České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra řídící techniky

Diplomová práce

Distribuovaný výpočet induktivních modelů

Bc. Jakub Špirk

Vedoucí práce: Ing. Ivan Šimeček, Ph.D.

Studijní program: Otevřená informatika

Obor: Počítačové inženýrství

9. května 2013

Poděkování

Rád bych na tomto místě poděkoval Ivanu Šimečkovi, Pavlu Kordíkovi, Honzovi Černému a Tomáši Bartoňovi za jejich trpělivost a neocenitelnou pomoc při psaní této práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 9. 5. 2013 .....

Abstract

Translation of Czech abstract into English. Development of good predictive regression models and classifiers is computationally very expensive. This thesis aims to present changes made to FAKE GAME software in order to enable distributive computation of models it generates. I propose two designs and implementations of necessary adjustments to the software; the first one changes the current process of model development to support multi-threading, the later is integration with Storm framework and distributive file system HDFS to support parallel execution across cluster systems.

Experiments performed on both real and artificial datasets confirm considerable speedup of development process for most real models as well as scalability of application on clusters.

Abstrakt

Tvorba kvalitních klasifikačních a regresních modelů je výpočetně velmi náročná operace. Moje práce pojednává o využití distribuovaného počítání pro zrychlení tvorby modelů generovaných aplikací FAKE GAME. Představuje dva návrhy a způsoby implementace paralelizace výpočtů pro urychlení tvorby modelů a jejich integraci s aplikací FAKE GAME. První předělává současný způsob tvorby modelu do vícevláknové podoby, druhý způsob umožňuje nasazení na distribuovaném systému s použitím frameworku Storm a distribuovaného filesystému HDFS.

Experimenty provedé na realných i umělých datech potvrzují zrychlení výpočtů pro většinu realných modelů a škálovatelnost aplikace pro distribuované systémy.

Obsah

1 Úvod **1**

2 Aplikace FAKE GAME **3**
2.1 Představení FAKE GAME . . . 3
2.2 Metaučení a šablony . . . 3

3 Proces dobývání znalostí - data mining **5**
3.1 Představení problému . . . 5
3.2 Historie . . . 6
3.3 Metodologie data miningu . . . 6
3.3.1 Metodologie 5A . . . 6
3.3.2 Metodologie SEMMA . . . 7
3.3.3 Metodologie CRISP-DM . . . 7
3.4 Podrobnější popis jednotlivých fází . . . 7
3.4.1 Porozumění problematice . . . 8
3.4.2 Porozumění datům . . . 8
3.4.3 Příprava dat . . . 8
3.4.4 Modelování . . . 9
3.4.5 Ohodnocení . . . 9
3.4.6 Nasazení modelu . . . 9
3.5 Princip modelování . . . 10
3.6 Modely aplikace FAKE GAME . . . 10
3.6.1 Základní algortimy . . . 10
3.6.1.1 Rozhodovací stromy . . . 11
3.6.1.2 K-Nejbližších sousedů . . . 11
3.6.1.3 Bayessovský klasifikátor - Bayess classifier . . . 11
3.6.1.4 Algoritmy podpůrných vektorů (SVM - Support Vector Machine) . . . 12
3.6.1.5 Neuronové sítě . . . 13
3.6.2 Složené (ensemble) modely . . . 14
3.6.2.1 Bagging (Bootstrap Aggregation) . . . 14
3.6.2.2 Boosting . . . 15
3.6.2.3 Stacking . . . 15
3.6.2.4 Cascade Generalization . . . 15
3.6.2.5 Arbitrating . . . 16

3.6.2.6 Cascading . . . 16
3.6.2.7 Delegating . . . 16
3.6.2.8 Area specialization . . . 16
3.6.2.9 Cluster divide . . . 17
3.6.2.10 Thresholding . . . 17

4 Paralelismus a paralelní programování 19
4.1 Paralelismus . . . 19
4.2 Architektura paralelních systémů . . . 19
4.2.1 Popis . . . 19
4.2.2 Flynnova taxonomie . . . 20
4.2.2.1 SISD model . . . 20
4.2.2.2 SIMD model . . . 20
4.2.2.3 MISD model . . . 21
4.2.2.4 MIMD model . . . 21
4.2.3 Paměťové architektury . . . 22
4.2.3.1 Sdílená paměť . . . 22
4.2.3.2 Distribuovaná paměť . . . 23
4.2.4 Co je potřeba vzít v úvahu při návrhu paralelního programu . . . 23
4.2.4.1 Amdahlovo pravidlo . . . 23
4.2.4.2 Problematika *Load Balancingu* a *Granularity* . . . 24
4.2.4.3 Datová konzistence . . . 25
4.2.4.4 Podmínky korektnosti paralelního programu . . . 25
4.3 Paralelní programování v jazyku JAVA . . . 25
4.3.1 Sdílená paměť - vlákna . . . 25
4.3.1.1 Plánování vláken . . . 26
4.3.1.2 Synchronizace vláken . . . 27
4.3.2 Distribuovaná paměť - projekt Storm . . . 28
4.3.2.1 Storm cluster . . . 28
4.3.2.2 Základní abstrakce . . . 29
4.3.2.3 Jednoduchá topologie . . . 29
4.3.2.4 Spolehlivost . . . 31
4.3.2.5 HDFS . . . 32

5 Návrh a realizace 33
5.1 Paralelizace FAKE GAME . . . 33
5.1.1 Návrh . . . 33
5.1.1.1 Rozhraní ensemblů v FAKE GAME . . . 33
5.1.1.2 Balíček game.parallel . . . 34
5.1.2 Realizace . . . 37
5.1.2.1 LearnManager . . . 37
5.1.2.2 Arbitrating . . . 38
5.1.2.3 Bagging . . . 38
5.1.2.4 Cluster divide . . . 40
5.1.2.5 Stacking . . . 40
5.1.2.6 Thresholding . . . 40

5.1.2.7 Classifier model . . . 40
5.2 Game-storm - návrh . . . 41
5.2.1 Popis realizace . . . 41
5.2.1.1 DataProducer . . . 42
5.2.1.2 CreateDataSpout . . . 42
5.2.1.3 DataProcessBolt . . . 43
5.2.1.4 LearningBolt . . . 43
5.2.1.5 EvaluateBolt . . . 44

6 Testování a zhodnocení zvoleného řešení 45
6.1 Porovnání paralelních implementací . . . 45
6.2 Zrychlení oproti sekvenční verzi . . . 45
6.2.1 Testovací data a šablony . . . 46
6.2.2 Změřené výsledky . . . 46
6.3 Velké datasety . . . 48

7 Závěr 51
7.1 Možná vylepšení . . . 51

A Obsah přiloženého CD 55

Seznam obrázků

2.1 Ukázková metalearningová šablona 4

3.1 Ukázková data pro rozhodovací strom 3.2 11
3.2 Rozhodovací strom vzniklý na základě dat 3.1 12
3.3 K-nejbližších sousedů 13
3.4 Model umělého neuronu 13
3.5 Schéma typického ensemble algoritmu 15

4.1 Kategorie paralelních systémů dle Flynnovy taxonomie 20
4.2 Model SISD architektury, blok PU symbolizuje procesor 20
4.3 Model SIMD architektury 21
4.4 Model MISD architektury 21
4.5 Model MIMD architektury 22
4.6 Model architektury se sdílenou pamětí 22
4.7 Model architektury s distribuovanou pamětí 23
4.8 Amdahlovo pravidlo 24
4.9 Storm cluster 29
4.10 Storm topologie - logický pohled 30
4.11 Storm topologie - fyzický pohled 31
4.12 Příklad topologie obsahující jeden spout a jeden typ boltů 31

5.1 Celkový pohled na aplikaci FAKE GAME 34
5.2 Struktura tříd pro klasifikátory 35
5.3 Výsledná struktura rozhraní 36
5.4 Rozhraní třídy LearnManager 37
5.5 Struktura tříd pro modely 39
5.6 GAME-Storm celkový pohled 42
5.7 Topologie GAME-Storm 43

6.1 Porovnání paralelního a sekvenčního výpočtu 48

Seznam tabulek

6.1 Rozdíl délky paralelního výpočtu bez a se Stormem 46
6.2 Testovací šablony a data 47
6.3 Zrychlení výpočtu při použití paralelizace 47

Kapitola 1

Úvod

Výpočet kvalitních klasifikátorů a regresních modelů je časově velmi náročný. Open source aplikace FAKE GAME, kterou vyvíjí skupina CIG na fakultě informačních technologií ČVUT, poskytuje uživateli možnost vyvíjet modely bez vysoké odborné znalosti problematiky dataminingu i bez hlubších znalostí struktury modelovaných dat. Paralelizaci aplikace FAKE GAME jsem se věnoval i ve své bakalářské práci. Ta však prošla v roce 2010 celkovým redesignem a vývoj modelů, které generuje, probíhá v současné době sekvenčně.

Mým prvním cílem je urychlit tento proces pomocí paralelizace výpočtu modelů v existující aplikaci pro systémy se sdílenou pamětí. Vedoucí projektu zároveň plánují vytvořit aplikaci, která přes webové rozhraní nabídne uživatelům možnost real-time zpracování modelů. Mým druhým cílem je tudíž navrhnout nový modul, fungující jako distribuovaný server pro počítání modelů z velkých objemů dat.

Pro úplnost textu zde znovu uvedu některé informace ze své bakalářské práce.

Na začátku práce představím aplikaci FAKE GAME a pro ni specifický způsob výpočtu modelů.

Druhou kapitolu budu věnovat popisu problematiky dobývání znalostí (data miningu), vysvětlení procesu a přiblížení některých metod při něm používaných.

Algoritmy modelování induktivních modelů jsou výpočetně velmi náročné. Proces paralelizace je proto, vedle optimalizace používaných algoritmů, další možností zrychlení výpočtu a tím i získání modelu. Následující kapitolu budu tedy věnovat nástinu problematiky paralelního programování.

Poté se zaměřím na rozbor možností paralelizace výpočtu modelů generovaných pomocí aplikace FAKE GAME, popíši způsob implementace zvolených možností a představím provedené experimenty a jejich výsledky.

V závěru zhodnotím zvolená řešení a postupy.

Kapitola 2

Aplikace FAKE GAME

2.1 Představení FAKE GAME

Projekt FAKE GAME je vyvíjen výzkumnou skupinou CIG (Computational Intelligence Group) na fakultě informačních technologií ČVUT. Cílem projektu je vývoj softwaru pro zautomatizování procesu extrakce informací z dat. V posledních letech prošla aplikace celou řadou změn, přičemž jednou z nejvýznamnějších je celková změna konstrukce modelů.

Původní FAKE GAME vychází z algoritmu Multilayered Iterative Algorithm (MIA) [14]. Model systému zde byl reprezentován formou sítě. Konstrukce modelu složitého systému probíhala vrstvu po vrstvě během procesu učení, přičemž učení - úprava parametrů přenosové sítě - probíhalo za použití dat popisujících modelovaný systém. Dnešní FAKE GAME využívá generování metalearningových šablon a ensemble modelů k získání finálního modelu.

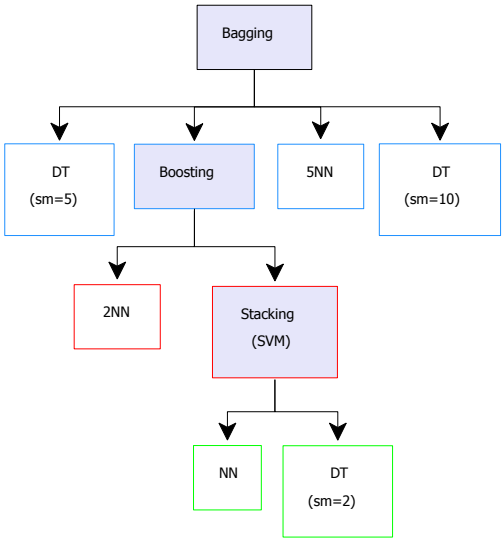
2.2 Metaučení a šablony

Metalearning (metaučení) je oblast strojového učení, kdy jsou učící algoritmy aplikovány na výsledky z experimentů. Jedná se tak vlastně o učení se o učení, kdy se učící systém pomocí metalearningových algoritmů snaží odhadnout nejvhodnější algoritmy pro modelování určitého typu dat. Podobně jako samotné trénování modelu probáhí i metaučení ve dvou fázích [17]:

- Fáze získávání znalostí (trénovací fáze):
  Probíhá jako získávání potřebných informací ze vstupních datových souborů (metaatributů a charakteristik). Výsledkem jsou metaznalosti o vstupním souboru a vhodný klasifikátor nebo množiny klasifikátorů (učící strategie) pro tento soubor.

- Fáze doporučovací (testovací fáze):
  V té jsou metainformace získané v trénovací fázi použity ke konfiguraci a doporučení optimální učící strategie. Metaatributy získané z aktuálního datového setu jsou porovnávány s metaatributy získanými v předchozí fázi a na základě jejich podobnosti dojde k výběru a doporučení optimální strategie.

Metalearningová šablona je množinou metalearningových algoritmů, modelovacích či klasifikačních algoritmů a jejich parametrů v hierarchické struktuře. Modely a klasifikátory níže v hierarchii jsou více specializované na konkrétní podmnožiny instancí dat nebo atributů. Tímto způsobem je tak vstupní problém dekomponován na menší podproblémy a výsledné řešení (model) se skládá z řešení těchto subproblémů. Nejlépe si lze šablony představit jako strom, kde listy tvoří základní algoritmy a uzly představují ensemble algoritmy.



Obrázek 2.1: *Obrázek ukazuje šablonu, při jejímž použití je na celý vstupní dataset aplikován algoritmus Bagging 3.6.2.1, který vygeneruje 4 podmnožiny dat pro členy ensemblu. Druhá podmnožina je pak použitá pro trénování KNN klasifikátoru 3.6.1.2 pomocí Boostingu, který zaručí, že vzorky, na kterých vykazoval KNN vysokou chybovost, budou s větší pravděpodobností použity v trénovacím datasetu Stackingu neuronové sítě a rozhodovacího stromu.*

Šablony mohou být vytvořeny ručně za použití expertních znalostí, cílem aplikace FAKE GAME je ale celý proces učení automatizovat [6]. FAKE GAME tak nabízí možnost automatické tvorby šablony pomocí genetického programování a zaručuje optimalitu pro konkrétní data set. Paralelizace procesu tvorby šablon je mimo rozsah této práce, zde se věnuji pouze paralelizaci procesu učení.

Kapitola 3

Proces dobývání znalostí - data mining

3.1 Představení problému

„Proces dobývání znalostí je proces netriviální extrakce implicitních, dříve neznámých a potenciálně užitečných informací z dat [21].“

Data mining je proces analyzování dat z různých perspektiv a jejich shrnutí do pro nás prospěšné, prakticky využitelné informace, která přinese užitek v podobě trefných prognóz a použitelných závěrů a klasifikací. Dnes má velmi široký rozsah využití - ať již ve vědeckém výzkumu při analýze genetické informace, při tvorbě modelů pro předpovědi počasí, v medicíně při hledaní faktorů ovlivňující výskyt chorob, v průmyslovém využití pro vyhledávače, plánovací systémy a finanční modeling, a v rámci marketingu při odhadu chovaní zákazníků či u řízení lidských zdrojů při hledaní vlastností úspěšných zaměstnanců.

Podle [16] mezi nejčastější úlohy řešené pomocí data miningu patří:

- Popis dat

  Vizualizace

  Sumarizace

- Hledání „nugetů“

  Dominantní struktury, asociační pravidla

  Segmentace, shluková analýza, popis rozdělení dat

- Predikce

  Klasifikace (predikce kategoriální proměnné)

  Regrese (predikce spojité proměnné)

  Časové řady (predikce závislé na čase)

3.2 Historie

Poprvé se aktivity, které dnes označujeme jako data mining, začínají objevovat v 60. letech minulého století. Šlo například o využívání techniky regresní analýzy s automatickým výběrem proměnných či rozhodovacími stromy, většinou se však jednalo o ojedinělé či spíše akademické záležitosti.

S rozvojem statistických metod a umělé inteligence, masivním používáním databázových aplikací a samozřejmě i technologickým rozvojem počítačů zaznamenávají data miningové postupy první praktické použití v průběhu sedmdesátých a osmdesátých let. Slovní spojení data mining tehdy ovšem stále mělo spíše hanlivý přídech: označovalo „vyzobávání rozinek“ z dat, hledání korelací ve velkých datových souborech, které – jak známo ze statistické teorie – je vystaveno obrovskému nebezpečí, že „objeví“ pouze nahodilé fluktuace v datech, bez možnosti zobecnění a praktického využití.

Výrazný rozvoj zaznamenává data mining počátkem 90. let 20. století. V té době již existují metody umožňující se výše zmíněnému problému vyhnout a navíc roste poptávka po nástrojích pro získávání potřebných informací ze stále se zvětšujících objemů dat, především z komerční sféry, což napomáhá k zavedení a uznání data miningu jakožto oboru aplikované vědy a k jeho širokému použití v komerční praxi. Nárůst množství aplikací v oblasti data miningu má za následek vznik řady specializovaných softwarových produktů či konzultačních firem zabývajících se touto problematikou.

Vzhledem k složitosti a rozdílnosti procesů spjatých s data miningem se během 90. let objevily metodologie snažící se tento proces formalizovat.

3.3 Metodologie data miningu

V [15] si můžeme v předmluvě přečíst následující motto: „Máte-li špatné údaje, ale dokonalou logiku, pak jsou vaše závěry zcela jistě mylné. Dopřejete-li si tudíž sem tam nějakou trhlinu v logickém uvažování, můžete díky náhodě dospět ke správnému závěru.“

Právě výše zmíněné motto vystihuje potřebu vzniku ucelené metodologie. Ty vznikají v průběhu 90. let jako produkt softwarových firem - např. metody 5A firmy SPSS, nebo metoda SEMMA společnosti SAS, a rovněž jako spolupráce výzkumných či komerčních institucí jako „softwarově nezávislé“ - např. CRISP-DM.

3.3.1 Metodologie 5A

Metodologii „5A“ nabízí firma SPSS jako svůj pohled na proces dobývání znalostí:

- Assess – posouzení potřeb projektu
- Access – shromáždění potřebných dat
- Analyze – provedení analýz
- Act – přeměna znalostí na akční znalosti
- Automate – převedení výsledků analýzy do praxe

3.3.2 Metodologie SEMMA

Softwarový produkt firmy SAS, Enterprise Miner, vychází z vlastní metodologie „SEMMA“ pro dobývání znalostí:

- Sample – vybírání vhodných objektů
- Explore – vizuální explorace a redukce dat
- Modify – seskupování objektů a hodnot atributů, datové transformace
- Model – analýza dat a vytvoření modelu
- Assess – porovnání modelů a interpretace

3.3.3 Metodologie CRISP-DM

Dle [2] vznikla metodologie CRISP-DM (CRoss-Industry Standard Process for Data Mining) v rámci výzkumného projektu Evropské komise. Cílem projektu je navrhnout univerzální postup (standardní model procesu dobývání znalostí), který bude použitelný v nejrůznějších komerčních aplikacích. Vytvoření takovéto metodologie umožní řešit rozsáhlé úlohy dobývání znalostí rychleji, efektivněji, spolehlivěji a s nižšími náklady. Kromě návrhu standardního postupu má CRISP-DM nabízet „průvodce“ potenciálními problémy a řešeními, které se mohou vyskytnout v reálných aplikacích. Na projektu spolupracují firmy NCR (přední dodavatel datových skladů), Daimler Chrysler, ISL (tvůrce systému Clementine) či OHRA (holandská pojišťovna). Všechny tyto firmy mají bohaté zkušenosti s reálnými úlohami dobývání znalostí z databází. Metodika CRISP-DM uvádí následující kroky:

- Business Understanding – porozumění problematice
- Data Understanding – porozumění datům
- Data Preparation – příprava dat
- Modeling – modelování
- Evaluation – vyhodnocení výsledků
- Deployment – využití výsledků

Terminologie zavedení v rámci této metodiky dnes tvoří standard v oboru.

3.4 Podrobnější popis jednotlivých fází

Všechny uvedené metodologie popisují v podstatě tentýž proces, který lze podle [16] podrobněji popsat následujícím způsobem:

3.4.1 Porozumění problematice

Počáteční fáze je zaměřena na porozumění cílům projektu z obchodního hlediska a poté na přeměnu těchto znalostí na definici dataminingového problému:

- Cíle podnikání – porozumět potřebám klienta, identifikovat cíle a jejich provázanost
- Průzkum situace – vyjasnění terminologie, analýza návratnosti investic, analýza prostředků a požadavků na projekt, zvážit omezení
- Stanovení cílů dataminingového projektu v technických pojmech – analýza obchodních požadavků, návrh řešení a typ model
- Vytvoření plánu projektu – jednotlivé fáze, jejich délka, vstupy, výstupy, zdroje, nástroje

3.4.2 Porozumění datům

Fáze porozumění datům začíná s původní kolekcí dat a pokračuje aktivitami směřujícími k seznámení se s daty, k identifikaci problémů kvality dat a k detekování důležitých vazeb pro získání potřebných informací:

- Sběr dat – získání úvodní kolekce dat (z databáze, datového skladu, vstupu) a jejich předzpracování pro využití analytickým softwarem
- Popis dat – množství dat, formát
- Průzkum dat – popisné statistiky, identifikace vztahů mezi atributy dat, významné subpopulace, odlehlé hodnoty
- Ověření dat – úplná data, chybějící data, šumy, neplatná data
- Eliminace chybných a odlehlých hodnot – nesmyslné hodnoty v datech (např. záporný čas), odlehlé hodnoty nejsou chybné, nicméně statisticky málo významné
- Zpracování chybějících dat – odstranit objekty s chybějícími daty, pro tvorbu modelu použít metody odolné vůči chybějícím datům, popř. chybějící data doplnit (mediánem, střední hodnotou ...)

3.4.3 Příprava dat

Tato fáze spočívá ve vytvoření konečného souhrnu dat, který bude použit pro tvorbu modelu:

- Odvození nových proměnných – shrnutí relevantních údajů a popřípadě transformace méně relevantních
- Normalizace dat – standardizace jednotek, optimalizace

- Selekce proměnných – výběr proměnných, které mají na modelovaný jev největší vliv, analytické metody jsou většinou výpočetně náročné a pro zbytečně velká datová pole by byly výpočty neúměrně složité
- Extrakce proměnných – vytvoření menšího počtu proměnných, které nesou maximum informací z původních dat
- Selekce relevantních objektů – podobně jako selekce a extrakce proměnných
- Formátování dat – úprava dat pro potřeby použitého modelovacího nástroje

3.4.4 Modelování

Fáze modelování zahrnuje vytvoření řady modelů za použití různých modelovacích technik:

- Výběr typu modelu a modelovací techniky – s ohledem na data, budoucí použití, srozumitelnost, obvykle kombinace několika modelů
- Stanovení testovacího plánu – testovací data, validace
- Tvorba modelu – ladění parametrů, zpracování vygenerovaných výsledků
- Hodnocení modelu – zvážení vypovídající hodnoty modelu z pohledu přesnosti a obecnosti

3.4.5 Ohodnocení

V této fázi projektu máme hotový model či modely, které by měly být dostatečně kvalitní a které slibují relevantní podklady k analýze. Před posledním krokem je ale ještě důležité znovu důkladně zvážit vypovídací hodnotu modelu a přesvědčit se, že model skutečně splňuje všechny obchodní požadavky, které byly zadány zadány. Rovněž je třeba se ujistit, že model není „přetrénovaný“ konkrétními testovacími daty, což je typické pro prokládání datových bodů polynomem vysokého stupně (aproximace neznámé funkce). Zároveň je již v této fázi třeba zvážit další kroky, jako jsou např. možnosti zlepšení do budoucna.

3.4.6 Nasazení modelu

Vytvořením modelu projekt nekončí, je potřeba stále vyřešit věci jako je integrace modelu či vyvinutého softwaru do podnikové infrastruktury, zajistit dosažitelnost získaných informací a zároveň model udržovat a sledovat, popř. obnovit či přestavit stávající model na model nový.

3.5 Princip modelování

Výpočetním jádrem celého procesu dobývání znalostí z databází je použití analytických metod. Vstupem do analytických procedur jsou předzpracovaná data, výstupem jsou znalosti. Všechny metody používané pro data mining vycházejí z předpokladu, že jednotlivé problémy (objekty) patřící k témuž konceptu (třídě) lze popsat pomocí podobných charakteristik (atributů), které jsou pro daný koncept společné - tyto metody také bývají označovány jako metody učení na základě podobností (*similarity-based learning*).

Učení na základě podobnosti pak vychází z úvahy, že objekty představující příklady téhož konceptu vytvářejí v tomto prostoru jakési shluky. Cílem modelování je tedy nalézt vhodnou reprezentaci těchto shluků, pomocí procesu učení modelu. Proces učení lze rozdělit dle použití „trenéra“ na metody s trenérem a metody bez trenéra - neboli jestli trénovací množina dat obsahuje ohodnocené příklady, či nikoliv. Dílem učení je pak nalézt popis jednotlivých tříd.

Požadavky na model při tomto procesu lze shrnout takto:

- Využitelnost při predikci
- Schopnost generalizace – nalézat obecně platné závislosti v datech
- Model nesmí být přeučený – nesmí se naučit „zdánlivé“ závislosti v datech nebo šum

Znalosti lze potom prezentovat různými způsoby - např. jako reprezentativní příklady (využívají analogické metody), funkce odpovídajícím jednotlivým shlukům či rozdělení prostoru atributů na snadno popsatelné, pravidelné útvary. V reálném životě jě častým problémem snaha analytiků používat pro všechny problémy stejné metody a nástroje, typicky ty, které nejlépe ovládají. To může vést až k příliš velkému ohýbání smyslu dat pouze za účelem možnosti využít „svůj“ nástroj. Ve statistice tuto vlastnost ilustruje např. oblíbenost některých modelů regresní analýzy, ovšem pokud vezmeme v potaz další metody a algoritmy, dostáváme velký rozsah prostředků, které umožňují výrazně zlepšit výsledky, popřípadě zefektivnit jednotlivé úlohy (proč využívat neuronové sítě na něco, co jednoduše zvládnou stromy, nač hledat v asociačních pravidlech finty pro práci s časovými údaji, když existují časové řady atd.).

3.6 Modely aplikace FAKE GAME

Aktuální implementace FAKE GAME nabízí dvě skupiny algoritmůalgoritmů: alforitmy pro základní modely, které již většinou nelze dále dekomponovat, a algoritmy, pro složené modely, tzv. ensembly.

3.6.1 Základní algortimy

Kromě škály regresních modelů realizovaných jako elementární funkce nabízí aplikace FAKE GAME také řadu klasifikačních algoritmů.

3.6.1.1 Rozhodovací stromy

Jednou z nejpoužívanějších metod reprezentace znalostí je metoda rozhodovacích stromů. Smysl tohoto uspořádání je velmi snadno pochopitelný a je znám z celé řady oblastí. Smyslem konstrukce stromu je rozdělování množiny trénovacích dat na menší podmnožiny tak, aby v každé podmnožině převládaly prvky jedné třídy. Od kořene stromu se na základě odpovědí na otázky (umístěné v nelistových uzlech) postupuje příslušnou větví stále hlouběji, až do listového uzlu, který odpovídá zařazení příkladu do třídy. Podrobnější informace o konstrukci rozhodovacích stromů jsou k dispozici např. v této online přednášce [7]. Následující příklad, který popisují obrázky 3.1 a 3.2, je převzat právě z uvedené přednášky.

Outlook	Temperature	Humidity	Windy	Play?
sunny	hot	high	false	No
sunny	hot	high	true	No
overcast	hot	high	false	Yes
rain	mild	high	false	Yes
rain	cool	normal	false	Yes
rain	cool	normal	true	No
overcast	cool	normal	true	Yes
sunny	mild	high	false	No
sunny	cool	normal	false	Yes
rain	mild	normal	false	Yes
sunny	mild	normal	true	Yes
overcast	mild	high	true	Yes
overcast	hot	normal	false	Yes
rain	mild	high	true	No

Obrázek 3.1: *Ukázková data pro rozhodovací strom 3.2*

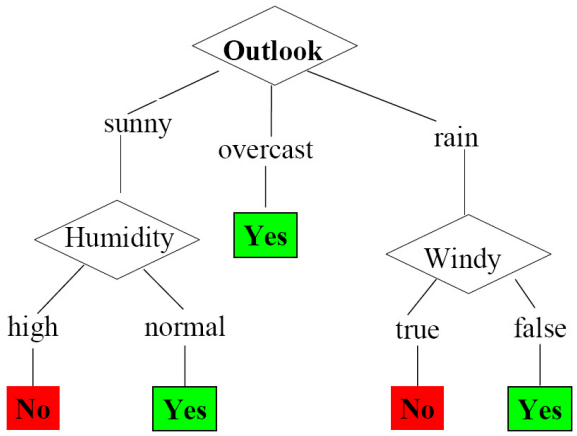
3.6.1.2 K-Nejbližších sousedů

V případě K-nejbližších sousedů jsou třídy reprezentovány svými typickými představiteli. V procesu klasifikace se pak nový případ zařadí do třídy na základě podobnosti (nejmenší vzdálenosti ke k-reprezentantům nějaké třídy). Obrázek 3.3 ilustruje způsob klasifikace.

3.6.1.3 Bayessovský klasifikátor - Bayess classifier

Bayesovský klasifikátor je založený na algortimu, který je podrobněji popsán v [13, 8] a který vychází z Bayessova teorému:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}, \tag{3.1}$$



Obrázek 3.2: *Rozhodovací strom vzniklý na základě dat 3.1*

kde $P(A)$ a $P(B)$ jsou pravděpodobnosti výskytu jevů A a B, $P(A|B)$ a $P(B|A)$ jsou pravděpodobnosti jevu A za předpokladu, že platí B a obráceně. Zároveň se předpokládá, že A a B jsou na sobě nezávislé.

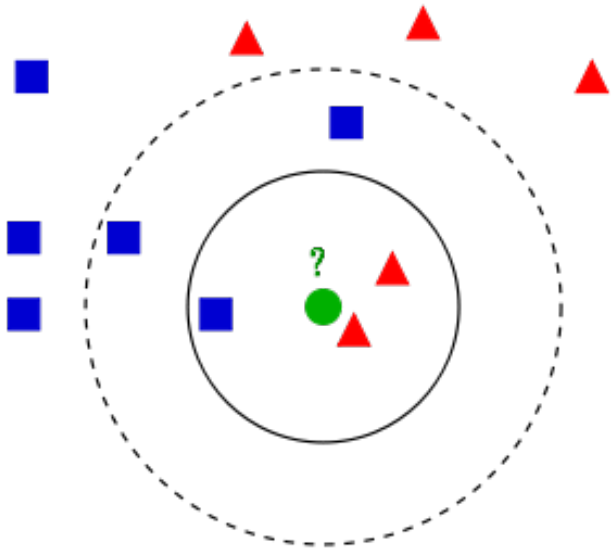
Při klasifikaci pomocí Bayssovksého klasifikátoru budeme hledat hypotézu s maximální posteriorní pravděpodobností - tj. největší pravděpodobností že daná hypotéza je na základě trénovacích dat platná.

$$A_{mmp} = maxP(B|A_i)P(A_i), \tag{3.2}$$

3.6.1.4 Algoritmy podpůrných vektorů (SVM - Support Vector Machine)

Základní princip této metody spočívá v převodu původního vstupního prostoru hodnot do prostoru nového, typicky vícedimenzionálního, kde lze oddělit různé třídy hodnot pomocí lineárních oddělovačů, neboť pro nalezení lineárních hranic existují efektivní algoritmy [12]. Lineárním oddělovačem bývá typicky nadrovina. Tu lze vždy najít při použití prostoru s dostatečným množstvím dimenzí: obecně platí, že pro N datových bodů je možné najít linearní oddělovač v prostoru s $N-1$ dimenzemi. U prostorů, kde se počet dimenzí D blíží N hrozí nebezpečí přetrénování lmodelu a SVM musí tím pádem hledat optimální lineární oddělovač. Lineární oddělovač poskytuje co nejširší pásmo mezi sebou samým a pozitivně ohodnocenými příklady na jedné straně, a mezi sebou samým a negativně ohodnocenými příklady na straně druhé.

Pro vyhledání optimálního oddělovače se v algoritmu SVM využívá metody kvadratického programování. Lze dokázat [12] že se pracuje pouze s těmi případy trénovací množiny (právě tzv. podpůrné vektory), které jsou pro výpočet oddělovače podstatné, čímž výrazně urychlují učení modelu.

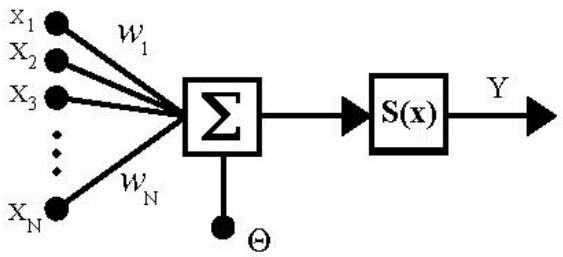


Obrázek 3.3: *K-nejbližších sousedů - nový prvek (zelený kruh) bude vyhodnocen jako červený trojúhelník pro k = 2, pro k = 3 pak jako modrý čtverec.*

3.6.1.5 Neuronové sítě

První modely neuronů a neuronových sítí se zkoumaly v rámci umělé inteligence již v 50. letech. Umělé neuronové sítě vycházejí z analogie s lidským mozkem. Z toho vyplývá, že podstatou návrhu neuronových sítí je modelování struktury (topologie) a vlastní činnosti biologických neuronových sítí. Podobně jako mozek jsou umělé neuronové sítě tvořeny množstvím navzájem propojených elementů - neuronů, určitým způsobem seskupených a mezi sebou propojených.

V umělých neuronových sítích je neuron (obr. 3.4) chápán jako buňka, která přijímá podněty od jiných neuronů, které jsou k ní připojeny „na vstupu“. Pokud souhrnný účinek těchto vstupních podnětů překročí určitý práh, neuron se aktivuje a sám začne svým výstupem působit na další neurony.



Obrázek 3.4: *Model umělého neuronu*

Každá neuronová síť jako celek realizuje určitou, pro ni typickou transformační funkci. Při modelování a řízení složitých, často silně nelineárních soustav býváme většinou postaveni před

problém, že daný proces není možné s uspokojivou přesností matematicky popsat, případně že exaktní matematický model procesu je tak složitý, že jeho případná algoritmizace je buď časově a programově velmi náročná, nebo dokonce nemožná. V tomto případě nám umělá neuronová síť poslouží jako univerzální funkční aproximátor. To znamená, že navržená a podle určitých matematických pravidel „naučená“ neuronová síť je následně schopná činnost výchozího procesu s určitou mírou přesnosti reprodukovat (simulovat) pro různé vstupní signály.

Obecný algoritmus učení neuronové sítě z přednášky [22]:

```
function NEURAL-NETWORK-LEARNING(examples) returns network
    network  a network with randomly assigned weights
    repeat
        for each e in examples do
            O     NEURAL-NETWORK-OUTPUT(network, e)
            T     the observed output values from e
        update the weights in network based on e, O, and T
        end
    until all examples are correctly predicted or stopping criterion is
    reached
return network
```

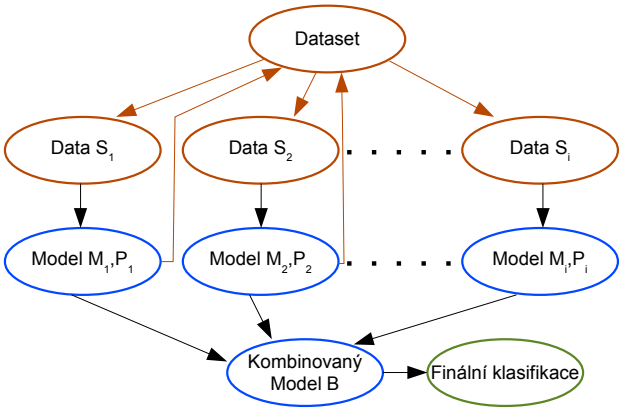
3.6.2 Složené (ensemble) modely

Složené algoritmy kombinují více základních algoritmů pro získání kvalitnějších modelů. Velkou úspěšnost mají u případů, kdy základní algortimy produkují modely s nízkou plasticitou, nebo modely přetrénované na konkrétní trénovací data [11].Ensemble techniky využívají toho, že každý model je dobrý v jiné oblasti dat a dělá jiné chyby. Zároveň se snaží této diverzity mezi svými modely aktivně dosáhnout. Ensemble model složený z modelů, jež dělají stejné chyby, nemá význam, neboť zde se touto technikou nedá nic získat (v tomto případě by lepší výsledek přinesl výběr modelu s nejnižší chybou a zbylé modely by sé vůbec nepoužily). Optimální diverzita je ta, která nejlépe vyvažuje komponenty tak, aby redukovala celkovou chybu.

Vedle základních (jednoduchých) algoritmů podporuje aplikace GAME následující ensemble algoritmy.

3.6.2.1 Bagging (Bootstrap Aggregation)

Bagging je nejjednodušší používaný ensemble algoritmus [6], který funguje na principu snižování rozptylu pomocí průměrování. Pro učení se nejprve náhodně vybere n vzorků z trénovacího datasetu (s opakováním) S_i a na těchto vzorcích se pak naučí modely M_i. Kombinovaný model B při predikci vybere ohodnocení, které dostane nejvíce hlasů (je predikováno nejvíce modely M_i). U regresních úloh je výstupem ensemblu vážený průměr.



Obrázek 3.5: *Schéma typického ensemble algoritmu*

3.6.2.2 Boosting

Myšlenka Boostingu spočívá v postupném trénování modelů tak, aby každý nový model byl natrénován vzorkem dat S_i, v němž jsou zvýrazněné instance, které předchozí model klasifikoval špatně. Na počátku mají všechny vzorky datasetu stejné váhy. První model je natrénován na tomto datasetu, je mu přiřazena přesnost P_i a u vzorků, které ohodnotil správně, je snížena jejich váha. Na takto upraveném datasetu je natrénován model následující. Tak se následující model specializuje více na instance, na které neumí předchozí modely správně odpovědět. Při predikci je pak vybráno ohodnocení na základě váženého (na základě přesnosti) maxima hlasů modelů u klasifikačních úloh či na vážený průměr u regresních úloh.

3.6.2.3 Stacking

Stacking, nebo též Stacked generalization, kombinuje výstupy různých modelů. V první fázi jsou modely (base learner) učené na vstupním datasetu, pro další fázi je pak tento dataset nahrazen novým, vzniklým z výstupů modelů v první fázi. Z toho datasetu se pak vytvoří finální model (metamodel). Při odpovědi na neznámou instanci se s touto instancí provede stejný postup jako při učení. Nejdříve je předána base learner modelům a z jejich výstupů je vytvořena instance pro metamodel. Výstupem ensemblu je poté výstup metamodelu.

3.6.2.4 Cascade Generalization

Cascade Generalization lze chápat jako speciální případ stackingu, hlavně díky jeho „vrstevnaté“ struktuře učení. Také využívá řetězení modelů, přičemž vzorek dat pro další model v řadě je základní dataset rozšířený o nové atributy, tj. výstupy předchozích modelů. Narozdíl od Stackingu tak všechny modely mají přístup k atributům původního datasetu.

3.6.2.5 Arbitrating

Arbitrating používá používá pro všechny modely metamodel zvaný rozhodčí (referee). Smyslem tohoto metamodelu je odhadnout pravděpodobnost s jakou daný model podá pravdivou předpověď. Základní myšlenka je, že každý z modelů je nejspolehlivější pro určitou subdoménu vstupních problémů (datasetu). Pro každý model je tedy zkonstruován rozhodčí, který popisuje tuto subdoménu a spolehlivost modelu v ní. Pro problém z konkrétní subdomény se pak vybere nejvěrohodnější model.

3.6.2.6 Cascading

Cascading lze považovat za modifikaci Boostingu s více druhy modelů. Cascading využívá, stejně jako Boosting, změnu distribuce učících vektorů, ale u Cascadingu se váhy mění na základě jistoty výstupu předchozího modelu. Cascading používá kaskádu modelů se zvětšující se složitostí. Úvodní rozložení učících vektorů je rovnoměrné a během iterací se zvyšuje váha těch instancí, u kterých si předchozí model není dostatečně jistý svým výstupem. Jistota výstupu je dána jako pravděpodobnost, že instance patří do dané třídy. První modely se obvykle volí semi-parametrické (například vícevrstvé neuronové sítě) a poslední model je vždy neparametrický, který vždy vybere nějakou třídu. Při klasifikaci neznámé instance se tato instance dává postupně všem modelům v kaskádě a jako výstup se vezme výstup prvního modelu, který na ní odpoví s větší jistotou, než je daný práh.

3.6.2.7 Delegating

Delegating je v principu podobný jako Cascading, s tím rozdílem, že každý model je učen pouze na delegovaných instancích, kterými si předchozí modely nebyly jisté. Delegating obsahuje, stejně jako Cascading, prahovou hodnotu, která určuje, od jaké hodnoty pravděpodobnosti jsou si modely výsledkem jisté. Učící proces se zastaví, pokud už nejsou žádné instance k delegování, nebo po fixním počtu iterací. Pro klasifikaci neznámé instance je výstupem první model, který ji dokáže určit s větší jistotou než je daný práh. Delegating má menší výpočetní náročnost než ostatní ensembly, protože počet instancí, na kterých se musí učit další modely, v každé iteraci klesá. Také celkový výstup se použitím Delegatingu nijak nekomplikuje, protože používá pouze výstup jednoho modelu učeného na nemodifikovaných datech.

3.6.2.8 Area specialization

Princip učení Area Specialization je stejný jako u Boostingu [9] (iterační, funguje pomocí úpravy vah instancí na základě toho, jak dobře na ně model odpovídá). Jediný rozdíl spočívá v parametru *modelSpecialization*, který určuje, jak moc se další modely budou specializovat na data, v nichž předchozí modely chybovaly. Výsledná odpověď, kterou ensemble vrátí, odpovídá tomu modelu, který je nejlepší v oblasti, kde se neznámý vektor nachází. Ke zjištění, který model je nejlepší pro danou oblast, využívá učící vektory. Pro neznámý vektor se nejdříve spočtou vzdálenosti ke všem učícím vektorům a vybere se n nejbližších, které budou rozhodovat o výstupu. Výstup je pak dán jako vážený průměr na základě chybovosti modelů.

3.6.2.9 Cluster divide

Cluster divide ensemble provede shlukovou analýzu, rozdělí data na shluky a přiřadí jeden model každému shluku. Výstup ensemblu pro neznámou instanci je určen výstupem modelu, který zodpovídá za daný shluk, do kterého tato instance patří. Tento přístup má dvě hlavní výhody. První výhodou je, že modely budou pravděpodobně podávat lepší výsledky, protože oblast, kterou mají postihnout, bude menší. Tím pádem mají větší šanci se modelovanému problému přizpůsobit. Druhá výhoda spočívá v tom, že učení většího množství modelů na méně datech je většinou časově méně náročné než učení jednoho model na všech datech. Toto platí, pokud je složitost učícího algoritmu větší než lineární vzhledem k počtu učebních vektorů.

3.6.2.10 Thresholding

Tento ensemble je velice vhodný pro vysoce nevybalancovaná data. Pracuje s modifikacemi vstupních dat tak, aby základní modely měly větší schopnost diskriminace vůči určité třídě. Smyslem tohoto postupu je rozdělit data dourčitých oblastí a pro každou oblast vytvořit specializovaný model, který bude dobře fungovat jen v dané oblasti.

Kapitola 4

Paralelismus a paralelní programování

Cílem mojí práce je urychlit výpočet modelů v aplikaci FAKE GAME pomocí paralelizace a navrhnout modul sloužící jako distribuovaný server pro počítání modelů, proto následující kapitolu věnuji paralelnímu programování.

4.1 Paralelismus

Paralelismus je model umožňující rychlejší řešení obsáhlých, složitých úloh. Vychází z myšlenky rozdělení daného problému na menší části (subproblémy), které se poté řeší zároveň, narozdíl od sekvenčního řešení problému, kdy se problém řeší krok za krokem. Základní schéma tak může být:

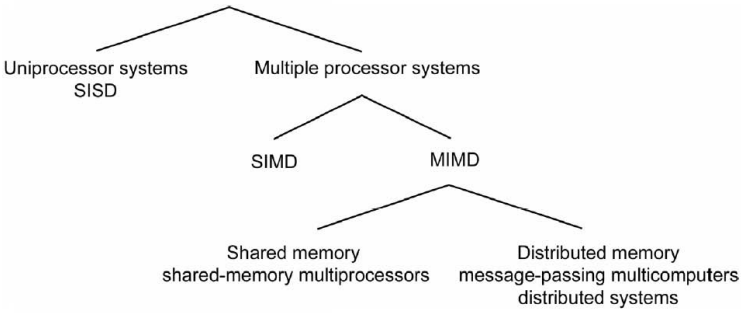
- Rozdělení úlohy na subproblémy
- Přidělení těchto problémů odlišným „pracovníkům“
- Koordinování „pracovníků“

4.2 Architektura paralelních systémů

4.2.1 Popis

Počítačové programy bývaly tradičně psány pro běh na sekvenčních počítačích tj. na počítačích s jedním procesorem (který u počítačů plní právě funkci „pracovníka“). S tím, jak stále roste složitost řešených úloh (problematika simulací, zpracování obrázků, videa, data miningu, modelování) a tím i požadavky na hardware, tento tradiční model počítače a programovacích technik již přestává být schopen dané úlohy řešit. Vývoj stále rychlejších procesorů začíná být neúměrně drahý oproti získanému zrychlení a lze hovořit o přiblížení se k limitu technologických možností dnes používaných metod konstrukce procesorů.

Paralelní počítače tak přinášejí relativně levnější cestu k získávání požadované výpočetní kapacity a i osobní počítače a notebooky jsou dnes již běžně vybavené vícejádrovými procesory. To samozřejmě vede k potřebě vytvářet programy schopné těchto technologických prostředků využít.



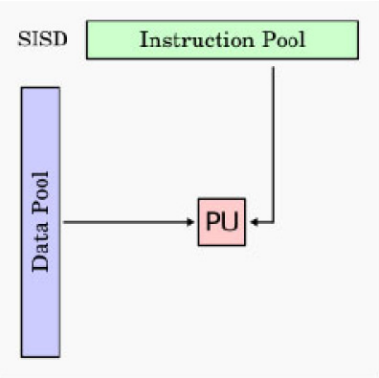
Obrázek 4.1: *Kategorie paralelních systémů dle Flynnovy taxonomie*

4.2.2 Flynnova taxonomie

Existuje několik metod, podle nichž lze popsat a rozdělit architekturu paralelních systémů. Jedna z nejpoužívanějších, Flynnova taxonomie [3], využívá ke kategorizaci systémů vztah mezi instrukcemi a daty programu 4.1.

4.2.2.1 SISD model

SISD - Single Instructions, Single Data stream představuje vlastně sekvenční počítač. Jedna instrukce je vykonávána každý hodinový cyklus a každá instrukce operuje nad jedním (resp. skalárním) datovým elementem. Výkon je tak limitován počtem instrukcí, které lze v daném časovém úseku zpracovat, je měřen v jednotkách MIPS (milios of instructions per second), nebo podle frekvence hodin v Hz (resp. GHz).

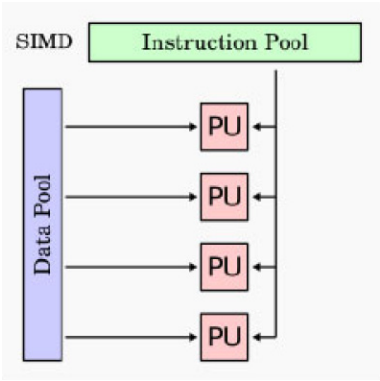


Obrázek 4.2: *Model SISD architektury, blok PU symbolizuje procesor*

4.2.2.2 SIMD model

U modelu Single Instructions, Multiple Data stream může každá instrukce pracovat s více než jedním datovým elementem - buďto v podobě vektoru (jedna instrukce tak upravuje více

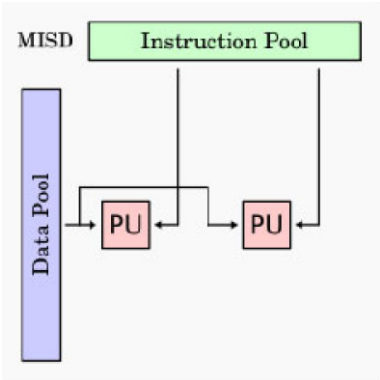
operandů tj. souřadnic vektoru), nebo ve formě rozdílných množin dat (jedna instrukce se vykonává na všech procesorech, které ale operují s různými množinami dat). Práce procesorů je synchronní a tudíž synchronizace programů nepředstavuje vážnější problém. Výkon je měřen v jednotkách MFLPOS (milions of floating point operations per second).



Obrázek 4.3: *Model SIMD architektury*

4.2.2.3 MISD model

Multiple Instructions, Single Data stream je velmi neobvyklá architektura která je používaná v podstatě pouze k vyloučení chyb. Rozdílné systémy pracují nad stejnými daty a musí se dobrat stejného výsledku.

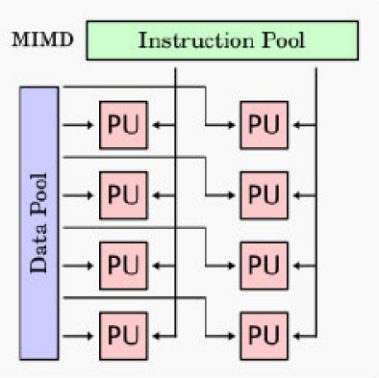


Obrázek 4.4: *Model MISD architektury*

4.2.2.4 MIMD model

Patrně nejpoužívanější architektura, Multiple Indstructions, Multiple Data stream, je spojením zcela nezávislých procesorů, kdy každý procesor zpracovává vlastní instrukce nezávislé na ostatních procesorech nad vlastními daty. Každý procesor tak vykonává vlastní operace bez

ohledu na činnost ostatních procesorů. Klíčovou otázkou běhu programu na této architektuře je tedy synchronizace procesorů na konci paralelních částí programu.



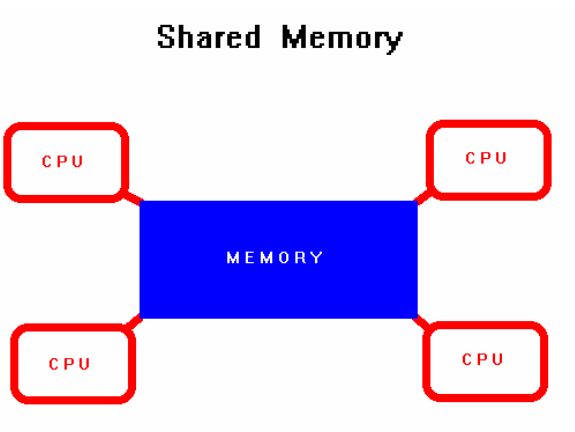
Obrázek 4.5: *Model MIMD architektury*

4.2.3 Paměťové architektury

Způsoby, jakými mezi sebou procesory komunikují, jsou závislé na architektuře paměti. Lze tak rozlišovat:

4.2.3.1 Sdílená paměť

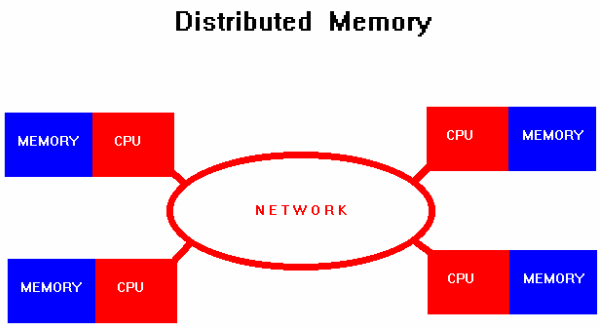
Jednotlivé procesory pracují nezávisle, ale sdílejí společný paměťový prostor. Problém synchronizace pak spočívá především v řízení přístupu jednotlivých procesorů k datům v této paměti, aby nedošlo k nekonzistenci dat či k „uzamknutí programu“. Klíčovou otázkou pro výkon této architektury je paměťová propustnost, a proto se tento model hodí k použití u nižšího počtu procesorů (řádově desítky).



Obrázek 4.6: *Model architektury se sdílenou pamětí*

4.2.3.2 Distribuovaná paměť

U této architektury má každý procesor vlastní paměťový prostor a data jsou sdílena pomocí zasílání zpráv po komunikační síti. Celkový výkon této architektury je závislý na rychlosti přenosové sítě.



Obrázek 4.7: *Model architektury s distribuovanou pamětí*

4.2.4 Co je potřeba vzít v úvahu při návrhu paralelního programu

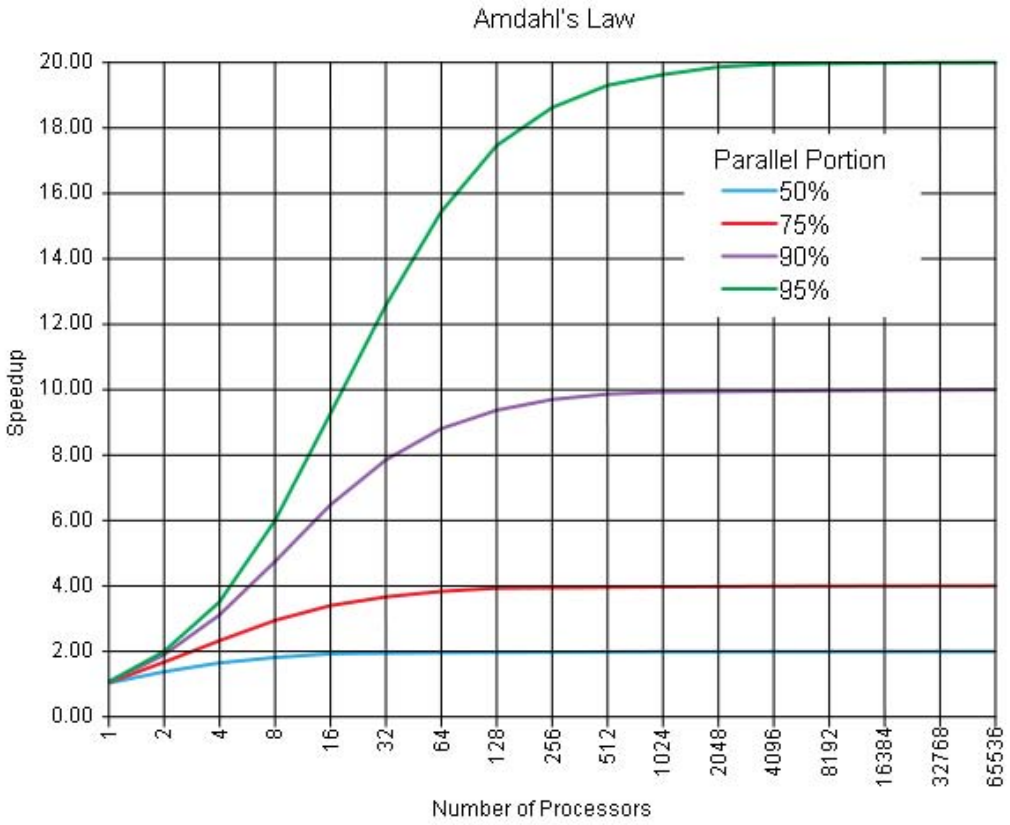
4.2.4.1 Amdahlovo pravidlo

Amdahlovo pravidlo [1], definované systémovým architektem Gene Amdalhem, je často používáno pro nalezení maximálního teoretického zrychlení výpočtu při použití paralelních systémů. Pravidlo je modelem chování mezi předpokládaným zrychlením paralelizovaného algoritmu oproti jeho sekvenční podobě za předpokladu, že rozsah problému je zachován. Pravidlo je definováno takto: pokud N je část programu, kterou lze paralelizovat, a $1 - N$ část, již paralelizovat nejde, potom je maximální možné zrychlení při použití P procesorů rovno právě:

$$\frac{1}{(1-N)+N/P} \tag{4.1}$$

Jak se P limitně blíží nekonečnu, maximální dosažitelné zrychlení je $1/(1-N)$. V praxi to znamená, že poměr výkon/cena strmě klesá (jak je vidět z obr. 4.8), jakmile existuje určitá neparalelizovatelná část programu. Uveďme si jednoduchý ilustrativní příklad. Představme si, že máme sečíst 1000000000 prvků vektoru a. Načtení prvků trvá 8% celkového výpočtového času, vlastní výpočet součtu trvá 90% celkového výpočtového času a tisk trvá 2% celkového výpočtového času. Je zřejmé, že jen výpočet součtu lze paralelizovat, t.j. $N = 90\%$ V případě použití P = 10 procesorů dostaneme z Amdahlova pravidla, že zrychlení výpočtu paralelizací je přibližně $5,2$-krát. Z Amdahlova pravidla a uvedeného příkladu plynou dva závěry:

- neplatí přímá úměra mezi počtem procesorů a urychlením výpočtu (užití 10 procesorů neznamená urychlení výpočtu 10-krát) a
- pro limitní případ $P \to \infty$ je urychlení výpočtu konečné a rovno $1/(1-N)$.



Obrázek 4.8: *Amdahlovo pravidlo - ilustrace snižování výhodnosti využívání příliš vysokého počtu procesorů.*

4.2.4.2 Problematika *Load Balancingu* a *Granularity*

Balancování zátěže (*Load Balancing*) znamená distribuci úkolů jednotlivým procesorům pro zaručení největší časové efektivity. Pokud totiž nejsou úkoly distribuovány balancovaně, může dojít ke zbytečným prodlevám, kdy se čeká na dokončení jedné úlohy, zatímco ostatní úkoly neběží. O přidělování jednotlivých úloh procesorům se stará jádro operačního systému, resp. plánovač, využívající dle operačního systému různé algoritmy pro vyvážené zatížení procesorů, např. FCFS, SJF, RR, SRT, prioritního plánování a další - více viz. přednášky na [5].

Problematika granularity vzniká z potřeby synchronizovat činnost různých procesorů pracujících na jednom problému. Synchronizace probíhá pomocí určité formy komunikace mezi procesory a právě podíl mezi časem věnovaným výpočtu a časem potřebným pro synchronizaci se nazývá granularita. Je to další z problémů, se kterým je potřeba počítat při návrhu paralelních systémů a programů. Příliš vysoká granularita totiž znamená příliš velkou spotřebu času na synchronizaci výpočtu a může vést až k tomu, že sekvenční řešení bude nakonec časově výhodnější než paralelní.

4.2.4.3 Datová konzistence

V okamžiku, kdy dochází k vícenásobnému užití stejného paměťového místa, se musíme zabývat otázkou konzistence dat z této adresy. Několik zároveň běžících procesů či vláken používá (čte/zapisuje) společné sdílené prostředky (např. sdílená paměť, sdílené proměnné, sdílené soubory, ...). Kvůli přepínání kontextu může dojít např. k pozměnění proměnné jedním procesem dříve, než ji stihne přečíst jiný proces, který ji taktéž potřeboval načíst. V důsledku je tak výsledek výpočtu závislý na přepínání kontextu jednotlivých procesů či vláken, které používají sdílené prostředky. Tyto chyby se velmi špatně detekují. Části programu, kde procesy či vlákna používají sdílené prostředky a kde tak může dojít k těmto chybám, se nazývají *kritické sekce*.

4.2.4.4 Podmínky korektnosti paralelního programu

Aby byl paralelní program korektní, musí splňovat tyto podmínky:

- Dva procesy se nesmí nacházet současně ve stejné kritické sekci.
- Na rychlost a počet procesorů nesmí být kladeny žádné předpoklady.
- Pokud proces běží mimo kritickou sekci, nesmí být blokován ostatní procesy.
- Žádný proces nesmí do nekonečna čekat na vstup do kritické sekce.

Splnění těchto podmínek lze pomocí určitých postupů, jako jsou aktivní čekání, vzájemné vyloučení pomocí blokování, zákazy přerušení, a k zajištění funkčnosti těchto postupů nabízí většina programovacích jazyků nejrůznější konstrukty synchronizačních prostředků, jako jsou mutexy, semafory, bariéry apod. Pro více informací o synchronizačních prostředcích jako celku doporučuji slidy z přednášek na stránce [5].

4.3 Paralelní programování v jazyku JAVA

4.3.1 Sdílená paměť - vlákna

Pro paralelizaci programů ve sdílené paměti nabízí JAVA koncept vláken. Vlákna jsou v JAVĚ vždy instancí třídy *java.lang.Thread* nebo jejího potomka, přičemž třída *Thread* nabízí základní metody, jako je spuštění vlákna, zastavení či ukončení.

Vlákna lze v JAVĚ vytvářet dvěma způsoby: buďto odvozením potomka od třídy *Thread* nebo implenetací rozhraní *Runnable*. Vlastní kód vlákna je dán metodou *run*(), kterou musíme v naší třídě implementovat a která je spouštěna pomocí volání metody *start*(), která nejprve vytvoří nové systémové vlákno a v něm pak spustí kód *run*. Jednoduché vlákno tedy může vypadat např. takto (datová položka semaphore je uvedena kvůli následujícím příkladům):

```
class MyThread extends Thread {
    Semaphore semaphore = new Semaphore();
    public MyThread(sem){
```

```
        semaphore = sem;
    }
    public void run() {
    //for semaphore synchronization -
    //if semaphore counter > 0 lower the counter
    //else this.wait()
    try{
     semaphore.acquire();
    }
    catch (InterruptedException ex) {
ex.printStackTrace();
    }
    //critical section
    . . .
    //increment semaphore counter,
    //notify() threads waiting on this semaphore
    semaphore.release();
    }
}
```

Vlákna v JAVĚ se nachází vždy v jednom ze čtyř stavů, přičemž přechody mezi nimi zajišťují metody třídy Thread. Stavy vláken, jak je popisuje [4]:

- Nové vlákno (new)
  Vlákno bylo vytvořeno, ale dosud nebyla spuštěna metoda `start()`.
- Běhuschopné (runnable) vlákno
  Metoda `start()` již proběhla, provádí se metoda `run()`.
- Blokované vlákno (blocked)
  Vlákno je blokováno a čeká na nějakém „zámku“.
- Čekající vlákno (waiting)
  Vlákno čeká na danou operaci jiného vlákna.
- Časově omezené čekání (timed_waiting)
  Vlákno čeká na operaci nějakého jiného vlákna, ale pouze po určený časový interval.
- Ukončené vlákno (terminated)
  Vlákno, jež dokončilo svůj výpočet.

4.3.1.1 Plánování vláken

Na systémech s jedním procesorem se multithreading (současný běh více vláken najednou) simuluje tak, že se vlákna o procesor dělí, a to podle pravidel, která určuje plánovač.

Java používá plánování podle priority: každé vlákno má přidělenu prioritu - číslo v rozmezí konstant MIN_PRIORITY až MAX_PRIORITY (definované ve třídě Thread), kde vyšší hodnota znamená vyšší prioritu. Od priority se odvíjejí tato pravidla:

1. Běžící vlákno musí mít nejvyšší prioritu.
2. Pokud je v běhuschopném stavu vlákno s vyšší prioritou, než má vlákno běžící, je běžící vlákno tímto okamžitě vystřídáno.
3. Vlákno s nižší prioritou může běžící vlákno vystřídat jen tehdy, pokud je běžící vlákno v neběhuschopném stavu nebo je ukončeno, a zároveň na přidělení procesoru nečeká jiné vlákno s vyšší prioritou.
4. Pokud na přidělení procesoru čeká více vláken se stejnou prioritou, je další vybráno tak, aby se postupně vystřídala všechna.
5. Běžící vlákno může navíc dobrovolně poskytnout procesor jiným vláknům se stejnou prioritou zavoláním metody yield().

Z uvedených pravidel tedy vyplývá, že běžící vlákno nemůže být „donuceno“ k vystřídání jiným vláknem se stejnou nebo nižší prioritou, ale pouze s prioritou vyšší.

JAVA je ovšem v tomto ohledu benevolentní a dovoluje na operačních systémech, které to podporují (Unix, Windows XP/NT), tzv. time slicing (sdílení času), kdy se vlákna se stejnou prioritou střídají po pevně přidělených časových intervalech.

4.3.1.2 Synchronizace vláken

Pro synchronizaci vláken umožňuje JAVA použít velmi jednoduchý způsob pracující na principu monitoru. V JAVĚ má každý objekt svůj jedinečný monitor. V něm jsou definována sdílená data a část programu, kterou může v daném monitoru v jednu chvíli vykonávat pouze jeden proces. Monitor je inicializován klíčovým slovem *synchronized*, kterým jsou označeny metody a bloky definující kritickou sekci. Vlákno, které vstoupí do této kritické sekce, aktivuje monitor, který se uzamkne, a tak je zaručena nepřerušitelnost této metody až do okamžiku, kdy vlákno monitor uvolní (vlákno vystoupí z kritické sekce).

Jinak je možné vlákna také synchronizovat přímo pomocí metod *wait*(), *notify*() či *notifyAll*(). Metoda *wait*() zastaví vlákno až do jeho probuzení metodou *notify*() nebo metodou *notifyAll*(), kterou volá odlišné vlákno. Výčet všech metod třídy *Thread* je uveden na stránkách [4].

Od uvedení J2SE 5.0 poskytuje JAVA řadu standardních synchronizačních prostředků nabízených v balíku *java.util.concurrent*. Kromě monitorů tak přibyly další synchronizační prostředky známé např. z jazyka c++, jako jsou semafory, mutexy a bariéry, nebo fronty zpráv. Ukázka kódu synchronizace pomocí metody *join*() a pomocí třídy *Semaphore*.

```
public static void main(String[] args){
    //creating and starting array of threads
    for (int i = 0; i < 30; i++){
        threads[i] = new MyThread();
        threads[i].start();
    }
    //synchronizing by waiting for all
    //the threads from whole array to finish
```

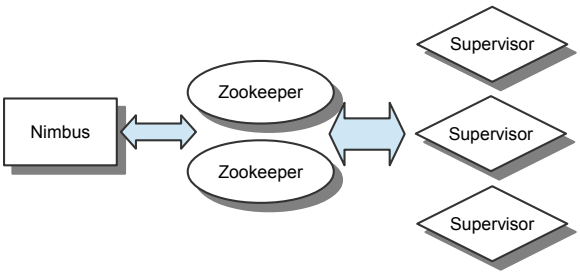
```
    for (int i = 0; i < 30; i++){
        try {
            threads[i].join();
        }
        catch (InterruptedException ex) {
            ex.printStackTrace();
        }
    }
}

public static void main(String[] args){
    //maximal number of running threads at a time
    int maxRunningThreads
    //inicializing Semaphore
    Semaphore semaphore = new Semaphore(maxRunningThreads, true)

    //creating and starting array of threads
    for (int i = 0; i < 30; i++){
        threads[i] = new MyThread(Semaphore);
        threads[i].start();
    }
}
```

4.3.2 Distribuovaná paměť - projekt Storm

Od uvedení projektu HADOOP roku 2005 [18] se objevuje množství frameworků umožňující snazší vývoj paralelních aplikací na systémech s distribuovanou pamětí. Typicky nabízejí rozhraní přibližující tvorbu programů na systémech s distribuovanou pamětí programování na systémech se sdílenou pamětí, přičemž o distribuci a částečně i synchronizaci jednotlivých paralelních úkolů se stará právě framework. Během implementační části své práce jsem kromě paralelizace výpočtu modelů ve sdílené paměti použil framework Storm jako základ pro nový modul aplikace FAKE GAME. Ten představuje realizaci druhého cíle práce - server, který umožňuje počítání modelů na distribuovaných systémech.

4.3.2.1 Storm cluster

Projekt Storm je open source systém vyvinutý pro distribuované zpracování úkolů v reálném čase. Ve Stormu existují dva typy nodů, z nichž se skládá každý Storm cluster. Jedná se o master node a worker node. Na master node běží proces zvaný Nimbus, který je zodpovědný za distribuci kódu jednotlivým worker nodům, přidělování práce worker nodům a monitoring chyb. Na každém worker nodu pak běží proces zvaný Supervisor, který očekává přidělenou práci a dle ní pak spouští samotné worker procesy. Worker procesy vykonávají části topologie. Běžící topologie se tak skládá z mnoha worker procesů běžících na různých strojích. Koordinaci komunikace mezi Nimbus a Supervisor procesy realizuje Zookeeper cluster [20] který tímto zajišťuje stabilitu systému a stará se se o případné pády, restarty apod. jednotlivých workerů.



Obrázek 4.9: *Storm cluster*

4.3.2.2 Základní abstrakce

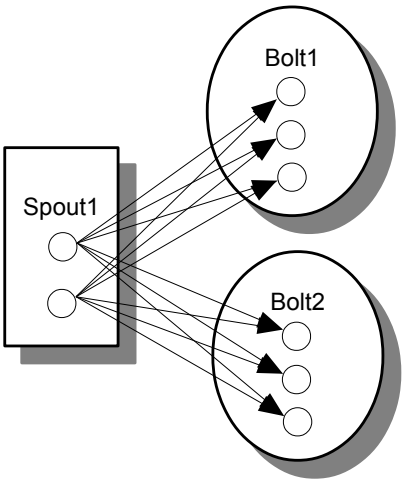
Stream (proud) je základní abstrakcí ve Stormu. Je to neomezená sekvence tzv. tuplů (párů). Tuple je pojmenovaný seznam hodnot, přičemž prvkem v tuplu může být libovolný objekt. Storm nabízí rozhraní pro transformace streamů, tzv. spouty (gejzíry) a bolty (blesky). Spouty slouží jako zdroje streamů: typicky načtou data z externího zdroje, transformují je a emitují do topologie. Bolty pak konzumují neomezené množství streamů, provádějí určité transformace a případně emitují nové streamy. Síť různě propojených spoutů a boltů tvoří topologii, která je spouštěna na Storm clusteru. Topologii lze nejlépe znázornit jako graf výpočtu, kdy každý uzel topologie zobrazuje spout nebo bolt a hrany zobrazují streamy mezi nimi. Každý uzel v Storm topologii běží paralelně, a v parametrech topologie lze specifikovat, kolik instancí kterého uzlu má být spuštěno. Storm následně určí, kolik vláken, v rámci kterého worker procesu a na kterých worker nodech v clusteru spustit. Obrázky 4.10 a 4.11 ukazují možný logický a fyzický pohled na uspořádání běžící topologie.

4.3.2.3 Jednoduchá topologie

Níže uvedený příklad je převzatý z projektu storm-starte, který slouží jako Storm tutorial. Pokusím se něm vysvětlit základní principy programování na Stormu. Uvedená topologie obsahuje dva bolty stejného typu a jeden spout. V metodách $setSpout(String, IRishSpout, int)$ a $setBolt(String, IRichBolt, int)$ definuje poslední parametr množství paralelizace, čili počet vláken realizujících kód daného prvku, který bude zpuštěn na worker nodech clusteru.

```
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout("words", new TestWordSpout(), 1);
builder.setBolt("exclaim1", new ExclamationBolt(), 3)
    .shuffleGrouping("words");
builder.setBolt("exclaim2", new ExclamationBolt(), 2)
    .shuffleGrouping("exclaim1");
```

Prvky této topologie jsou organizovány do řady, připojení boltů na konkrétní streamy a strategie doručování tuplů ze streamů na bolty určuje tzv. Stream grouping. $TestWordSpout$



Obrázek 4.10: *Logický pohled na topologii obsahující jeden spout s mírou paralelizace 2 a dva bolty s mírou paralelizace 3.*

tak emituje tuply do streamu „words“. Na něj jsou připojena 3 vlákna, ve kterých běží *ExclamationBolt*. Tuply emitované do tohoto kanálu jsou jim přidělovány podle strategie *shuffleGrouping*, čili náhodně, ale tak, aby bolty dostaly stejný počet tuplů. Tyto bolty pak emitují tuply do kanálu „exclaim1“, na který jsou připojeny 2 vlákna, v nichž běží opět *ExclamationBolt*. Storm definuje 7 základních způsobů grupování streamů, vlastní způsoby lze vyvinout implementací rozhraní CustomStreamGrouping.

1. Shuffle grouping: tuply jsou náhodně distribuovány všem boltům tak, aby všechny bolty dostaly stejný počet tuplů.

2. Fields grouping: stream je rozdělen podle hodnoty prvku tuplu tak, aby všechny tuply se stejnou hodnotou daného prvku byly zpracovávány stejným boltem.

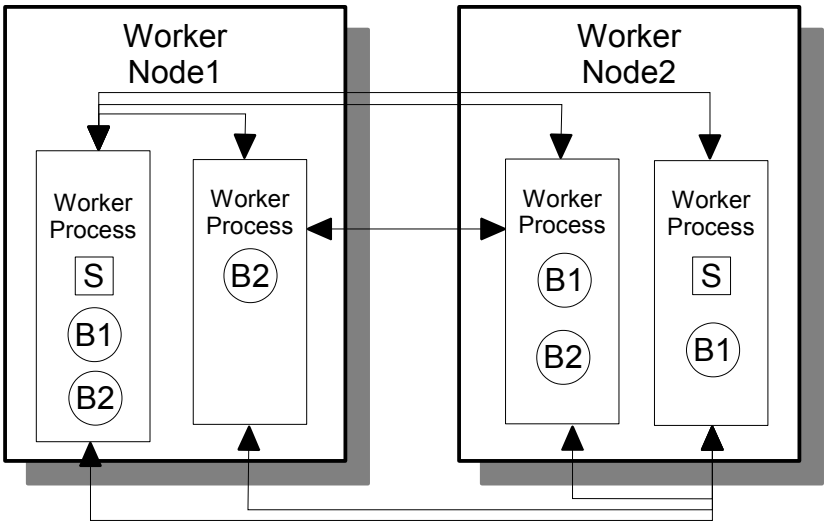
3. All grouping: tuply jsou replikovány pro všechny bolty.

4. Global grouping: celý stream jde jen do jednoho boltu. Pokud na přidělení procesoru čeká více vláken se stejnou prioritou, je další vybráno tak, aby se postupně vystřídala všechna.

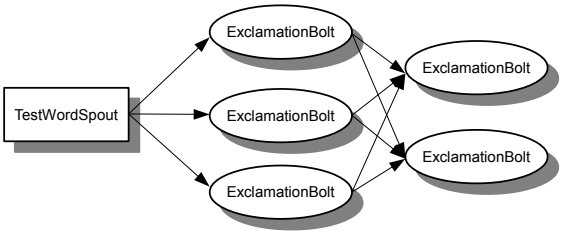
5. None grouping: v aktuální verzi Stormu totéž jako Shuffle grouping.

6. Direct grouping: jedná se o speciální případ groupingu, kdy zdroj přímo rozhoduje o tom, který bolt bude daný tuple zpracovávat. Tento grouping se dá používat pouze pro streamy které byly deklarovány jako direct streamy.

7. Local or shuffle grouping: pokud cílový bolt již běží v nějakém worker procesu, budou tuply směřovány přímo do něj. Jinak funguje stejně jako Shuffle grouping.



Obrázek 4.11: *Možné fyzické uspořádání vláken při výpočtu topologie z obrázku 4.10 na clusteru se dvěma worker nódy čtyřmi worker procesy. Symboly S a B značí výpočetní vlákna spoutu a boltů.*



Obrázek 4.12: *Příklad topologie obsahující jeden spout a jeden typ boltů. TestWordSpout produkuje náhodná slova a ExclamationBolt přečte slovo na vstupu, přidá k němu znak "!" a výsledek emituje. Topologie s takto definovanými streamy tak bude produkovat výstup ve tvaru <slovo>!!*

4.3.2.4 Spolehlivost

Storm zaručuje, že každý tuple prošel topologií. To je zaručeno sledováním stromu života tuplů a rozpoznáním, kdy byl strom ukončen. Topologie mají definovanou maximální povolenou délku doby výpočtu, po kterou musí být tuple zpracován. Pokud během maximální povolené doby zpracován není, Storm zajistí zopakování daného tuplu. Pro využití spolehlivosti tak musí programátor sdělit, kdy vznikají nové větve ve stromu života tuplu a kdy byly ukončeny. K tomu slouží metoda *ack*() rozhraní *OutputCollector*.

Následující příklad ukazuje implementaci ExclamationBolt z předchozí kapitoly. V me-

todě *execute*() je je ukázáno, jak se definuje nová větev života tuplu – svázáním výstupu se vstupním tuplem.

```
public static class ExclamationBolt extends BaseRichBolt {
    OutputCollector _collector;

    public void prepare(Map conf, TopologyContext context,
                            OutputCollector collector) {
        _collector = collector;
    }

    public void execute(Tuple tuple) {
        _collector.emit(tuple, new Values(tuple.getString(0) + "!"));
        _collector.ack(tuple);
    }

    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word"));
    }
}
```

4.3.2.5 HDFS

Poslední infrastrukturní komponentou mého řešení je HDFS - Hadoop Distributed File System. Jedná se o distribuovaný filesystém vytvořený tak, aby běžel na běžném hardwaru. V mojí implementaci bude sloužit jako perzistentní vrstva při běhu topologie. HDFS je optimalizován pro velké objemy dat a přístupy typu „jednou zapiš, mnohokrát čti“ [19], což odpovídá potřebám mé aplikace. HDFS je implementován jako JAVA aplikace. Ovládání filesystému je umožněno prostřednictvím HDFS klienta nabízejícího podobné rozhraní jako bash. Kromě toho existuje také C a pro HDFS nativní JAVA API, což je i mnou používaný způsob. API je podobné známým IO a NIO JAVA rozhraním.

Kapitola 5

Návrh a realizace

5.1 Paralelizace FAKE GAME

5.1.1 Návrh

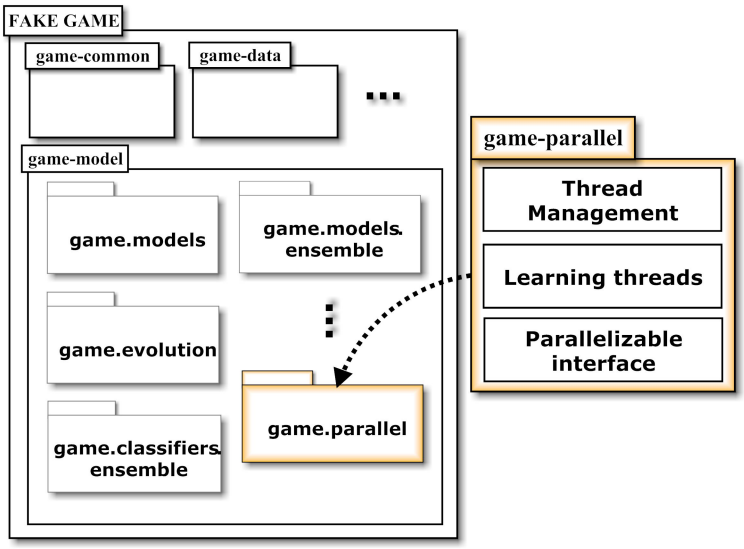
Jak je ukázáno v 4.2.4.1, největšího zrychlení lze dosáhnout paralelizováním těch částí výpočtu, které jsou časově nejnáročnější. Výpočetně nejnáročnější částí je proces učení modelů [9]. Učení modelů podle šablony probíhá způsobem popsaným v 2.2, přičemž řada ensemble algoritmů umožňuje učit své základní modely nezávisle na sobě, paralelně, a tudíž efektivněji.

5.1.1.1 Rozhraní ensemblů v FAKE GAME

Při návrhu obou částí řešení jsem vycházel z pokynů vedoucích projektu FAKE GAME. Pro první část byla hlavním požadavkem jednoduchost, malé množství zbytečného kódu starajícího se o paralelizaci a možnost stále používat sekvenční učení, tudíž jsem navrhl následující způsob. Aplikace FAKE GAME je složena z 5 modulů, přičemž o tvorbu modelů se stará modul game-models, který obsahuje kód pro konfiguraci, inicializaci a učení modelů, rozdělený podle logického uspořádání do jednotlivých java balíčků (package). Rozhodl jsem se, že první část řešení navrhnu jako balíček obsahující třídy, které se starají o plánování a synchronizaci vláken. Vlákna budou realizovat učení jednotlivých základních a ensemble modelů. Obrázek 5.1 schematicky znázorňuje situaci.

Algoritmy učení ensemble jsou specifické pro dané ensembly. Celkový pohled na rozhraní ensemble modelů v GAME uvádí obrázek 5.5 pro problémy regresní a 5.2 pro klasifikační. Metody realizující učení modelů jsou definovány v rozhraních $Classifier$ a $ClassifierEnsemble$ pro klasifikační problémy a $ModelLearnable$, resp. $ModelEnsemble$ pro problémy regresní. Jedná se o metody $learn()$, $relearn()$ a $learn(int)$. Metoda $learn()$ zajišťuje naučení daného modelu, popř. všech modelů v ensemblu, které ještě nejsou naučené, metoda $relearn()$ pak znovu naučení modelů, které již naučené jsou, a metoda $learn(int)$ je specifická pouze pro ensembly a definuje naučení jednoho konkrétního modelu z ensemblu.

Samotná implementace metod je závislá na konkrétním typu ensemblu nebo single modelu. Typicky u ensemble modelů tyto metody definují způsob přípravy dat pro modely v ensemblu a pořadí, v němž jsou modely v ensemblu učeny. Následující příklad uvádí jednoduchou metodu $learn()$ pro Bagging ensemble.



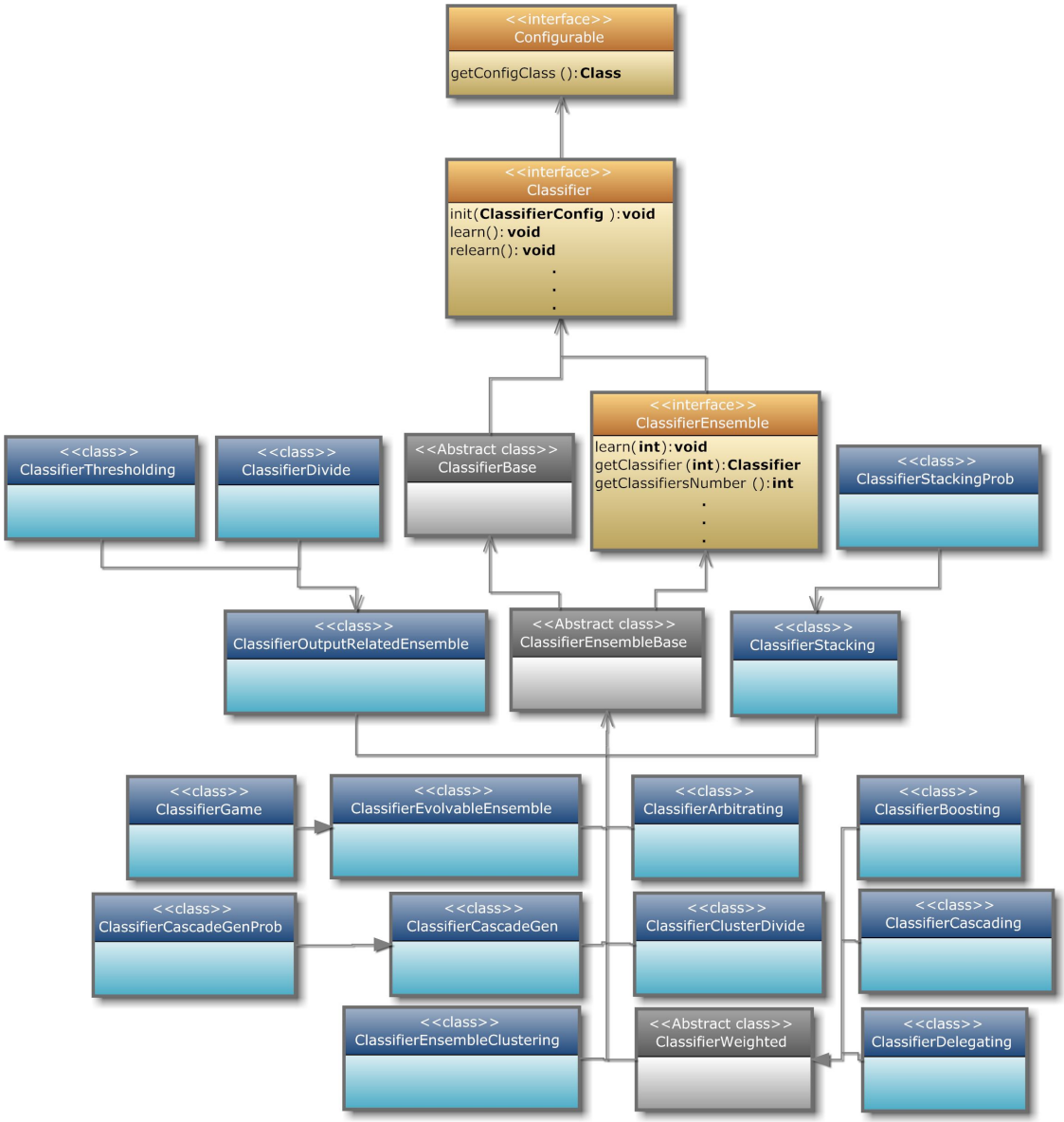
Obrázek 5.1: *Obrázek ukazuje celkový pohled na moduly aplikace FAKE GAME. Game-parallel je mnou navržený modul starající se o paralelizaci výpočtu modelů.*

```
public void learn() {
    Classifier cls;
    Random random = new Random();
    //for all classifiers in ensemble
    for(int i=0;i<numClassifiers;i++) {
        cls = ensClassifiers.get(i);
        //learn model if its not already learned
        if (!cls.isLearned()) {
            //prepare data
            prepareData(cls,random);
            //learn classifier
            cls.learn();
        }
    }
    //clean up
    postLearnActions();
}
```

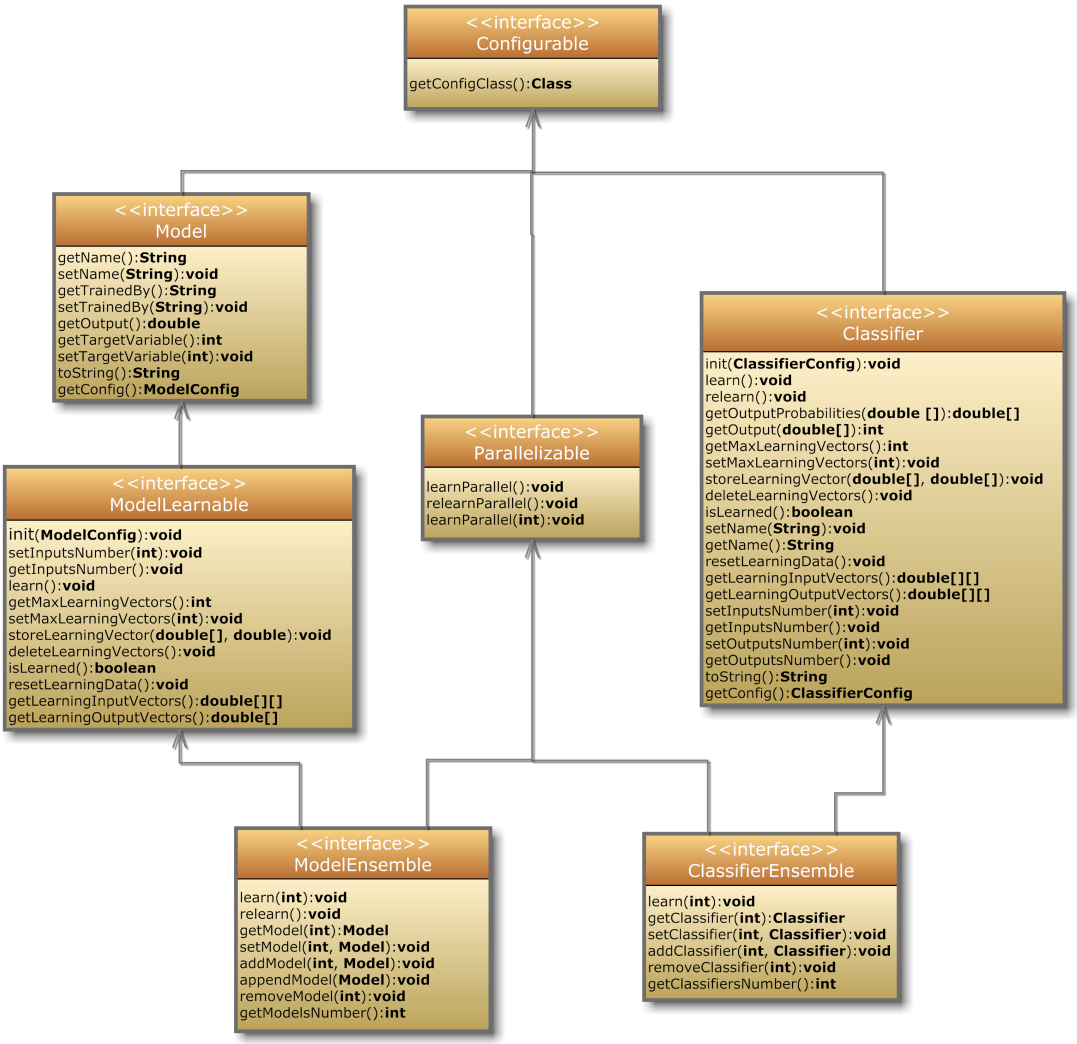
5.1.1.2 Balíček game.parallel

JAVA package game.parallel obsahuje třídy pro vlákna realizující učení, synchronizaci a plánování vláken, a rozhraní *Parallelizable*, ze kterého dědí rozhraní *ModelEnsemble*, resp. *ClassifierEnsemble*. Jednotlivé implementace jsem přidal přímo do tříd ensemblů. Rozhraní definuje metody $learnParallel(LearnManager)$, $relearnParallel(LearnManager)$ a

$learnParallel(LearnManager, int)$, jejich implementace je závislá na konkrétním ensemble algoritmu. Myšlenka všech implementací je stejná: rozdělit podmodely v ensemblu na skupiny, které lze učit najednou, a ty pak učit paralelně v rozdílných vláknech. Finální sktrukturu modelů uvádí obrázek 5.3.



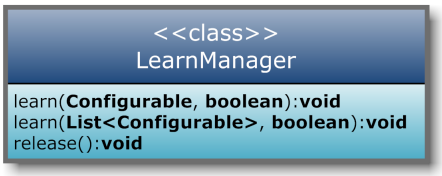
Obrázek 5.2: *Struktura tříd pro klasifikátory*



Obrázek 5.3: *Výsledná struktura rozhraní*

Základem kódu realizujícího paralelní výpočet je třída *LearnManager*. Ta vytváří vlákna pro jednotlivé modely a hlídá, aby během výpočtu nedošlo k vyčerpání maximálního množství vláken dřív, než se začne počítat základní model (list v šabloně), čímž by došlo k uváznutí aplikace. Možnost definovat maximální počet současně běžících vláken je nutný pro kontrolu alokace systémových zdrojů aplikaci. Třída nabízí dvě veřejné metody. Její rozhraní zobrazuje obrázek 5.4.

LearnThread je třída realizující učení jednoho konkrétního modelu. Její instance při nastartovaní vytvoří nové systémové vlákno a v něm volá příslušné učící metody podle třídy modelu a proměnné relearn. Pro modely které implementují rozhraní *Parallelizable*, volá *learnParallel*(), resp. *relearnParallel*(), pro *Classifier learn*() či *relearn*(), a pro základní *Model* pak *learn*().



Obrázek 5.4: *Rozhraní třídy LearnManager*

5.1.2 Realizace

Realizace spočívá v implementaci pozměněných učícíh ensemble algoritmů. Algoritmy lze rozdělit na dvě skupiny, ty které jdou paralelizovat a ty které nikoliv. U neparalelizovatlelných algoritmů metoda *learnParallel* funguje stejně jako její sekvenční varianta, jednotlivé modely se ale taktéž učí prostřednictvím LearnManageru aby se zařídilo zpropagování informace, že probíhá paralelní zpracování. Ensemble model níže v hierarchii šablony může být paralelizovatelný.

5.1.2.1 LearnManager

$LearnManager$ řídí paralelizaci. Poskytuje public metody pro učení jednoho, nebo skupiny modelů a hlídá počet právě běžících vláken. Třída má dvě public metody $learn(List < Configurable >, boolean)$ a $learn(Configurable, boolean)$ které pro danou množinu modelů vytvoří vlákna v nichž pak probíhá učení modelu. Maximální množství současně běžících modelů je kontrolováno pomocí semaforu ($java.util.concurrent.Semaphore$. Pokud se vyčerpá povolené množství vláken dřív než se začne učit nějaký model v listu hierarchické šablony, dojde k uváznutí. Tato situace je ošetřena tak, že aktuální vlákno, z nějž se volá $LearnManager$, zvýší počítadlo semaforu předtím, než nastartuje učící vlákna. To si lze dovolit, neboť toto vlákno pak čeká na dokončení vláken která startuje. Stejný postup je uplatněn i pokud chce své modely učit ensemble, který nejde paralelizovat. Jen místo skupiny modelů předává model jeden. Pro něj se opět stejným způsobem vytvoří nové vlákna a volající vlákno čeká na jeho dokončení. Dále popisuji jen paralelizovatelné algoritmy.

Ukázka kódu metody *learn* třídy $LearnManager$:

```
public void learn(List<Configurable> confs, boolean relearn) {
List<LearnThread> threads = new ArrayList<LearnThread>();
    boolean permitGiven = false;
    try {
        //create threads
        for (int i=0;i<confs.size();i++){
            Configurable cnf = confs.get(i);
            LearnThread lthread = new LearnThread(cnf, this, relearn);
            threads.add(lthread);
        }
        Integer actPermits = 0;
        synchronized(actPermits){
```

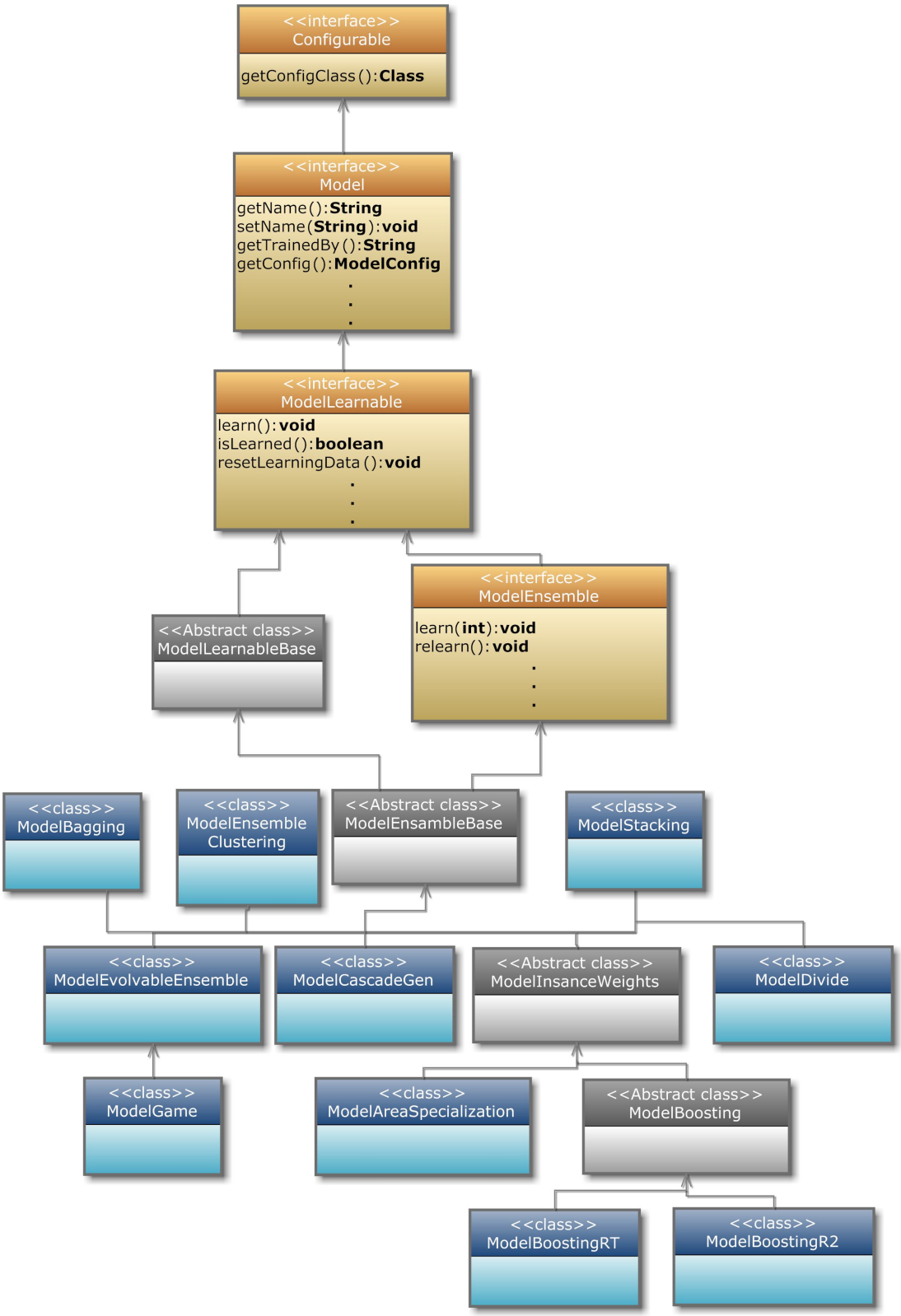
```
            actPermits = sem.availablePermits();
            if (actPermits < this.maxPermits) {
                //need to do this to prevent deadlock in case no
                //permits are available and we are not in any leaf
                //node of template - can afford to do it, since this
                //thread is going to wait for others to finish
                this.release();
                permitGiven = true;
            }
        }
        //start these threads using executeThreads method
        this.executeThreads(threads);
    } finally {
        //wait for threads to finish
        for (int i=0; i<threads.size(); i++){
            try {
                threads.get(i).join();
            } catch (InterruptedException ex) {
                Logger.getLogger(LearnManager.class.getName())
                .log(Level.SEVERE, null, ex);
            }
        }
        //continue, so acquire given permit back
        if (permitGiven) {
            this.acquire();
        }
    }
}
```

5.1.2.2 Arbitrating

Arbitrating je ensemble algoritmus pouze pro klasifikační úlohy. Modely v ensemblu jsou rozdělené na dvě skupiny. Sudé modely fungují jako „rozhodčí“ pro liché modely 3.6.2.5. Modely v jedné skupině se navzájem nijak neovlivňují, závislost je pouze mezi modely z různých skupin. Paralelní verze algoritmu tak nejdříve naučí liché modely, po dokončení jejich učení proběhne učení rozhodčích.

5.1.2.3 Bagging

Bagging vytvoří vzorek dat pomocí náhodného výběru s opakováním pro každý model v ensemblu. Výstup je dán průměrem u regresních úloh, nebo se používá hlasování u úloh klasifikačních. Paralelní verze učení vypadá stejně. Aktuálně běžící vlákno vytvoří vzorky dat, a dvojici model plus data uloží do seznamu modelů které lze učit paralelně. Seznam poté předá *LearnManageru* a čeká na dokončení jejich exekuce.



Obrázek 5.5: *Struktura tříd pro modely*

5.1.2.4 Cluster divide

Algoritmus Cluster divide ensemble rozděluji vstupní množinu dat na shluky, které přiřazuje modelům v ensemblu. Modely v ensemblu se navzájem nijak neovlivňují, paralelní verze vypadá, kromě způsobu přípravy dat, stejně jako u Baggingu.

5.1.2.5 Stacking

U Stackingu se nejdříve všechny modely v ensemblu, kromě posledního, učí na svém vzorku dat, poslední model se pak učí z výstupů těchto modelů. Paralelní verze pro všechny modely kromě posledního vypadá opět stejně jako u Baggingu. Pro poslední model jsem udělal metodu learnMetamodelParalel, která kopírují svojí sekvenční verzi, jen pro učení modelu použije *LearnManager*.

5.1.2.6 Thresholding

V případě Thresholding lze učit najednou ty modely, které nespadají do stejné oblasti. Paralelní verze použije stejné metody jako verze sekvenční pro vytvoření oblastí, učení modelů z různých oblastí pak pouští paralelně.

5.1.2.7 Classifier model

Classifier model je speciální typ ensemblu, který pro každou třídu vytvoří zvláštní dataset a tím převede klasifikační úlohu na úlohu regresní. Jednotlivé modely v ensemblu pak řeší tyto regresní úlohy. Modely se opět nijak neovlivňují, lze je tedy učit paralelně. Následující ukázka zobrazuje metodu *learnParallel* pro Classifier model.

```
public void learnParallel(LearnManager lte) {
    ModelLearnable LearnableModel;
    List<Configurable> confs = new ArrayList<Configurable>();
        //pro all models in ensemble
        for(int i=0;i<numModels;i++) {
            if (classifierModels.get(i) instanceof ModelLearnable) {
                LearnableModel = (ModelLearnable)classifierModels.get(i);
                //learn model if its not already learned
                if (!LearnableModel.isLearned()) {
                    //prepare data, ie. pick one class
                    prepareData(LearnableModel);
                    //add to list for parallel processing
                    confs.add(LearnableModel);
                }
            }
        }
        //learn models on LearnManager
        lte.learn(confs, false);
        //clean up input data
```

```
        postLearnActions();
    }
```

5.2 Game-storm - návrh

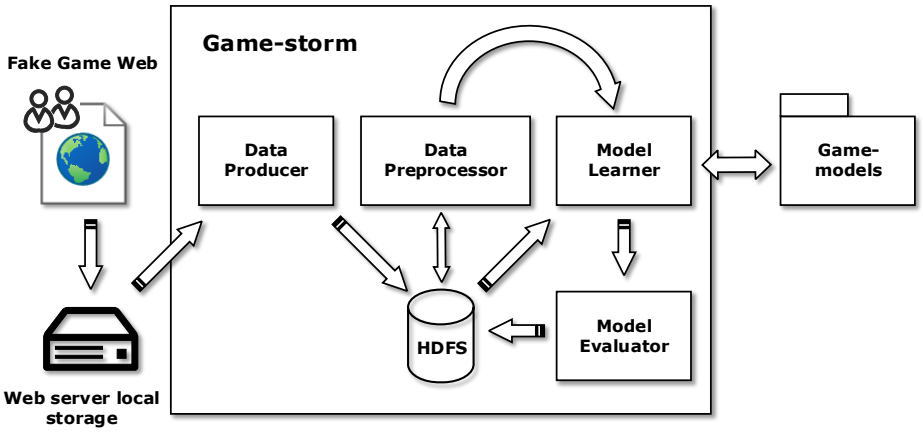
Jak jsem již uvedl v úvodu práce 1, chtějí správci projektu vytvořit aplikaci, která bude nabízet přes jednoduché webové rozhraní vytvoření modelu v reálném čase. Proto jsem se rozhodli vytvořit pro FAKE GAME nový modul, který bude sloužit jako výpočetní server pro tuto aplikaci. Kromě počítání modelů v reálném čase bude zároveň schopný počítat modely pro velké objemy vstupních dat. Ensemble algoritmy implementované v GAME mají totiž kromě vysokých nároků na výpočetní výkon i vysoké nároky paměťové. Teoreticky mají paměťovou náročnost přinejmenším $(N * M)$, kde N je množství učících dat a M je počet uzlů mezi kořenem a listy stromu šablony. Většina ensemble algoritmů však vstupní data nějakým způsobem modifikuje a vytváří další instance [10]. Testy na vybraných šablonách, uvedených v [6], a umělých datech vytvořených pouhou multiplikací příslušných datasetů, ze kterých byly tyto šablony vytvořeny, ukazují, že velikost java heap potřebná pro výpočet modelu je většinou $\sum_{i=0}^{M} K_i * N$, kde M je výška stromu šablony, K_i je počet modelů v ensemblech v i-té vrstvě stromu a N je množství vstupních dat. Pro typické šablony se tak jedná o zhruba sedmdesáti- až sto padesáti-násobky velikosti vstupního datasetu.

Jako základ modulu jsem zvolil framework Storm, kvůli jeho specializaci na realtime počítání a přívětivé rozhraní umožňující snadno a efektivně řešit moji úlohu. Storm cluster již byl testován v laboratoři skupiny CIG. Pro rychlý přístup jednotlivých komponent systému k datům, omezení datových přenosů a snížení nutnosti serializace velkých datových celků jsem se rozhodl pro přenos a uchovávání dat použít distribuovaný filesystém HDFS. Stručné představení frameworku Storm jsem uvedl v kapitole 4.3.2.

Modul jsem navrhl způsobem, který ilustruje obrázek 5.6. Skládá se ze čtyř hlavních komponent. Data Producer zajišťuje čtení dat z lokálního úložiště web serveru a ukládá je do distribuovaného filesystému HDFS. Komponenta Data Preprocessor tato data čte, v případě, že jsou příliš velká pro výpočet modelu najednou, tak provede jejich samplování. Poté je uloží zpět do HDFS a pošle zprávu Model Learneru. Ten načte data, zavolá učení modelu a výsledné modely posílá Model Evaluatoru. Model Evaluator provede ohodnocení modelu a uloží ohodnocené modely na DHFS úložiště. Kromě těchto hlavních komponent bylo zapotřebí vyvinout podpůrnou komponentu sloužící jako HDFS klient, který zajišťuje komunikaci s HDFS úložištěm. Komponenty jsou navržené tak, aby každá komponenta mohla tvořit samostatnou jednotku (Bolt nebo Spout) topologie Stormu. Jedinou výjimkou je Data Producer. Důvodem je skutečnost, že přidělování komponent topologie worker nodům si řídí framework sám. Všechny worker nody by musely mít zaručen přístup na lokální úložiště web serveru.

5.2.1 Popis realizace

Při implementaci jsem vycházel z návrhu. Implementoval jsem topologii, ve které komponentám uvedeným v návrhu odpovídá jedna komponenta topologie. Strukturu topologie uvádí obrázek 5.7.Data producer není implementován jako součást topologie kvůli výše uvedenému důvodu.



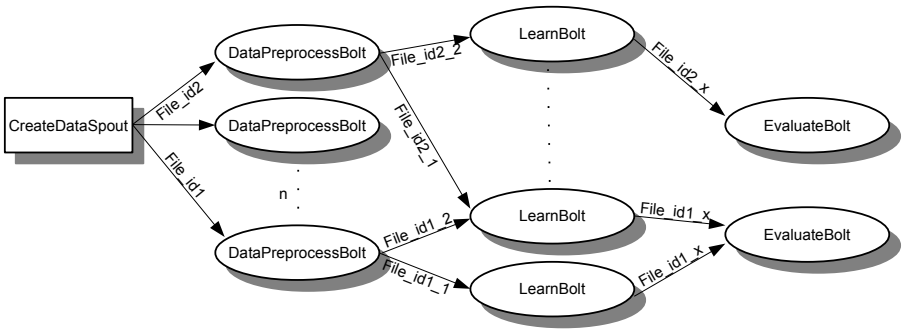
Obrázek 5.6: *Návrh komponent modulu Game-storm. Data Producer zajišťuje čtení dat z lokálního úložiště web serveru a ukládá je do distribuovaného filesystému HDFS. Komponenta Data Preprocessor tato data čte, v případě že jsou příliš velká pro výpočet modelu najednou tak provede jejich samplování. Poté je uloží zpět do HDFS a pošle zprávu Model Learneru. Ten načte data, volá učení modelu a výsledné modely posílá Model Evaluatoru. Model Evaluator provede ohodnocení modelu a uloží ohodnocené modely na DHFS úložiště.*

5.2.1.1 DataProducer

Data Producer implementuje třída *DataProducer*. Tu lze spustit jako samostatný proces, nebo jako vlákno běžící mimo topologii. Periodicky kontroluje obsah zadaného adresáře na lokálním filesystému a v případě, že přibudou nová data, tato data kopíruje do HDFS úložiště. Existující datové adresáře si spolu s údaji o jejich zprocesování udržuje v mapě dat. Data musí mít odpovídající strukturu - tři soubory obsahující trénovací data, konfiguraci šablony a trénovací data v samostatném adresáři. Typ dat je určen příponou souboru. Pro komunikaci s HDFS jsem vytvořil třídu HDFSClient poskytující nutné funkce jako kopírování souboru z lokálního filesystému, zápis a čtení dat apod. Při návrhu aplikace, kdy více vláken může najednou přistupovat do HDFS je zapotřebí si uvědomit, že deskriptor HDFS filesystému je singleton, a tudíž jeho zavření zavře všechny aktuální konexe do filesystému.

5.2.1.2 CreateDataSpout

Spout *CreateDataSpout* funguje podobně jako DataProducer, nicméně již implementuje rozhraní *BaseRichSpout* 4.3.2.2 a slouží jako zdroj dat pro topologii. Taktéž periodicky kontroluje zadanou adresářovou strukturu jestli nepřibyla nová data, tentokrát ale na HDFS úložišti. Pokud existují nová data, emituje do streamu *configData* HDFS cestu k datům a identifikátor úkolu, aby bylo posléze možné přiřadit spočítané modely k zadaným datům.



Obrázek 5.7: *Topologie GAME-Storm. Obrázek znázorňuje mnou navrženou topologii pro výpočet modelů. Popis funkcí komponent uvádí kapitola 5.2.1.*

5.2.1.3 DataProcessBolt

Na stream do nějž emituje *CreateDataSpout*, je připojeno n instancí *DataProcessBoltu*, kterým jsou v náhodném pořadí přidělovány tuply ze streamu dle pravidel definovaných pro shuffle grouping 4.3.2.3. V případě příliš velkého objemu vstupních dat zajišťují jejich náhodné samplování (vzorkování) a uložení vzorků v HDFS. Do streamu *processedData* emituje tuply obsahující vše potřebné k výpočtu modelu, tedy cestu ke vzorku trénovacích dat, cestu k šabloně a k testovacícm datům. Kromě toho tuply díle propagují identifikátor původního datového souboru, celkový počet vzorků a pořadí vzorku. Pokud bolt zpracoval dataset, který nebylo třeba samplovat, potvrdí zpracování tuplu, až když dostane potvrzení od následujícího boltu v řadě. Tím je v případě chyby zajištěno opakované odeslání příslušného tuplu z *CreateDataSpoutu* - princip spolehlivosti Storm topologie je vysvětlen v kapitole 4.3.2.4. V případě, že bolt zpracovával datový soubor, tkerý bylo zapotřebí samplovat, odesílá potvrzení hned po emitování svého tuplu. Zabraňuje se tím zahlcení topologie v případě špatné konfigurace, např. vinou příliš vysoké maximální velikosti vzorku dat. Pokud dojde k chybě u jednoho z mnoha vzorků, stále budou k dispozici modely z ostatních vzorků. Počet neúspěšných výpočtů je patrný z počtu spočítaných modelů.

5.2.1.4 LearningBolt

Tuply z výstupního streamu *DataProcessBolt* jsou opět dle pravidel shuffle grouping přidělovány *LearningBoltm*. *LearningBolt* vytvoří dle cesty k data samplům a šabloně příslušné GAME objekty a zavolá ModelFactory, resp. ClassifierFactory, definované v modulu game-models pro vytvoření modelů. Pro tvorbu modelu používá mojí paralelní implementaci popsanou výše 5.1.2. Bolt běží jako vlákno ve Worker procesu 4.3.2.1 v jedné konkrétní JVM na jednom worker nodu, není tudíž důvod paralelizaci pro sdílenou paměť nepoužít. Potvrzení o zpracování tuplu posílá až když dostane potvrzení o zpracování svého tuplu.

5.2.1.5 EvaluateBolt

EvaluateBolty jsou připojeny na výstupní stream do nějž zapisují *LearningBolty*. Narozdíl od předchozích boltů nezpracovávají tuply náhodně, ale díky použití *Fieldsgrouping* je zaručeno, že všechny tuply se stejným identifikátorem úkolu bude zpracovávat stejný bolt. Každý *EvaluteBolt* si drží stav jím zpracovávaných úkolů a po přijetí všech modelů patřících k danému úkolu udělá ohodnocení modelů (v této verzi implementován RMSE - Root Mean Squared Error) a zapíše všechny ohodnocené modely do příslušného adresáře se vstupními daty v HDFS. Nejlepších N modelů je zvýrazněné jinými názvy souborů. Po zapsání výsledků posílá bolt potvrzení o zpracování přijatého tuplu.

Kapitola 6

Testování a zhodnocení zvoleného řešení

Důvodem paralelizace aplikace GAME bylo zrychlení výpočtu modelů oproti výpočtu sekvenčnímu.

6.1 Porovnání paralelních implementací

Požadavek na zrychlení byl ověřen srovnáním délky sekvenčního a paralelního výpočtu na stejném testovacím hardwaru při použití vícevláknové verze aplikace bez Stormu a při použití Stormu běžícího v local clusteru módu (vývojový mód pro Storm umožňující běh clusteru jako vláken v rámci jedné JVM).

Nejprve jsem provedl srovnávací měření mezi oběma paralelními verzemi. Pro srovnání jsem použil datasety o velikosti cca. 512kB (local cluster Stormu v současné verzi neumožňuje konfigurovat velikost JAVA heap) a mnou předgenerované šablony, u nichž lze očekávat dobrou paralelizovatelnos - obsahují většinu paralelizovatelných ensemble modelů. Local cluster byl nakonfigurován tak, aby byl spuštěn pouze jeden bolt realizující výpočet, a nejlépe tak simuloval činnost jednoho worker nódu ve storm clusteru pro porovnání režije potřebné pro Storm worker nód oproti verzi bez Stormu. Srovnání délky výpočtu ukazuje následující tabulka 6.1. Testování probíhalo na počítači se čtuřjádrovým procesorem Intel Core i5 na frekvenci 3,8GHz se 16GB RAM.

6.2 Zrychlení oproti sekvenční verzi

Zrychlení je definováno takto:

$$S = \frac{T_{seq}}{T_{par}} \tag{6.1}$$

kde T_{seq} je délka běhu programu sekvenčně a T_{par} je délka běhu paralelně.

Kvůli v podstatě stejnému výkonu obou paralelních verzí aplikace jsem se pro porovnání s verzí sekvenční rozhodl použít verzi bez Stormu. Porovnávat délku výpočtu sekvenční verze,

Měření	Bez Stormu	Se Stormem	**Rozdíl**
1	420,8	426,3	**1,03**
2	107,2	107,5	**0,3**
3	52,43	52,70	**0,5**
4	721,2	728,9	**1,1**

Tabulka 6.1: *Tabulka ukazuje rozdíl délky paralelního výpočtu mezi verzí se Stormem a bez Stormu. Časy jsou uvedené v sekundách, jedná se vždy o průměr z dvaceti měření. Sloupec „rozdíl“ udává percentuální zpomalení, pohybuje se do maxima jednotek procent. Lze tedy říci, že na jednom počítači dosahují obě verze téměř stejných výsledků.*

běžící na jednom počítači, oproti verzi se Stormem nasazené na clusteru, nemá smysl. Vzhledem ke způsobu implementace (model se počítá v rámci jednoho worker nódu na clusteru 5.2) a výsledkům z předchozího měření bude zrychlení 6.1 stejné jako u verze bez Stormu při použití stejného HW pro worker nódy clusteru. Clusterová verze byla vytvořena převážně kvůli možnosti zpracovávat větší množství modelů najednou a kvůli výpočtům modelů na větším objemu učících dat, které by na verzích bez distribuované paměti, vzhledem k paměťové náročnosti výpočtů, nebylo možné provádět.

Provedl jsem 6 pokusů s různě složitými vstupními daty a různými šablonami; kombinace jsem převzal z [6], aby odpovídaly realitě. Pro přímé měření délky výpočtu jsem použil metodu $System.currentTimeMillis()$. Do testování jsem zahrnul jednu šablonu, která nemá žádný ensemble algoritmus, jež by profitoval z paralelizace, pro ukázku jaký overhead (režii) má moje paralelizace oproti čistě sekvenční variantě. Jedná se data set Bosthouse a šablonu BoostingRTModel(tr=0.1) [2x CascadeGenModel 4x ExpModel, 2x ModelAreaSpecialization 8x SineModel].

6.2.1 Testovací data a šablony

Jako testovací data jsem používal sady reálných dat multiplikovaných na velikost cca 3 MB. Tyot sady jsou k dispozici na stránkách UCI repository. Jejich přehled uvádí tabulka 6.2. Zvolené kombinace datasetů a modelů mi umožnily provádět měření v rozsahu od 14 s u datasetu Breast až po cca 1000 s u Iris. Měření jsou opakována dvacetkrát: více měření vzhledem ke skutečnosti že šablony zaručují podobnost výsledných modelů, a tudíž i doba potřebná pro jejich učení bude velmi podobná, nemá smysl.

6.2.2 Změřené výsledky

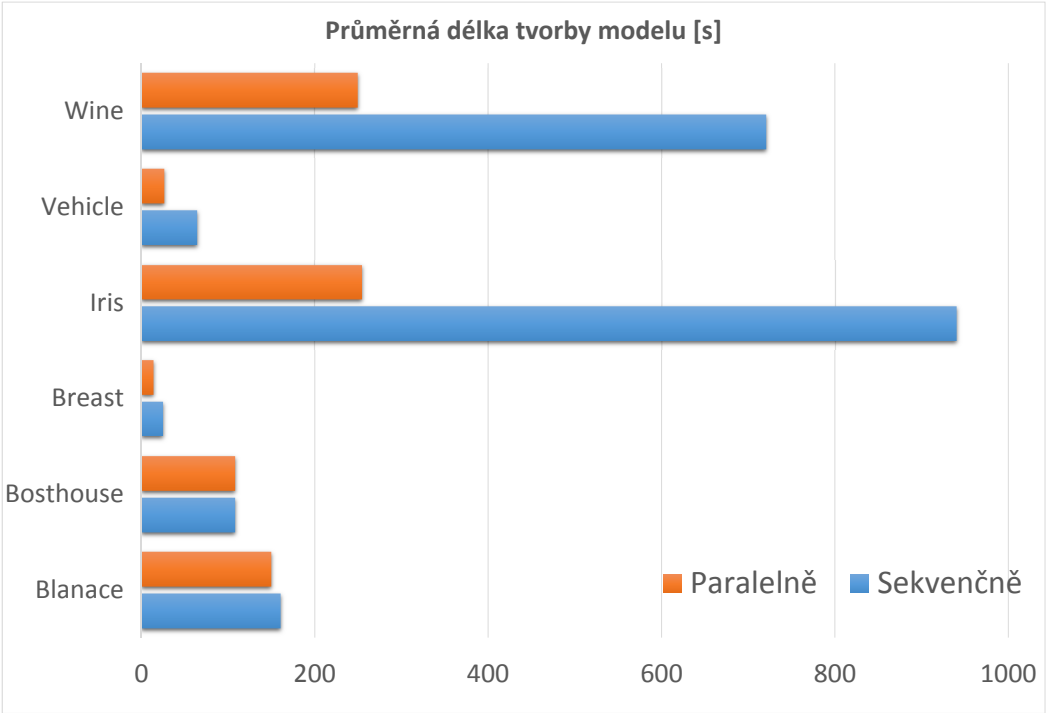
Datové soubory jsou uloženy na přiloženém CD v adresáři \testing\data. Tabulka s naměřenými hodnotami je k nahlédnutí v adresáři \testing. Graf 6.1 ukazuje souhrnné porovnání naměřených hodnot délky běhu pro všech 6 měření. Zobrazuje průměrné hodnoty z dvaceti měření, použití jiného zobrazení dat nemá, vzhledem k použití šablon, význam. Z grafu a následující tabulky je patrné zrychlení při paralelním zpracování. V tabulce 6.3 jsou vždy uvedeny průměrné hodnoty pro každé měření a zrychlení je spočítáno podle vzorce 6.1.

Data Set	Konfigurace šablony
Balance	Boosting57x ClassifierModel<outputs>x PolynomialModel(degree=4)
Boshouse	BoostingRTModel(tr=0.1)[2x CascadeGenModel4x ExpModel, 2xModelAreaSpecialization8x SineModel]
Breast	ClassifierModel<outputs>x CascadeGenModel[7x CascadeGenModel[5x GaussianModel]]
Iris	ClassifierArbitrating8x ClassifierBagging3x ClassifierModel<outputs>x [GaussianMultiModelConfig, SigmoidModelConfig]
Vehicle	ClassifierArbitrating6x ClassifierModel<outputs>x DivideModel(mult=6.68)[7x PolynomialModel(degree=3)]
Wine	CascadeGenProb9x ClassifierModel<outputs>x BoostingRTModel(tr=0.1)[8xGaussianModel]

Tabulka 6.2: *Testovací šablony a data*

Data set	Délka sekvenčního běhu	Délka paralelního běhu	**Zrychlení**
Balance	160	149	**1,07**
Bosthouse	107	107	**1**
Breast	24,7	13,9	**1,78**
Iris	939	255	**3,69**
Vehicle	64,3	26,4	**2,44**
Wine	720	249	**2,88**

Tabulka 6.3: *Tabulka ukazuje zrychlení výpočtu, časové údaje jsou v sekundách. Šablona pro dataset Bosthouse neobsahuje žádné paralelizovatelné ensembly a byla přidána uměle na ukázku toho, jakou režii má paralelní zpracování oproti sekvenčnímu. Naměřené zrychlení se pohybuje v rozmezí 1,07 pro nejhůře paralelizovatelnou šablonu do 3,69 u šablony nejlépe paralelizovatelné.*



Obrázek 6.1: *Porovnání naměřených hodnot. Délka výpočtu modelu sekvenčně je uvedena modrou barvou, délka paralelního zpracování oranžově. U čtyř ze šesti měření je patrné výrazné zrychlení.*

Nejnižší zrychlení, kromě záměrně zařazené šablony, bylo naměřenoé u šablony Boosting{57x ClassifierModel{<outputs>x PolynomialModel(degree=4)}} na datech balance. To je dáno tím, že boosting není paralelizovatelný, modely v ensemblu se učí postupně, a jediná paralelizace v této šabloně tak byla současný výpočet tří základních polynomiálních modelů najednou - *ClassifierModel* rozdělí data podle tříd a ke každé třídě pak přistupuje jako k regresnímu problému. Dodatečné testy ukázaly, že naučení základního polynomiálního modelu trvá řadově desítky až stovky milisekund a většinu výpočetního času zabere modifikace dat v Boostingu: zde tudíž k vysokému zrychlení ani dojít nemůže. U ostatních testovacích šablony se zrychlení pohybovalo v rozmezí 1,78 - 3,69.

6.3 Velké datasety

Distribuovaná verze aplikace byla testována na Storm clusteru se čtyřmi worker nódy, nasazenými na počítačích se čtyřjádrovými procesory Intel Core i5 s frekvencí 3,2 GHz a operačním systémem Debian Linux. Testování probíhalo s různými konfiguracemi počtu worker nódů, počtu procesů běžících v rámci jednoho nódu a počtu vláken. Testy prokázaly škálovatelnost aplikace dle teoretických předpokladů: nejlepší výsledky pro real time zpracování velkého

množství malých datasetů či velkých dat samplovaných na malé vzorky vykazovala aplikace při konfiguracích s počtem procesů i vláken v každém procesu mírně vyšších než je počet procesorů worker nódu. V případě potřeby zpracovávat větší vzorky najednou je zapotřebí omezit paralelizace uvnitř procesů kvůli velikosti spotřebovávané paměti. Maximální velikost paměti přidělené jednomu worker procesu nejde v současné verzi Stormu konfigurovat pro jednotlivé topologie, je nastavena při startu clusteru.

V konfiguraci se čtyřmi worker procesy na každém nódu, přičemž pro učení modelů bylo alokováno 11 procesů a učení probíhalo v maximálně 5 vláknech v každém procesu, zpracovala aplikace během necelých dvou hodin, přesněji 116 minut, data o objemu 320MB. Maximální velikost vzorku byla nastavena na 3 MB, přičemž velikost JAVA heap byla 1024 MB. U vyšších velikostí vzorků docházelo k vyčerpání paměti přidělené učícím procesům.

Kapitola 7

Závěr

Podařilo se mi implementovat dvě změny umožňující paralelní výpočet modelů generovaných aplikací FAKE GAME. Hlavním cílem paralelizace bylo zrychlení výpočtu modelů, umožnění automatizovaného počítání většího množství modelů najednou a počítání modelů pro velké trénovací množiny dat. Paralelizací výpočtu ensemble modelů Arbitrating, Bagging, Classifier Model, Cluster Divide, Stacking a Thresholding se mi podařilo dosáhnout zrychlení jejich výpočtu.

Z měření uvedených v předchozí kapitole je patrné, že mnou implementovaná paralelizace přináší ve většině případů znatelné zrychlení průměrné doby tvorby modelu. Měření prokázala, že zrychlení na čtyřjádrovém procesoru se pohybuje v rozmezí $1,07 - 3,69$ na vybraných reálných datasetech a jim odpovídajících hierarchických šablonách.

Testování aplikace na Storm clusteru prokázala její schopnost počítat modely pro řádově stovky megabytů trénovacích dat. Vlastnosti mnou navrženého řešení lze snadno upravovat konfigurací počtu worker nódů ve Storm clusteru, změnou počtu worker procesům na worker nodech, úpravou velkosti paměti přidělené a počtem maximálního počtu vláken běžících v jednom procesu. Systém je tak možné optimalizovat pro real time výpočty modelů z mnoha menších datových sad, i pro počítání modelů z velkých datasetů.

Zdrojové kódy aplikace jsem uveřejnil na stránkách Sourceforge.net jako novou větev Game-Storm v úložišti projektu FAKE GAME.

7.1 Možná vylepšení

Jako budoucí vylepšení mnou navrženého a implementovaného řešení se nabízí vyvinutí většího množství Storm boltů realizujících jiné způsoby samplování vstupních dat či přidání boltů realizujících jednotlivé učící algoritmy, což by umožnilo výraznější paralelizaci výpočtu jednoho modelu.

U vícevláknové verze pro sdílenou paměť se jako možné vylepšení nabízí vytvoření plánovače vláken, který bude plánovat spouštění výpočetních vláken na základě odhadu složitosti modelu, čímž by se minimalizovalo čekání na dokončení výpočtu modelů v ensemblu.

Literatura

[1] Amdahl's law.
http://en.wikipedia.org/wiki/Amdahl's_law.

[2] CRISP-DM - hlavní stránka.
http://www.crisp-dm.org/index.htm.

[3] Flynn's taxonomy.
http://en.wikipedia.org/wiki/Flynn's_taxonomy.

[4] API specification for version 6 of the Java Platform, Standard Edition.
http://java.sun.com/javase/6/docs/api/java/lang/Thread.html.

[5] Oficiální stránka předmětu X36OSY.
http://service.felk.cvut.cz/courses/X36OSY/.

[6] Pavel Kordík and Jan Černý. Building Predictive Models in Two Stages with Meta-Learning Templates. paper for KDD 2013, Chicago, IL, USA.

[7] Dr. Gregory Piatetsky-Shapiro and Prof. Gary Parker. Data Mining Course, 2006.
http://www.kdnuggets.com/data_mining_course/.

[8] Geoffrey I. Webb and Janice R. Boughton and Zhihai Wang. Not so naive Bayes: Aggregating one-dependence estimators. *Machine Learning*. 2004, s. 1–4.

[9] Jan Černý. Metody pro kombinování modelů a klasifikátorů, Květen 2010.

[10] Jan Černý. Metody pro kombinování modelů a klasifikátorů, Květen 2010.

[11] Lior Rokach. Ensemble Methods in Supervised Learning. In MAIMOM, O. – ROKACH, L. (Ed.) *Data mining and knowledge discovery handbook*. 2010. s. 959–979. ISBN 978-0-387-09822.

[12] Ovidiu Ivanciuc. *SVM - Support Vector Machines* [online]. 2005.
http://www.support-vector-machines.org.

[13] P. Kordík and J. Motl. Přednáška 7: Bayesovská klasifikace. Přednáška předmětu Vytěžování znalostí z dat, 2011.
https://edux.fit.cvut.cz/oppa/BI-VZD/prednasky/p7-Bayes.pdf.

[14] Pavel Kordík, Miroslav Šnorek. Progress in inductive modelling. 2004.
http://neuron.felk.cvut.cz/game/doc/game.pdf.

[15] Petr Berka. *Dobývání znalostí z databází.* Prague : Academia, 1 edition, 2003. ISBN 80-200-1062-9.

[16] Petr Pošík. Data Mining, Část I. Co je data mining?, 2005. `http://cyber.felk.cvut.cz/gerstner/teaching/zbd/DataMining1-hout.pdf`.

[17] Richardo Vilalta and Christophe Giraud-Carrier and Pavel Brázdil. Meta-learning: Concepts and techniques. In MAIMOM, O. – ROKACH, L. (Ed.) *Data mining and knowledge discovery handbook.* 2010. s. 717–731. ISBN 978-0-387-09822.

[18] The Apache Software Foundation. *Welcome to Apache Hadoop* [online]. 2013. `http://hadoop.apache.org/`.

[19] The Apache Software Foundation. *HDFS Architecture Guide* [online]. 2013. `http://hadoop.apache.org/docs/stable/hdfs_design.html`.

[20] The Apache Software Foundation. *Apache Zookeeper Home* [online]. 2013. `http://zookeeper.apache.org/`.

[21] Usama M. Fayyad and Gregory Piatetsky-Shapiro and Padhraic Smyth. *From data mining to knowledge discovery in databases.* American Association of Artificial Intelligence, 1996.

[22] Václav Hlaváč. Umělé neuronové sítě z pohledu rozpoznávání. `cmp.felk.cvut.cz/~hlavac/TeachPresCz/31Rozp/72UmeleNN.ppt`.

Příloha A

Obsah přiloženého CD

- Adresář src obsahuje kompletní zdrojové soubory aplikace FAKE GAME.
- Adresář dist obsahuje distribuci aplikace FAKE GAME ve formě jar souborů.
- Adresář data obsahuje tesovací data a soubory šablon.
- Adresář dp obsahuje zdrojové soubory pro vygenerování této práce v systému LaTeX.
- Elektronická verze této práce je ve formátu PDF uložena v root adresáři.