VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

MODUL PRO KLASIFIKACI VÝSLEDKŮ V RÁMCI E-LEARNINGOVÉHO SYSTÉMU

A MODULE FOR CLASSIFICATION OF RESULTS IN AN E-LEARNING SYSTEM

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. JAKUB KOČVARA**
AUTHOR

VEDOUCÍ PRÁCE **Ing. VLADIMÍR BARTÍK, Ph.D.**
SUPERVISOR

BRNO 2017

Abstrakt

V této práci se snažíme pomocí metod strojového učení predikovat výslednou známku studenta ve výukovém informačním systému na základě jeho chování během semestru. Cílem je zjistit optimální techniky při extrakci dat, jejich úpravě a učení predikčního modelu. Poté celý systém implementovat jako modul, který budeme moct ke stávajícímu systému připojit.

Abstract

In this thesis we try using machine learning techniques to predict final grade of a student in a learning management system on the basis of his behavior during the semester. The aim is to determine the optimal technology for the extraction, treatment and machine learning on data. The whole system would then be implemented as a module that we will be able to plug in the existing system.

Klíčová slova

Big data, dolování dat, strojové učení, klasifikace, predikce, neuronové sítě, databázové systémy, datový sklad, informační systém

Keywords

Big data, data mining, machine learning, classification, prediction, neural networks, database systems, data warehouse, information system

Citace

KOČVARA, Jakub. *Modul pro klasifikaci výsledků v rámci e-learningového systému*. Brno, 2017. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Bartík Vladimír.

Modul pro klasifikaci výsledků v rámci e-learningového systému

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Vladimíra Bartíka, PhD. Další informace mi poskytl Ing. Mirek Melichar a Ing. Augustin Židek. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Jakub Kočvara
24. května 2017

Poděkování

Děkuji Ing. Vladimíru Bartíkovi, Ph.D. za konzultace při tvorbě práce. Dále Ing. Mirku Melicharovi a Ing. Augustinu Židkovi za pomoc s výběrem algoritmu strojového učení.

Obsah

Kapitola 1

Úvod

V poslední době je velký důraz kladen na zpracování dat ve firmách všeho druhu. Na aplikace, které dokáží agregovat miliony datových vstupů uživatelů v realném čase na zmapování dopravní situace nebo předpovídají, o které produkty by mohl mít zákazník zájem na základě uživatelů s podobným chováním, jsme v 21. století už všichni zvyklí. Nastala doba, v které si každá větší společnost uvědomuje, že získávání znalostí dat může rapidně zlepšit „user experience“ i kvalitu manažerských rozhodnutí v chodu firmy. Investice do získání znalostí z velkých objemů dat zvyšují retenci uživatelů a efektivitu managementu, a tím i zisky ve firmě.

1.1 Cíl práce

Přestože v např. ve zdravotnictví, marketingu nebo finančnictví je práce s daty velmi rozšířená, ve vzdělávání se pokročilejší metody získávání znalostí z dat příliš nepoužívají. Tato práce je vytvářena ve spolupráci s firmou, která dodává americkým univerzitám tzv. LMS neboli *learning management systems*. Jedná se o rozšířený informační systém, který slouží zejména pro studenty studující dálkově nebo přímo pro online univerzity. Tento systém umožňuje instruktorům a studentům provádět online přednášky, přístup k výukovým materiálům, odevzdávání úkolů i psaní testů. Aktivita instruktorů i studentů byla během posledních tří let monitorována a logována na pozdější zpracování. Cílem práce je na základě těchto dat vymyslet novou funkctionalitu, která uživatelům může pomoct. Předběžný plán je, že bychom pomocí dat o chování uživatelů v systému chtěli předpovědět jejich konečnou známku na konci semestru.

Kapitola 2

Analýza dat

V této kapitole si objasníme několik základních termínu a technik z oblasti datové analýzy. Tato disciplína se rapidně vyvíjí již několik let a pojmy jsou často do určité míry zaměnitelné. Podívame se na ty, které se používají nejčastěji.

2.1 Big data

Zatímco termín „big data“ je relativně nový, proces shromažďování a uchovávání velkého množství informací pro případnou následnou analýzu nic nového není. Digitalizace dat analýzu velmi usnadňuje. Koncept nabral na obrátkách na začátku 21. století, když datový analytik Doug Laney formuloval definici zpracování velkých objemů dat. Teorii pojmenoval 3V, anglicky *volume, velocity, variety.*[12] Tyto kritéria ještě často bývají doplněna o *variabilitu* a *komplexitu.*[9] Vypadají tedy následovně:

- **Objem**
  Organizace sbírá data z různých zdrojů, včetně obchodních transakcí, sociálních médií a informací ze senzorů nebo strojových logů dat. V minulosti by bylo skladování takových obnosů dat problémem, ale nové technologie distribuovaných úložišť jako Apache Hadoop pomáhají uvolnit zátěž.

- **Rychlost**
  Datové toky přicházejí nebývalou rychlostí a musí být zpracovávány nebo alespoň ukládány velmi rychle. Inteligentních senzory, RFID tagy, web nebo mobilní aplikace dokáží produkovat kvanta dat extrémním tempem a je potřeba se jimi zabývat v téměř reálném čase.

- **Různorodost**
  Data přicházejí ve všech formátech - od strukturovaných číselných nebo textových dat v tradičních databázích po nestrukturované textové dokumenty, e-mail, video, audio, burzovní data a finanční transakce.

- **Variabilita**
  Kromě zvyšující se rychlosti a různorodosti dat mohou datové toky přicházet v nepravidelných intervalech. Tok dat se může měnit periodicky (denní nebo sezónní cyklus) nebo nárazově (mimořádné události). V těchto špičkách může být datové zatížení náročné zvládnout. Pokud jsou data nestrukturovaná, je to ještě náročnější.

- **Složitost**
  Data mohou pocházet z různých zdrojů, což ztěžuje manipulaci s nimi, je nutné je transformovat a sjednotit napříč systémy. To vše při zachování jejich informační hodnoty a vztahů.

Tradiční analytické nástroje nejsou vhodné pro zachycení poznatků při zpracovávání velkých objemů dat. Ta jsou příliš rozměrná pro komplexní analýzu a hledání potenciálních korelací a vztahů mezi různorodými vlastnostmi dat. Jejich objem a komplexita jsou moc velké pro lidské testování všech hypotéz, a proto bychom tímto způsobem jen těžko odvodili celkovou hodnotu ukrytou v datech.

Základní analytické metody používané ve firemní sféře a vytváření výkazů lze provádět pomocí programů jako je MS Excel nebo MS Access, popřípadě použitím jednoduchých SQL příkazů. Takto se dají realizovat operace jako sumace, aritmetické průměry, mediány, atd. OLAP (Online Analytical Processing) je pouze systematizované rozšíření těchto základních analytik, které stále spoléhá na nasměrování člověka, který díky svým zkušenostem ví, co a jak počítat. Řešením tohoto problému je technika, které se říká dolování dat.

2.2 Dolování dat

Přesto, že by tomu název napovídal, při dolování dat ve skutečnosti samotná data odnikud nedolujeme, ale hledáme v nich na první pohled skryté souvislosti. Tento pojem je úzce spjat se statistikou, umělou inteligencí a strojovým učením (machine learning). Často se pod tímto pojmem skrývá kombinace některých kroků při zpracování informací:[5]

- sběr
- extrakce
- skladování
- analýza
- vyhodnocení

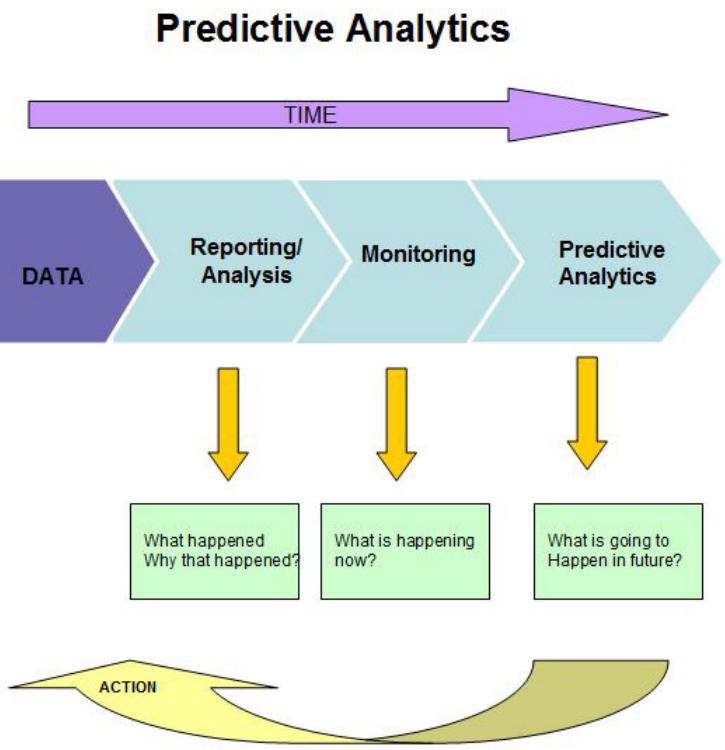
Krok analýzy se při dolování dat řeší pomocí pokročilých algoritmů, které se zastřešují pojmem *strojové učení*.

2.3 Strojové učení

Strojové učení (anglicky *machine learning*) je ideální technika pro nalezení souvislostí ukrytých ve velkých datech. Tyto algoritmy v dnešní době řeší problémy, které lze exaktně jen velmi obtížně formulovat, mimo jiné počítačové vidění nebo rozpoznávání řeči. Strojové učení je revoluční právě tím, že na rozdíl od tradiční analýzy lze model naučit bez explicitně naprogramovaného chování.[24] Čím více dat do systému přivedeme, tím více a přesněji se dokáže učit a aplikovat výsledky ke zvýšení kvality modelu.

2.3.1 Prediktivní analýza

Proces, při kterém pomocí strojového učení ze známých dat vyrábíme model a následně se snažíme odhadnout, jak se v našem systému zachovají nově příchozí data se jmenuje *predikce*. Prediktivní analýza je termín, který se používá v korporátním prostředí pro cyklus, při kterém se na základě firemních dat snažíme naučit prediktivní model, který nám může pomoct při dalším rozhodování.[18]



Obrázek 2.1: Diagram průběhu prediktivní analýzy.

Kapitola 3

Metody strojového učení

Pojem strojového učení byl na konci 90. let formálně definován Tomem M. Mitchellem následovně [14]:

> O počítačovém programu můžeme říct, že se učí ze zkušeností E s ohledem na nějakou třídu úkolů T a měření výkonu P, pokud se jeho výkon P při úkolech T zvyšuje se zkušenostmi E.

3.1 Feature vector

Množině vstupů do algoritmů říkame anglicky *feature vector*. V češtině se vstupům říká atributy, rysy, proměnné a někdy dokonce fíčury. Kardinalita množiny rysů je označována jako dimenze vektoru. Vektor rysů $\{age, sex, salary\}$ je 3-dimenzionální. it Redukce dimenzionality vektoru rysů může v některých případech zlepšit chování modelu oproti původnímu vektoru. Stejně tak se můžeme k lepším výsledkům dopracovat pomocí opačného postupu, kterému se říká it konstrukce rysů. Pokud vytvoříme nový rys operací nad stávajícími, můžeme tím dát učícímu algoritmu více informací.

Například při zkoumání závislosti rysů jednotlivých pasažéru na Titanicu s jejich přežitím lze předpokládat, že jejich jméno a příjmení jsou rysy, které jejich přežití nijak neovlivní a dal by se proto vyloučit. Pokud bychom mluvili o jméně samotném byla by to pravda, jenže pomocí shodných jmen pasažérů vytvořit nový rys: počet rodinných příslušníků na palubě, který se ukázal být jako velmi důležitý [19].

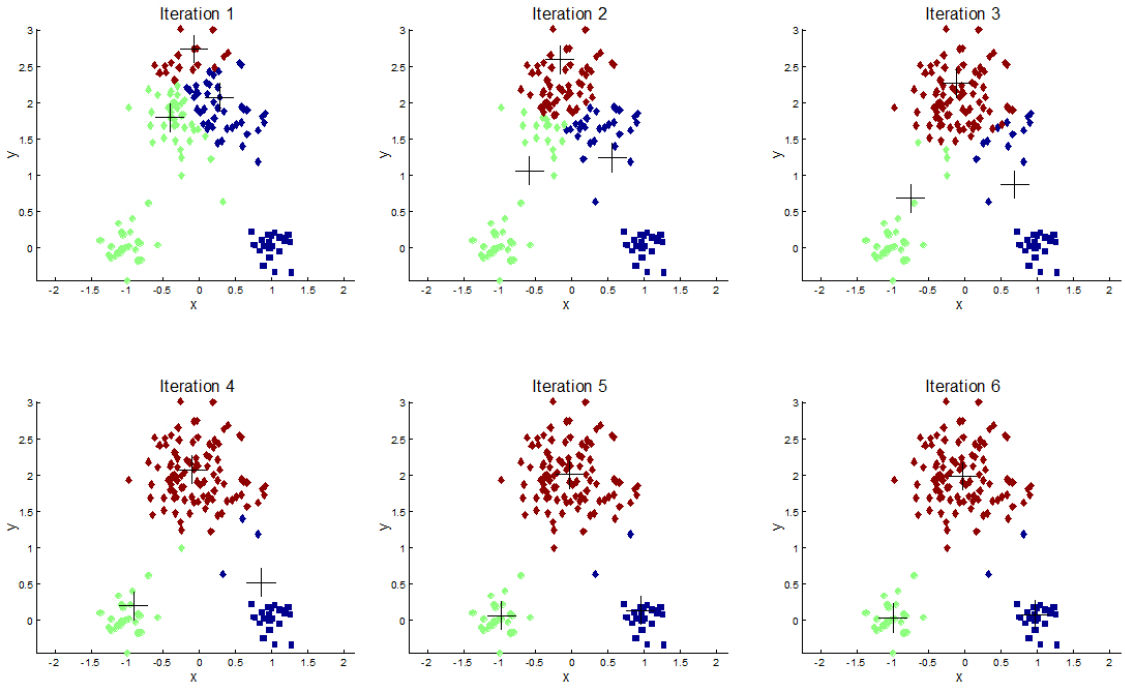
3.2 Rozdělení typů strojového učení

Rozdělení algoritmů pro strojové učení existuje mnoho, v této práci se však podíváme na ty základní. Podle typu úkolu, který chceme strojovým učením řešit ho můžeme rozdělit na dva základní typy: *učení bez učitele* a *učení s učitelem.*[22]

3.2.1 Učení bez učitele (unsupervised learning)

Pokud se učíme na datech, ke kterým nemáme výstupy a nejsme tedy schopni určit kvalitu modelu, mluvíme o učení bez učitele. To v podstatě znamená, že se pouze pomocí algoritmů snažíme najít na první pohled skrytou strukturu v datech. Mezi nejpoužívanější praktiky patří hledání shluků nebo detekce anomálií v datech. Mezi nejznámější techniky sloužící ke

hledání shluků patří *k-means* algoritmus. Ten se snaží data rozdělit na k shluků podle jejich vzdáleností k nejbližším geometrickým středům.



Obrázek 3.1: Průběh algoritmu k-means na datech při k=3. Potenciální shluky jsou na začátku rozmístěny náhodně a iteračně konvergují ke geometrickým středům jednotlivých shluků.

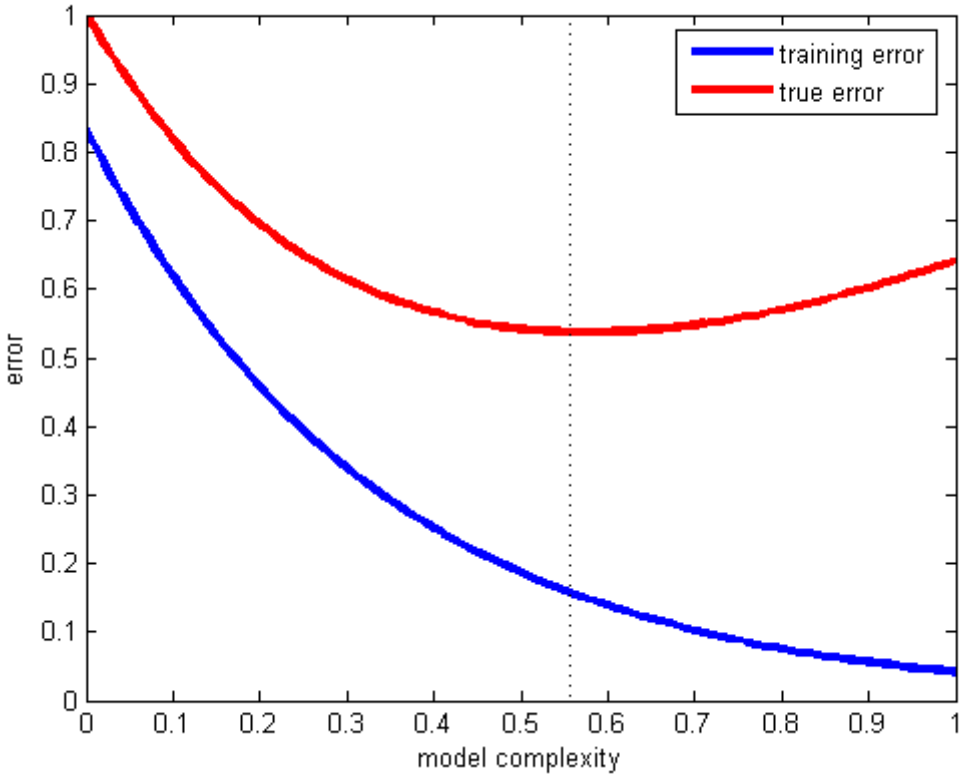
3.2.2 Učení s učitelem (supervised learning)

Pro vstupní trénovací data máme jasně dané správné výstupy. Model si v tomto případě můžeme představit jako funkci, která má na vstupu určitý počet proměnných (dimenze vektoru rysů) a k nim máme zadaný příslušný výstup. Učení v tomto případě znamená, že se snažíme tuto funkci (model) upravovat tak dlouho, aby byl výstup modelu na trénovacích datech co nejpodobnější skutečnosti.

Podívejme se ještě jednou na zadání naší úlohy: **víme výslednou známku studenta a snažíme se ji předpovědět u nových studentů**. Z toho vyplývá, že náš problém spadá do kategorie učení s učitelem.

3.2.3 Přeučení (overfitting)

Jeden z nežádoucích efektů při učení z trénovacích dat je tzv. *přeučení*. To nastává, pokud je model moc „šitý na míru“ pro trénovací data. To znamená, že i když je na první pohled model přesnější, od určité komplexity začíná na testovacích datech vykazovat horší výsledky. Proto musíme vědět, kdy je algoritmus optimálně naučen, aby predikce na testovacích datech vykazovala nejlepší výsledky. Ze statistického hlediska to znamená, že model vykazuje vysoký *rozptyl* (kvadrát směrodatné odchylky). Opakem přeučení je tzv. „underfitting“. [8]



Obrázek 3.2: Závislost chyby predikce na komplexitě modelu. Trénovací data jsou vyznačena modře, testovací červeně. Přerušovaná čára značí optimum, v kterém by se mělo učení zastavit.

Kapitola 4

Základní algoritmy strojového učení s učitelem

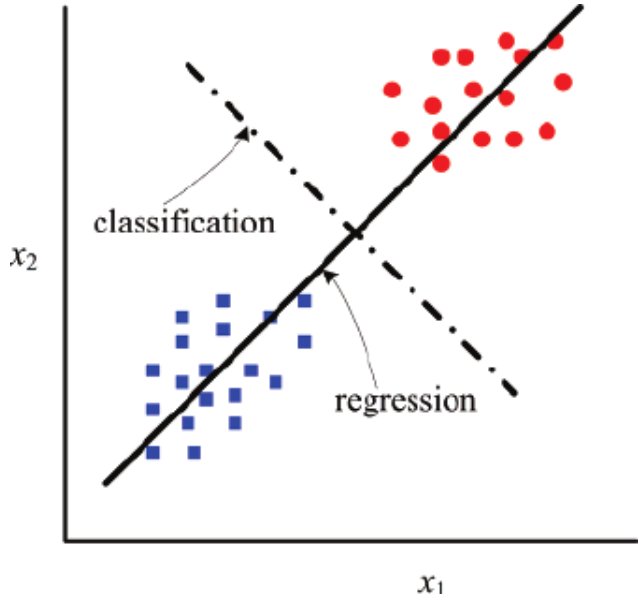
Tento typ učení se spoléhá na to, že nám je při daných vstupních proměnných znám požadovaný výstup algoritmu. Podle typu výstupní proměnné však rozdělujeme tyto problémy do dvou základních tříd: [1]

- **Klasifikace**
  Výstupní proměnná nabývá diskrétních hodnot z předem daného intervalu. Jinými slovy má příslušnost do jedné z množiny kategorií. Například příslušnost ke krevní skupině, identifikace odrůdy květiny nebo volba politické strany. Speciálním případem klasifikace je *binární klasifikace*, při které může výstup patřit pouze do dvou možných kategorií (true/false, 0/1, ...), např.: přežití člověka, úspěch nebo neúspěch studenta, přítomnost onemocnění, atd.

- **Regrese**
  Výstupní proměnná nabývá spojitých hodnot. Prediktivní funkce je zadána pro všechny vstupy z definičního oboru. Mezi příklady užití patří predikce ceny nemovitostí, věku nebo přijmu osob.

V tuto chvíli není jasné, který přístup se bude pro náš případ hodit. Známky, které studenti na konci semestru dostávají jsou diskrétní hodnoty, zpravidla na stupnici od A do F. V systému jsou však známky uloženy jako percentuální hodnoty mezi 0 a 100. Záleží na tom jak si instruktor pro svůj kurz nastaví hodnotící stupnici (grading system).

Tyto dva typy úkolů nejsou však vzájemně výlučné. Z regresního problému, lze pomocí sloučení intervalů výstupních hodnot vytvořit problém podobný klasifikaci a naopak. V následující části práce si ukážeme některé nejpoužívanější praktiky v klasifikaci a regresi.



Obrázek 4.1: Rozdíl mezi klasifikací a regresí.

4.1 Zobecněný lineární model (GLM)

Tento pojem zastřešuje metody vycházející z lineární regrese i pro výstupní proměnné, které mají jiné než normální rozdělení chyby. Patří sem kromě lineární například logistická nebo Poissonova regrese. Na sestavení GLM modelu potřebujeme tři prvky: [16]

- předpokládané rozdělení pravděpodobnosti z exponenciální rodiny (normální, binomické, gamma, Poissonovo, ...)
- lineární prediktor (lineární funkce parametrů modelu)

  $$\vec{\eta} = X\vec{\beta} + \vec{\epsilon}$$

  X je vstupní matice, jejíž řádky odpovídají jednotlivým měřením a sloupce tvoří vektor rysů, $\vec{\beta}$ je vektor vyjadřující vliv příslušných rysů na výstup, $\vec{\epsilon}$ je „offset“, neboli vektor případných vedlejších vlivů, které známe a nemusíme je modelem odhadovat
- spojovací funkce μ, která dokáže namapovat lineární prediktor na požadované rozložení pravděpodobnosti

4.1.1 Lineární regrese

Základním triviálním případem regresní analýzy je regrese pomocí lineární funkce ve tvaru:

$$y = \alpha x + \beta$$

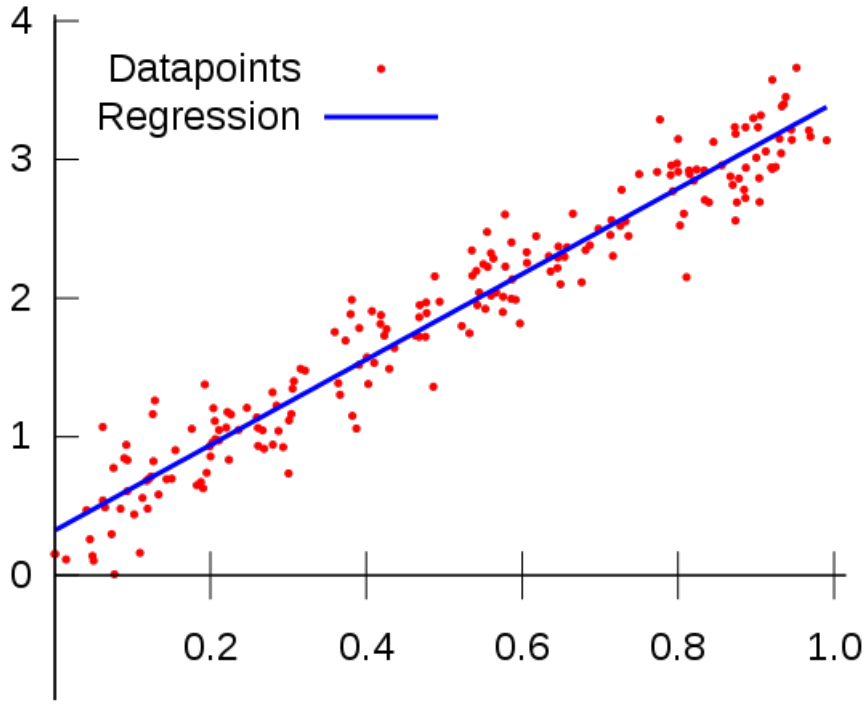
Proměnná α určuje sklon regresní přímky a β její posun na ose y. Snažíme se o proložení této přímky datovými body co nejpřesněji. K určení optimálních parametrů α a β slouží několik metod, které mohou dojít k různým výsledkům. Jedna z nejpoužívanějších technik na evaluaci regresní přímky s minimální chybou je **metoda nejmenších čtverců**. Cílem této metody je, aby výsledné řešení minimalizovalo součet čtverců odchylek vůči každé z

rovnic. Výpočet parametrů, který získáme derivací součtu čtverců mezi původními body a těmi aproximovanými přímkou vypadá takto (n je počet bodů ve vzorku, a proto musí platit $n \geq 2$):

$$\alpha = \frac{n\sum x_i y_i - \sum x_i \sum y_i}{n\sum x_i^2 - (\sum x_i^2)}$$

$$\beta = \frac{n\sum x_i y_i - \sum x_i \sum x_i y_i}{n\sum x_i^2 - (\sum x_i^2)}$$

Tato metoda je vhodná pro data, která jsou *homoskedastičná*, tzn. jejich rozptyl je homogenní. Pokud je rozptyl v datech závislý na parametru, může být predikce pomocí lineární regrese velmi nepřesná.



Obrázek 4.2: Regresní přímka na 1-dimenzionálních datech s homogenním rozptylem.

4.1.2 Logistická regrese

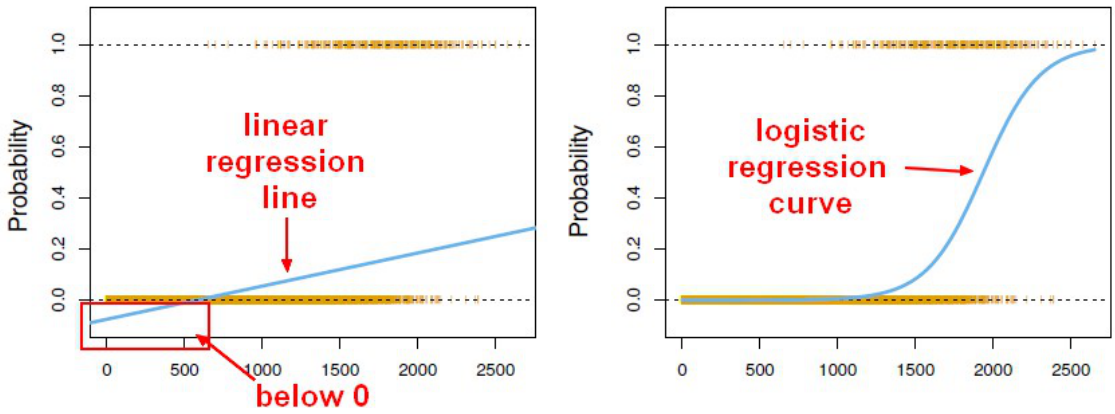
Existují data, pro které není lineární regrese vhodná. Jeden z těchto případu je pokud chceme provést regresní analýzu na datech, v kterých je výstupní proměnná binární a patří tedy pouze do dvou kategorií. Lineární regrese by byla velmi nepřesná a její predikce by nespadaly do intervalu od 0 do 1 (binární data). Ideální pro binární data je regrese logistická, kde závislost výstupní proměnné na základě vstupních můžeme vyjádřit pomocí logistické funkce, která vypadá takto:

$$\mu = \frac{e^{X\vec{\beta}+\vec{\epsilon}}}{1 + e^{X\vec{\beta}+\vec{\epsilon}}}$$

Po úpravě vychází:

$$X\vec{\beta} + \vec{\epsilon} = ln(\frac{\mu}{1-\mu})$$

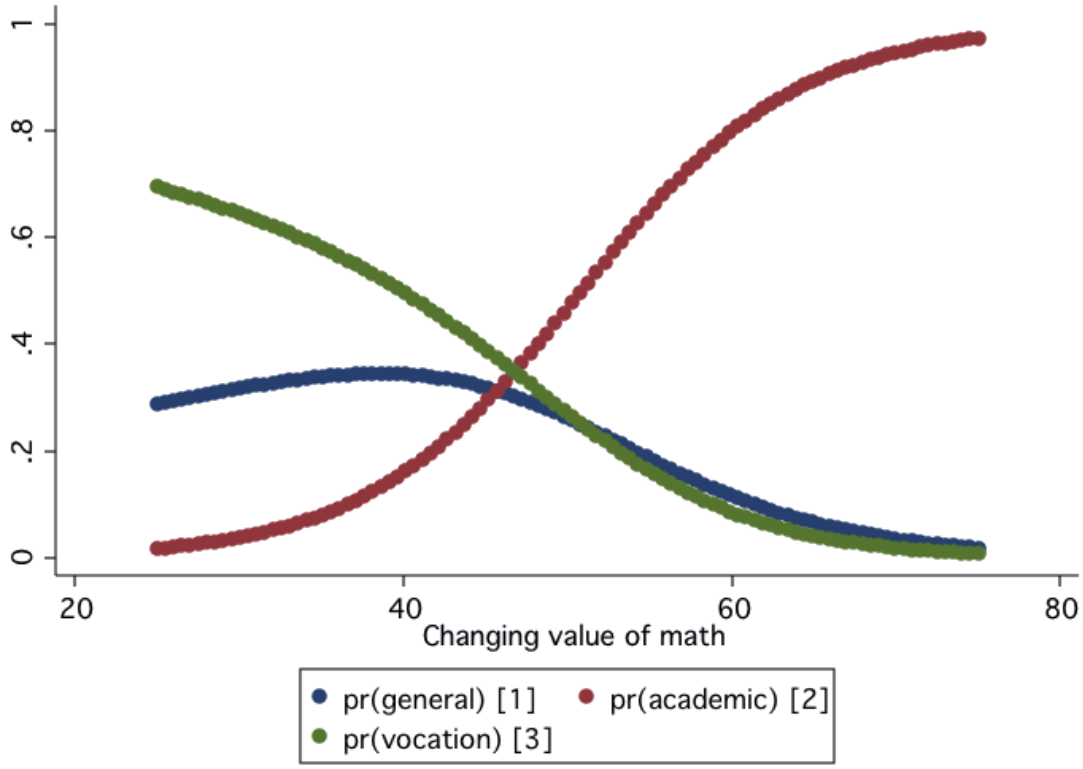
Výrazu $ln(\frac{\mu}{1-\mu})$ se říká *logit*. K výpočtu hodnot parametrů vektoru $\vec{\beta}$ a $\vec{\epsilon}$ je opět použita iterativní metoda nejmenších čtverců. Takto získáme optimální tvar logistické (sigmoid) křivky. Predikce u této metody probíhá tak, že po zadání vstupů získáme číslo mezi 0 a 1, což je pravděpodobnost příslušnosti do jednotlivých kategorií. Například v předmětu, kde 0 znamená neúspěch a 1 úspěch, kde nám při predikci vyjde pravděpodobnost $0,84$, můžeme s 84% jistotou říct, že člověk předmět splnil.



Obrázek 4.3: Na obrázku vlevo vidíme nevhodnost lineární regrese na binárních datech (dokonce je částí mimo obor hodnot). Křivka logistické regrese určuje pravděpodobnost příslušnosti do kategorií.

4.1.3 Multinomiální logistická regrese

Klasická logistická regrese pracuje pouze s binárními daty (2 kategorie). Zobecněním této metody, které dokáže pracovat s daty, patřící do více kategorií je multinomiální logistická regrese. Funguje velmi podobně, s tím rozdílem, že se jednu z výsledných kategorií označíme jako referenční a vůči ní budeme porovnávat pravděpodobnost ostatních kategorií. Pokud existuje s kategorií, tak vektor $\vec{\beta}_i, i \in \{1, 2, \ldots, s-1\}$ představuje vliv jednotlivých vstupních proměnných na to, zda bude výstup v i-té kategorii nebo v kategorii referenční. Vytváříme tedy model postupně po dvojicích a výsledkem je soubor logistických funkcí s parametry pro každou kategorii zvlášť.



Obrázek 4.4: Tři logistické křivky, kde každá určuje pravděpodobnost příslušnosti studenta do určité kategorie na základě jeho výsledků v matematice.

4.2 Učení pomocí rozhodovacích stromů

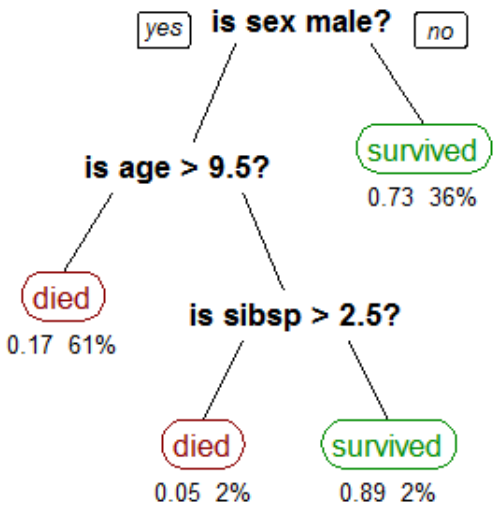
Často používané techniky při strojovém učení používají rozhodovací stromy (decision trees). Ty jsou populární kvůli své přehlednosti a snadné interpretovatelnosti výsledků. V kombinaci s meta-algoritmy můžeme získat velmi přesný prediktor.

Každý uzel rozhodovacího stromu představuje rozcestí, na kterém se rozhodujeme podle vlastnosti některé ze vstupních proměnných (např. $age \geq 50$). Každá větev reprezentuje výsledek tohoto testu. Listy stromu jsou kategorie, do kterých může daný prvek příslušet. Cesta stromem od kořene k listu reprezentuje množinu klasifikačních pravidel.

Strom se sestavuje tak, aby od sebe uzel objekty maximálně odlišoval. To lze vypočítat pomocí *Shannonovy entropie* na pravděpodobnostech výsledků uzlu.

$$H = -\sum_{i=1}^{n} p_i \log_2 p_i$$

Nejpoužívanější algoritmy jsou ID3, C4.5 a C5.0. Zjednodušeně fungují tak, že se pomocí upraveného výpočtu entropií pro atributy rozhodnou vytvořit uzly stromu a poté se rekurzivně aplikují na nově vzniklé uzly a vytváří podstromy. [26]



Obrázek 4.5: Jednoduchý rozhodovací strom, v kterém pomocí informací o pasažérovi získáváme pravděpodobnost jeho úmrtí na Titanicu.

4.2.1 Meta-algoritmy (ensemble methods)

Samotný rozhodovací strom není zpravidla dost silný predikční model. Meta-algoritmy kombinují více učících algoritmů pro získání lepšího prediktivní výkonu, než by bylo možno získat z predikce pomocí jeho složkových algoritmů.

Bagging

Zkrácené slovo pro pojem „**b**ootstrap **agg**regat**ing**“. Za tímto pojmem se skrývá technika, která kombinuje množství rozhodovacích stromů k vytvoření stabilnějšího a přesnějšího prediktoru. Z trénovacích dat jsou uniformě vybrány podmnožiny objektů o předem určené kardinalitě. Pro každý vzorek je vytvořen rozhodovací strom. Na testovacích datech se snažíme výsledek predikovat pomocí každého z rozhodovacích stromů. Konečný výsledek je při klasifikaci kategorie s nejvyšším počtem „hlasů“ a při regresi je to průměr regresních křivek. Tímto se také vyhýbáme silnému přeučení.[2]

Velmi populární machine learning algoritmus **Random forest** vychází z baggingu s tím rozdílem, že kromě vzorků z dat vytváří i náhodné vzorky rysů, pomocí kterých vytváří rozhodovací stromy.

Boosting

Technika, při které se snažíme pomocí slabých klasifikátorů vytvořit jeden silný. Při přidání slabého klasifikátoru do systému zkontrolujeme správnost celkové klasifikace. Objektům, které byly predikovány mylně zvýšíme váhu a objektům, které byly predikovány správně váhu snížíme. Tato váha zaručí, že se v budoucnosti slabé klasifikátory více zaměří na objekty s vyšší vahou. Tímto by měl časem systém konvergovat a chyba při predikci se snižovat.[11]

Nejznámějším algoritmem používajícím boosting je AdaBoost. V práci se budeme zabývat algoritmem **GBM** (gradient boosted models), který boosting používá v kombinaci s rozhodovacími stromy.

Kapitola 5

Strojové učení pomocí algoritmu gradient descent

Abychom pochopili princip „učení“ a vytváření modelu, musíme si uvědomit, o co se vlastně snažíme. Matematicky vzato je model funkce, která má několik vstupních proměnných a jeden nebo více výstupů. Pomocí parametrů této funkce (označujeme je zpravidla *váhy*) můžeme model upravit tak, aby co nejlépe imitoval chování trénovacích dat (někdy to však není ideální, viz overfitting 3.2.3). Při učení porovnáváme hodnoty predikované naším modelem s očekávanými výstupy a snažíme se z toho vyvodit přesnost našeho modelu. Způsob, kterým tyto hodnoty porovnáváme označujeme jako *účelovou funkci* (anglicky *loss* nebo *cost function*). Chceme najít minimum této funkce, učení modelu je tedy optimalizační problém [25].

5.1 Hledání minima funkce

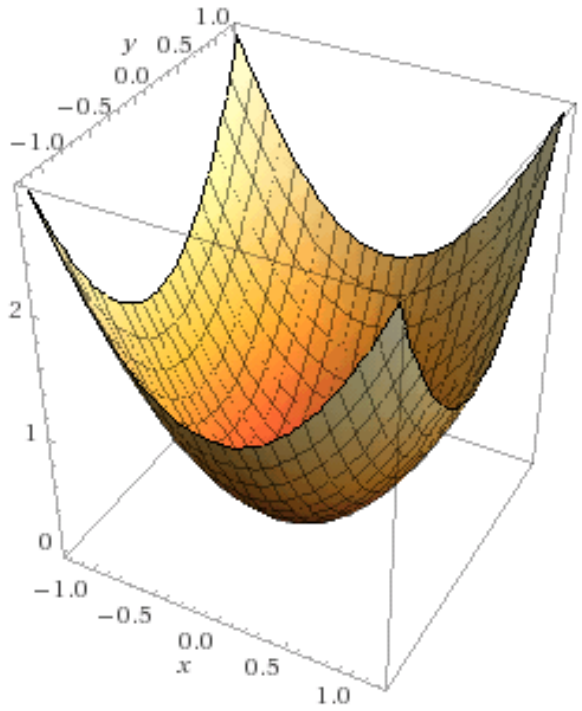
Derivace funkce určuje rychlost změny výstupu této funkce v závislosti na změně vstupu. U dvourozměrných grafů určuje směrnici tečny k původní funkci v určitém bodě. Lokální extrémy (minima nebo maxima) najdeme tak, že hledáme bod na ose x, ve kterém se $f'(x) = 0$. Zda se jedná o maximum či minimum lze poznat pomocí druhé derivace.

U funkcí s více neznámými je postup podobný, ale je nutné využít tzv. *parciální derivace*, kdy derivujeme funkci pouze podle jedné proměnné (v jedné dimenzi) a s ostatními proměnnými zacházíme jako s konstantami. Můžeme si to také představit jako řez 3D grafem, ze kterého nám vznikne 2d graf. Na něm poté provedeme derivaci a hledáme hodnotu proměnné, pro kterou se rovná nule. Pokud tento postup provedeme pro každou proměnnou, získáme souřadnice extrému.

Jako příklad si uvedeme funkci $z = x^2 + y^2$. Provedeme derivaci podle x a y:

$$\frac{\partial z}{\partial x} = 2x, \frac{\partial z}{\partial y} = 2y$$

$$2x = 0 \wedge 2y = 0 \Rightarrow min(z) = [0, 0]$$



Obrázek 5.1: 3D znázornění funkce $x^2 + y^2$. Zde můžeme jednoduše vidět, že minimum se nachází v bodě $x = 0, y = 0$.

5.2 Gradient descent

Pro zjednodušení si představme, že účelová funkce L, kterou chceme minimalizovat má pouze dvě vstupní proměnné. Gradient je ve své podstatě zobecnění derivace pro funkce s více proměnnými. Jedná se o vektor všech parciálních derivací takové funkce. Podobně jako u dvourozměrných grafů gradient určuje „sklon“ funkce, avšak u vyšších dimenzí si již nedokážeme představit jak funkce, ani její sklon vypadá.

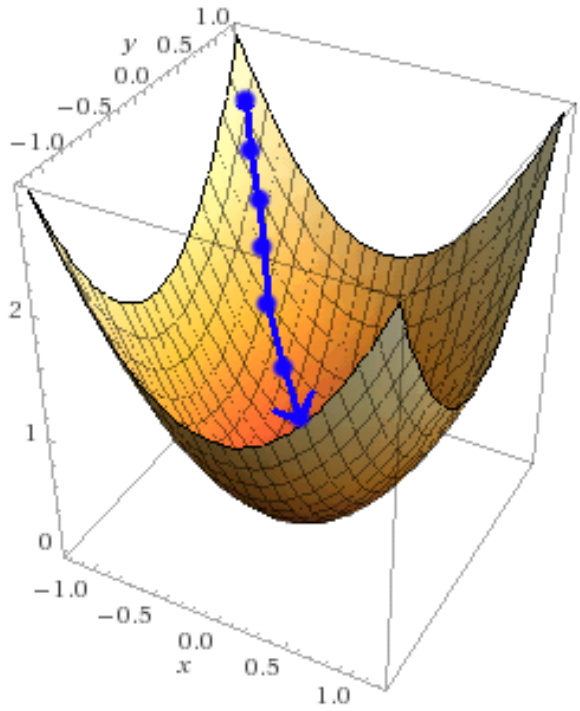
Abychom našli lokální minimum funkce pomocí gradient descent, musíme provádět malé kroky úměrné zápornému gradientu (nebo jeho aproximaci) v určitém bodě funkce. Pokud bychom hledali lokální maximum, provedli bychom to samé, ale s kladným gradientem, tento postup se nazývý analogicky „gradient ascent“.

Tento algoritmus si tak můžeme jednoduše znázornit ve třech dimenzích, jako hledání nejnižšího bodu v údolí (nebo nejvyššího bodu pohoří pro maxima).

Uvedeme případ, kdy máme pouze dvě vstupující proměnné (pouze pro ilustraci, v praxi můžeme mít až tisíce vstupních proměnných). Vztah mezi vstupními proměnnými a výstupem z účelové funkce můžeme popsat následujícím vzorcem, kde C je účelová funkce a $v1$ a $v2$ jsou vstupní proměnné:

$$\Delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2$$

Pro n proměnných by zápis vypadal analogicky takto:



Obrázek 5.2: Ilustrace gradient descent na funkci $x^2 + y^2$. Bod se iterativně přibližuje minimu.

$$\Delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2 + \cdots + \frac{\partial C}{\partial v_n}\Delta v_n$$

Tento zápis není pro další výpočty příliš vhodný. Zavedeme tedy notaci vektoru gradientů jako ∇C. Ten bude obsahovat všechny parciální derivace. Vektor Δv bude obsahovat všechny změny ve vstupních proměnných. Mezi nimi provedeme skalární součin.

$$\Delta C \approx \nabla C \cdot \Delta v$$

$$\nabla C = \begin{bmatrix} \frac{\partial C}{\partial v_1} \\ \frac{\partial C}{\partial v_2} \\ \vdots \\ \frac{\partial C}{\partial v_n} \end{bmatrix}$$

$$\Delta v = [\Delta v_1, \Delta v_1, \ldots, \Delta v_n]$$

Z tohoto zápisu jasněji pochopíme jaký mají mezi sebou vektory vztah. Chceme docílit, aby se hodnota C snižovala, neboli $\Delta C \leq 0$. Toho dosáhneme tak, že jako vybereme vhodné Δv takto:

$$\Delta v = -\eta \nabla C$$

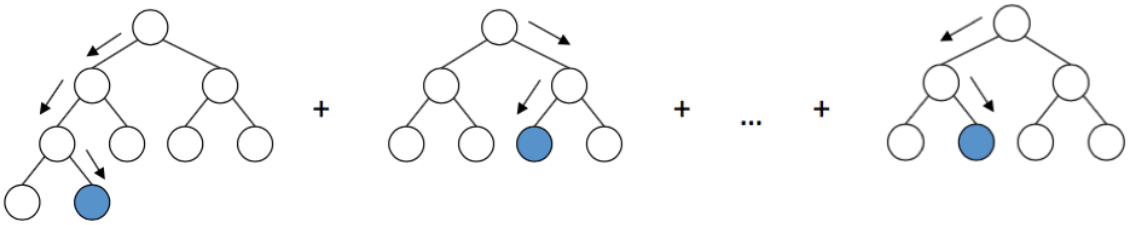
η je malý hyperparametr, který můžeme při procesu učení vybrat a určuje, jak velké kroky bude algoritmus při učení provádět neboli rychlost učení (parametr známý jako *learning rate*). Pokud dosadíme do vzorce, dostaneme $\Delta C \approx \nabla C \cdot (-\eta \nabla C)$, což lze upravit na $\Delta C \approx \eta |\nabla C|^2$.

Protože $|\nabla C|^2 \geq 0$ bude vždy $\Delta C \leq 0$, a tedy zaručíme, že vždy půjdeme správným směrem, a to k minimu účelové funkce. Pokud se pro zjednodušení vrátíme k trojrozměrnému nákresu gradient descent na obrázku 5.2, můžeme pohyb v jednom kroku bodu na pozici v popsat jako $v \to v' = v - \eta \nabla C$. Tuto operaci budeme opakovat tolikrát, dokud uznáme, že jsme v minimu účelové funkce [17].

5.3 Gradient Boosted Models

Velmi populární meta-algoritmus pro klasifikaci a regresi pomocí rozhodovacích stromů je GBM neboli gradient boosted models. *gradient boosting* je technika, kterou zpopularizoval na konci 90. let statistik Leo Breiman [3].

Gradient boosting je metoda strojového učení, která využívá kombinace slabých prediktorů, což jsou v případě GBM rozhodovací stromy omezené velikosti. Pokud je T počet koncových uzlů (listů) v dílčím stromu, jejich počet určuje úroveň interakce mezi vstupními proměnnými. Pokud si představíme strom s $T = 2$, není mezi proměnnými možná jakákoli interakce. Těmto speciálním případům se říká rozhodovací „pařezy“ (decision stumps). Empiricky bylo dokázáno, že $T \in \langle 4, 8 \rangle$ je pro gradient boosting optimální [7].



Obrázek 5.3: Ilustrace rozhodování pomocí slabých klasifikačních rozhodovacích stromů při algoritmu *gradient boosted models*.

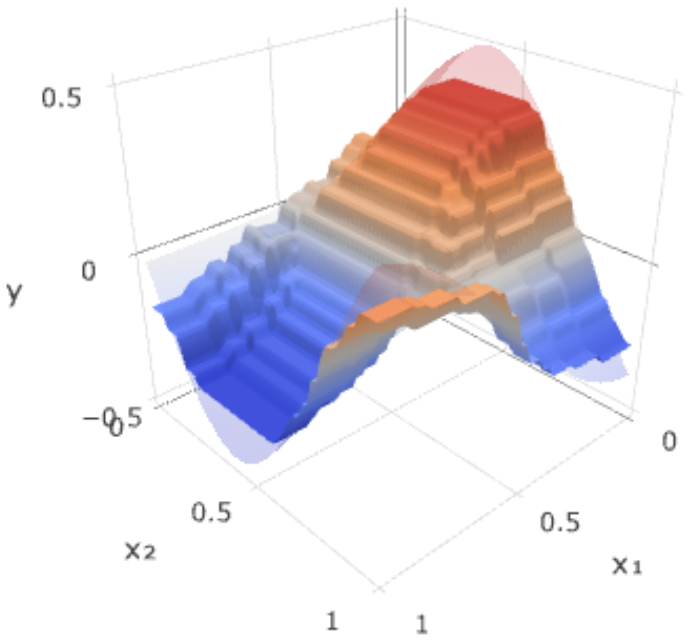
Pokud se snažíme model naučit model M predikovat hodnoty jako $\hat{y} = M(x)$, můžeme použít jednu ze základních účelových funkcí *MRE* (mean squared error), která se vypočítá tak, že pro každý vstup z trénovací množiny vypočítáme rozdíl mezi očekávanou a predikovanou hodnotou na druhou: $(\hat{y} - y)^2$. Tu potom vydělíme počtem prvků v trénovací množině.

V každé iteraci i v algoritmu gradient boosting máme k dispozici model M_i, o kterém předpokládáme, že není perfektní. Gradient boosting vezme M_i a snaží se ho nadstavit tak, aby dosáhl lepší přesnosti. Z předchozí kapitoly víme, že je potřeba přičíst derivaci účelové funkce, abychom se více přiblížili k požadovanému výsledku. Tento vztah můžeme obecně zapsat jako $M_{i+1}(x) = M_i(x) + h(x)$. Pomocí jednoduché úvahy je možné ideální funkci h odvodit takto:

$$M_{i+1}(x) = M_i(x) + h(x) = y$$

$$h(x) = y - M_i(x)$$

Výraz $y - M_i(x)$ vyjadřuje chybu v predikci mezi y a $\hat{y}$. Pokud zintegrujeme $y - M_i(x)$, získáme $\frac{1}{2}(y - M_i(x))^2 + C$ neboli zpět naši účelovou funkci MSE. Vidíme tedy, že gradient boosting je ve své podstatě *gradient descent* algoritmem [7].



Obrázek 5.4: Graf aproximace funkce pomocí GBM s 9 stromy o hloubce 3. Vidíme, že aproximace je schodovitá, ale funkci relativně přesně kopíruje.

5.3.1 Regularizace

Naivní implementace *gradient boosted models* často vede k přeučení. Regularizační techniky slouží k omezení tohoto jevu při strojovém učení.

Jednou ze základních technik regularizace v GBM je omezení iterací algoritmu. Pokud je počet stromů příliš velký, model je „ušitý na míru“ trénovacímu datasetu a na testovacích datech selže. Abychom dokázali odhadnout optimální počet iterací, je vhodné mít kromě trénovacího a testovacího datasetu ještě množinu *validační*. Její prvky leží mimo trénovací množinu a v průběhu učení se pro ni snažíme predikovat pomocí momentální iterace modelu. Sledujeme chybu jak trénovacího a validačního datasetu při každé iteraci a v momentě, kdy chyba trénovacích dat klesá, ale validačních začíná stoupat, ukončíme učení v optimální čas.

5.3.2 Shrinkage

Důležitou technikou regularizace GBM je také *shrinkage*. Jedná se úpravu *learning rate* neboli koeficientu, kterým regulujeme kroky, po kterých se algoritmus učí. Je vyzkoušeno, že hodnoty $\eta < 0,1$ dokáží model lépe generalizovat než modely naučené bez použití *shrinkage* ($\eta = 1$). Tím se snižuje přeučení, ale za cenu vyšší výpočetní náročnosti.

5.3.3 Stochastic Gradient Boosting

Heuristikou nad algoritmem gradient boosting je jeho stochastická varianta. Využívá techniky *bagging* tak, že v každé iteraci se počítá přesnost modelu pouze na podmnožině tréno-

vacího datasetu. Ukázalo se, že uvedení náhodného vybírání podmnožin v průběhu učení dokáže výrazně zvýšit přesnost i rychlost algoritmu.

Velikost těchto podmnožin je udávána pomocí parametru f, který představuje zlomek velikosti původních trénovacích dat. Pokud $f = 1$, chová se algoritmus stejně jako bez úpravy (deterministicky). Optimální hodnoty byly empiricky vyhodnoceny jako $f \in \langle 0,5; 0,8\rangle$ [6]. Tradičně se f nastavuje jako $0,5$.

Použitím této metody můžeme sledovat ještě jeden typ chyby, tzv. *out-of-bag error* (OOB). Jedná se o výpočet přesnosti predikce pro prvky, které nebyly vybrány do trénovací podmnožiny. OOB se tak chová jako vnitřní validační dataset.

5.4 Umělé neuronové sítě

Momentálně nejpopulárnější a nejmocnější klasifikační modely používané ve výzkumných oborech i v komerčních aplikacích jsou tzv. *umělé neuronové sítě* . Jsou založeny na propojení jednoduchých rozhodovacích jednotek (umělých neuronů) do komplexní sítě. Inspirací je lidský mozek a síť neuronů propojená axony, analogická k té umělé. Existují hypotézy, že jsme na správné cestě k vyřešení problému „pravé“ umělé inteligence a lidský mozek funguje na podobném principu, s tím, že lidský mozek má převahu pouze v počtu neuronů a dokonalejší učící algoritmy a heuristiky (např.: schopnost dítěte klasifikovat objekty pouze na základě několika málo příkladů). Přesto se v několika posledních letech výkon neuronových sítí zlepšil natolik, že jsou schopné řešit velmi složité úkoly, které byly ještě donedávna považovány za řešitelné pouze člověkem. Mezi tyto odvětví patří kromě dolování dat například počítačové vidění, rozpoznávání řeči, simultánní tlumočení, autonomní vozidla, hraní her nebo předpověď vývoje akcií.

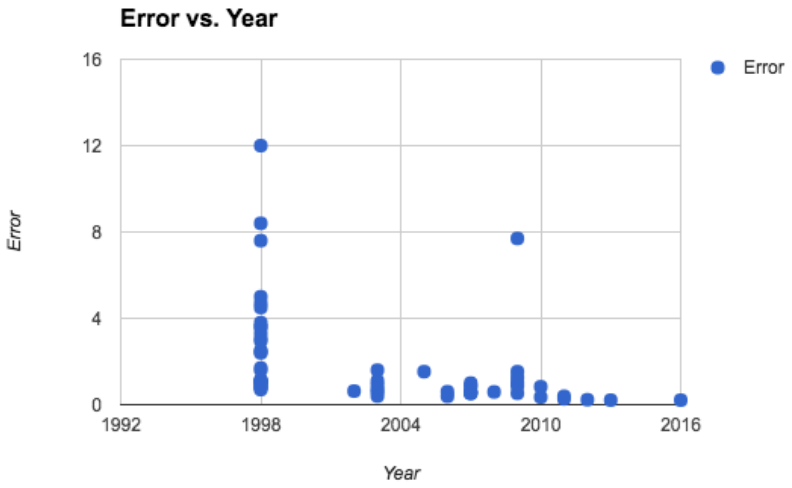
Stručná historie

Počátky neuronových sítí můžeme nalézt už na konci 50. let 20. století, kdy byl vytvořen koncept perceptronů, které se chovaly jako lineární klasifikátory. Počítačoví vědci se tehdy několik takových jednotek snažili spojit paralelně, a tím se zrodily primitivní jednovrstvé neuronové sítě. Po počátečním optimismu obklopujícím novou technologii se ukázalo, že taková síť není schopná řešit některé elementární problémy (jako například exkluzivní OR obvod). Výzkum neurálních sítí začal stagnovat v 70. letech a čekalo se, zda se toto odvětví neposune dál s příchodem vyšší výpočetní síly počítačů budoucnosti. Tomuto období se přezdívá „AI winter“.[27]

Na konci 70. let se objevily první náznaky algoritmů, které dokáží neuronovou síť učit za pomocí zpětné propagace chyby - technika, která je dnes známá jako *backpropagation*. V roce 1986 vědci dokázali vyprodukovat první využitelnou neuronovou síť schopnou řešit některé jednoduché úkoly podobně jako tehdejší zaběhlé predikční modely.[21] V roce 1993 se pomocí neuronové sítě podařilo zvítězit v soutěži o nejpřesnější predikci časových sérií. Druhá vlna popularity neuronových sítí přišla v nedávné minulosti (kolem roku 2008) se zvýšením dostupnosti GPGPU (General Purpose Graphical Processing Unit) a technologie CUDA od společnosti Nvidia (případně open-source OpenCL), která umožňuje paralelizaci výpočtů za pomocí grafických karet. Tato technologie dokázala řádově snížit čas potřebný k naučení neuronových sítí. Tento vývoj otevřel dveře tzv. *hlubokým* neuronovým sítím (deep neural networks), které obsahují několik (někdy i několik desítek) vrstev neuronů. Vychází z nich také konvoluční a rekurentní neuronové sítě, které se v současnosti těší výborným výsledkům a komerčnímu využití.

V roce 2016 společnost DeepMind vytvořila program AlphaGo, určený k hraní tradiční čínské hry Go. Tato umělá inteligence je kombinací tradičních rozhodovacích algoritmů pro hraní her, využívajících hrubou sílu (pomocí kterých se podařilo porazit například nejlepšího šachového velmistra Garryho Kasparova v roce 1996) a specializovaných neuronových sítí, naučených na milionech historických partií. Variace tohoto algoritmu poté nechali hrát proti sobě a vítěze takto „šlechtili“ mezi sebou. Výsledný program dokázal porazit světového šampiona v Go. Tato hra je spojována s nutností použít abstrakci a intuici podobnou té lidské, a proto ji někteří experti na umělou inteligenci ještě před několika lety považovali za nedoknutelnou.[23]

Na následujícím grafu můžete vidět historický vývoj přesnosti klasifikátorů na datasetu MNIST, který obsahuje tisíce ručně psaných číslic a úkolem modelu je jejich rozpoznání. Jednoduché lineární klasifikátory dosahovaly přesnosti kolem 92%, algoritmus K-nearest neighbors 98% a rozpoznávání pomocí Haarových rysů (používáno např. pro detekci obličejů) bylo účinné kolem 98,5%. Vedou však jednoznačně konvoluční neurální sítě, které v současnosti dokáží číslice klasifikovat přesněji než člověk - s úspěšností 99,77%.[4]



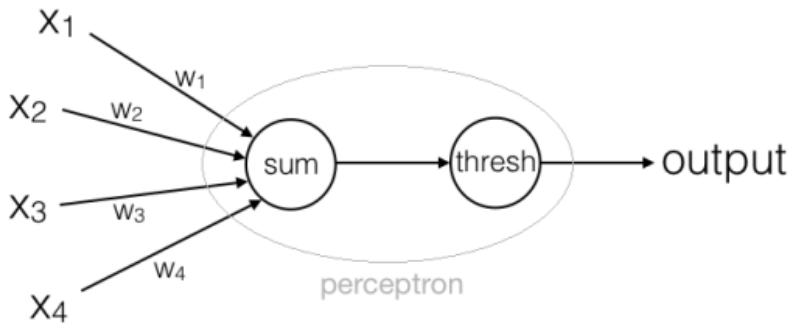
Obrázek 5.5: Časový průběh výkonnosti algoritmů pro klasifikaci datasetu MNIST. Nejlepších výsledků dosahují konvoluční neuronové sítě.

5.4.1 Perceptron

Perceptron je nejzákladnější typ umělého neuronu. Jedná se o lineární klasifikátor, který funguje na základě kombinace množiny vah $\{w_1, w_2, ..., w_i\}$ s vektorem binárních vstupů $\{x_1, x_2, ..., x_i\}$ a produkuje jeden binární výstup. Ten je vypočítán tak, že se vytvoří součin vstupů s jejich příslušnými hodnotami, a z nich se udělá součet. Tomuto postupu se říká *vážená suma* a její vzorec je $\sum_i w_i x_i$. Pokud je výsledek větší než předem daný práh (threshold), je výstupem perceptronu 1, v opačném případě 0.

Následující funkce popisuje chování perceptronu:

$$f = \begin{cases} 1, & \sum_i w_i x_i \geq threshold \\ 0, & \sum_i w_i x_i < threshold \end{cases}$$

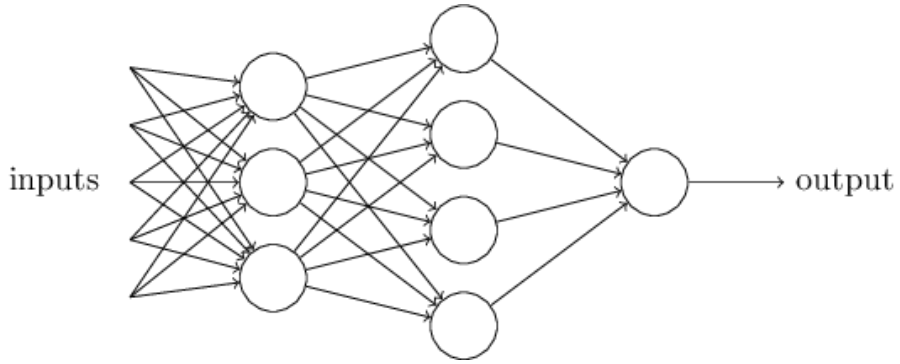


Obrázek 5.6: Jednoduché schéma perceptronu. Na levé straně vstupy s váhami, napravo výstup.

Vidíme, že perceptron představuje velmi jednoduchý matematický model. Je analogický k tomu, jak děláme rozhodnutí v našem životě na základě přiřazení vah jednotlivým okolnostem. Například se chci rozhodnout, zda se zúčastním venkovního koncertu. Hlavní faktory, které jsou pro mě relevantní jsou např. počasí a cena koncertu. Cena je pro mě důležitá, ale počasí je rozhodující faktor, a proto mu přiřadím větší *váhu*. Vytvoříme si nějaký arbitrární práh, který musíme překročit, aby pro nás mělo na koncert cenu jet, pokud ho nedosáhneme, zůstaneme doma. Denně takových rozhodnutí činíme několik a mohli bychom tak říct, že si v hlavě nevědomky sestavujeme perceptrony.

Je zřejmé, že perceptron není úplný model lidského rozhodování, ale příklad ilustruje, jak může perceptron zvážit různé druhy okolností, aby mohl dojít k rozhodnutí. A zdá se, že by složitější síť perceptronů mohla provádět velmi jemná rozhodnutí u komplexnějších a abstraktnějších problémů. Jednalo by se o sít, kde se výstupy neuronů propojeny se vstupy neuronů v další vrstvě. Takto každý neuron rozhodne pouze vlastní problém, ale jejich kombinací získáváme výkonnější model (např.: šel jsem na koncert i do divadla, takže později nepojedu na festival). Přestože má perceptron ve schématu pouze jeden výstup, na diagramu perceptronové sítě 5.7 vidíme, že z nich vede výstupů více. Jedná se však pouze o kopii stejného výstupu určenou pro vstup pro každý neuron v další vrstvě.

Neuronům, jejichž vstupem je původní vektor rysů říkáme *vstupní vrstva*. Na výstupu může být jeden neuron, sloužící pouze pro binární klasifikaci nebo více (výstupní vrstva), pro klasifikaci do několika tříd. Vrstvám, které nejsou ani vstupní ani výstupním říkáme **skryté vrstvy**. Těch může být i několik desítek (viz deep learning).



Obrázek 5.7: Neuronová síť s jednou skrytou vrstvou.

Bias a vektorová notace

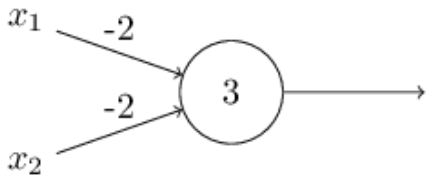
Fungování perceptronu je sice v předchozí kapitole popsáno správně, avšak v praxi se nepoužívá porovnávání s prahem, které je trochu těžkopádné, ale dává se přednost zavedení systémové chyby do perceptronu, pro kterou se používá anglické označení *bias*. V praxi označujeme bias jako b a platí, že $b = -threshold$.

Dalším zjednodušením zápisu je převést sumaci $\sum_i w_i x_i$ do vektorové notace. Vynásobení vstupů s váhami a jejich následné sečtení odpovídá aplikaci skalárního součinu na vstupní a váhový vektor. Po úpravě tedy dostaneme následující funkci perceptronu:

$$f = \begin{cases} 1, & w \cdot x + b \geq 0 \\ 0, & w \cdot x + b < 0 \end{cases}$$

Z tohoto zápisu můžeme vidět, že v perceptronu s velkým biasem bude výstup neuronu velmi těžké přepnout z kladného stavu a naopak. Na první pohled nemusí být jasné v čem je zápis s biasem prospěšný, ale při výpočtech při učení sítě dochází k velkým zjednodušením.

Funkci perceptronu si nyní můžeme představit alternativně jako výpočet základních logických funkcí (pomocí jednoho perceptronu lze vytvořit AND, OR nebo NAND). Na následujícím obrázku vidíme perceptron, který vypočítá logickou funkci NAND. Ta vrací *false* pouze pokud jsou oba vstupy *true*. Náš perceptron funguje podobně, a proto vrací 0 pouze pokud $(-2) * 1 + (-2) * 1 + 3 < 0$. NAND brána je důležitá, protože pomocí ní lze vypočítat všechny Booleovské funkce. Například XOR, který pomocí jednoho perceptronu nelze simulovat, lze vytvořit pomocí 4 NAND perceptronů.



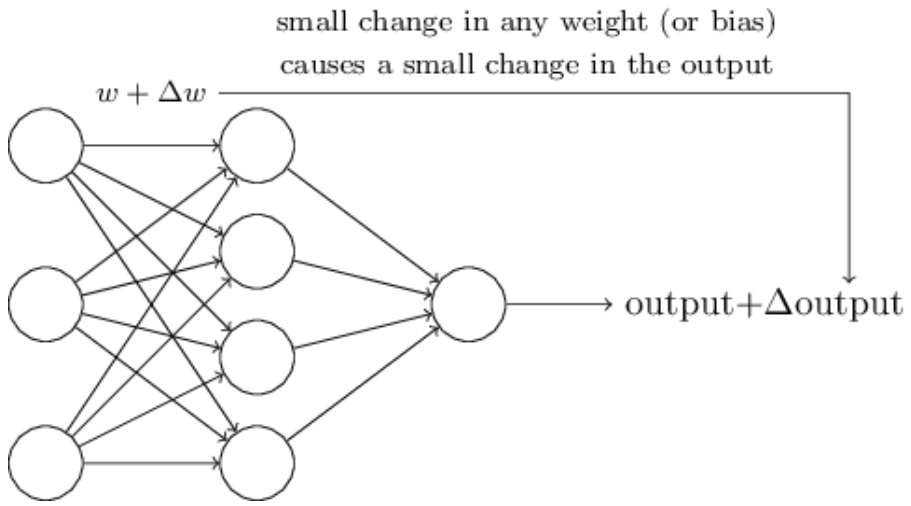
Obrázek 5.8: NAND brána simulována pomocí perceptronu.

5.4.2 Aktivační funkce

Koncept neuronových sítí se zatím nejeví jako nic užitečného. Váhy a biasy, které jsme dosud používali, jsme museli vybrat manuálně (viz příklad s koncertem). Ve skutečnosti chceme,

aby se neuronová síť chovala stejně jako ostatní metody strojového učení a potřebujeme vymyslet algoritmus, který se pomocí příkladů vstupních vektorů a výstupů dokáže naučit na datech. Síť by tak byla schopna odvodit ideální váhy a biasy, aby její model nejlépe odpovídal skutečnosti (trénovacím datům).

Velmi jednoduchý způsob by byl se snažit manuálně pozměnovat hodnoty vah a pozorovat, zda se výstup mění podle našich představ. Tento postup se poté samozřejmě nahradí učícím algoritmem, který váhy změní systematicky. Pro to, aby však cokoli podobného fungovalo potřebujeme, aby se malá změna ve váze kdekoli v síti promítla jako malá změna ve výstupu. Pokud se například snažíme klasifikovat číslice z datasetu MNIST a máme problém s přesností určení jedné z číslic, mohli bychom se pokusit najít váhu, kterou je potřeba pozměnit ke zlepšení přesnosti klasifikace pro tuto číslici. Zde však narazíme na zásadní problém v architektuře perceptronů. V praxi může malá změna váhy nebo biasu jednotlivého perceptronu v síti někdy způsobit, že výstup tohoto perceptronu se úplně převrátí, například rovnou z 0 na 1. To může způsobit, že se chování zbytku sítě úplně změní v nepředvídatelném směru a velmi radikálně. Takže i pokud problémová číslice je nyní klasifikována správně, chování sítě na ostatních obrázcích bude pravděpodobně zcela změněno způsobem, který nelze jednoduše zvrátit. Proto je obtížné předem vidět, jak postupně upravovat váhy a biasy, aby se síť přiblížila požadovanému chování.



Obrázek 5.9: Chceme, aby malá změna ve váze kdekoli v síti odpovídala malé změně ve výstupu.

Potřebujeme změnit lineární chování perceptronů, kdy při každém překročení prahu přeskočí hodnota z 0 na 1 nebo naopak. Zavedeme tedy tzv. *aktivační funkci*, do které jako vstup přichází výsledek skalárního součinu vektorů vstupů a vah sečtený s biasem pro korespondující vrstvu neuronů. Výstup z aktivační funkce se pohybuje zpravidla pouze v omezeném intervalem hodnot, nejčastěji funkce sigmoid s intervalem $(0, 1)$. Výstup z této funkce pošleme dále do sítě na vstup dalších neuronů. Tím docílíme zjemnění vlivu jednotlivých vah na konečný výsledek neuronové sítě.

Pokud označíme aktivační funkci jako σ, bude předpis funkce neuronů nyní vypadat takto:

$$f = \sigma(w \cdot x + b)$$

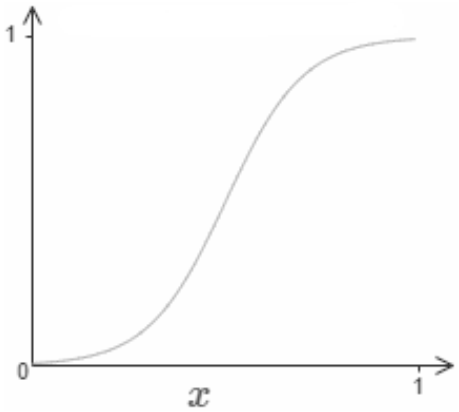
Sigmoid neurony

Tento typ neuronu je pouze klasický perceptron, jehož výsledek před odesláním dál do sítě obalíme funkcí sigmoid (někdy zvaná *logistická funkce*, viz logistická regrese 4.1.2). Byla vybrána kvůli svým příhodným vlastnostem, její obor hodnot je $(0, 1)$, a tudíž můžeme její výstup interpretovat jako pravděpodobnost příslušnosti k určité třídě. Pokud se například snažíme klasifikovat, zda student prospěje či nikoli (binární klasifikátor), a z výstupního neuronu získáme hodnotu $0,83$, můžeme říct, že model předpovídá, že student uspěje. Předpis aktivační funkce sigmoid je rovnice 5.1, po dosazení výpočtu neuronu rovnice 5.2.

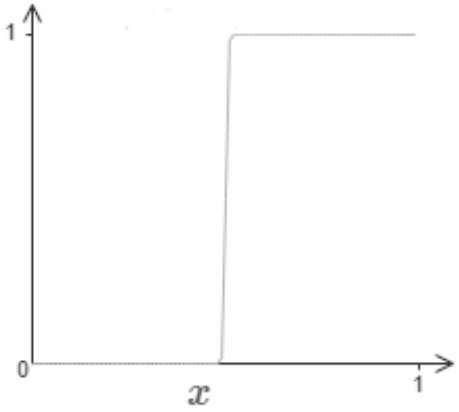
$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{5.1}$$

$$\sigma(w \cdot x + b) = \frac{1}{1 + e^{-w \cdot x - b}} \tag{5.2}$$

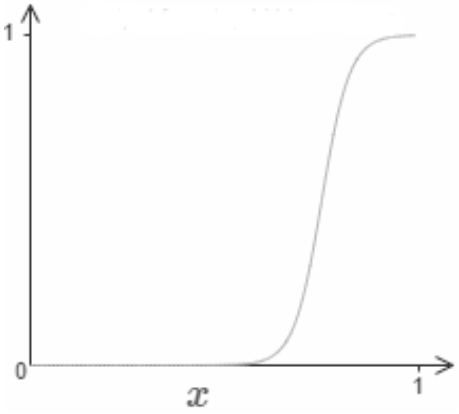
Vidíme tedy, že pokud $z = w \cdot x + b$ je velké kladné číslo, e^{-z} se blíží nule, a výstup neuronu bude tedy 1. Na druhou stranu, pokud $z = w \cdot x + b$ je velké záporné číslo, e^{-z} se blíží nekonečnu, a výstup neuronu bude tedy 0. Toto chování v extrémech je velmi podobné perceptronu. Zájímavé chování má sigmoid neuron při menších hodnotách. Tvar funkce je závislý na vstupních vahách w a biasu b. Váha ovlivňuje jak rychle funkce stoupá a následně klesá, což znamená čím nižší váha, tím více je sigmoid funkce *hladká*, a při klasifikaci si bude síť méně jistá. V momentě, kdy $w \to \infty$ mění se sigmoid funkce v tzv. *jednotkový skok* (neboli Heavisidova funkce), kdy od určité hodnoty x určené biasem b výstup přeskočí z 0 na 1. Toto chování je shodné s perceptronem bez aktivační funkce. Bias zjednodušeně určuje kde na ose x dojde k přelomu z 0 na 1. Pokud $w = -2b$, tak se $\sigma(\frac{1}{2})$ vždy rovná $\frac{1}{2}$.



Obrázek 5.10: Sigmoid funkce s váhou $w = 10$ a biasem $b = -5$.



Obrázek 5.11: Sigmoid funkce s váhou $w = 400$ a biasem $b = -200$. Vidíme, že s vysokými hodnotami w a b začíná sigmoid připomínat jednotkový skok.



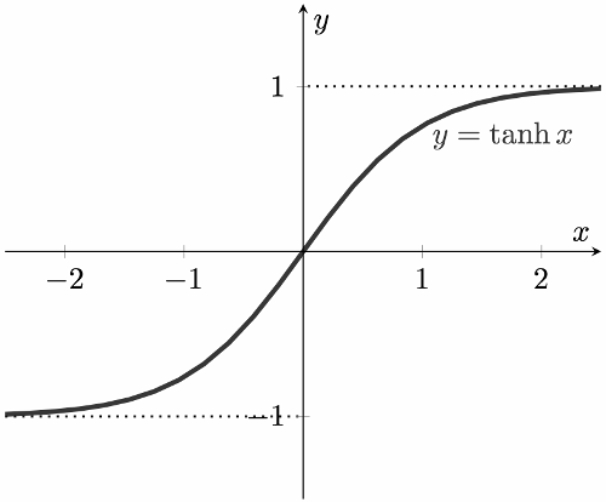
Obrázek 5.12: Sigmoid funkce s váhou $w = 28$ a biasem $b = -21$. Díky biasu vidíme posunutí přechodu po ose x.

Jiné aktivační funkce

Sigmoid nemusí být vhodný pro všechny aplikace v neuronových sítích. UKázalo se, že v některých případech se může sigmoidová neuronová síť „zaseknout“ v procesu učení. Pro některé aplikace tedy existují vhodné alternativy aktivačních funkcí. Dvě nejpoužívanější jsou hyperbolický tangens a ReLU [15].

Hyperbolický tangens

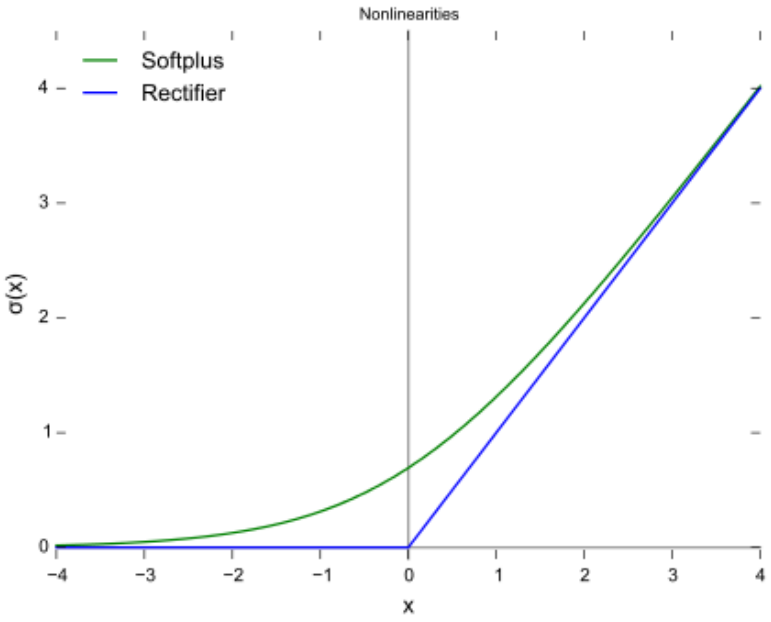
Podobně jako sigmoid je hyperbolický tangens funkce ve tvaru „S“. Rozdíl je však v tom, že výstup extrémních negativních hodnot se neblíží 0, ale -1. Vhodná vlastnost je také, že vstupy blížící se nule se budou pohybovat kolem nuly i na výstupu. Tyto vlastnosti znamenají, že při učení je neuronová síť méně náchylná k „zaseknutí“.



Obrázek 5.13: Funkce $f(x) = tanh(x)$. Pokud $x \to \infty$, tak $f(x) \approx 1$ a naopak. V případě, že $x \approx 0$, tak $f(x) \approx 0$.

ReLU

Rectifier Linear Unit je typ neuronu, který pro aktivaci používá *náběhovou funkci*. Jedná se o zjednodušenou aproximaci funkce *softplus*: $f(x) = ln(1+e^x)$. Mezi její výhody patří řídká aktivace, což znamená, že většinou pouze 50% neuronů ve skrytých vrstvách je aktivováno. Je také značně rychlejší na výpočet. Používá se hojně v oblasti vícevrstvých (hlubokých) neuronových sítí, počítačového vidění a rozpoznávání řeči [15].



Obrázek 5.14: Z tohoto grafu vidíme, že funkce $ln(1 + e^x)$ je hladkou aproximací náběhové funkce.

5.4.3 Backpropagation

Nejrozšířenější metoda učení umělých neuronových sítí, která se používá při učení s učitelem, avšak lze ji použít i při učení bez učitele, například u autoenkoderů. Je používána ve spojení s minimalizací účelové funkce (loss function) pomocí výpočtu vektoru parciálních derivací. Nejpoužívanější metodou je právě *gradient descent*. Tento algoritmus pracuje ve dvou fázích, šíření a aktualizace vah a biasů.

Fáze 1

Vstupní vektor se postupně propaguje v síti po jednotlivých vrstvách dopředu, dokud nedosáhne výstupní vrstvy. Výstup sítě se poté porovná s požadovaným výstupem pomocí účelové funkce a vypočte se hodnota chyby pro každý z neuronů ve výstupní vrstvě. Hodnoty chyb jsou pak šířeny zpětně od výstupu, dokud každý neuron nemá přidruženou hodnotu chyby, která zhruba představuje, jak moc neuron ovlivnil celkový výstup.

Fáze 2

Tyto hodnoty využijeme k výpočtu gradientu účelové funkce, pomocí kterého je optimalizační metoda schopna váhy aktualizovat tak, aby došlo k minimalizaci účelové funkce.

Důležitost tohoto procesu spočívá v tom, že při učení sítě se neurony ve skrytých vrstvách organizují tak, aby se určité neurony naučily rozpoznávat určité charakteristiky ze vstupního vektoru (případně vícedimenzionálního vstupu). Pokud je po naučení na vstupu přítomen libovolný vektor, který může být neúplný nebo obsahuje šum, neurony ve skryté vrstvě sítě se aktivují, pokud nový vstup obsahuje rysy, které připomínají ty, které se jednotlivé neurony naučily rozpoznávat na trénovacích datech.

Princip učení

Abychom mohli hodnotit výkon naší neuronové sítě, musíme si zavést konkrétní účelovou funkci na měření chyby. Mezi ty nejjednodušší patří *kvadratická účelová funkce*:

$$C(w,b) = \frac{1}{2n}\sum_{x}(y(x) - a_x)^2$$

Zde w označuje kolekci všech vah v neuronové síti a b označuje kolekci všech biasů. Proměnná n je počet vstupů při trénování. Pro každý vstupní vektor x z trénovací množiny vypočítáme rozdíl mezi výstupní hodnotou ze sítě $y(x)$a očekávaným výstupem a z datasetu. Chybu vždy umocníme na druhou a sečteme ji pro všechny vektory x. Kvadratická účelová funkce je jen jedna z mnoha účelových funkcí, které se v praxi používají, ale je užitečnější než naivní metody (jako například jednoduchá suma rozdílů) hlavně proto, že malé změny v síti se markantněji promítnou do přírůstku nebo úbytku výsledku účelové funkce.

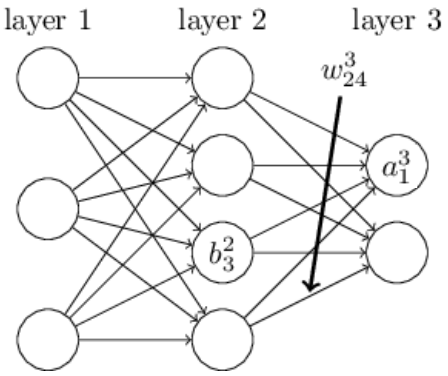
Vidíme tak, že výsledek účelové funkce se bude snižovat, pokud se budou výstupy neuronové sítě pro určitý vektor blížit k očekávaným výstupům z datasetu. Gradient descent se snaží najít optimální kolekci w a b, tak aby rozdíl mezi očekávaným a naším výstupem byl co nejmenší, a ideálně dosáhnout $C(w,b) \approx 0$. To ve výsledku znamená zpřesnění klasifikace neuronové sítě.

Aplikace gradient descent na neuronové sítě

Nyní se podíváme na to, jak vypadá gradient descent v praxi, po aplikaci na neuronovou síť. Začneme notací, jakou se označují neurony v síti.

Váha spojení mezi dvěma neurony se označuje jako w_{ij}^l. Jedná se o váhu mezi j-tým neuronem vrstvy $l-1$ a i-tým neuronem l-té vrstvy. Může se nejprve zdát, že jsou indexy i a j prohozeny, ale tato notace pomůže se zápisem vzorců při výpočtu chyb na jednotlivých neuronech. Bias neuronu je označen jako b_i^l, kde l je vrstva, v které neuron leží a i jeho pořadí ve vrstvě.

Dále si uvedeme ještě dvě proměnné z_i^l a a_i^l. Písmenem z označujeme hodnotu neuronu před vstupem do aktivační funkce (suma součinů vstupů s jejich váhami přičtená s biasem) a a označuje hodnotu neuronu po aktivaci, neboli jeho výstup, který posíláme do neuronů další vrstvy.



Obrázek 5.15: Na diagramu je zobrazeno značení jednotlivých prvků neuronové sítě a jejich indexace. w je váha, b je bias a a je výstup z aktivační funkce neuronu.

Nyní můžeme vytvořit vzorec, kterým získáme výstup jednoho neuronu z jeho vstupů 5.3 a jeho přepis do vektorové varianty 5.4.

$$a_j^l = \sigma\left(\sum_j w_{ij}^l a_j^{l-1} + b_i^l\right) \tag{5.3}$$

$$a^l = \sigma\left(w^l a^{l-1} + b^l\right) \tag{5.4}$$

Abychom mohli minimalizovat funkci, chceme vědět jak změnit všechny váhy a biasy v síti. Znamená to, že potřebujeme vypočítat parciální derivace účelové funkce C pro každou váhu a bias, tedy $\frac{\partial C}{\partial w_{ij}^l}$ a $\frac{\partial C}{\partial b_j^l}$. Chceme tedy zjednodušeně vědět, jak moc určitý neuron negativně přispívá k celkovému výstupu. Této vlastnosti se říká *chyba neuronu*. Neuron j ve vrstvě l má chybu:

$$\delta_i^l = \frac{\partial C}{\partial z_i^l}$$

Pokud bychom chtěli vypočítat chybu neuronu δ_i^L ve výstupní vrstvě L, musíme vynásobit parciální derivaci účelové funkce C podle výstupu z tohoto neuronu a_i^L (jak moc je C mění v závislosti na a_i^L). Potřebujeme také zjistit, jak rychle se aktivační funkce σ mění

v bodě z_i^L. Tuto hodnotu získáme derivací aktivační funkce. Vzorec pro chybu ve výstupní vrstvě poté vypadá takto:

$$\delta_i^L = \frac{\partial C}{\partial a_i^L}\sigma'(z_i^L)$$

Vektorovou verzi tohoto algoritmu lze jednoduše napsat pomocí operace $\circ$ (Hadamardův součin), která vynásobí mezi sebou prvky se stejnými indexy v obou maticích a vrátí matici stejných rozměrů se součiny na těchto indexech:

$$\delta^L = \nabla_a C \circ \sigma'(z^L) \tag{5.5}$$

Vektor gradientu $\nabla_a C$ obsahuje parciální derivace ve tvaru $\frac{\partial C}{\partial a_i^L}$, jako v předchozím vzorci. Pro naši kvadratickou účelovou funkci $C = \frac{1}{2}\sum_i (y - a_i^L)^2$ můžeme vypočítat derivaci v bodě a_i^L jako $\frac{\partial C}{\partial a_i^L} = a_i^L - y_i$. Pokud tedy dosadíme do vektorového zápisu kvadratickou účelovou funkci, vznikne:

$$\delta^L = (a_i^L - y_i) \circ \sigma'(z^L)$$

Tímto se dostáváme k obecnému zápisu vektoru chyby vrstvy v závislosti na chybách v předchozí vrstvě, který použijeme k *propagaci* chyby zpět v síti (odsud jméno algoritmu *backpropagation*):

$$\delta^l = (w^{l+1})^T \delta^{l+1} \circ \sigma'(z^l) \tag{5.6}$$

Zde vidíme výraz $(w^{l+1})^T$, který představuje transponovanou matici všech vah z vrstvy $l+1$. Ta je dvourozměrná, protože si musíme představit, že pokud je každá vrstva síťe úplně propojena, také každý neuron je propojen s každým z další vrstvy. Tímto postupem dokážeme získat přehled o chybách na výstupu l-té vrstvy. Aplikací Hadamardova součinu s $\sigma'(z^l)$ procházíme zpět skrz aktivační funkci ve vrstvě l. Pokud použijeme nejprve výpočet 5.5, a poté aplikujeme rovnici 5.6 kolikrát bude potřeba, jsme schopni získat chybu na jakémkoli neuronu v síti.

Posledním krokem je převést chybu neuronu na příslušné změny ve váhách a biasu, které se ho týkají. Ty můžeme napsat jako následující parciální derivace:

$$\frac{\partial C}{\partial w_{ij}^l} = al - 1_j \delta_i^l \tag{5.7}$$

$$\frac{\partial C}{\partial b_i^l} = \delta_i^l \tag{5.8}$$

Opakováním tohoto postupu po krocích η (learning rate) bychom se měli dostat k minimu účelové funkce, jinými slovy k naučení neuronové sítě. Vzorce 5.5, 5.5, 5.7 a 5.8 jsou základní pilíře, pomocí kterých lze implementovat učící algoritmus *backpropagation*. V této kapitole jsme však příliš nevysvětlili, jak byly odvozeny. Důkazy těchto rovnic jsou podle mého názoru již nad rámec seznámení s neuronovými sítěmi.

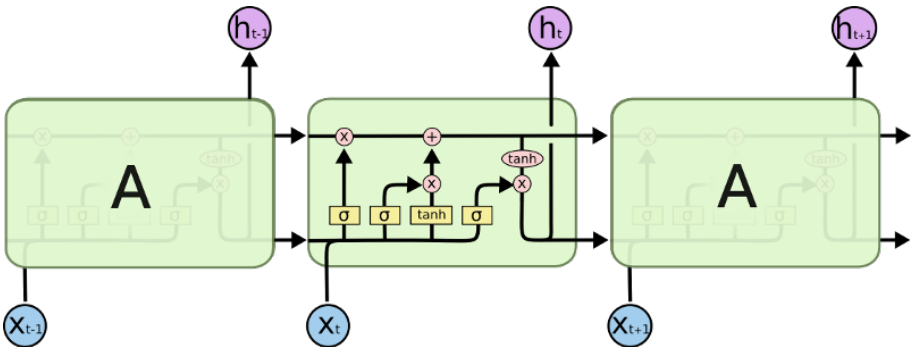
Typy neuronových sítí

Síťi, kterou jsme probírali v předchozí kapitole se přezdívá *feed-forward network*, což znamená, že výstupy z neuronů vždy putují jen do následující vrstvy a nikdy se nevrací zpět. Data v síťi tedy putují jen jedním směrem. Popsali jsme také v principu, jak lze učit sítě jakýchkoli rozměrů, tedy i sítě s velkým počtem vrstev používaných při *deep learningu*.

Rekurentní neuronové sítě (RNN)

Obohacením feed-forward neuronových sítí vznikají sítě *rekurentní*. Znamená to, že výstup z neuronu může putovat zpět do sítě a ovlivnit později jeho vstup. Vzniká tak v síti orientovaná kružnice. Tento princip slouží k simulování paměti v síti. V některých oblastech se nelze rozhodovat pouze za pomocí momentálního vstupu, ale i na základě historie [20].

Používají se zejména při zpracovávání mluveného slova, rozpoznávání rukopisů a generování textu. Dokonalejší verzí klasických RNN jsou LSTM (long short-term memory) neuronové sítě, které v této oblasti dosáhly velkých pokroků a používají je pro svoje hlasové asistenty Google, Microsoft i Apple [10].

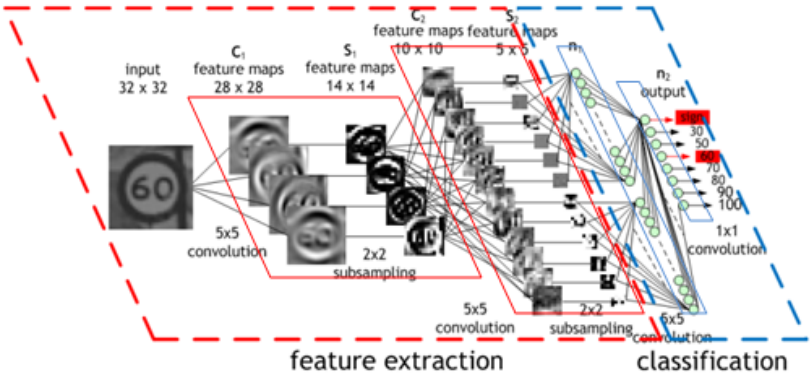


Obrázek 5.16: Diagram jednotky v LSTM síti, můžeme vidět, že se výstup vrací po „sběrnici“ zpět a může být použit na vstupu.

Konvoluční neuronové sítě (CNN)

Konvoluční sítě jsou variací na vícevrstvé neuronové sítě a jsou volně inspirovány zrakovou oblastí mozkové kůry. Vypočítavají konvoluci dvou překrývajících se částí dat (např. částí obrázku). Jsou používány zejména v oblasti počítačového vidění (detekce obsahu obrázků, rozpoznávání emocí, autonomní vozidla).

Obsahují minimální předzpracování dat, aby síť v obrázku sama našla vzory, podle kterých bude určovat jeho obsah. Čím více se blížíme výstupu, tím složitější tvary je skrytá vrstva na základě informací ze svých vstupů rozeznat. Uvnitř sítě provádíme snižování rozlišení různých části obrázku, abychom síť nezahltili daty.[13] Existuje množství implementací již naučených CNN na rozpoznání obsahu fotografií, mezi nejznámější patří Google Inception.



Obrázek 5.17: Konvoluční neuronová síť postupně provádí konvoluci na blocích 5x5 pixelů, získá z nich rysy a ty potom pošle do klasifikační vícevrstvé neuronové sítě. Ve výsledku zjistí, že na obrázku je značka s číslem 60.

Kapitola 6

Získání dat z databáze

Abychom se měli vůbec z čeho učit, musíme nějak definovat, co znamená „chování uživatele“ v systému. Pokusíme se různými operacemi nad OLTP databází získat proměnné, které by mohli kvantifikovat, jak se uživateli po určité době, která v semestru uplynula, daří. Tyto údaje bychom periodicky ukládali do datového skladu, kde by čekali na další zpracování strojovým učením. Údaje o uživatelích jsou průběžně logovány do SQL databáze. Pomocí technologie AJAX se z webového informačního systému ukládají informace o jednotlivých přihlášeních uživatelů, jak často otevírají naučné materiály, atd. Tyto údaje jsou uloženy v dedikovaných tabulkách s informacemi o čase události a uživatele a kurzu, kterých se tato událost týká. Další data, která nás mohou při strojovém učení zajímat lze najít v tabulce, která obsahuje všechny oznámkované práce a testy v semestru. Zde můžeme zkontrolovat, zda je v systému záznam o odevzdání (napsání testu) do nejpozdějšího data odevzdání. Pokud ne, můžeme ho považovat za zameškaný a tím získáme vstupní proměnnou *missed items*. Jednotlivé hodnocené položky mají však v informačním systému přidělenou váhu, podle jejich důležitosti v kontextu kurzu. Tu udává pro každý test nebo domácí úkol váha určená instruktorem. Každý zmeškaný úkol je tedy vynásoben jeho váhou, která určuje kolik procent celkové známky znamená, a tedy jestli je jeho zameškání pro výslednou známku důležité či nikoli.

Nejnáročnější na výpočet je tzv. *running average*. Instruktoři na začátku semestru dají každému testu i úkolu váhu, která určuje percentuální vliv na výslednou známku. Náš algoritmus získá tento průměr tak, že vezme všechny hodnocení v kurzu mezi začátkem semestru a momentálním datem. Z těchto hodnocení vytvoříme vážený průměr a získáme tak náš momentální „running average“. Pokud si položíme proměnnou R, známky jsou označeny proměnnou g, jejich počet n a jejich váhy w, vypadá vzorec pro running average následovně:

$$R = \frac{1}{n} \sum_{i=1}^{n} g_i w_i$$

Data vzorkujeme vždy po 5% uběhnutého semestru. Pokud tento algoritmus spustíme i pro data z minulosti od doby, kdy jsme začali informace o uživatelích logovat, získáme data o chování uživatelů během více než 3 let. Jeden z problémů byl předpoklad, že v principu každý kurz probíhá stejně a trvá stejně dlouho. To neplatí mezi různými institucemi, ale ani v rámci jedné instituce. Chování kurzů je specifické, a proto je ukládání faktů o nich načasováno individuálně. V praxi to funguje tak, že se každý den kontroluje, které kurzy se dostali do dalších 5% v rámci svého trvání a jeho snapshot se uloží do tabulky faktů v

datovém skladu. To znamená, že časová osa je pro každý kurz individuální a nedá se tedy porovnávat. Diagram datového skladu najdete v příloze A.1.

6.1 Příprava dat pro učení

Abychom mohli predikovat známky studentů v současnosti, musíme mít k dispozici model, který odpovídá kurzu, jehož je student právě součástí. Kurzy probíhají každý rok v zimním nebo letním semestru, a to tak, že nemusí mít stejný průběh, začátek ani konec. Pokud je kurz nový, nemáme samozřejmě k dispozici historická data a nemůžeme tedy vytvořit trénovací dataset.

Data je nutné „normalizovat“, k tomu slouží právě převod průběhu kurzu na procenta. Takto můžeme momentální percentuální podíl kurzu, ve kterém se student právě nachází, vložit do algoritmu strojového učení. Není tak pro nás důležité, zda je letní nebo zimní semestr, ani rok konání kurzu. Nejprve bylo nutné spustit skript na tvoření snapshotů do historie v databázi. To se ukázalo jako velmi složité jak implementačně, tak časově. Informace o výkonu uživatelů v kurzu jsou v desítkách tabulek. Například pro výpočet zameškaných úkolů nebo testů (missed items) je nutné provést následující kroky:

1. normalizace časové linky kurzu
2. kontrola splněných úkolů v daný čas
3. porovnání s nejzazším datem odevzdání
4. vynásobení zameškaných položek s korespondujícími váhami
5. sečtení a uložení do faktu *missed items*

Některé z těchto operací vyžadují `JOIN` několika tabulek a jejich zpracování. Vyhodnocování, zda je úkol zameškaný je extrémně náročná operace. Tento jev je následkem neoptimálního návrhu legacy databáze. Po spuštění skriptu, který měl vyhodnotit fakta a uložit je do datového skladu jsem zjistil že se data ukládají extrémně pomalu. To je kombinací náročnosti operace, zastaralé databáze a staršího databázového serveru. Vyhodnocování faktů všech institucí od začátku logování (cca 2013) by tak zabralo odhadem několik týdnů až měsíců. Rozhodl jsem se tak vytvořit trénovací data jen z jedné instituce, která měla o nový modul největší zájem. Vytvořil jsem zálohu databáze, kterou jsem nainstaloval na virtuální server vytvořený jen pro tento účel. I tento proces však trval více než 14 dní s častými výpadky záložní databáze a jinými technickými potíži. Samotné zpracování dat na jedné instituci o velikosti zhruba 8000 studentů trval asi 4 dny nepřetržitého provozu.

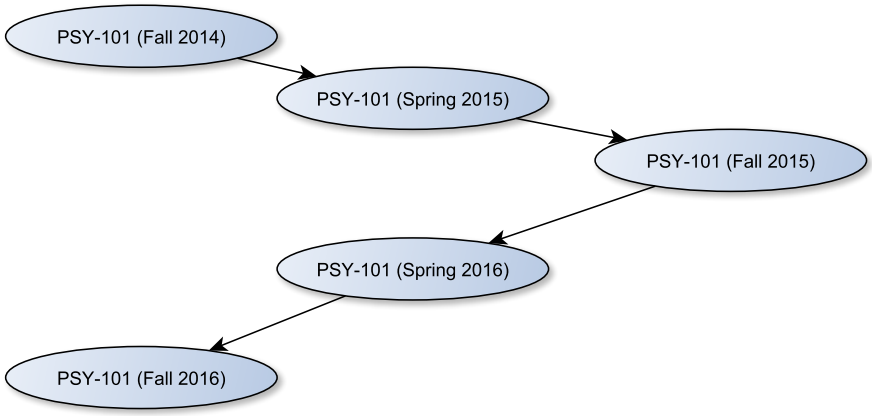
6.2 Sdružování kurzů

Vzhledem k tomu, že v jednom kurzu je zpravidla jen několik desítek studentů, nebylo by zřejmě příliš vhodné predikovat na základě modelu, který byl naučen pouze na stejném kurzu, který probíhal v minulém semestru. Proto v systému existuje systém replikací, který instruktoři hojně využívají. Funguje tak, že instruktor může při vytváření nového kurzu použít jakýkoli historický kurz jako šablonu a předvyplní se mu jeho data. Primární využití této funkce je právě pokud se kurz po semestru (nebo za rok) opakuje a instruktor chce využít materiály, osnovu, příspěvky na fóru a jiná data z předchozího kurzu, který vyučoval.

Záznam o replikaci se zaznamenává do databáze, a u každého kurzu lze jednoduše najít, zda je originální nebo má nějakého „předka“. Pomocí těchto informací lze sdružovat data z kurzů do jednoho trénovacího datasetu, aby vznikl model, který bude přesnější díky většímu počtu trénovacích dat. To bude ovšem fungovat jen za předpokladu, že se chování kurzu (počet písemek, úkolů, atd.) meziročně radikálně nezměnil, případně někdo nereplikoval kurz, který nemá s novým nic společného. Typy sdružených kurzů jsou většinou dvojí. V práci je pojmenujeme *řetěz* a *hvězda*.

6.2.1 Struktura řetěz

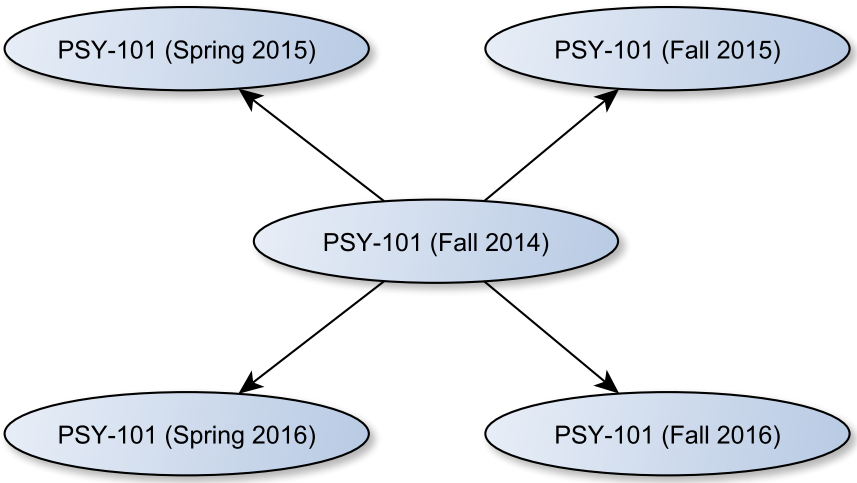
Původně jsem předpokládal, že struktura těchto shluků bude jednoznačná: instruktor každý rok replikuje svůj kurz z minulého semestru (roku) a navazuje na něj. Tímto by vznikla struktura připomínající řetěz, kdy každý kurz má jen jednoho předka, ten má jen jednoho předka, atd. Jako analogickou datovou strukturu si můžeme představit jednosměrný lineární seznam, kde každá položka ukazuje na svého předka.



Obrázek 6.1: Řetězová replikace modelových kurzů psychologie.

6.2.2 Struktura hvězda

Po zobrazení dat o replikacích z databáze do orientovaných grafů jsem zjistil, že další populární systém replikací funguje tak, že instruktoři kopírují každý semestr ze stejného kurzu. Můžeme si ho představit jako „protokurz“, ze kterého vycházejí všechny ostatní kurzy (mají vždy jen jednoho stejného předka). Graf, který vznikne tak připomíná hvězdu. Tento způsob dědění kurzů je dokonce více rozšířený než *řetěz* (asi 60% všech shluků).



Obrázek 6.2: Replikace ve tvaru hvězdy modelových kurzů psychologie.

6.2.3 Potíže se sdruženými kurzy

Při zpracování těchto shluků kurzů se objevil jeden zajímavý jev. Často vznikaly obrovské shluky, které obsahovaly několik desítek kurzů a faktové tabulky tak byly velmi objemné (až desítky MB). Strojové učení na těchto datech bylo nejen extrémně časově náročné, ale vytvořený predikční model byl velmi nepřesný (klasifikace často hraničila s náhodným odhadem).

Příčinou bylo to, že některé instituce využívaly jednoho dedikovaného kurzu jako šablonu pro ostatní. Mohlo jim to ušetřit práci s vyplňováním údajů společných pro celou školu. Například, aby se jim automaticky zkopírovalo logo nebo společné formátování určené vedením školy. Tato anomálie však narušovala všechny počáteční předpoklady a původní algoritmus sdružoval kurzy, které spolu neměly obsahově nic společného. Po důkladnějším zkoumání bylo zjištěno, že problémem byly vždy jen první iterace daného kurzu. Ty čerpaly ze šablony, ale dále se již replikovalo uvnitř kurzu (řetěz).

Řešením bylo oddělit tyto „protokurzy“ tak, že se sdružují kurzy jen se stejným *kódem kurzu* (např PSY-101). Také graf *hvězda* byla omezen tak, že po odstranění hlavního uzlu nesmí obsahovat podgrafy s více než jedním uzlem. Tím se zamezí shlukům s kurzy z rozdílných oblastí.

Kapitola 7

Experimenty s daty

Použijeme data, která byla získána z databáze, předzpracována a sdružena postupy z předchozí kapitoly. Dalším důležitým faktorem je v jaké části semestru jsme. Pochopitelně na začátku semestru, kdy má student minimum známek je nemožné přesně určit jeho finální známku na konci semestru. Stejně tak ke konci semestru tušíme, že bude výsledná známka záležet z většiny jen na momentálním průměru studenta. Teoreticky by model pro každý moment v semestru, ale to by znamenalo extrémní množství modelů, které by díky malému vzorku dat byly velmi nepřesné. Proto se k této informaci budeme chovat jako k další vstupní proměnné. Vstupní vektor rysů pro jednotlivé studenty vypadá takto:

- **Running average**
  Průběžný průměr studenta, který je v datovém skladu vypočítán pomocí vzorce, který vynásobí jednotlivé známky váhami určenými instruktorem (známka z domácího úkolu má menší váhu než závěrečný test) a následně je zprůměruje.

- **Percentage**
  Percentuální určení průběhu semestru. Například hodnota 60 znamená, že má za sebou student 60% kurzu.

- **Logins**
  Počet přihlášení do kurzu od začátku semestru. Předpokládáme, že lepší studenti tráví více času v systému.

- **Materials opened**
  Počet naučných materiálů poskytnutých instruktorem, který si student v systému otevřel. Předpoklad je opět podobný jako při počtu přihlášení: čím více, tím lepší známka.

- **Forum posts**
  Počet příspěvků, které student napsal během semestru na fórum kurzu. Podobně jako u ostatních metrik doufáme, že se vyšší počet příspěvků kladně promítne na výsledné známce (ne však drasticky).

- **Missed assignments**
  Počet zameškaných klasifikovaných úkolů v semestru vynásobených korespondujícími váhami, které představují důležitost úkolu při známkování. Tento rys by měl mít negativní vliv na výslednou známku.

Výsledkem je **Final grade**, což je spojitá proměnná nabývající hodnoty od 0 do 100, která může být pomocí příslušného hodnotícího systému převedena na známku.

7.1 Porovnání metod strojového učení na reálných datech

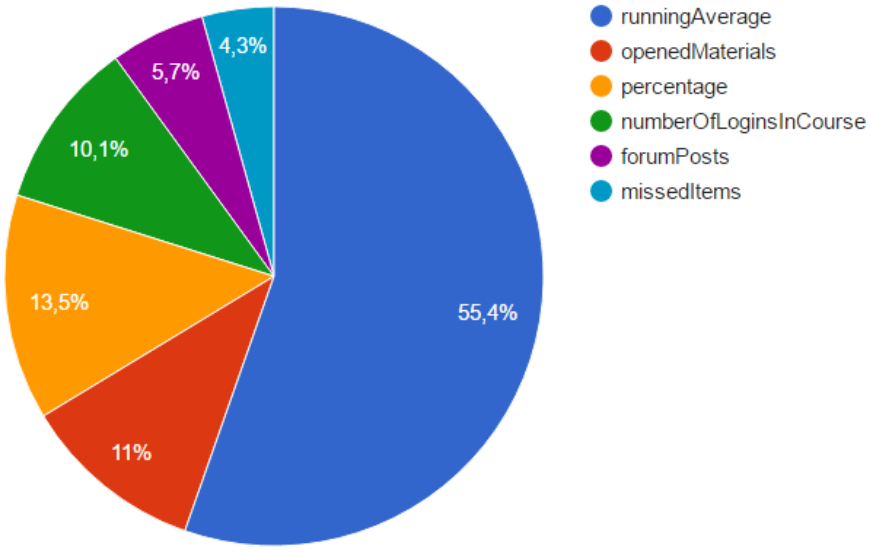
Původně byla celá práce koncipována tak, že se bude učit pomocí neuronových sítí, ale tento přístup se ukázal jako nevhodný. Kromě neuronových sítí byly vyzkoušeny metody *Generalized Linear Models* (GLM) a *Gradient Boosted Models* (GBM). Každá z těchto metod má své výhody a nevýhody v kontextu predikce známek a v této kapitole se budeme těmito rozdíly zabývat.

7.2 Učení pomocí GLM

První experimenty provedeme pomocí knihovny `glm` v jazyce R. Model se bude učit pomocí multionmiální logistické regrese. Testovací data náhodně promícháme a rozdělíme je v poměru 80/20 na trénovací a testovací vzorek. Data, na kterých provádíme benchmark jsou ze sdružených 7 kurzů psychologie z jedné instituce. Metoda učení `glm` vyžaduje pouze jeden důležitý parametr - distribuci výsledku, budeme předpokládat normální (`gaussian`).

7.2.1 Relativní vliv

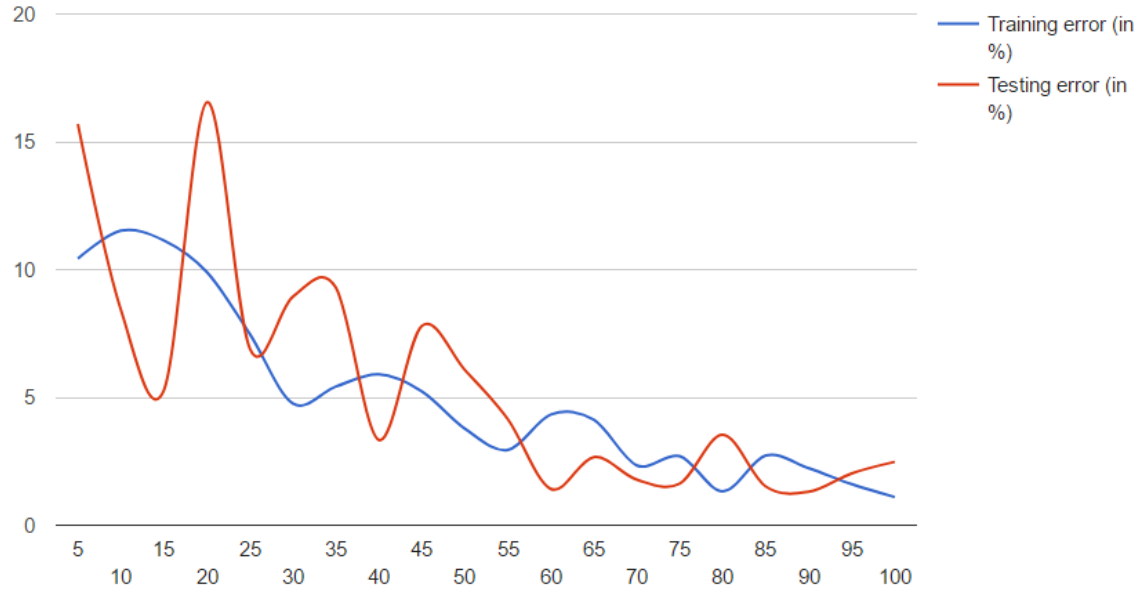
Na následujícího koláčového grafu vidíme vliv jednotlivých vstupních proměnných na výsledek:



Obrázek 7.1: Vidíme, že největší vliv na výsledek má podle očekávání známkový průměr. Nejmenší vliv v tomto datovém vzorku mají zameškaná odevzdání.

7.2.2 Přesnost predikce

Protože čekáme, že predikce bude méně přesná na začátku semestru, vytvoříme graf závislosti mezi částí uběhlého semestru a percentuální chybou v předpovězených bodech. Z následujícího grafu vidíme jednoznačnou souvislost.



Obrázek 7.2: Na následujícím grafu vidíme, že zhruba do 30% kurzu nemá GLM predikce žádný smysl, od poloviny semestru se pohybuje pod 5%

7.3 Učení pomocí GBM

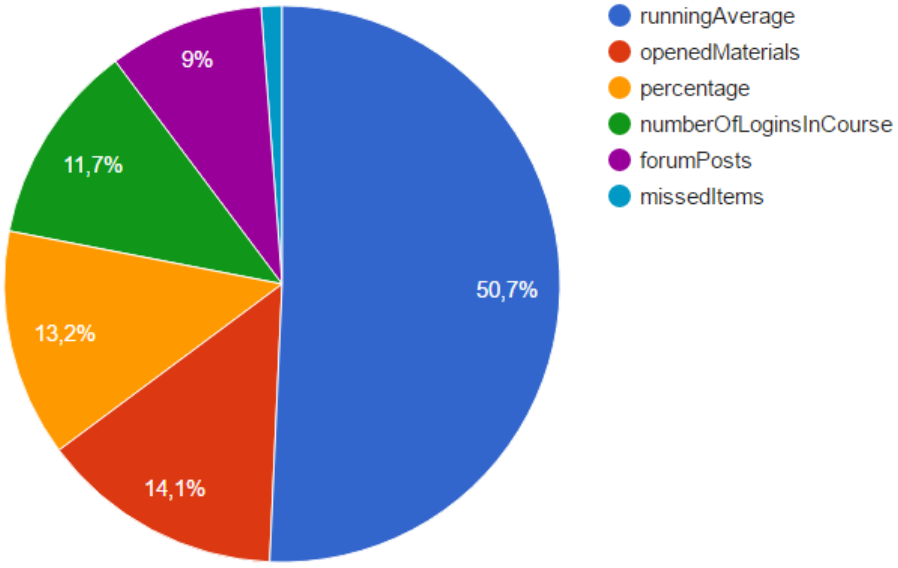
Pro další demonstraci strojového učení použijeme metodu *generalized boosted models*. Tu lze implementovat jednoduše pomocí balíčku `gbm` v jazyce R. Data jsou rozdělena stejně jako v předchozím případě.

V GBM ještě musíme nastavit zejména tyto důležité hyperparametry:

- `distribution` - Pomocí tohoto parametru lze nastavit předpokládanou distribuci dat a podle toho také způsob výpočtu cenové funkce, např.: `bernoulli` pro binární data, `multinomial` pro kategorický výstup nebo `poisson`. My předpokládáme normální rozložení, a proto vybereme `gaussian`.
- `ntrees` - maximální počet stromů, podle kterých se chceme v modelu rozhodovat (předpokládáme, že optimální počet stromů bude méně než toto číslo). Po několika spuštěních učení modelu vybírám číslo 5000.
- `shrinkage` - určuje iterativní krok při učení modelu. Menší číslo může znamenat přesnější model, avšak za cenu zdlouhavého učení. Volíme výchozí hodnotu $0,01$
- `interaction.depth` - určuje maximální interakci mezi vstupními proměnnými. Necháváme výchozí hodnotu 3.

7.3.1 Relativní vliv

Na následujícího grafu vidíme vliv jednotlivých vstupních proměnných na výsledek:



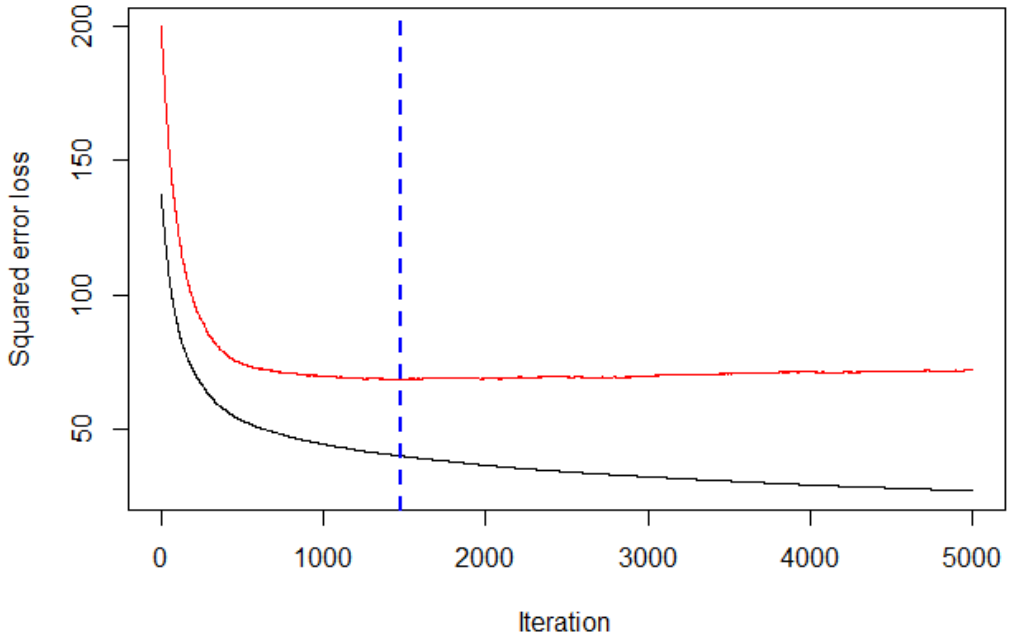
Obrázek 7.3: Graf je podobný jako u GLM, avšak vidíme znatelné rozdíly ve využití *running average* a *missed items*.

7.3.2 Optimální počet rozhodovacích stromů

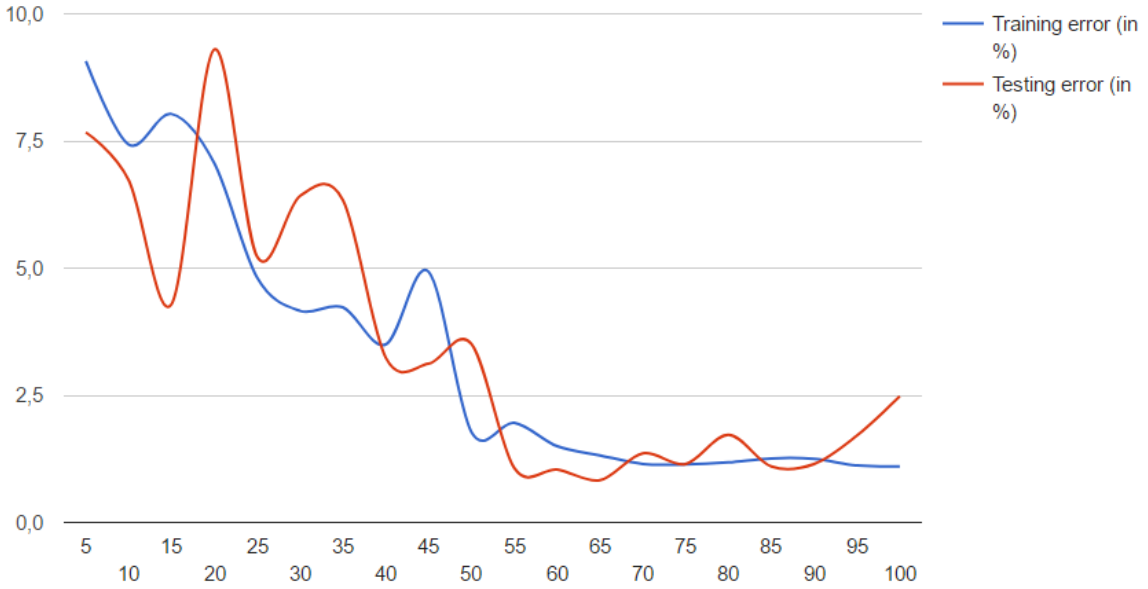
Abychom se vyhnuli přeučení, chceme učení zastavit v momentě, kdy se predikce na testovacích datech začne zhoršovat. Pro výpočet tohoto prahu má knihovna funkci `gbm.perf`, která při učení průběžne porovnává predikci na trénovacích a testovacích datech a dokáže jej najít. Na grafu 7.4 vidíme průběh cenové funkce na trénovacích a testovacích datech.

7.3.3 Přesnost predikce

Z grafu 7.5 průměrné chyby při predikci vidíme, že si GBM vede jednoznačně lépe na začátku semestru, kdy v podstatě hádá s větší přesností. Poté je na tom také o poznání lépe.



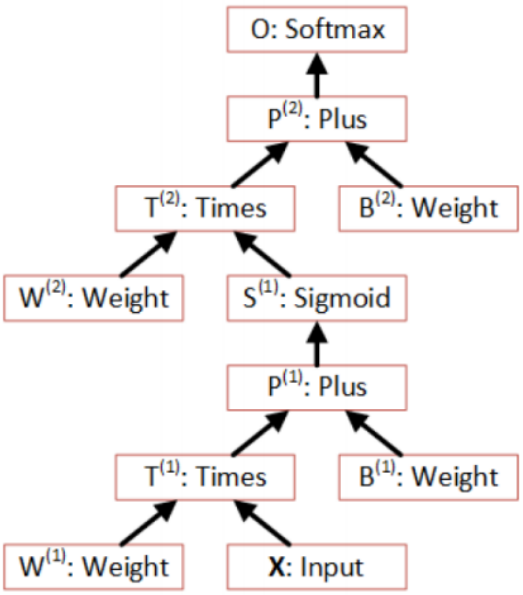
Obrázek 7.4: Černě je vyznačena chyba při preikování trénvacích dat, červeně testovacích. Ideální počet stromů, abychom se v tomto případě vyhnuli přeučení je 1389.



Obrázek 7.5: Z grafu vyplývá, že zhruba od poloviny kurzu začínáme získávat přesné predikce s průměrnou chybou přibližně ±2%. To je z pohledu ostatních kurzů výjimečně dobrý výsledek.

7.3.4 Učení pomocí neuronových sítí

Originální volba pro způsob učení. K implementaci byla použita populární knihovna TensorFlow od společnosti Google. Ta pracuje na principu vytváření výpočetních grafů (computation graph), kde uzly jsou funkce a vstupy a výstupy jsou tensory. Tensor si můžeme zjednodušeně představit jako vícedimenzionální pole (například vektor vyjadřuje 1-dimenzionální tensor). Vidíme, jak se tento přístup nabízí k tvorbě neuronových sítí.



Obrázek 7.6: Výpočetní graf. Některé z funkcí v uzlech jsme probírali v kapitole o neuronových sítích. V závorce je znázorněna arita operací (případně dimenze tensorů).

Knihovna TensorFlow má však v sobě různé druhy neuronových sítí již implementovány. Pro testování byl použitý framework TFLearn, který knihovnu TensorFlow obaluje a umožňuje tak tvořit neuronové sítě jednodušeji pomocí několika příkazů, avšak s nevýhodou ztráty absolutní kontroly nad vnitřním pracováním neuronové sítě (což pro náš problém není potřeba).

Úvaha nad rekurentní neuronovou sítí

Na začátku mého seznámení s neuronovými síťemi a strojovým učením jsem předpokládal, že pro tento problém bude ideální použít rekurentní neuronovou síť. Chceme u studentů sledovat trend v průběhu semestru, to jak si vedou závísí na jejich předchozím výkonu. Vše napovídá tomu, že budeme analyzovat časovou řadu. Na tyto problémy se v praxi RNN využívají, ale od konzultantů z branže (DeepMind) jsem slyšel, že pro můj problém je rekurentní neuronová síť nevhodná. Hlavním problémem v aplikaci RNN na problém predikce chování studentů v kurzu je **nedostatek dat**. RNN se učí velmi pomalu a potřebují velké množství dat k tomu, aby konvergovaly ke správnému řešení. Z těchto důvodů jsem rekurentní přístup zavrhnul a vydal jsem jiným směrem, který mi byl doporučen. Čas budu chápat při učení jako jen další ze vstupů ve vektoru rysů - *percentage*.

Implementace vícevrstvé neuronové sítě

Všechny neuronové síťe jsem však ještě neodmítnul. Nabízí se jednodušší postup pomocí *feed-forward* neuronové sítě. Ta by měla vytvořit model, který se bude chovat podobně jako u ostatních metod strojového učení. Záhy jsme ale narazili na jednu zřejmou nevýhodu: rychlost učení vícevrstvých neuronových sítí.

Učení jediného shluku kurzů trvalo na CPU mého notebooku (Intel i7-4702MQ) i několik minut. Stejný model trvalo vytvořit například algoritmu GBM jen několik sekund. Výpočty vah v sítích, které jsou plně propojené dokáží velmi těžit z paralelizace. TensorFlow tak umí použít GPGPU knihovnu CUDA k použití nízkoúrovňových operací grafické karty. Nainstalování CUDA ovladačů a využití dedikované karty (Nvidia GTX 760M) zrychlilo učení řádově, ale pořád nedosahovalo rychlosti jednodušších algoritmů. Pokud by neuronové sítě dosahovaly lepších výsledků než GBM a GLM dalo by se ospravedlnit pronajmutí virtuální instance s GPU kartou například u Microsoft Azure, avšak výsledky byly velmi podobné. Navíc provoz těchto instancí je extrémně drahý (i ke 100 Kč/h), a proto se jejich integrace do systému jednoduše nevyplatí.

Zde je tabulka porovnávající dobu učení a průměrnou chybu predikce různých metod na 139 středně velkých sdružených kurzech:

Tabulka 7.1: Porovnání metod pro učení

Použitá metoda	**Čas (v s)**	**Průměrná chyba (v %)**
Generalized Linear Model (logistic regression)	112	8,55%
Gradient Boosted Model	179	5,62%
Neuronová síť - 2 vrstvy x 8 neuronů (GPU)	$\sim$ 900	5,71%
Neuronová síť - 2 vrstvy x 16 neuronů (GPU)	$\sim$ 3000	6,1%

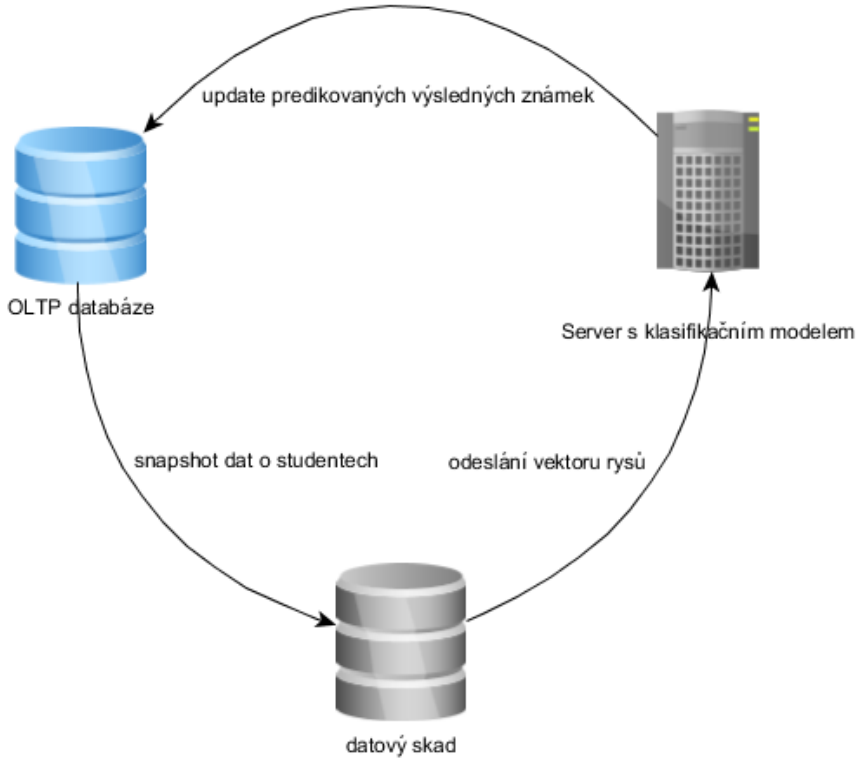
Vidíme, že neuronové sítě jsou výrazně pomalejší a dosahují podobných výsledků. Anomálie je síť s dvěma vrstvami a 16 neurony, která se učila několik desítek minut a dospěla k méně přesným výsledkům než síť s 8 neurony ve vrstvě. Dokáži si to vysvětlit jen jako výsledek přeučení. To navazuje na další nevýhodu *black box* povahy neuronových sítí. Velmi těžko se dá zjistit, proč nedosahují správných výsledků. TensorFlow obsahuje sice utilitu TensorBoard k vykreslení výpočetních grafů a metrik neuronových sítí, ale stejně je objasnění chování sítě velmi problematické. Chybí grafy relativního vlivu jako u knihoven GLM a GBM v jazyce R.

Kapitola 8

Implementace systému

8.1 Cyklus zpracování dat

Cílem je ve výsledku implementovat systém, který z OLTP databáze bude periodicky získávat kvantitativní data o uživatelích a uloží je do datového skladu. Odsud budou zaslány na server s implementací prediktivního modelu, který bude pro každého uživatele v kurzu předpovídat jeho výslednou známku. Tu následně uložíme zpět do online databáze, aby byla pro systém instantně k dispozici pomocí SQL a mohli ji zobrazit instruktorům.



Obrázek 8.1: Životní cyklus dat v systému.

8.2 Backend

Aplikace je na backendu integrována do komplexního informačního systému pro dálkové studenty (přesněji learning management system). Databáze je implementována v MSSQL na SQL Serveru 2012. Tato verze umožňuje také práci s datovými sklady, provádění periodických snapshotů a vykreslování diagramů faktových tabulek, jaké můžete vidět v příloze. Databáze je propojena s backend serverem napsaném v jazyce C# pomocí frameworku Microsoft .NET. Komunikace přes internet používá REST rozhraní implementované .NET knihovnou WebAPI.

8.2.1 Příprava dat

Import modul je schopný vytáhnout z datového skladu informace o studentech z celé instituce. Sjednotí podobné kurzy (viz kapitola o získávání dat) a připraví je pro strojové učení. Data jsme z databáze obdrželi jako C# objekty, a protože používáme na učení jazyk R, musíme data serializovat. Pro konverzi použijeme jednoduchý vlastní serializátor, který hodnoty na řádcích separuje čárkou a řádek ukončí pomocí `\n`. Získáme tak CSV (comma seperated values) tabulky obsahující řádky s vektorem rysů a výslednou známkou. Tyto soubory vložíme do ZIP archivu, který vložíme do virtuální složky přístupné ze sítě.

8.2.2 Učení

Proces učení je implementován tak, aby se dal spouštět vzdáleně. Pokud je na vzdáleném počítači nainstalována WebAPI aplikace, jazyk R a má přístup k databázi, lze na něj dedikovat všechny náročné výpočty spojené s učením modelů. Je na něm spuštěn webový server, na kterém lze zavolat REST endpoint s následujícími parametry: cesta k ZIP souboru obsahujícím CSV soubory a číslo transakce učení (ukládá se v databázi, abychom mohli porovnávat výkon různých verzí modelů v čase). HTTP endpoint se jmenuje `LearnDataPrepared`. Celý HTTP call má tento tvar:

```
http:{domain-name}/api/AI/LearnDataPrepared
```

Query parametry s cestou k archivu a číslem transakce:

```
?ZipFilePath={path-to-archive}&ID={transaction-id}
```

Na to zareaguje WebAPI a zavolá se C# funkce `LearnDataPrepared`, která okamžitě odpoví na HTTP zavolání pozitivně kódem 200 (učení začalo) a svůj kód provádí v asynchronním vlákně na pozadí. ZIP archiv stáhne a lokální cestu k němu pošle učícímu skriptu `learnp.R`. Ten provede učení a vytvořené modely s metadaty vloží do nového archivu, který pošle pomocí HTTP endpointu `StoreModel` zpět odesílateli. Ten extrahuje modely a uloží si je do složky, v které je bude mít k dispozici na predikci. Je potřeba také uložit do databáze údaje o tom, pro které kurzy máme již model připraven (případně informace o jeho kvalitě). Iniciátorem učení je *analytický modul* a části systému, která provádí učení se přezdívá *AI modul*. Ve skutečnosti se jedná o jiné části stejného .NET projektu, který je nainstalován na dvou různých počítačích připojených po internetu. Cílem je výpočetně náročné učení provést na pronajatém virtuálním stroji s několika výkonnými CPU, který po splnění úkolů může opět zaniknout. Sekvenční diagram tohoto postupu najdete v příloze A.2.

8.2.3 Predikce

Predikování není zdaleka tak náročná operace jako učení. Vše tedy může probíhat lokálně. Modely máme k dispozici z učícího procesu a v databázi máme uloženo, pro který kurz byl vytvořen který model. Analytický modul může periodicky (jednou za 5% kurzu - o časování se bude starat scheduler) vytvořit snapshot momentálních faktů uživatelů v jednom kurzu v databázi a poslat je jako objekt do funkce `PredictOneCourse` v AI modulu (ten při predikci běží lokálně). Po predikci pomocí R skriptu `predict.R` se vrátí předpovězené hodnoty v CSV formátu. Objekt se obohatí o nový sloupec *predictions* a pošle zpět analytickému modulu. Ten si predikce aktualizuje v databázi. Tento proces probíhá synchronně (čeká se na návrat funkce).

8.3 Skripty v jazyce R

R je jazyk a prostředí pro matematické a statistické výpočty. Obsahuje také velké množství balíčků na globálním repozitáři CRAN. Mezi ně patří mimo jiné `gbm`, které implementuje metodu strojového učení gradient boosted models. Na zobrazování grafů lze použít vestavěné metody nebo pokročilejší balíček `ggplot2`.

8.3.1 Paralelní učení

Po tom, co předá C# kontrolu programu učícímu skriptu R, nezačne vytváření modelu hned. Učení je náročná operace, u které můžeme těžit z paralelismu tak, že skript pustíme na každém jádře zvlášť. K tomu slouží v R balíček `parallel`. Ten umí detekovat počet jader a vytvořit cluster procesů, kde každý z nich bude vytvářet jeden model simultánně. Syntax je velmi jednoduchá: na začátku programu určíme počet možných procesů a spustíme funkci `parLapply` s polem CSV souborů a funkcí s algoritmem na strojové učení. Tato funkce dokáže paralelně aplikovat učení na každý CSV soubor v poli a jejich výsledky uložit do výstupního pole.

Výstupem učení je model `{id}.RData` a soubor s metadaty `{id}.json` pro každý shluk kurzů s číslem *id*. Dále se vytvoří soubor `metadata.csv`, který obsahuje informace o jednotlivých modelech v jedné tabulce. Na každém řádku je číslo modelu, relativní vliv proměnných na jeho výstup a průměrné chyby predikce pro trénovací a testovací set.

8.3.2 Predikce

V `gbm` je predikce pomocí modelu velmi jednoduchá. Stačí nám pouze načíst model, který budeme k predikci používat pomocí *id*, které dostaneme jako argument. Pokud na model zavoláme metodu `predict()`, R automaticky pozná algoritmus, kterým byl model naučen a použije knihovnu `gbm` k predikci. Výstupem skriptu je původní CSV obohacené o sloupec s predikcemi.

8.3.3 Frontend

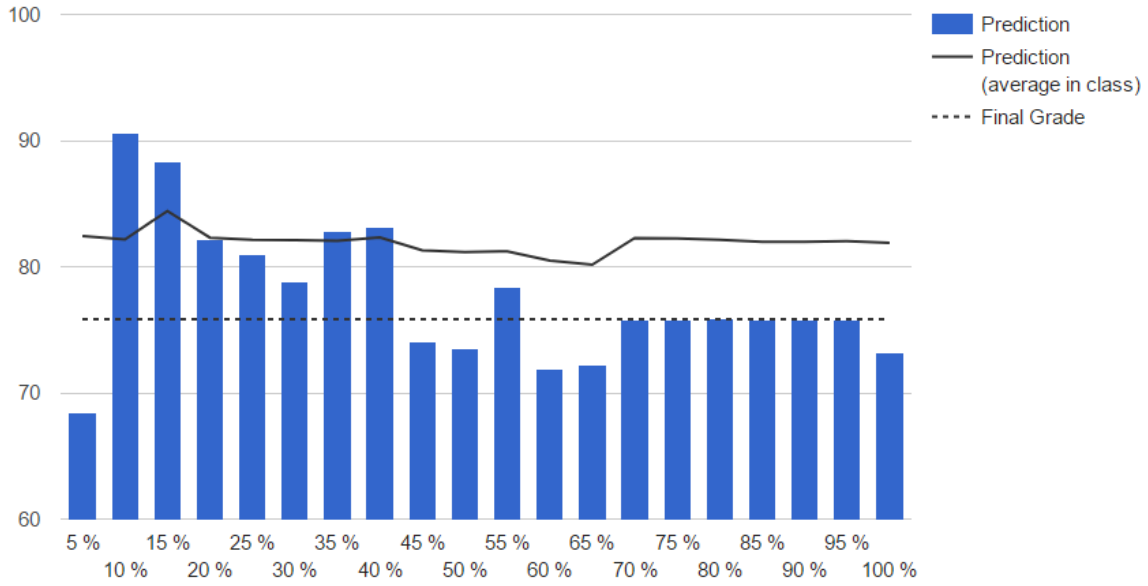
Samotné modely jsou pro predikci dostačující, ale často se chovají jako černá skříňka a není jednoduché určit, jakou účinnost model má. Jeho chování se může zdát často na první pohled nelogické, ale po bližším přezkoumání dat uvidíme, že to může být netradiční strukturou kurzů, na kterých se učil.

Informace o průběhu kurzu pro jednotlivé studenty i celou třídu by bylo vhodné graficky znázornit. Proto jsem vytvořil jednoduchý frontend, který dokáže zpracovat JSON soubor s metadaty o modelu a kurzu a zobrazit důležité informace. Frontend byl implementován jako webová aplikace pomocí JavaScript frameworku AngularJS a knihovny `angular-google-charts`.

Funkce frontendu

Nejprve se zobrazí seznam kurzů, které jsou v modelu sdruženy (viz kapitola o replikacích) s informací o počtu studentů v kurzu. Po vybrání jednoho z kurzů vidíme seznam studentů, kteří byli v tomtu kurzu zapsáni. Mají u jména napsanou svoji predikovanou (nebo po skončení kurzu finální) známku v barvě od červené po zelenou podle jejich výkonu. Po otevření si můžeme prohlédnout grafy, které ukazují tendenci predikce v předchozích časových obdobích. Můžeme tak vidět trend, kterým se studentův výkon ubírá. Kromě predikce můžeme sledovat i kumulativní změnu v ostatních metrikách - počet loginů, otevřených materiálů, zameškaných úkolů a příspěvků na fóru. Tyto hodnoty můžeme porovnat s průměrem v kurzu, který je vždy vyznačen do grafu.

Aplikace také dokáže zobrazit graf chyby při predikci v závislosti na procentu kurzu, který uběhl od začátku semestru. Chybu vykreslí jak pro trénovací, tak pro testovací data. Poslední vizualizací je koláčový graf, který zobrazuje relativní vliv vstupů na model vytvořený pomocí algoritmu GBM. Tyto informace jsou pouze pro vnitřní použití, jejich zobrazení v systému bude podmíněno speciálním uživatelským účtem. Instruktoři však budou mít přístup k predikcím a datům o průběhu studentů. Z nich by se daly vyčíst sestupné trendy a včasně zakročit u problémových studentů. Screenshot aplikace můžete vidět v příloze A.4.



Obrázek 8.2: Sloupcový graf pro jednoho studenta, znázorňující jakou známku pro něj model predikoval v průběhu semestru. Přerušovaně je známka, kterou na konci student dostal. Černě je vyznačen průměr všech studentů v kurzu.

Kapitola 9

Závěr

Z experimentů vidíme slibné výsledky, podle kterých by se na základě chování studentů v průběhu kurzu dala s určitou užitečnou přesností předpovědět jejich výsledná známka, což by instruktorům pomohlo při identifikaci problémových studentů. Zjistili jsme, že průměrně v první třetině kurzu je predikce velmi nepřesná, protože se opírá o nedostatečná data. Je otázkou zda predikci instruktorům v této fázi vůbec zobrazovat. Některé kurzy byly bohužel k naučení modelu nepoužitelné, protože jejich struktura neodpovídala předpokladům. Například kurzy, které nejsou známkované, nebo nenají v průběhu semestru úkoly a na konci studenti obdrží známku pouze na základě docházky. Tyto kurzy je nutné odstranit z procesu učení a jednoduše instruktorům oznámit, že jejich kurz je pro predikci nevhodný. Bylo by také zajímavé obohatit vektor rysů o demografická data o studentech, to je však momentálně z technických důvodů nemožné. Navíc je zde i potíž s etikou a povolením o sbírání dat o studentech. O prediktivní analýzu v e-learningových systémech se snaží již několik společností z branže. Tyto systémy jsou však proprietární a jejich fungování není objasněno. Tato práce ilustruje přístup, který využívá moderní algoritmy strojového učení a podle mého názoru by některé zde použité techniky mohly tuto disciplínu obohatit.

Základní princip učení a predikce funguje podle návrhu. Spojení technologií C# a R není sice příliš hladké, avšak dokud nebude v jazyce C# takový počet balíčků na statistiku a strojové učení, nemáme na výběr a musíme na backendu kombinovat technologie. Příjemným překvapením byl balíček `parallel` v R, který dokáže velmi jednoduše program replikovat a spustit ho paralelně na několika jádrech (při použití hyperthreadingu i vláknech). To dokáže proces učení značně zrychlit. Učení navíc funguje i vzdáleně pomocí HTTP rozhraní, což původně nebylo v plánu a zvyšuje to modularitu.

Bohužel nebyly využity neuronové sítě, jak bylo původně zamýšleno. Ukázalo se, že jsou pro naše využití zbytečně složité. Jejich učení trvalo řádově déle než u jednodušších metod a dosahovaly podobných výsledků. Díky jejich nepraktičnosti jsem dal přednost metodě *gradient boosted models*, která využívá rozhodovacích stromů. Ta splnila svoji funkci výborně. Poměr rychlosti a přesnosti byl v porovnání s ostatními metodami nejlepší. Metoda GBM vykazovala značně lepší výsledky než metoda GLM (generalized linear model), bylo by zajímavé zjistit proč tomu tak bylo, avšak to by zřejmě nebyl jednoduchý matematický problém a určitě nad rámec práce.

Dalším krokem bude implementovat automatizaci tohoto procesu, kdy bude dedikovaný server s modelem pomocí scheduleru periodicky predikovat známky pro všechny studenty a pomocí nové API aktualizovat předpokládané známky v databázi. Dále je potřeba zapracovat na hlubší integraci do systému a revizi GUI pro zobrazování dat o studentech instruktorům. Po dostatečném otestování v praxi bude na řadě vytvořit notifikační systém,

který bude automaticky hlídat výkon studentů v kurzech a podle zadaných kritérií bude instruktorům (případně i samotným studentům) posílat varování o případném propadu známky.

Literatura

[1] Alpaydin, E.: *Introduction to Machine Learning*. MIT Press, 2004, ISBN 978-0-262-01243-0.

[2] Breiman, L.: *Bagging predictors*. 1996, doi:10.1007/BF00058655.

[3] Breiman, L.: *Arcing The Edge*. 1997.
URL
http://statistics.berkeley.edu/sites/default/files/tech-reports/486.pdf

[4] Ciresan, D. C.: *Multi-column Deep Neural Networks for Image Classification*. 2012.

[5] Fayyad, U.: *From Data Mining to Knowledge Discovery in Databases*. 1996.
URL http://www.kdnuggets.com/gpspubs/aimag-kdd-overview-1996-Fayyad.pdf

[6] Friedman, J. H.: *Stochastic Gradient Boosting*. 1999.
URL https://statweb.stanford.edu/~jhf/ftp/stobst.pdf

[7] Hastie, T.: *The Elements of Statistical Learning*. Springer Publishing, 2009, ISBN ISBN 0-387-84857-6.

[8] Hawkins, D.: *The problem of overfitting*. 2004, ISBN 978-0137903955.

[9] Hilbert, M.: *3D Data Management: Controlling Data Volume, Velocity and Variety*. 2014, doi:10.1111/dpr.12142.
URL http://www.martinhilbert.net/wp-content/uploads/2015/01/BigData4Dev_Hilbert2014.pdf

[10] Hochreiter, S.: *Long short-term memory*. 1997, doi:10.1162/neco.1997.9.8.1735.

[11] Kearns, M.: *Thoughts on Hypothesis Boosting*. 1988.
URL http://www.cis.upenn.edu/~mkearns/papers/boostnote.pdf

[12] Laney, D.: *3D Data Management: Controlling Data Volume, Velocity and Variety*. 2001.
URL https://blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-Velocity-and-Variety.pdf

[13] LeCun, Y.: *LeNet-5, convolutional neural networks*. 2013.
URL http://yann.lecun.com/exdb/lenet/

[14] Mitchell, T.: *Machine Learning*. McGraw Hill, 1997, ISBN 0-07-042807-7.

[15] Nair, V.: *Rectified linear units improve restricted Boltzmann machines*. 2010. URL http://machinelearning.wustl.edu/mlpapers/paper_files/icml2010_NairH10.pdf

[16] Nelder, J.: *Generalized Linear Models*. Blackwell Publishing, 1972, doi:10.2307/2344614.

[17] Nielsen, M. A.: *Neural Networks and Deep Learning*. Determination Press, 2015.

[18] Nyce, C.: *Predictive Analytics White Paper*. 2007. URL http://www.hedgechatter.com/wp-content/uploads/2014/09/predictivemodelingwhitepaper.pdf

[19] Risdal, M.: *Exploring the Titanic Dataset*. 2016. URL https://www.kaggle.com/mrisdal/titanic/exploring-survival-on-the-titanic/notebook

[20] Rojas, R.: *Neural networks: a systematic introduction*. Springer Publishing, 1996.

[21] Rumelhart, D. E.: *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*. Cambridge: MIT Press, 1986.

[22] Russell, S.: *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2003, ISBN ISBN 978-0137903955.

[23] Silver, D.: *AlphaGo: Mastering the ancient game of Go with Machine Learning*. 2016. URL https://research.googleblog.com/2016/01/alphago-mastering-ancient-game-of-go.html

[24] Simon, P.: *Too Big to Ignore: The Business Case for Big Data*. Wiley, 2013, ISBN 978-1-118-63817-0.

[25] Snyman, J. A.: *Practical Mathematical Optimization: An Introduction to Basic Optimization Theory and Classical and New Gradient-Based Algorithms*. Springer Publishing, 2005, ISBN 0-387-24348-8.

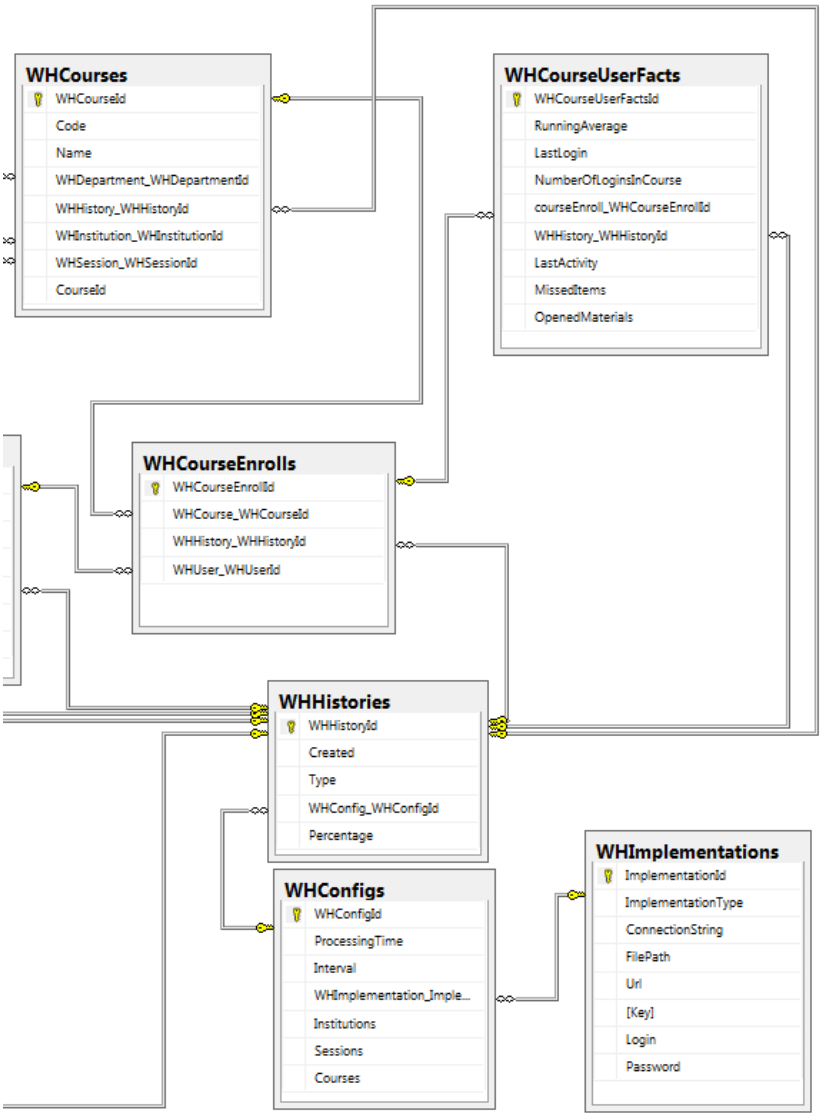
[26] Utgoff, P.: *Incremental induction of decision trees*. 1989, doi:10.1023/A:1022699900025.

[27] Werbos, P.: *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. 1975.

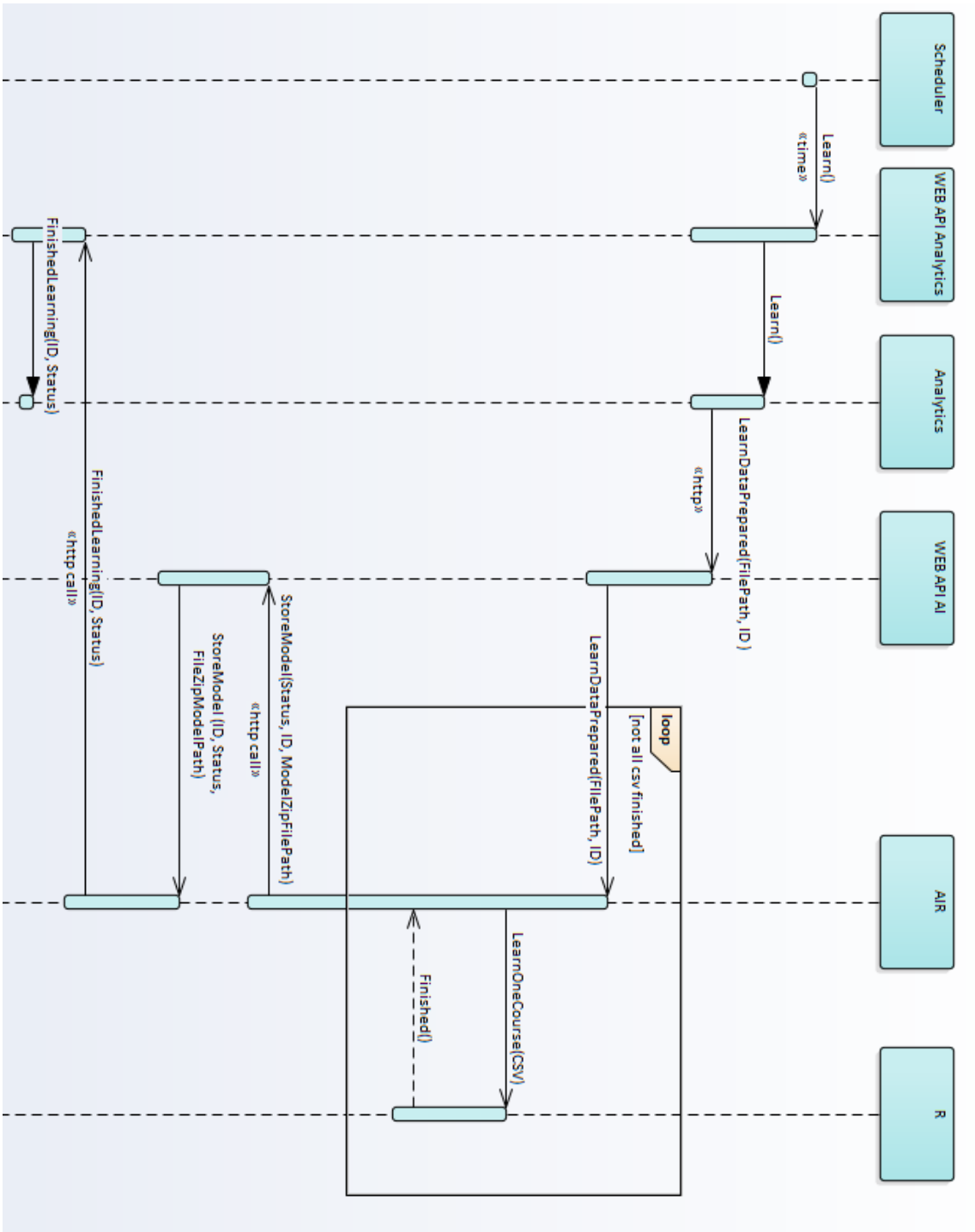
Přílohy

Příloha A

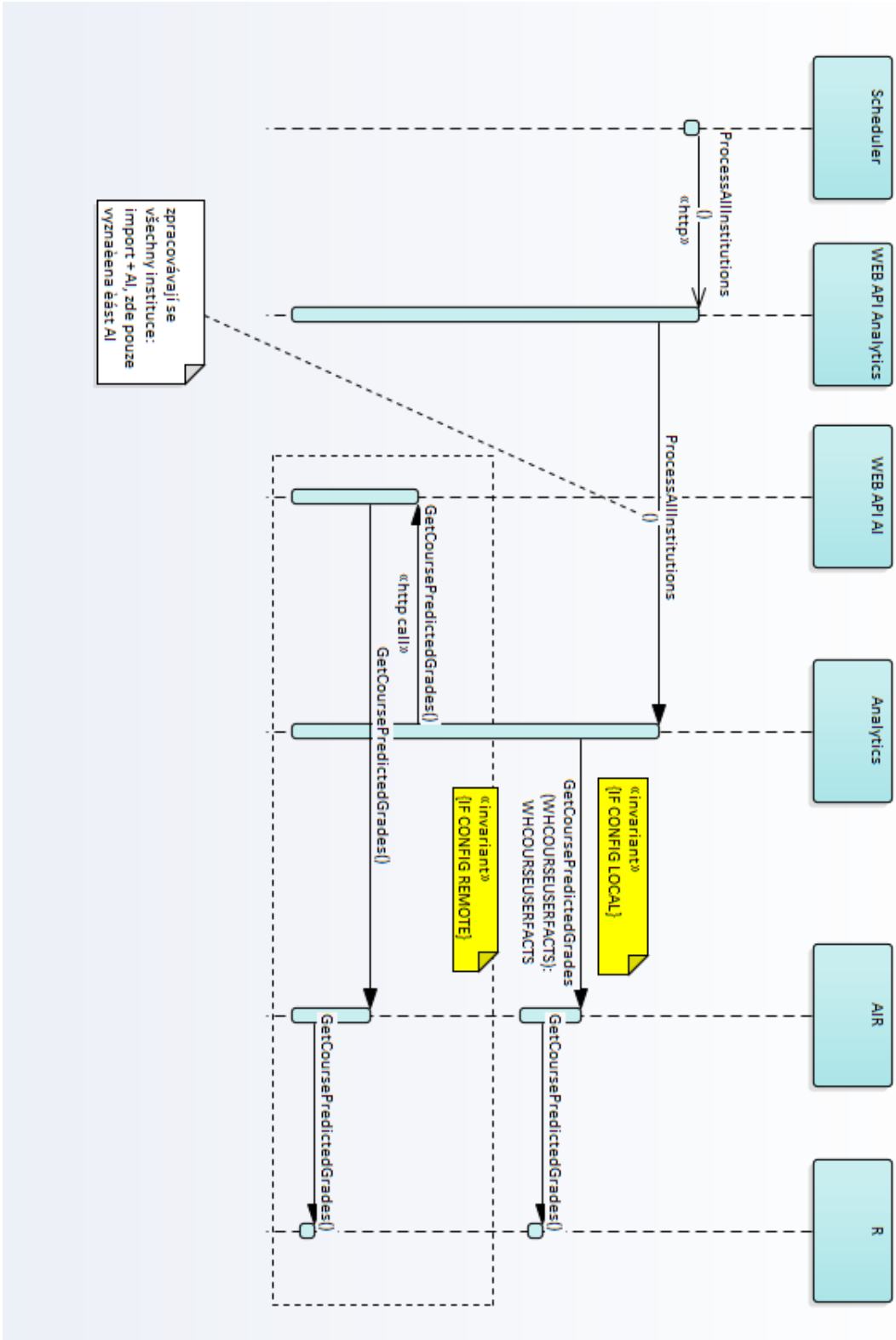
Diagram datového skladu



Obrázek A.1: Diagram datového skladu (nedůležitá část vlevo je ořezána). Vektor rysů je uložen v tabulce WHCouseUserFacts.



Obrázek A.2: Sekvenční diagram procesu učení. Volá se přes HTTP na vzdálený server, který má obsahuje nainstalované R a skripty na učení modelů. Lze spustit i lokálně, ale chceme využít paralelismu na vzdáleném virtuálním stroji s více jádry.



Obrázek A.3: Sekvenční diagram procesu predikce. Backend doplní objekt o sloupec s predikcí a vrátí ho.

Obrázek A.4: GUI frontendové aplikace. Sloupcový graf pro jednoho studenta, znázorňující kumulativní počet otevřených materiálů jednoho studenta v průběhu kurzu