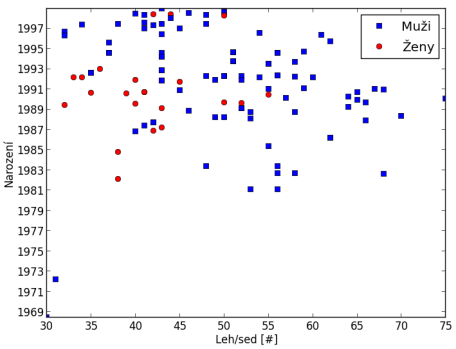
Obrázek A8



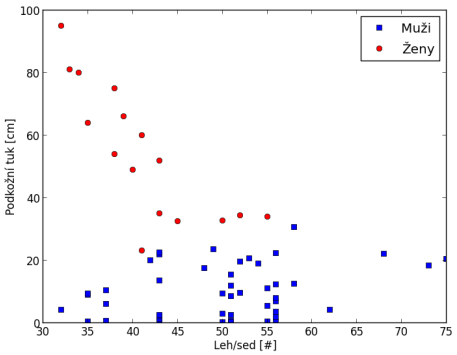
Obrázek A9



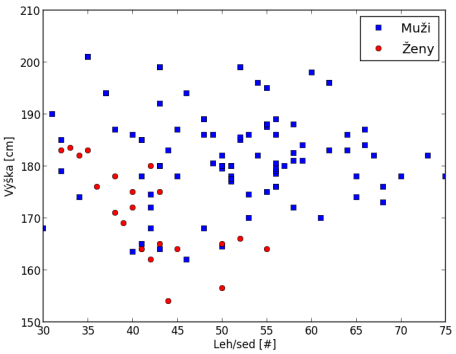
Obrázek A10



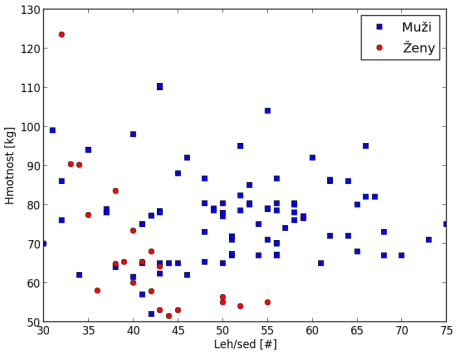
Obrázek A11



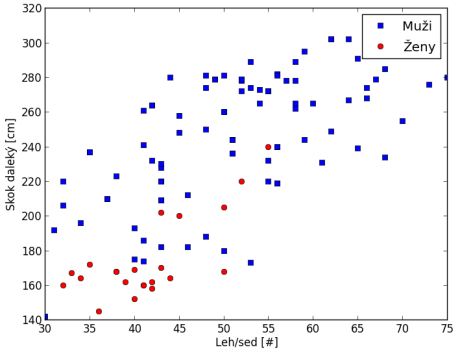
Obrázek A12



Obrázek A13



Obrázek A14



Obrázek A15

A.3 Rozhodovací stromy

Implementace algoritmu ID3 a jeho použití na data UNIFITTEST včetně funkcí použitých pro vizualicaci nalezeného rozhodovacího stromu.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import matplotlib
import matplotlib.pyplot as plt
from matplotlib.legend import Legend
from math import log
from numpy import float

def id3(examples, attributes):
    # rozděl příklady na pozitivní a negativní
    examples_pos, examples_neg = split_examples_by_out(examples)

    # zvol nejčastější hodnotu v testovací sadě
    most_common = int(len(examples_pos) > len(examples_neg))

    # pokud jsou všechny příklady pozitivní, vrať 1
    if examples_pos and not examples_neg: return 1

    # pokud jsou všechny příklady negativní, vrať 0
    elif examples_neg and not examples_pos: return 0

    # pokud jsou atributy prázdné, vrať nejčastější hodnotu
    elif not attributes: return most_common

    # jinak konstruuj podstrom
    else:
        # Najdi atribut s největším informačním ziskem
        max_information_gain = None

        # pro každý atribut
        for idx, attr in enumerate(attributes):
            # spočítej informační zisk
            attribute_information_gain = information_gain(attr, examples_pos, examples_neg)

            if attribute_information_gain > max_information_gain:
                # zkoumaný atribut má zatím nejlepší informační zisk
                best_attr = attr
                max_information_gain = attribute_information_gain
                attributes_without_best = attributes[:idx] + attributes[idx+1:]

        # rozděl příklady dle hodnoty zvoleného atributu
        examples_by_best = split_examples_by_attr(examples, best_attr)

        tree = [best_attr[0], {}]  # nový podstrom

        # pro každou možnou hodnotu zvoleného atributu vytvoř hranu
        for possible_value, examples_with_value in examples_by_best.iteritems():
            # pokud pro tuto hodnotu existují příklady
            if examples_with_value:
                # pod novou hranou je strom vytvořený rekurzivním voláním
                tree[1][possible_value] = id3(examples_with_value, attributes_without_best)
            else:
                # pod novou hranou je list
                tree[1][possible_value] = most_common

        return tree
```

```
57
58
59 def entropy(*args):
60     out = 0
61     total = sum(args)
62     for count in args:
63         if count != 0:
64            p = float(count)/total
65            out -= p * log(p, 2)
66     return out
67
68
69 def information_gain(attribute, examples_pos, examples_neg):
70     division_pos = {}
71     division_neg = {}
72
73     for examples, examples_division in ((examples_pos, division_pos), (examples_neg, division_neg)):
74         for possible_value in attribute[1]:
75             examples_division[possible_value] = 0
76
77         for example in examples:
78             examples_division[example[0][attribute[0]]] += 1
79
80     gain = entropy(len(examples_pos), len(examples_neg))
81
82     for possible_value in attribute[1]:
83         gain_delta = (division_pos[possible_value] + division_neg[possible_value])
84         gain_delta *= entropy(division_pos[possible_value], division_neg[possible_value])
85         gain_delta /= len(examples_pos) + len(examples_neg)
86         gain -= gain_delta
87     return gain
88
89
90 def split_examples_by_out(examples):
91     examples_pos = []
92     examples_neg = []
93     for example in examples:
94         if example[1]:
95             examples_pos += [example]
96         else:
97             examples_neg += [example]
98     return examples_pos, examples_neg
99
100 def split_examples_by_attr(examples, attr):
101         examples_by_attr = {}
102         for possible_value in attr[1]:
103             examples_by_attr[possible_value] = []
104
105         for example in examples:
106             examples_by_attr[example[0][attr[0]]] += [example]
107         return examples_by_attr
108
109
110 def calculate_subtree_widths(tree):
111     if isinstance(tree, int):
112         width = 1
113     else:
114         width = sum([calculate_subtree_widths(subtree) for subtree in tree[1].values()])
115         tree.append(width)
116
117     return width
118
```

```
119
120 def plot_tree(tree):
121     calculate_subtree_widths(tree)
122     plot_subtree(tree, 1, 1)
123     plt.gca().invert_yaxis()
124     plt.savefig('../graphs/tree/tree.png', bbox_inches='tight', pad_inches=0)
125
126
127 def plot_subtree(tree, top, left, parent=None, attr_value=None):
128     if isinstance(tree, int):
129         center = left
130         plt.plot(center, top, marker='.', color='white')
131         plt.text(center, top, out[tree],
132             backgroundcolor='w', bbox={'pad': 5, 'ec': 'w'}, va='bottom', ha='center')
133     else:
134         center = left + (tree[2] * 0.5) - 0.5
135         plt.plot(center, top, marker='.', color='white')
136         plt.text(center, top, attr_names[tree[0]][0],
137             backgroundcolor='w', bbox={'pad': 5, 'ec': 'w'}, va='bottom', ha='center')
138
139         for val, subtree in tree[1].iteritems():
140             plot_subtree(subtree, top+1, left, center, attr_names[tree[0]][1][val])
141             left += 1 if isinstance(subtree, int) else subtree[2]
142
143     if parent:
144         ax.arrow(
145             parent,
146             top-1,
147             (center-parent),
148             1,
149             edgecolor='gray',
150             linewidth=0.1,
151         )
152         if attr_value:
153             plt.text((parent+center)/2.0, top-0.5, attr_value,
154                 backgroundcolor='w', bbox={'pad': 5, 'ec': 'w'}, va='bottom', ha='center')
155
156
157
158 if __name__ == '__main__':
159     import xlsx_import
160     data = xlsx_import.parse_file(r'../data/data2.xlsx')
161
162     attr_names = [
163         (u'hmotnost', (u'n', u's', u'v')),
164         (u'výška', (u'n', u's', u'v')),
165         (u'tuk', (u'n', u's', u'v')),
166         (u'skok', (u'n', u's', u'v')),
167         (u'leh/sed', (u'n', u's', u'v')),
168     ]
169     out = (u'm', u'ž')
170     attributes = [(idx, range(len(attr[1]))) for (idx, attr) in enumerate(attr_names)]
171
172     examples = []
173     IN_ATTR_COUNT = len(attr_names)
174     examples_min = [float('inf') for ii in range(IN_ATTR_COUNT)]
175     examples_max = [float('-inf') for ii in range(IN_ATTR_COUNT)]
176     for item in data:
177         example = (
178             [
179                 item.subject_weight,
180                 item.subject_height,
```

```
                item.subject_body_fat,
                item.test_1,
                item.test_2,
            ],
            int(item.subject_sex)
        )
        for idx in xrange(IN_ATTR_COUNT):
            if not (example[0][idx] is None):
                examples_min[idx] = min(examples_min[idx], example[0][idx])
                examples_max[idx] = max(examples_max[idx], example[0][idx])

        examples += [example]

    # rozděl data do kategorií
    for example in examples:
        for idx in xrange(IN_ATTR_COUNT):
            categories = len(attributes[idx][1])
            if example[0][idx] is None:
                norm_value = int(categories/2)
            else:
                norm_value = min(
                    int(
                        (example[0][idx] - examples_min[idx]) * (categories+1) \
                        / (examples_max[idx] - examples_min[idx])
                    ),
                     categories-1
                )
            example[0][idx] = norm_value

    tree = id3(examples, attributes)

    fig = plt.figure()
    ax = fig.add_subplot(111, xmargin=0.01, ymargin=0.01)
    plt.axis('off')
    matplotlib.rcParams.update({'font.size': 8})

    plot_tree(tree)
```

A.4 Perceptron

Implementace perceptronu a ukázka jeho učení na příkladu booleovských funkcí.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
from random import random

# Vstup:
# n ... počet vstupů
# examples ... seznam příkladů každý příklad ve tvaru dvojice:
#  ((vstup_1, vstup_2, ..., vstup_n), ocekavany_vystup)
# learning_rate ... učící konstanta, neopovinný parametr
# max_loops ... maximální počet cyklů před neúspěšným ukončením
#
# Výstup:
# n-tice reprezentující váhy perceptronu
def train_perceptron(n, examples, learning_rate=0.1, max_loops=1000):
    w = [random() for i in range(n)]  # inicializace vah náhodným číslem

    E = True  # inicializace chyby E
    loop = 0  # čítač cyklů
    while E:  # dokud je chyba alespoň na jednom vstupu
        loop += 1
        if loop > max_loops:  # při překročení maximálního počtu cyklů
            return None  # ukonči s prázdným výstupem

        E = False
        for x, t in examples:  # vstupní vektor x, očekávaný výstup t
            e = t - o(x, w)  # hodnota chyby pro tento příklad
            E = E or (e != 0)  # pokud chyba, nastav i globální chybu

            if E:
                for i in range(n):  # pro každý vstup/váhu
                    w[i] += e * x[i] * learning_rate  # upravit váhy w
    return w

# výstupní funkce
def o(x, w):
    # test zda je vektorový součin vah větší než daná mez
    return sum([w[i]*x[i] for i in range(len(w))]) > 0.5


# testovací příklady
boolean_functions = [
    ('FALSE', (0, 0, 0, 0)),
    ('AND', (0, 0, 0, 1)),
    ('X > Y', (0, 0, 1, 0)),
    ('X', (0, 0, 1, 1)),
    ('Y > X', (0, 1, 0, 0)),
    ('Y', (0, 1, 0, 1)),
    ('XOR', (0, 1, 1, 0)),
    ('OR', (0, 1, 1, 1)),
    ('NOR', (1, 0, 0, 0)),
    ('XNOR', (1, 0, 0, 1)),
    ('NOT Y', (1, 0, 1, 0)),
    ('Y -> X', (1, 0, 1, 1)),
    ('NOT X', (1, 1, 0, 0)),
    ('X -> Y', (1, 1, 0, 1)),
    ('NAND', (1, 1, 1, 0)),
```

```
57      ('TRUE', (1, 1, 1, 1)),
58  ]
59
60  for name, (x1, x2, x3, x4) in boolean_functions:
61      # trénovací příklady
62      # první vstupní hodnota x
63      # druhá vstupní hodnota y
64      # třetí vstupní hodnota vždy 1
65      #    přidává perceptronu stupeň volnosti, který nahrazuje předsudek (bias)
66      examples = [
67          ((0, 0, 1), x1),
68          ((0, 1, 1), x2),
69          ((1, 0, 1), x3),
70          ((1, 1, 1), x4),
71      ]
72
73      print name
74      w = train_perceptron(3, examples)
75      if w:
76          for (x, y, _), expected_result in examples:
77              calculated_result = int(o([x, y, _], w))
78              if calculated_result == expected_result:
79                  info = 'ok'
80              else:
81                  info = 'chyba!'
82              print '[%s, %s] -> %s (%s)' % (x, y, calculated_result, info)
83      else:  # perceptron se nepodařilo vytrénovat
84          print 'zacykleno'  # pravděpodobně není lineárně oddělitelné
85      print ''
```

A.4.1 Perceptron - výsledky testů

```
FALSE
[0, 0] -> 0 (ok)
[0, 1] -> 0 (ok)
[1, 0] -> 0 (ok)
[1, 1] -> 0 (ok)

AND
[0, 0] -> 0 (ok)
[0, 1] -> 0 (ok)
[1, 0] -> 0 (ok)
[1, 1] -> 1 (ok)

X > Y
[0, 0] -> 0 (ok)
[0, 1] -> 0 (ok)
[1, 0] -> 1 (ok)
[1, 1] -> 0 (ok)

X
[0, 0] -> 0 (ok)
[0, 1] -> 0 (ok)
[1, 0] -> 1 (ok)
[1, 1] -> 1 (ok)

Y > X
[0, 0] -> 0 (ok)
[0, 1] -> 1 (ok)
[1, 0] -> 0 (ok)
[1, 1] -> 0 (ok)

Y
[0, 0] -> 0 (ok)
[0, 1] -> 1 (ok)
[1, 0] -> 0 (ok)
[1, 1] -> 1 (ok)

XOR
zacykleno

OR
[0, 0] -> 0 (ok)
[0, 1] -> 1 (ok)
[1, 0] -> 1 (ok)
[1, 1] -> 1 (ok)

NOR
[0, 0] -> 1 (ok)
[0, 1] -> 0 (ok)
[1, 0] -> 0 (ok)
[1, 1] -> 0 (ok)

XNOR
zacykleno

NOT Y
[0, 0] -> 1 (ok)
[0, 1] -> 0 (ok)
[1, 0] -> 1 (ok)
[1, 1] -> 0 (ok)

Y -> X
[0, 0] -> 1 (ok)
[0, 1] -> 0 (ok)
[1, 0] -> 1 (ok)
[1, 1] -> 1 (ok)

NOT X
[0, 0] -> 1 (ok)
[0, 1] -> 1 (ok)
[1, 0] -> 0 (ok)
[1, 1] -> 0 (ok)

X -> Y
[0, 0] -> 1 (ok)
[0, 1] -> 1 (ok)
[1, 0] -> 0 (ok)
[1, 1] -> 1 (ok)

NAND
[0, 0] -> 1 (ok)
[0, 1] -> 1 (ok)
[1, 0] -> 1 (ok)
[1, 1] -> 0 (ok)

TRUE
[0, 0] -> 1 (ok)
[0, 1] -> 1 (ok)
[1, 0] -> 1 (ok)
[1, 1] -> 1 (ok)
```

A.5 Neuronové sítě

A.5.1 Neuronová síť - vlastní implementace

Vlastní implementace algoritmu Backpropagation.

```
# -*- coding: utf-8 -*-
from random import random, shuffle
from utils import sgn
import math

class Ann(object):
    def __init__(self, layer_sizes, learning_rate=0.5):
        self.learning_rate = learning_rate
        self.init_values(layer_sizes)
        self.init_weights()

    def init_values(self, layer_sizes):
        self.values = []
        for size in layer_sizes:
            self.values += [[None for ii in range(size)]]

    def init_weights(self):
        self.weights = {}
        for layer in self.walk_layers(exclude_last=True):
            self.weights[layer] = {}
            for ii in range(len(self.values[layer])):
                self.weights[layer][ii] = {}
                for jj in range(len(self.values[layer+1])):
                    self.weights[layer][ii][jj] = random() - 0.5

    def last_layer(self):
        return len(self.values) - 1

    def walk_layers(self, exclude_last=False, exclude_first=False, reverse=False):
        start = 0
        end = len(self.values)
        if exclude_first:
            start += 1
        if exclude_last:
            end -= 1
        if reverse:
            return xrange(end-1, start-1, -1)
        else:
            return xrange(start, end)

    def walk_units(self, layer):
        return xrange(len(self.values[layer]))

    def walk_weights(self, layer):
        assert(0<=layer and layer<len(self.values))
        for ii in xrange(len(self.values[layer])):
            for jj in xrange(len(self.values[layer+1])):
                yield ii, jj

    def get_output(self, layer, ii):
        val = self.values[layer][ii]
        try:
            return 1.0/(1+math.exp(-val))
        except OverflowError:
            if val < 0:
```

```
56                 return 0
57             else:
58                 return 1
59
60     def evaluate(self, data):
61         assert len(data) == len(self.values[0])
62         self.values[0] = data
63
64         for layer in self.walk_layers(exclude_last=True):
65             for jj in self.walk_units(layer+1):
66                 data_in = [self.get_output(layer, ii) * self.weights[layer][ii][jj]\
67                     for ii in self.walk_units(layer)]
68                 self.values[layer+1][jj] = sum(data_in)
69
70     def train(self, training_data, repetitions=1):
71         self.error_list = []
72         import math
73         for i in range(repetitions):
74             shuffle(training_data)
75             for data_in, data_out in training_data:
76                 self.train_instance(data_in, data_out)
77                 # self.learning_rate *= 0.99
78
79     def train_instance(self, data_in, data_out):
80         self.evaluate(data_in)
81         self.errors = {}
82
83         last_layer = self.last_layer()
84         self.errors[last_layer] = {}
85         for ii in self.walk_units(last_layer):
86             o = self.get_output(last_layer, ii)
87             t = data_out[ii]
88             self.errors[last_layer][ii] = o * (1 - o) * (t - o)
89
90         for layer in self.walk_layers(exclude_last=True, exclude_first=True, reverse=True):
91             self.errors[layer] = {}
92             for ii in self.walk_units(layer):
93                 o = self.get_output(layer, ii)
94                 self.errors[layer][ii] = o * (1 - o) * sum([(self.weights[layer][ii][jj] \
95                     * self.errors[layer+1][jj]) for jj in self.walk_units(layer+1)])
96
97         for layer in self.walk_layers(exclude_last=True):
98             for ii, jj in self.walk_weights(layer):
99                 self.weights[layer][ii][jj] += \
100                     self.learning_rate * \
101                     self.errors[layer+1][jj] * \
102                     self.get_output(layer, ii) * \
103                     self.weights[layer][ii][jj]
104
105
106
107 if __name__ == '__main__':
108     ann = Ann(layer_sizes=[6, 5, 1])
109     data = xlsx_import.parse_file(r'../data/data2.xlsx')
110
111     for row in data:
112         if row.subject_body_fat:
113             data_in = [
114                 int(row.subject_body_fat),
115                 time.mktime(row.subject_birth.timetuple()),
116                 row.subject_weight,
117                 row.subject_height,
```

```
                row.test_1,
                row.test_2
            ]
            data_out = [int(row.subject_sex)]

    ann.train(training_data, repetitions=1000)
    ann.visualise_error_list()
    ann.visualise()
```

A.5.2 Neuronová síť - vlastní implementace - vizualizace

```
# -*- coding: utf-8 -*-
import math
import matplotlib.pyplot as plt
from matplotlib.legend import Legend

from ann import Ann
class AnnVisualised(Ann):
    def visualise(self):
        fig = plt.figure()
        ax = fig.add_subplot(111, xmargin=0.01, ymargin=0.01)
        ax.set_yscale('log')
        for layer in range(len(self.values)):
            for ii in range(len(self.values[layer])):
                plt.plot(*self.visualise_unit_position(layer, ii), marker='o', color='b')

        for layer in range(len(self.values) - 1):
            for ii in range(len(self.values[layer])):
                for jj in range(len(self.values[layer+1])):
                    start = self.visualise_unit_position(layer, ii)
                    end = self.visualise_unit_position(layer+1, jj)

                    weight = self.weights[layer][ii][jj]
                    linewidth = '%s' % (self.weights[layer][ii][jj])
                    linestyle = 'solid'
                    if weight>0:
                        edgecolor = 'g'
                    else:
                        edgecolor = 'r'
                    ax.arrow(
                        start[0], start[1], (end[0]-start[0]) * 0.93, (end[1] - start[1]) * 0.93,
                        edgecolor=edgecolor, linestyle=linestyle, linewidth=linewidth,
                        fill=False, head_width=0.03, length_includes_head=True)
        plt.show()

    def visualise_unit_position(self, layer, idx):
        return idx - len(self.values[layer])/2.0, len(self.values) - layer

    def train(self, *args, **kwargs):
        self.error_list = []
        super(AnnVisualised, self).train(*args, **kwargs)

    def train_instance(self, data_in, data_out, *args, **kwargs):
        super(AnnVisualised, self).train_instance(data_in, data_out, *args, **kwargs)
        self.error_list += [math.fabs(self.errors[self.last_layer()][0])]


    def visualise_error_list(self):
        fig = plt.figure()
        ax = fig.add_subplot(111, xmargin=0.01, ymargin=0.01)
```

```
50          ax.plot(range(len(self.error_list)), self.error_list, 'ro-')
51          plt.show()
```

A.5.3 Neuronová síť - využití PyBrain

Ukázka využití knihovny PyBrain na data UNIFITTEST.

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3  import os
4  import math
5  import pickle
6  from os.path import join
7  import time
8  from random import random, randint
9
10 from pybrain.tools.shortcuts import buildNetwork
11 from pybrain.datasets import SupervisedDataSet
12 from pybrain.supervised.trainers import BackpropTrainer
13 from pybrain.structure import FeedForwardNetwork, LinearLayer, SigmoidLayer, FullConnection
14
15 import xlsx_import
16 from utils import get_average
17 from nn_visualise import visualise_net, visualise_errors
18
19 if __name__ == '__main__':
20     out_dir = '../graphs/nn/%s/' % time.strftime("%Y%m%d_%H%M%S")
21     os.mkdir(out_dir)
22
23     net = buildNetwork(5, 5, 1, hiddenclass=SigmoidLayer, outclass=SigmoidLayer, bias=True)
24
25     # create dataset
26     ds = SupervisedDataSet(5, 1)
27     data = xlsx_import.parse_file(r'../data/data2.xlsx')
28
29     BODY_FAT_DEFAULT = get_average(data, 'subject_body_fat')
30     for row in data:
31         data_in = [
32             1.0/(int(row.subject_body_fat or 0) or BODY_FAT_DEFAULT),
33             1.0/row.subject_weight,
34             1.0/row.subject_height,
35             1.0/row.test_1,
36             1.0/row.test_2
37         ]
38         if row.subject_sex:
39             data_out = [0.9]
40         else:
41             data_out = [0.1]
42
43         ds.addSample(data_in, data_out)
44
45     pickle.dump(ds, open(join(out_dir, 'data.pickle'), 'w'))
46
47     # create trainer
48     trainer = BackpropTrainer(net, ds)
49
50     # run training cycle
51     trainingErrors, validationErrors = trainer.trainUntilConvergence()
52
53     # visualise errors
```

```
    visualise_errors(
        trainingErrors[5:],
        validationErrors[5:],
        save_as=join(out_dir, 'errors_all.png')
    )
    visualise_errors(
        trainingErrors[-int(len(trainingErrors)/10.0):],
        validationErrors[-int(len(validationErrors)/10.0):],
        save_as=join(out_dir, 'errors_end.png')
    )

    # visualise network
    visualise_net(net, save_as=join(out_dir, 'network.png'))

    # save network for future use
    pickle.dump(net, open(join(out_dir, 'net.pickle'), 'w'))

    # save trainer history
    pickle.dump(trainingErrors, open(join(out_dir, 'training_errors.pickle'), 'w'))
    pickle.dump(validationErrors, open(join(out_dir, 'validation_errors.pickle'), 'w'))
```

A.5.4 Neuronová síť - využití PyBrain - vizualizace

```
# -*- coding: utf-8 -*-
import time
import math
import matplotlib.pyplot as plt
from matplotlib.legend import Legend


def visualise_net(net, show=True, save_as=None):
    fig = plt.figure()
    ax = fig.add_subplot(111, xmargin=0.01, ymargin=0.01)
    ax.axes.get_xaxis().set_visible(False)
    ax.axes.get_yaxis().set_visible(False)

    for module_idx, module in enumerate(net.modulesSorted):
        try:
            next_module = net.modulesSorted[module_idx+1]
        except IndexError:
            next_module = None

        for idx in range(module.dim):
            plt.plot(
                idx - module.dim / 2.0, len(net.modules) - module_idx,
                marker='o',
                color='b'
            )

        for connections in net.connections[module]:
            for ii in range(len(connections.params)):
                start_idx, end_idx = connections.whichBuffers(ii)
                start_x = start_idx - module.dim / 2.0
                start_y = len(net.modules) - module_idx

                end_x = end_idx - next_module.dim  / 2.0
                end_y = start_y - 1

                weight = connections.params[ii]
                linewidth = '%s' % (weight)
```

```
38                 linestyle = 'solid'
39                 if weight>0:
40                     edgecolor = 'g'
41                 else:
42                     edgecolor = 'r'
43                 ax.arrow(
44                     start_x,
45                     start_y,
46                     (end_x-start_x) * 0.93,
47                     (end_y - start_y) * 0.93,
48                     edgecolor=edgecolor,
49                     linestyle=linestyle,
50                     linewidth=linewidth,
51                     fill=False,
52                     head_width=0.03,
53                     length_includes_head=True)
54
55     if save_as:
56         plt.savefig(save_as)
57
58     if show:
59         plt.show()
60     plt.clf()
61
62
63 def visualise_errors(training_errors, validation_errors, show=True, save_as=None):
64     fig = plt.figure()
65     ax = fig.add_subplot(111, xmargin=0.01, ymargin=0.01)
66     plt.plot(range(len(training_errors)), training_errors, 'r.-')
67     plt.plot(range(len(validation_errors)), validation_errors, 'b.-')
68     if save_as:
69         plt.savefig(save_as)
70     if show:
71         plt.show()
72
73     plt.clf()
```