Univerzita Hradec Králové
Fakulta informatiky a managementu
Katedra informatiky a kvantitativních metod

Strojové učení na platformě Apache Spark

Bakalářská práce

Autor: František, Hylmar
Studijní obor: Aplikovaná informatika

Vedoucí práce: prof. RNDr. PhDr. Antonín Slabý, CSc.

Odborný konzultant: Ing. Ondřej Klapka

Hradec Králové březen 2020

Prohlášení:

Prohlašuji, že jsem bakalářskou práci zpracoval samostatně a s použitím uvedené literatury.

V Hradci Králové dne 10.4.2020 František Hylmar

Poděkování:

Děkuji vedoucímu bakalářské práce prof. RNDr. PhDr. Antonínu Slabému, CSc. za metodické vedení práce a neocenitelnou pomoc při její tvorbě.

Anotace

Bakalářská práce se zaměřuje na aplikaci postupů strojového učení na velké objemy dat pomocí platformy Apache Spark. V práci jsou popsány metody doporučování produktů pomocí algoritmů pro kolaborativní filtrování. Vybraný algoritmus je implementovaný v programovacím jazyce Scala a nasazen na virtuální Hadoop clustr. Pro dataset obsahující implicitní data o poslechu hudebních interpretů je na základě implementovaného algoritmu vytrénován a optimalizován příslušný model. Přesnost predikcí modelu je následně ohodnocena jak strojově, tak i manuálně.

Annotation

Title: Machine Learning with Apache Spark

This bachelor thesis focuses on the application of machine learning procedures to large volumes of data using the Apache Spark platform. This work describes methods of product recommendation using collaborative filtering algorithms. Selected algorithm is implemented in Scala programming language and deployed on virtual Hadoop cluster. For a dataset containing implicit music data, the model is trained and optimized based on the implemented algorithm. The accuracy of model predictions is then evaluated both automatically and manually.

Obsah

1. Úvod

Podíl online distribuce na celkových objemech neustále roste a to i v segmentech, které byly donedávna doménou kamenných obchodů. Tento růst má za následek tlak, kdy jednotliví obchodníci soupeří o větší podíl na trhu. U většiny úspěšných společností je chování zákazníka pečlivě sledováno. Každý zákazník za sebou zanechává digitální stopu, tato stopa je následně uložena a analyzována. Výsledky takových výpočtů mohou být dále použity pro vylepšení další interakce se zákazníky. Zlepšení uživatelské zkušenosti může nabývat mnoho podob, jedním z nejzřetelnějších je snaha nabídnout zákazníkům takové produkty, které by se jim mohly zamlouvat.

Množství dat, které je nutné zpracovat je často tak velké, že přesahuje možnosti standartních nástrojů jako jsou například relační databáze. Výpočty takového rozsahu si vyžádaly nové přístupy jak v samotném ukládání dat tak i v jejich následném zpracování.

V roce 2003 vydala společnost Google dokument Google File System [1] popisující strukturu distribuovaného souborového systému GFS, o rok později vydala tatáž společnost dokument MapReduce [2], popisující jednoduchý programovací model pro paralelní zpracování dat nad GFS. Nejen na základě těchto článků vznikla iniciativa, která měla jako cíl vytvořit volně dostupnou platformu pro zpracování velkých objemů dat. Tento projekt, známý pod jménem Hadoop [3] aktuálně zahrnuje celý ekosystém nástrojů určených pro široké spektrum použití. Jedná se zejména o HDFS, open source distribuovaný souborový systém. Další z těchto nástrojů je i velice populární univerzální rámec pro paralelní výpočty nad HDFS, Spark [4].

Účelem této práce je experimentální aplikace strojového učení na platformě Spark. Pro experiment byly vybrány doporučovací systémy, konkrétně skupina algoritmů pro kolaborativní filtrování implicitních datasetů. Vybraný algoritmus by měly být implementován s důrazem na paralelní výpočet a škálovatelnost. Pro algoritmus by měla být navržena metodika ohodnocení přesnosti jednotlivých výpočtů. Závěrem budou výsledky posouzeny z hlediska přesnosti, rychlosti a škálovatelnosti.

2. Strojové učení

Strojové učení je zatím nepřesně definovaný pojem, obecně ale můžeme říct, že strojové učení je věda o programování počítačů tak, aby se byly schopné učit z dat. Aplikace strojového učení již pronikla do našeho každodenního života, jedním z viditelnějších případů je například filtrování spamu. Při tvorbě takového systému tradičním přístupem by programátor nadefinoval sadu pravidel, která by na základě obsahu daného emailu rozhodla zda se jedná o spam nebo ne. Takový systém by ale zdaleka nebyl triviální a musel by se průběžně přizpůsobovat měnícím se podmínkám. Sada vestavěných pravidel by se stávala čím dál tím více komplexnější s narůstajícími požadavky na údržbu. Spam filtr založený na strojovém učení používá rozdílný přístup. Automaticky se naučí, která slova a fráze spam typicky obsahuje. Zřejmý problém je zde fráze 'automaticky se naučí'. Učícímu algoritmu je nutné poskytnout podmínky za kterých se bude schopný sám učit. V případě spam filtru učícímu algoritmu poskytneme databázi historických emailů. Emaily samotné nám ale nestačí, pro potřeby učícího algoritmu ještě potřebujeme označení zda je daný historický email spam nebo ne. Obecně můžeme říct, že se snažíme vytvořit abstraktní model, který nám na základě zadaného, předem nepoznaného, vstupu poskytne nějakou formu výstupu. Skupinu postupů, kdy k učení použijeme data obsahující vstup a požadovaný výstup, nazýváme učení s učitelem. Učení s učitelem je pouze jednou z kategorií v rámci technik strojového učení. Účelem této práce není poskytnout kompletní přehled všech těchto technik, tento je možné dohledat například zde [5, 6, 7]. Pro aplikaci strojového učení aktuálně existuje celá řada nástrojů. Mezi nejpoužívanější patří MATLAB, knihovna Scikit-Learn pro Python nebo RStudio.

Vytvoření funkčního modelu na základě vstupních dat je komplexní postup, který není specifický pro žádnou konkrétní platformu. Tento postup je možné rozdělit do několika základních kroků [7]:

1. Zkoumání a příprava dat
2. Výběr modelu
3. Vytvoření a vyhodnocení modelu

2.1. Zkoumání a příprava dat

Vstupní data nemusí a obvykle nebudou připravená pro strojové zpracování. Toto platí zejména pro případ implicitních dat, která mohou být v podobě různých logovacích souborů, obsahujících další nerelevantní údaje, nebo v podobě databázových tabulek. Z takových zdrojů je třeba extrahovat potřebná data do struktur vhodnějších pro další

zpracování. Dále je nutné vstupní data prozkoumat a porozumět jejich základním charakteristikám. K tomu slouží různé vizualizační a dotazovací nástroje, které nám umožní porozumět distribucím hodnot jednotlivých vlastností, korelacím mezi vlastnostmi atd. V rámci této fáze by se měly objevit nedostatky a chyby ve vstupních datech, které by měly za následek pokřivení výsledneho modelu. Dále je třeba upravit vlastnosti vstupních dat tak aby jejich hodnoty byly vhodné pro strojové zpracovaní. Typicky převodem vlastností na numerické hodnoty a úpravou škály jejich hodnot. Dva základní postupy pro úpravu škály numerických hodnot jsou normalizce a standardizace. U normalizace odečteme minimum a dělíme rozsahem. Toto nám převede hodnoty do intervalu <0, 1>. Normalizaci definujeme jako:

$$x_{norm} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

Standardizace, někdy též nazývaná z-score převede hodnoty tak aby měly nulový průměr a směrodatnou odchylku rovnou 1. Nejprve od jednotlivých hodnot odečteme jejich průměr $\overline{x}$ a tímto je vycentrujeme kolem nuly, následně tyto hodnoty dělíme směrodatnou odchylkou. Směrodatná odchylka σ odpovídá odmocnině z rozptylu hodnot, rozptyl Var definujeme jako $\frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^2$. Výsledný vzorec pro standardizaci (z-score) je tedy:

$$x_{zscore} = \frac{x_i - \overline{x}}{\sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^2}}$$

Obecně se preferuje standardizace před normalizací. Normalizace je citlivější na extrémní hodnoty, které potom tlačí majoritní část hodnot směrem k nule.

2.2. Výběr modelu

Jaký typ modelu zvolit je plně závislé na zvoleném úkolu. Pro některé úlohy bude vhodnější lineární model a pro jiné neuronová síť. A priori není žádný typ modelu nevýhodnější [7]. Typickým postupem je vytvořit model pro několik nejvhodnějších kandidátů a následně vyhodnotit ten nejúspěšnější.

2.3. Vytvoření a vyhodnocení modelu

V této fázi máme připravená data pro daný úkol a vybraný typ modelu, na jejichž základě vytvoříme či vytrénujeme cílový model. Tento model by měl být schopný předpovídat výstupní hodnoty na základě vstupních hodnot, které nebyly použity pro jeho vytvoření. To nám ovšem nestačí, potřebujeme nějak zjistit nakolik je náš model přesný. Jestli se můžeme na jeho předpovědi spolehnout například pro důležitá obchodní rozhodnutí.

Potřebujeme tedy nějakou metodiku jak změřit jeho přesnost. Dále pokud model obsahuje nějaké parametry, které ovlivňují jeho chování, je nutné nalézt jejich optimální hodnoty.

2.3.1. Ohodnocení přesnosti modelu

Reálnou přesnost modelu je možné ověřit pouze na datech, které nebyla použita k jeho vytvoření resp. k jeho učení. Obvyklá metoda je rozdělit zdrojový dataset na dvě části, na takzvanou tréninkovou a testovací sadu. Model se, jak už název napovídá, učí pouze na tréninkové sadě a jeho přesnost se následně ověří na sadě testovací. Obvyklý postup je použít na tréninkovou sadu 70-80% zdrojových dat. Chyba naměřená na testovací sadě nám ukáže jak by se mohl náš model chovat v reálném nasazení. To, že je chyba naměřená na tréninkové sadě nízká a na testovací vysoká může mít několik příčin. Nejpravděpodobnější je, že je náš model takzvaně přeučený. To znamená, že je model příliš specializovaný na tréninkovou sadu a nereflektuje realitu. Tento problém je možné eliminovat některým způsobem regularizace [7, 5]. Dalším problémem může být nesprávně zvolená tréninková a testovací sada, kdy ta testovací obsahuje aspekt, který nebyl zohledněn v rámci učení. Pro vyhodnocení přesnosti modelu použijeme naměřenou hodnotu y_i a předpovězenou hodnotu modelu pro dané měření $\hat{y}_i$. Rozdíl těchto hodnot odpovídá chybě v předpovědi pro dané měření. Průměrná hodnota přes chyby všech měření nám následně ukáže chybu celkovou. Základní metodiky postavené na tomto principu a používané pro ohodnocení modelu jsou následující:

RMSE (Root Mean Squared Error)

Z chyb jednotlivých měření spočítáme druhou mocninu a jejich sumu zprůměrujeme. Celková chyba odpovídá druhé odmocnině z této sumy:

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

MAE (Mean Absolute Error)

Celková chyba odpovídá průměru ze sumy absolutních hodnot chyb jednotlivých měření:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}\left|y_i - \hat{y}_i\right|$$

RMSE a MAE jsou si dost podobné kdy ukazují průměrnou chybu nehledě na její orientaci. To znamená, že nezáleží na tom zda je chyba měření kladná nebo záporná, v obou případech se agreguje do výsledné chyby. RMSE která se zdá v odborné literatuře preferovanější metodikou [7, 8], více penalizuje velké chyby než MAE.

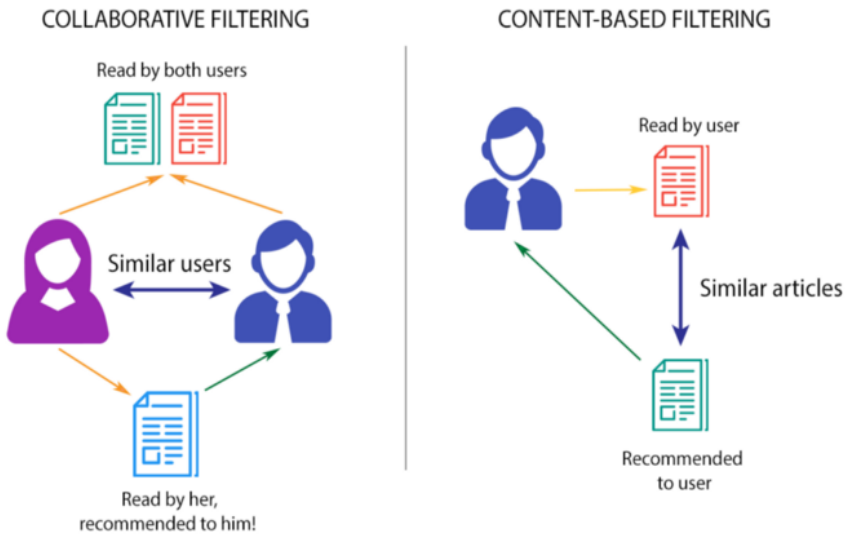
2.3.2. Hyper parametry

Ve většině případů model obsahuje dodatečné parametry ovlivňující jeho výsledné chování, takzvané hyper parametry. Optimální hodnota těchto parametrů není předem známá a je typicky závislá na podstatě vstupních dat. Tuto hodnotu tedy určujeme během tréninkové fáze. Zde se ale opakuje stejný problém jako při vyčlenění testovacích dat. Při určování hodnoty pouze na jedné sadě dat, není jisté zda je tato hodnota optimální pouze pro tréninková data nebo obecně. Jedním z postupů jak vyřešit tento problém je vyčlenit další část z treninkové sady, takzvanou validační sadu. Přesnost modelu pro různé hodnoty hyper parametrů následně ověřujeme na této sadě. Zde ale už může docházet k degradaci modelu z důvodu nedostatečnosti treninkových dat, kdy jsme se vědomě zbavili poloviny těchto dat pro dodatečné ověřování přesnosti modelu. Další často používaná metodika je takzvaná křížová validace. Tréninková sada je nejprve rozdělena na sadu k stejných částí. Následné provedeme k tréninkových fází, kdy použijeme $k-1$ částí jako trénigovou sadu a jednu část použijeme pro ověření přesnosti. Výsledná průměrná chyba ze všech k měření nám poskytne realističtější výslednou hodnotu. Zásadní nevýhodou křížové validace je její nadměrná výpočetní náročnost. Vzorec pro křížovou validaci (KV) s použitím RMSE:

$$RMSE_{KV} = \frac{1}{k}\sum_{i=1}^{k} RMSE_i$$

3. Doporučovací systémy

Doporučovací systémy jsou příkladem strojového učení s jehož aplikací se setkala naprostá většina lidí. Tyto systémy mají dnes široké spektrum použití od sociálních sítí přes streamovací služby až po online prodejce. Úspěšné nasazení těchto systémů sahá daleko do minulosti. Průkopníky v této oblasti jsou komerčně úspěšné společnosti jako Amazon, Spotify nebo Netflix. Vzhledem k dlouhé historii existuje mnoho různých, kvalitně zdokumentovaných přístupů. Doporučovací systémy jsou založeny na dvou základních strategiích. Jedná se o obsahově zaměřené systémy a systémy kolaborativního filtrování. Viz. Obr. 1.



Obr. 1. Porovnání strategií doporučovacích systémů

U obsahově zaměřených systémů je nutné nejprve vytvořit profil jednotlivých produktů na základě jejich vlastností. Například profil knihy může obsahovat autora a žánrovou příšlušnost. Na základě těchto informací může systém následně doporučovat uživateli produkty, které mají shodné nebo podobné charakteristiky. Výhodou těchto systémů je jednoduchost a přesnost. Na druhou stranu spravovat detailní informace o velkém množství jednotlivých produktů může být časově či finančně náročné. Druhá strategie je kolaborativní filtrování. Kolaborativní filtrování je založeno na předchozí interakci uživatele se systémem. Na základě této interakce se systém snaží identifikovat nové vazby mezi zákazníky a produkty. Pro systémy kolaborativního filtrování existují dva základní druhy vstupu:

Explicitní

Uživatel explicitně zadává zpětnou vazbu. Například zadá nějakou formou numerického hodnocení produktu - například počet hvězd. Nebo zadá binární hodnocení - líbí/nelíbí. Většina starších systémů byla založena na analýze

explicitního vstupu. Velkou nevýhodou těchto systémů je nutnost spolupráce uživatele. Naprostá většina z nich žádné explicitní hodnocení neposkytuje.

Implicitní

Zde se analyzuje předchozí interakce se systémem např. zakoupené knihy, puštěné filmy atd. Systémy založené na implicitní zpětné vazbě jsou široce používané ve společnostech jako Netflix nebo Amazon. Tato práce se bude dále zabývat pouze algoritmy tohoto typu.

3.1. Kolaborativní filtrování pro implicitní datasety

Implicitní data vznikají samotnou interakcí uživatele se systémem. Tato interakce může nabývat mnoho různých podob závislých na konkrétní podobě daného systému. Například pro společnosti poskytující streamovaný obsah jako Netflix nebo Spotify to může být shlédnutí daného obsahu. Ale také zda byl tento obsah shlédnutý celý, zda a kdy bylo shlédnutí přerušeno popřípadě počet shlédnutí. Pro internetové knihkupectví jako Amazon toto budou primárně zakoupené tituly. Tyto data se ukládají a následně vyhodnocují tak aby se dosáhlo co možná nejpřesnějších doporučení. Implicitní data mají tedy tendenci být komplexnější než data explicitní což sebou přináší i řadu problémů, které je nutné v daném algoritmu zohlednit:

1. Pouze pozitivní zpětná vazba. Pokud evidujeme pouze položky, které si uživatel vybral, je obtížné rozlišit ty, které opravdu preferuje. Například pokud uživatel neshlédl daný film, může to být způsobeno tím že o něm dosud neví ale také tím, že ho nemá rád.
2. Implicitní datasety obsahují velký podíl šumu. Pokud pasivně sledujeme chování uživatele, můžeme pouze hádat jejich preference a pravé motivy. Například i přesto, že uživatel zhlédl některý film, nemusel se mu doopravdy líbit [9].
3. Studený start. Nově přidané produkty do systému mají nulový počet interakcí. Systém by měl být schopný doporučovat i takové produkty.

Pro tvorbu doporučení na základě kolaborativního filtrování existují dva základní postupy, nacházení nejbližších sousedů a faktorizace matic.

3.1.1. Nacházení nejbližších sousedů

Systémy založené na nacházení nejbližších sousedů se snaží na základě vypočtené vzdálenosti doporučit nejblíže položené položky. Tyto systémy jsou uživatelsky nebo produktově orientované. V prvním případě se vypočítává vzdálenost mezi uživateli ve

druhém mezi produkty. Produktově založený přístup je obvykle preferován před uživatelským, podstata produktů se obvykle příliš nemění takže model může být předpočítán a následně použit bez konstantního přepočítávání.

3.1.2. Faktorizace matic

V systémech založených na faktorizaci matic je interakce uživatelů s produkty definována jako matice, ve které řádky obsahují všechny uživatele a sloupce všechny produkty v katalogu. Jednotlivé buňky pak obsahují ohodnocení produktu uživateli. Tato matice je typicky velmi řídká, většina možných kombinací uživatelů s produkty v datasetu chybí. Algoritmus faktorizuje matici na uživatelskou a produktovou faktorovou matici s omezeným počtem rozměrů. Tyto rozměry reprezentují takzvané latentní faktory, které odpovídají například žánrům v daném datasetu. Úkolem algoritmů je odhadnout chybějící hodnoty v matici, proto jsou také někdy nazývány jako algoritmy pro doplňování matic. Narozdíl od zdrojové matice kde většina hodnot chybí je matice, která vznikne jako produkt faktorových matic hustá. Tato matice už obsahuje odhadnuté nenulové hodnoty pro kombinace uživatelů s produkty, které ve zdrojové matici chyběly [10].

Uživatel	Produkt	Hodnocení
Eliška	John Wick: 3	1
Eliška	Parazit	5
Jan	Bad Boys: Navždy	4
Jan	Terminátor: T. osud	3
Jakub	John Wick: 3	5
Jakub	Parazit	1
Tereza	Terminátor: T. osud	1
Tereza	Parazit	1

Obr. 2. Přehled hodnocení

Na Obr. 2 jsou zobrazeny hodnocení pro čtyři uživatele, kde každý hodnotil dvakrát z celkového počtu čtyř filmů. Z přehledu je patrné, že Eliška má ráda komedie ale nemá ráda akční filmy. Jan má rád komedie i akční filmy. Jakub má rád pouze akční filmy a Tereza nemá ráda nic.

	John Wick: 3	Bad Boys: Navždy	Terminátor: T. osud	Parazit
Eliska	1			5
Jan		4	3	
Jakub	5			1
Tereza			1	1

Obr. 3. Zdrojová matice

Na Obr. 3 jsou tyto hodnocení zapsány v maticovém tvaru, kde je řádek pro každého

uživatele a sloupec pro každý film. Tato matice má vyplněných 50% hodnot, v reálném použití by tato matice obsahovala stovky tisíc řádků a sloupců a byla by daleko řidší (obvykle méně než 1%).

Matice s uživatelskými faktory

	John Wick: 3	Bad Boys: Navždy	Terminátor: T. osud	Parazit
Akce	2.1	1.5	1.9	0.1
Komedie	0.1	1.2	0.1	2.2

Matice s produktovými faktory

	Akce	Komedie
Eliška	0.1	2.3
Jan	1.2	1.8
Jakub	2.1	0.2
Tereza	0.1	0.2

Doplněná matice

	John Wick: 3	Bad Boys: Navždy	Terminátor: T. osud	Parazit
Eliska	0.44	2.91	0.42	5.07
Jan	2.7	3.96	2.46	4.08
Jakub	4.43	3.39	4.01	0.65
Tereza	0.23	0.39	0.21	0.45

Obr. 4. Faktorové matice

Na Obr. 4 jsou zobrazeny výsledné faktorové matice pro dva faktory. V tomto případě byl každému faktoru přiřazen konkrétní význam, jeden pro komedie a druhý pro akční filmy. Při reálném použití jsou konkrétní významy faktorů skryté a z hlediska algoritmu nejsou důležité. Matice s uživatelskými faktory obsahuje pro každého uživatele hodnotu nakolik je v daných faktorech zastoupen. Stejně tak i matice s produktovými faktory obsahuje pro každý produkt hodnotu stejných faktorů. Výsledkem maticového součinu těchto dvou faktorových matic je doplněná matice obsahující predikce pro všechny uživatele a produkty.

Singular value decomposition (SVD) je vhodný algoritmus, který odebere šum ze zdrojové matice R a zároveň přesně odhadne chybějící hodnoty [11, 12]. Další algoritmus, popularizovaný během slavné Netflix výzvy, je Alternating Square Roots (ALS) [13].

3.2. K nejbližších sousedů

Algoritmus spočítá vzdálenost mezi cílovým produktem a všemi ostatními produkty. Následně ohodnotí tuto vzdálenost a nalezne k nejbližších produktů jako výsledné doporučení. Definujme matici $R_{m,n}$ kde m určuje počet uživatelů a n počet produktů v katalogu. Každý produkt můžeme následně interpretovat jako vektor s m rozměry, tj. jeden rozměr pro každého uživatele. Pro výpočet vzdálenosti mezi dvěma vektory existuje více postupů. Standartní norm-2 neboli euklidovská vzdálenost není pro výpočet vzdálenosti vektorů s velkým počtem rozměrů doporučená [14]. Doporučená je takzvaná kosinová podobnost.

3.3. Alternating Least Square (ALS)

ALS z anglického názvu Alternating Least Squares, volně přeloženo jako střídavá metoda nejmenších čtverců, je algoritmus pro faktorizaci matic viz. 3.1.2. Definujme matici R_{ui} kde u určuje počet uživatelů a i počet položek v katalogu. Elementy $r_{ui} \in \mathbb{R}$ matice R definují ohodnocení uživatele u položky i. Předpokládáme, že matice R je rozsáhlá ale zároveň velice řídká, typicky méně než 1 % polí má přiřazenou hodnotu.

$$R_{m,\,n} = \begin{pmatrix} r_{1,\,1} & r_{1,\,2} & \cdots & r_{1,\,n} \\ r_{2,\,1} & r_{2,\,2} & \cdots & r_{2,\,n} \\ \vdots & \vdots & \ddots & \vdots \\ r_{m,\,1} & r_{m,\,2} & \cdots & r_{m,\,n} \end{pmatrix}$$

Matici R rozložíme na 2 matice X_{uk} a Y_{ik} tak, že $XY^T \approx R$. Tyto matice obsahující řádky pro každého uživatele resp. produkt se nazývají faktorové. Vektory v těchto faktorových maticích mají typicky nízký počet rozměrů k. Každý tento rozměr odpovídá latentní vlastnosti modelu a jejich hodnoty odpovídají tomu nakolik jsou uživatelé a produkty spjaty s touto skrytou vlastností. Na rozdíl od R je výsledná matice XY^T hustá, doplněné hodnoty obsahují odhad ohodnocení mezi uživateli a položkami. Soustava rovnic daná $XY^T = R$ nemá pravděpodobně řešení. Důvodem je příliš mnoho rovnic u pro X resp. i pro Y a pouze k neznámých. Řešením je použít metodu nejmenších čtverců, kdy hledáme řešení s minimální chybou:

$$\| R - XY^T \|_2$$

Na začátku nastavíme hodnoty matice X na náhodná malá čísla a spočítáme Y. Následně střídavě počítáme X a Y do té doby, než se stabilizuje hodnota $\| R - XY^T \|_2$.

Autoři [9] rozšířili popsaný algoritmus tak aby se výpočet dal efektivně distribuovat. Hodnoty jednotlivých hodnocení r_{ui} byly nahrazeny odvozenou binární proměnnou p_{ui} tak, že platí:

$$p_{ui} = \begin{cases} 0 & r_{ui} = 0 \\ 1 & r_{ui} > 0 \end{cases} \qquad (1)$$

Pokud uživatel dosud produkt neužil, proměnná nabývá hodnotu 0 v opačném případě hodnotu 1. Důvěra v tuto hodnotu je vysoce proměnlivá. Jak již bylo řečeno pro nulové hodnoty to může znamenat, že uživatel o produktu zatím neví. Pro každého uživatele a produkt tedy platí, že součin příslušných faktorových vektorů x_u a y_i odpovídá této binární proměnné:

$$p_{ui} = x_u^T y_i$$

Čím je počet konzumací vyšší tím více narůstá důvěra v hodnotu $p_{ui} = 1$. Autoři článku

vyjádřili důvěru v hodnotu p_{ui} následovně:

$$c_{ui} = 1 + \alpha r_{ui}$$

Existuje tedy minimální důvěra 1 pro zatím nezkonzumované produkty. Hodnota proměnné lineárně narůstá s počtem užití produktu. Poměr nárustu je dán konstantou α, přesná hodnota tohoto parametru je specifická pro konkrétní dataset a bude odvozena v rámci křížové validace viz. 2.3.2. Pro každého uživatele máme k dispozici i hodnot c, jednu pro každý produkt. Tyto hodnoty převedeme do diagonální matice C_u s rozměry $i x i$. V rovnici bude tato matice figurovat jako váhy:

$$C_u Y x_u = C_u p_u$$

Rovnici násobíme transpozicí položkové faktorové matice. Tímto rovnici převedeme do normální vážené formy:

$$Y^T C_u Y x_u = Y^T C_u p_u$$

Výsledná rovnice, po zahrnutí regularizačního parametru, která vyjadřuje výpočet faktoru pro jediného uživatele odpovídá:

$$x_u = (Y^T C_u Y + \lambda I)^{-1} Y^T C_u p_u$$

Problém při výpočtu je výraz $Y^T C_u Y$. Zde by se muselo opakovaně provádět pro každého uživatele násobení produktové faktorové matice a diagonální matice s preferencemi daného uživatele. Faktorová matice je v rámci distribuovaného výpočtu typicky rozdělená na více výpočetních uzlů a její násobení vyžaduje přeskládání bloků dat v rámci výpočetního clustru. Autoři článku vyřešili tento problém převedením výrazu $Y^T C_u Y$ na výraz:

$$Y^T Y + Y^T (C_u - I) Y$$

Zde nejprve předpočítáme $Y^T Y$ a následně přičítáme pro každého uživatele $Y^T (C_u - I) Y$. Z výpočtu vypadnou všechny produkty, které nemají pro daného uživatele žádné hodnocení, ty mají v diagonální matici C_u hodnotu 1 ($c_{ui} = 1 + \alpha r_{ui}$) a odečtením jednotkové matice získáme matici kde jsou v příslušných řádcích samé nuly. Analogický případ je i výraz $Y^T C_u p_u$, kde také neřešíme chybějící hodnocení pro které obsahuje vektor p_u nulové hodnoty. V rámci distribuovaného výpočtu můžeme držet při výpočtu hodnocení daného uživatele na jednom výpočetním uzlu a provádět výpočet cílových faktorů lokálně. Analogicky rovnice, která vyjadřuje výpočet faktorů pro jediný produkt odpovídá:

$$y_i = (X^T C_i X + \lambda I)^{-1} X^T C_i p_i$$

4. Apache Spark

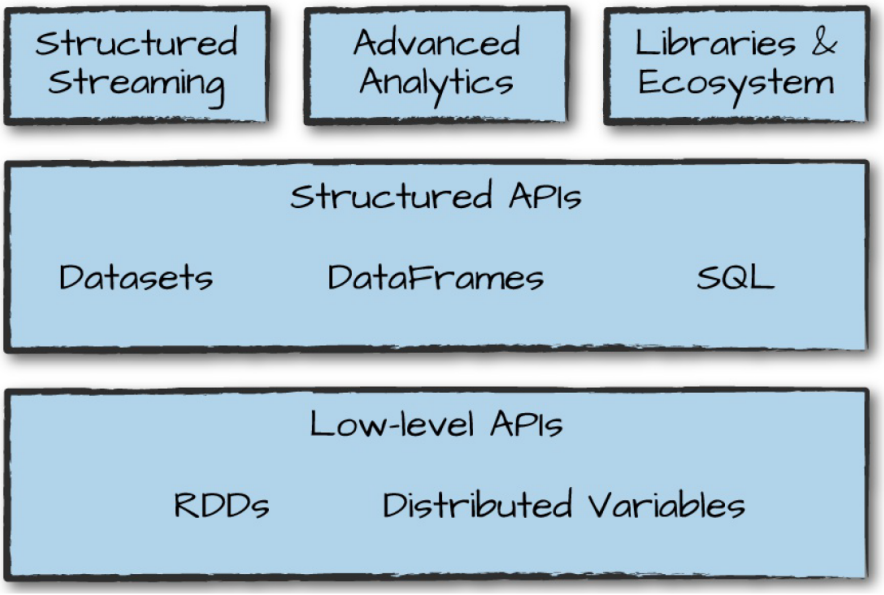
Apache Spark [4] je unifikovaný výpočetní systém pro paralelní zpracování dat na počítačových clustrech. Spark nabízí bohaté API pro datové operace jako je filtrování, spojování (join), seskupování a agregace. Toto API je dostupné pro řadu populárních programovacích jazyků jako jsou Java, Python, C# a R. Spark je aktuálně nejaktivněji vyvíjeným open source projektem v této oblasti s více než tisícem aktivních vývojářů.

Filozofie Sparku je rozdílná od předcházejících platforem pro zpracovávání velkých objemů dat jako je například Hadoop, ten zahrnuje jak výpočetní systém (MapReduce) tak i úložiště dat (HDFS). Obě tyto části jsou spolu úzce provázané a je obtížné provozovat jednu část bez té druhé. Ačkoliv je možné Spark bez problémů provozovat nad HDFS není na tomto úložném systému nijak závislý a je možné ho používat i spolu s jinými zdroji dat. Jednou z motivací tohoto přístupu je, že data které je potřeba analyzovat, jsou typicky již uložena v rozdílných formátech v řadě různých úložných systémů. Přesouvání těchto dat pro analytické účely může být zejména při vyšších objemech nepraktické. Spark je proto postaven tak aby byl přístup k datům co nejvíce transparentní.

Klíčovou vlastností Sparku je, že samotné provedení sekvence datových operací je nejprve optimalizováno. Tato optimalizace zajistí co možná nejefektivnější řetězec zpracování s využitím operační paměti pro mezivýsledky. To je velkou výhodou v rychlosti výpočtu oproti MapReduce kde se v každém kroku výsledky perzistují a je nutné je v následujícím kroku řetězce znovu načíst.

V diagramu Obr. 5 je zobrazena základní struktura Sparku. Základem je nízko-úrovňové API pro práci s datasety, RDD (Resilient Distributed Dataset) volně přeloženo jako pružný distribuovaný dataset. Dalším patrem je strukturované API, přidané v druhé generaci Sparku. Na RDD a strukturovaném API je postavena řada specifických modulů, které jsou součástí standartní distribuce Sparku. Mezi tyto moduly patří:

- MLlib pro podporu strojové učení.
- Structured Spark Streaming pro podporu datových proudů.
- GraphX pro podporu analýzy grafů.



Obr. 5. Struktura Sparku [8]

4.1. Spark API

Pro vytváření vlastních aplikací existují ve Sparku dvě programové rozhraní (API). Jedná se základní RDD API a jeho nadstavbu, strukturované API viz. Obr. 5.

4.1.1. RDD

RDD (Resilient Distributed Dataset) je základní konstrukt Spark API. Představuje neměnnou kolekci záznamů, rozdělenou do částí, které mohou být nezávisle paralelně zpracovány. Tyto části, takzvané partitions, jsou typicky rozložené na více uzlů v rámci výpočetního clustru. Jednotlivé záznamy jsou klasické Java, Python nebo Scala objekty. Při operaci nad daným RDD je každé partititon přiřazen právě jeden výkonný proces. RDD API dále disponuje řadou operací pro manipulaci s daty, které se dělí do dvou základních skupin:

Transformace

Ve Sparku jsou zdrojová data typicky neměnná. Pokud chceme provést úpravy nad RDD, definujeme jednu nebo více takzvaných transformací. Tyto transformace instruují Spark jak má změnit zdrojová data. Důležitý je fakt, že samotná transformace pouze definuje jak se mají data změnit, transformace samotná se ale instantně neiniciuje. Komplexní operace nad daty většinou zahrnují celý řetězec atomických transformací. Spark počká na akci která si vyžádá transformovaný RDD, z řetězce transformací následně automaticky vyhodnotí optimální plán provedení a teprve potom transformuje zdrojový RDD. Transformace se dále dělí na dva základní typy, na úzké a široké. Úzké transformace se aplikují na jednu

partition nezávisle na ostatních. Výsledkem je právě jedna cílová partition, typicky uložená na stejném počítači v rámci clustru. Úzké transformace se tedy nechají jednoduše paralelizovat. Při široké transformaci je transformovaná partition závislá na ostatních partition, typicky uložených na ostatních počítačích. Je tedy nutné provést takzvané přeskládání (shuffle) a v rámci clustrové sítě přenést potřebná data.

Akce

Akce jsou metody RDD API, které spouští samotný výpočet, při kterém se aplikují definované transformace na zdrojový RDD.

V následujícím jednoduchém příkladu spočítáme počet lichých a sudých čísel v datasetu:

```
1    val rdd = spark.sparkContext.parallelize(Seq(1, 2, 3, 4, 5, 6, 7, 8, 9))
2      .map(v => (v%2, v))
3      .groupByKey()
4      .mapValues(v=>v.size)
```

Pro tento příklad vyhodnotí Spark následující plán zpracování, kde se čísla na konci řádků odkazují na řádky ve uvedeném zdrojovém kódu:

```
(2) MapPartitionsRDD[3] at mapValues at 4 ④
 |  ShuffledRDD[2] at groupByKey at 3 ③
 +-(2) MapPartitionsRDD[1] at map at 2 ②
    |  ParallelCollectionRDD[0] at parallelize at 1 ①
```

① Na řádku 1 nejprve vytvoříme zdrojový dataset. Ten v tomto případě inicializujeme přímo v kódu na hodnoty v intervalu <1, 9> ačkoliv typickým použitím je načíst zdrojová data z perzistentního úložiště.

② Na řádku 2. použijeme RDD metodu **map**, tato metoda má jako argument funkci, která trasnsformuje jednu z hodnot zdrojového RDD. V plánu zpracování je jako výsledek tohoto kroku MapPartitionsRDD, to znamená, že se transformace provede lokálně na úrovni jednotlivých partition. Jako výsledek transformace je RDD dvojic kde je první hodnota zbytek po celočíselném dělení dvěmi. Pokud jsou v RDD dvojice, Spark automaticky interpretuje první hodnotu jako klíč.

③ Na řádku 3. pomocí RDD metody **groupByKey** seskupíme hodnoty se stejným klíčem. Výsledkem tedy bude RDD dvojice s příslušným klíčem, kde je jako hodnota kolekce všech hodnot, které mají ve zdrojovém RDD daný klíč. V plánu zpracování je jako výsledek tohoto kroku ShuffledRDD, to znamená, že je třeba provést přeskládání v rámci výpočetního clustru a seskupit hodnoty se stejným klíčem.

④ Na řádku č. pomocí RDD metody **mapValues** transformujeme pouze hodnoty, klíč ve dvojici zůstává nezměněn. Zde je jako hodnota kolekce čísel odpovídající danému zbytku po celočíselném dělení. V rámci této operace tuto kolekci převedeme na celé číslo odpovídající počtu členů této kolekce. V plánu zpracovaní je jako výsledek tohoto kroku MapPartitionsRDD, transformace se tedy také provede lokálně na úrovni jednotlivých partition.

Výsledkem je 5 lichých hodnot (klíč 1) a 4 sudé hodnoty (klíč 0):

```
(1,5)
(0,4)
```

4.1.2. Strukturované API

Do druhé generace přidali autoři Sparku takzvané strukturované API. Základním konstruktem tohoto API je DataFrame. DataFrame reprezentuje tabulku složenou z řádků a sloupců, tato tabulka se ale, stejně jako RDD, může rozkládat na mnoha počítačích v rámci výpočetního clustru. DataFrame také jako RDD používá pro operace s daty akce a transformace. Strukturované API by mělo být preferovaným způsobem používání Sparku, je uživatelsky přívětivější, vysoce optimalizované a odstiňuje uživatele od náročnějších detailů. Nicméně pokud je třeba mít přímou kontrolu nad tím, jak jsou data fyzicky uložena v rámci clustru, je nutné použít RDD API. Stejný koncept jako DataFrame, omezený na jediný počítač, používají API jako Python Pandas nebo R DataFrames. Toto usnadňuje používání Sparku uživatelům se znalostmi těchto nástrojů, například jako doplňující nástroj pro práci s velkými objemy dat.

4.2. Spark Aplikace

Spark aplikace se skládá z řídícího procesu a sady výkonných procesů. Řídící proces je zodpovědný za analýzu a distribuci jednotlivých úkolů výkonným procesům. Výkonné procesy jsou zodpovědné za zpracování úkolů, které jim přiřadí řídící proces a za reportování stavu tohoto úkolu zpět řídícímu procesu. Clustr počítačů, které Spark využívá pro vykonání dané aplikace je řízený clustr manažerem. Manažerský proces řídí přístup k prostředkům clustru a přiřazuje jeho zdroje jednotlivým aplikacím. V rámci jednoho clustru tedy může být spuštěno více Spark aplikací zároveň. Spark není závislý na jednom konkrétním clustr manažeru, v době psaní práce podporoval Hadoop YARN, Apache Mesos a také vlastní manažer, omezený pouze na jediný počítač. Vývojář Spark aplikace je odstíněný od toho na jaké úkoly bude aplikace rozdělena nebo na kterých počítačích v rámci clustru budou tyto úkoly vykonány. O vše se transparentně postará

Spark spolu s použitým clustr manažerem. Průběh Spark aplikace lze rozdělit do několika fází:

Inicializace

Aplikace samotná je typicky java knihovna obsahující spustitelnou třídu. Způsobů jak spustit aplikaci je více, základní z nich je použít utilitu spark-submit z příkazové řádky. V této chvíli spouštíme proces na klientském počítači, tento proces kontaktuje příslušný clustr manažer a zažádá si o prostředky pro řídící proces. Clustr manažer umístí řídící proces samotný na některý z dostupných počítačů clustru. Proces spuštěný z příkazové řádky na klientském počítači je ukončen a aplikace je spustěna. Průběžný stav aplikace je možné sledovat pomocí dodatečných dotazů na příšlušný clustr manažer. V této chvíli je tedy řídící proces umístěn na některém počítači v clustru, řídící proces následně aktivuje uživatelský kód (spustitelná třída v rámci odeslaného JARu). Tento kód musí obsahovat inicializaci třídy **SparkSession**. SparkSession následně komunikuje s clustr manažerem a vyžádá si spuštění jednotlivých výkonných procesů. Po inicializaci a startu těchto výkonných procesů odešle clustr manažer relevantní informace o jejich umístění zpět řídícímu procesu.

Vykonání

V této chvíli máme inicializovaný Spark clustr složený z jednoho řídícího procesu a sady výkonných procesů. Řádící proces přiděluje jednotlivé úkoly výkonným procesům. Výkonné procesy komunikují mezi sebou, vykonávají přidělené úkoly a vyměňují si potřebná data. Po dokončení přiděleného úkolu reportují výsledný status zpět řídícímu procesu.

Dokončení

Po dokončení běhu aplikace, je řídící proces ukončen s výsledným stavem. Clustr manažer následně ukončí všechny přidělené výkonné procesy. V této chvíli je možné zjistit konečný status aplikace dotazem na příslušný clustr manažer.

4.3. Spark a strojové učení

Základním kamenem pro pokročilé analytické výpočty na platformě Apache Spark je jeho standartní knihovna MLlib. Tato knihovna obsahuje širokou podporu postupů strojového učení a jeho jednotlivých fází viz. 2. Dále knihovna nabízí konzistentní API s praktickou podporou pro řetězení částí výpočtů. Vývojář se soustředí pouze na implementaci konkrétního algoritmu a následně ho propojí do tréninkového řetězce s dostupnými komponentami pro škálování hodnot, ladění hyper parametrů,

vyhodnocování přesnosti vytvořeného modelu atd. [8]. Knihovna obsahuje tyto základní prvky:

Model

Model je komponenta která dokáže nějakým způsobem transformovat vstupní data. Obsahuje jedinou metodu **transform** se vstupním parametrem typu DataFrame viz. 4.1.2. Model dále obsahuje podporu pro definici konfiguračních parametrů, například jméno sloupce pod kterým model přidá predikce ke vstupním datům. Další praktická vlastnost modelu je podpora prezistence. Vytvoření modelu může být výpočetně náročné, je tedy potřeba mít možnost takový model uložit a později znovu načíst pro opakované použití.

Estimator

Estimator obsahuje jedinou metodu **fit** se vstupním parametrem typu DataFrame viz. 4.1.2 a návratovým typem Model. Estimator provede analýzu vstupních data a na jejich základě inicializuje příslušný Model.

Typickým použitím dvojice Estimator a Model je že, Estimator nejprve na základě tréninkové sady inicializuje Model, který následně přidá do datasetu další sloupec, obsahující predikce pro nová data.

4.4. Spark a Hadoop

Apache Hadoop je open source implementací technik popsaných v materiálech vydaných společností Google [1, 2]. Jádro tohoto projektu obsahuje tři základní komponenty. Distribuovaný souborový systém HDFS, systém na správu prostředků počítačových clustrů YARN a výpočetní systém MapReduce. Od svého vzniku se k těmto nástrojům přidala řada dalších nástrojů a knihoven adresujících další oblasti užití. Jedním z těchto nástrojů je i Apache Spark, aktuálně jeden z nejpopulárnějších výpočetních nástrojů v celém Hadoop ekosystému. Při nasazení v rámci Hadoopu je Spark závislý na clustr manažeru YARN, dále primárně pracuje s daty uloženými v HDFS ačkoliv může využívat i data z jiných zdrojů.

4.4.1. HDFS

Hadoop Distributed File System (HDFS) je distribuovaný souborový systém pro Hadoop, optimalizovaný pro ukládání velkých objemů dat [3]. Při ukládání dat, HDFS rozdělí soubor do bloků konstantní, nakonfigurované délky, standartně 128 MB. Následně uloží repliky každého bloku na nakonfigurovaný počet počítačů v clustru, standartně 3. Replikace bloků dat se provádí jednak z důvodu zálohy, ale také pro umožnění

paralelních výpočtů nad daty na více počítačích v clustru. Každý z počítačů v clustru má nainstalovanou HDFS službu DataNode, která je zodpovědná za ukládání dat na disk a jejich následné načítání. DataNode služba zná jenom bloky, které má uložené a jejich identifikátory, ale neudržuje informace o tom, které bloky patří ke kterým uloženým souborům. Tyto informace jsou udržovány koordinační službou NameNode, která udržuje informace o mapování souboru do bloků, dále také udržuje metadata o souborech jako jsou například přístupová práva.

HDFS má vysokou průchodnost. Pokud chce klient uložit soubor, nejprve kontaktuje NameNode a obdrží list DataNode služeb pro každý blok. Samotný zápis následně probíhá mezi klientem a DataNode. Po zapsání bloku na první DataNode tento automaticky replikuje tento blok na další počítače a neblokuje klienta v dalším zápisu. Stejně tak při načítání souboru klient komunikuje přímo s DataNode.

HDFS je tolerantní k chybám. Pokud dojde k havárii disku, počítače nebo dokonce celého racku, NameNode úkoluje jednotlivé DataNode služby, které drží repliky ztracených dat, aby rozkopírovaly ztracené bloky na další počítače v clustru. Tím se zajistí nastavený replikační faktor pro každý blok [15].

4.4.2. YARN

Yet Another Resource Negotiator (YARN) je centralizovaný clustr manažer pro Hadoop. Každý z počítačů v clustru má nainstalovanou YARN službu NodeManager, která komunikuje s řídící službou ResourceManager. Každý NodeManager reportuje službě ResourceManager kolik zdrojů v podobě operační paměti a procesorových jader je dostupných na daném počítači. Zdroje na jednotlivých počítačích jsou rozdělené do logických celků takzvaných kontejnerů, kde má každý kontejner přidělené určité množství zdrojů (například 4 procesorová jádra a 8GB RAM). NodeManager je zodpovědný za vytváření a monitorovani kontejnerů na daném počítači a jejich ukončení pokud překročí přidělené zdroje. Aplikace které potřebují provést výpočet v rámci clustru nejprve kontaktují službu ResourceManager a zažádají si o jeden kontejner na kterém spustí vlastní koordinační proces nazývaný ApplicationMaster (AM). ApplicationMaster si následně zažádá ResourceManager o potřebné kontejnery na kterých provede výpočet samotný [15].

Jak již bylo řečeno, jednotlivé soubory jsou rozdělené v HDFS do bloků. Při zpracování těchto souborů Sparkem se tyto bloky mapují přímo na Spark partition. Každý z těchto bloků/partitions zpracovává právě jeden YARN kontejner obsahující výkonný proces Sparku. Při zpracování jsou tyto procesy umístěné lokálně k datům aby se co nejvíce eliminovala síťová komunikace.

5. Implementace vybraných algoritmů v řešené oblasti

5.1. Výběr vhodných technologií

Před samotnou implementací je třeba zvolit vhodné technologie. Jedná se primárně o volbu programovacího jazyka a příslušných vývojových nástrojů. Pro vývoj v prostředí Apache Spark existuje řada technologií, které se svojí funkcionalitou částečně nebo úplně překrývají. Jejich výběru je proto třeba věnovat zvýšenou pozornost.

5.1.1. Výběr programovacího jazyka

Spark podporuje několik různých jazyků a to jsou Scala, Java, Python a R. Scala je nativní jazyk Sparku a jeho přidružených knihoven. Jedná se o takzvaný JMV jazyk, který potřebuje ke svému běhu nainstalovanou Javu, konkrétně její běhové prostředí. Zdrojový kód v jazyce Scala se při kompilaci převede do Java byte kódu. To znamená, že se aplikace distribuuje ve formě java archivů takzvaných jarů. K vývoji Spark aplikací můžeme tedy použít také přímo Javu samotnou, aplikace napsané v Javě mohou používat knihovny napsané ve Scale a naopak. Zejména podpora funkcionálního programování a ad-hoc struktur činí ze Scaly nejvhodnější jazyk pro Spark. Python je velice populární jazyk určený pro rychlé prototypování. V posledních letech se stal víceméně standardem pro experimentování v oblasti umělé inteligence a strojového učení. Python má k dispozici mnoho specializovaných knihoven jako jsou Sciki-learn nebo TensorFlow. Python není nativním jazykem Sparku, místo toho používá specializovanou knihovnu Py4J pro interakci s JVM. Jazyk R je naopak velice populární v rámci statistické komunity. Pro práci s tímto jazykem existují dokonce dvě knihovny, SparkR distribuovaná spolu se Sparkem a dále sparklyr, vyvíjená týmem nástroje RStudio.

Pro vývoj aplikace byl vyhodnocen jako nejvhodnější jazyk Scala. Při vývoji se budeme moci nechat inspirovat praktikami použitými při vývoji Sparku samotného, zejména jeho knihoven určených pro strojové učení. Dále budeme používat Python v rámci experimentální fáze projektu k optimalizaci hyper parametrů.

5.1.2. Výběr nástroje pro automatizaci buildů aplikace

Nástroj pro automatizaci buildů zaštiťuje kompletní cyklus sestavení aplikace. Toto zahrnuje zejména kompilaci zdrojového kódu a spuštění kompletní sady projektových testů. Na závěr cyklu nástroj vytvoří konečnou verzi aplikace a uloží ji pod příslušným identifikátorem verze do sdíleného repositáře. Pro aplikace vyvíjené v jazyce Scala

existuje specifický nástroj **sbt**, dále je možné použít některý z nástrojů určený přímo pro jazyk Java spolu s příslušným rozšířením pro Scalu. Těchto nástrojů je celá řada, zřejmě nejpoužívanější jsou Apache Maven a Gradle. Pro náš projekt byl vybrán nástroj Maven, zejména proto, že je použit i v rámci vývoje Sparku samotného.

5.1.3. Výběr IDE

Vedle nástroje pro automatizaci buildů budeme dále potřebovat integrované vývojové prostředí (IDE). Takové prostředí usnadňuje psaní zdrojového kódu, spouštění jednotlivých unit testů atd. Na toto prostředí existuje několik základních požadavků, které musí splňovat pro vývoj Spark aplikací:

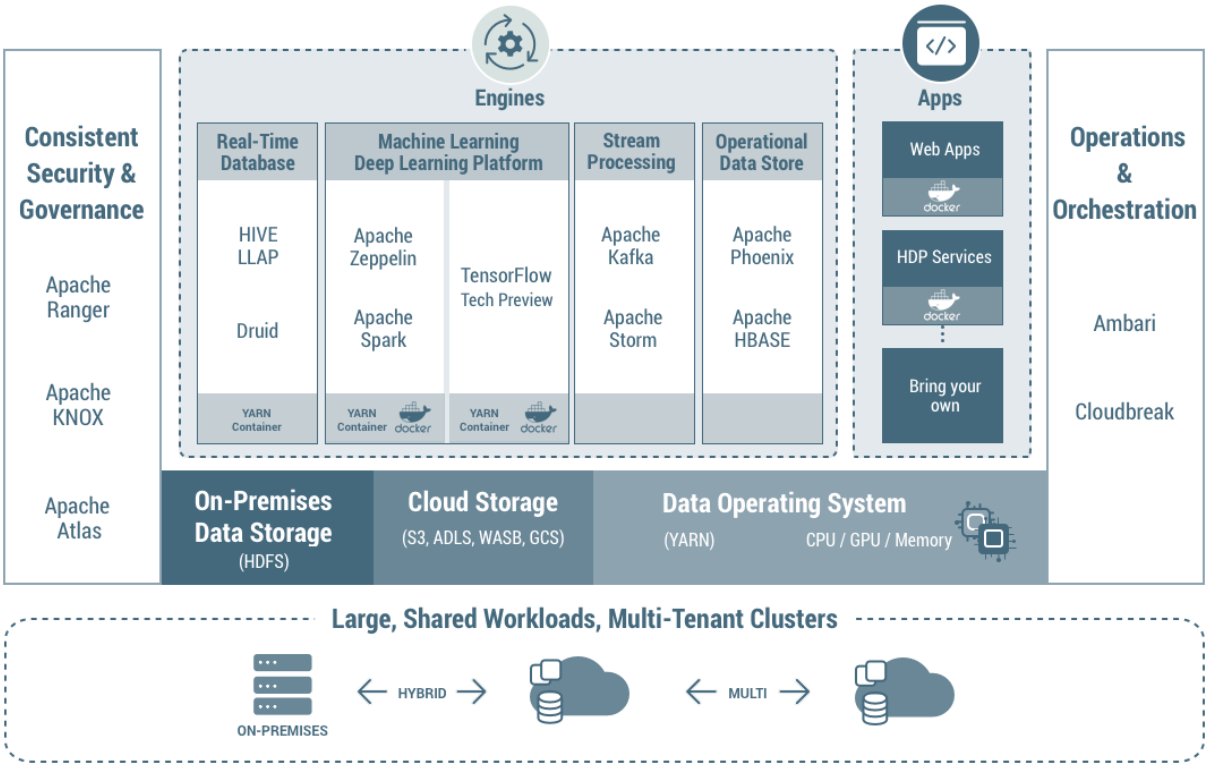
1. Vestavěná podpora Apache Maven. IDE by mělo být schopné automaticky naimportovat informace z Maven deskriptoru aplikace (pom.xml). Primárně se jedná o sadu knihoven, závislostí projektu, která by měla být nadefinovaná na jediném místě. Vzhledem ke své popularitě, je podpora Apache Maven zahrnuta ve většině současných Java IDE.
2. Podpora Scaly. Scala je jen jeden z mnoha JVM jazyků a je mnohem méně používaná než Java samotná. Kvalitní podpora toho jazyka není úplně bežná.
3. Podpora Scala testovacího rámce. Ačkoliv je Apache Maven skvělý nástroj a při sestavování aplikace spustí kompletní sadu unit testů. Při implementaci samotné potřebujeme jemnější kontrolu nad tím, který test spouštíme. Zvolené IDE by mělo tuto možnost podporovat a umožnit nám spouštět testy jednotlivě dle potřeby. V našem projektu použijeme stejný testovací rámec jako je použitý ve Sparku samotném. To nám dovolí použít stejné konstrukty, bázové třídy unit testů, usnadňující automatické vytváření a rušení testovacích instancí Sparku.

Do užšího výběru postoupili dva kandidáti. První je Scala IDE, vývojové prostředí postavené na populární platformě Eclipse. Druhé je neméně populární vývojové prostředí IntelliJ IDEA. Scala IDE ve verzi 4.7.0 splňovalo první dva z uvedených požadavků, nicméně se i přes urputnou snahu nepodařila uspokojivě vyřešit podpora unit testů. IntelliJ IDEA spolu se Scala pluginem splnila všechny tři požadavky na výbornou. Pro vývoj aplikace tedy použijeme nástroj IntelliJ IDEA.

5.2. Instalace Apache Spark

Pro vývoj a testování vyvíjené aplikace je třeba nainstalovat některou z distribucí Sparku. Nejjednodušší na instalaci je používat Spark v takzvaném stadalone módu. Tato distribuce používá vlastní clustr manažer, určený pro použití na jediném počítači. V

tomto módu se typicky používá lokální souborový systém a výpočet je distribuován výhradně na jádra jediného procesoru. Vzhledem k tomu, že se tato práce zabývá distribuovanými výpočty, byl by tento mód pro vývoj aplikace příliš zjednodušující. Vhodnější je zvolit některou z forem clustrových distribucí. Zde se nabízí použít Spark v rámci některé Hadoop distribuce, toto je také nejběžnější forma nasazení Sparku v rámci organizací zabývajících se zpracováním velkých objemů dat. Největší společnosti nabízející distribuce Hadoopu jsou Cloudera a Hortnonworks. Ačkoliv při produkčním použití jsou jejich produkty licencovány, obě dvě společnosti nabízí volně dostupné repozitáře pro hlavní distribuce linuxu. Dále také nabízejí nástroje pro usnadnění instalace jednotlivých služeb. Typicky takový nástroj automaticky nainstaluje kompletní Hadoop clustr dle zvolené topologie. Další variantou je použít některou z variant Hadoopu, které jsou nabízeny jako služba v cloudových systémech. Největší hráči Microsoft a Amazon aktuálně nabízejí Hadooop resp. Spark v rámci jejich portfolia služeb. U Microsoftu se služba jmenuje Azure HDInsight a u Amazonu Amazon EMR. Obrovskou výhodou cloudového řešení je, že se obejdeme bez nutnosti instalovat jednotlivé komponenty na jednotlivé počítače v clustru. Celý Hadoop ekosystém zahrnuje velké množství na sobě závislých služeb, které je nutné nainstalovat podle zvolené topologie na jednotlivé počítače. Dále je nutné jednotlivé služby nakonfigurovat tak aby mohly vzájemně spolupracovat. I v případě, že chceme používat Spark samotný, je třeba na celém clustru nainstalovat distribuovaný souborový systém HDFS a clustr manažer YARN. Toto vše udělá cloud za nás a celý systém je možný používat v řádech desítek minut. Nevýhodou je ovšem v tomto případě cena. I když zvolíme minimální variantu, nedostaneme se pod desítky Euro měsíčně což není pro akademické použití příliš vhodné. Po zvážení všech důvodů se pro testování nasazení aplikace použil virtuální clustr s distribucí Hadoopu od společnosti Hortonworks. Tato společnost nazývá svůj produkt HDP (Hortonworks Data Platform).



Obr. 6. Hortonworks Data Platform

Na Obr. 6 je zobrazen ekosystém služeb HDP. Tento ekosystém zahrnuje mimo základních prvků Hadoopu jako jsou Spark, HDFS, YARN mnoho dalších služeb jako například NoSQL databáze HBASE, framework pro podporu neuronových sítí TensorFlow atd. Pro testování aplikace ale nainstalujeme pouze bezpodmínečně nutné služby. Pro usnadnění správy clustru je v HDP použita aplikace Apache Ambari viz. Obr. 6 sekce Operations & Orchestration. Tato aplikace má takzvanou master-slave architekturu. Na všech počítačích v clustru je nainstalována klientská služba. Ta má za úkol instalovat vybrané balíky na daný počítač, dále tato služba reportuje systémové informace pro účely monitoringu. Na jeden vybraný uzel v klastru je nainstalována serverová (master) služba aplikace. Ta dle potřeby instruuje jednotlivé připojené klienty a sbírá data pro monitoring, tato služba dále disponuje REST a webovým rozhraním. Přes webové rozhraní je možné jednoduše spravovat jednotlivé služby v rámci celého clustru. Přes REST rozhraní je možné nainstalovat kompletně celý clustr dle zadané definice. Tyto definice jsou v JSON formátu, takzvané Ambari Blueprints.

5.2.1. Instalace HDP

Jako virtualizační platforma pro hostitelský počítač byl vybrán VirtualBox od společnosti Oracle. Tato platforma je volně dostupná a podporuje většinu operačních systémů. Pro usnadnění práce s virtuálními operačními systémy je inicializace řízena utilitou Vagrant. Vagrant umožnuje jednoduše definovat jaké virtuální stroje je potřeba vytvořit, jaké mají

mít paramtery, sdílené adresáře atd. Samotná práce s VirtualBoxem se tím výrazně zjednodušuje. Filozofie za tímto nástrojem je taková, že je tyto prostředí potřeba vytvářet opakovaně a plně automaticky, tak aby vývojáři nebyli nuceni neustále opakovat stejné konfigurační úkony. Úkony jako instalace databází, webových a aplikačních serverů jsou místo toho definovány ve skriptech a Vagrant je při vytváření virtuálních strojů automaticky aplikuje. Vagrant podporuje nejznámější konfigurační rámce jako jsou Puppet nebo Chief, podporuje ale také obyčejné shell skripty. Jako operační systém byla vybrána distribuce linuxu Centos 7, pro kterou společnost Hortonworks spravuje volně dostupné Yum repozitáře. Pro instalalaci testovacího prostředí byly použity následující repozitáře a verze:

Tab. 1. HDP Yum repozitáře pro Centos 7

Jméno	**Verze**	**URL**
Ambari	2.7.4.0	http://public-repo-1.hortonworks.com/ambari/centos7/2.x/updates/2.7.4.0/ambari.repo
HDP	3.1.4.0	http://public-repo-1.hortonworks.com/HDP/centos7/3.x/updates/3.1.4.0/hdp.repo
HDP-UTILS	1.1.0.22	http://public-repo-1.hortonworks.com/HDP-UTILS-1.1.0.22/repos/centos7

Pro zrychlení instalace a omezení množství stahovaných dat při opakované instalaci je výhodné použít lokální repozitáře. Jednotlivé Yum repositáře stáhneme v komprimované podobě, tyto lokální repozitáře následně uložíme na hostitelském stroji a přes sdílené adresáře učiníme dostupné všem virtualním strojům. Dále je nutné stáhnout některé knihovny, které jsou sice zadarmo ale vyžadují registraci u společnosti Oracle. Jedná se o distribuci Java SDK a dále java konektor pro databázi MySQL. Konečná adresářová struktura v souborovém systém hostitelského počítače by tedy měla vypadat následovně:

```
/yum
    /repo
        /ambari ①
        /hdp
            /HDP          ②
            /HDP-UTILS    ③
        /common
         |- jdk-8u231-linux-x64.tar.gz  ④
         |- mysql-connector-java.jar    ⑤
```

① Obsah Yum repozitáře Ambari. Viz Tab. 1

② Obsah Yum repozitáře HDP. Viz Tab. 1

③ Obsah Yum repozitáře HDP-UTILS. Viz Tab. 1

④ Distribuce Oracle Java 8 SDK (build 231).

⑤ Java konektor pro databázi MySQL.

Clustr je složen ze čtyř virtuálních počítačů viz. Tab. 2. Počítač c7201 obsahuje všechny řídící procesy pro HDFS a Yarn, dále také obsahuje utility pro volání služeb clustru. V produkčních clustrech jsou typicky tyto dvě role rozděleny mezi různé počítače. Dále clustr obsahuje tři naprosto identické počítače c7202, c7203, c7204. Tyto počítače budou zpracovávat jednotlivé úlohy distribuovaných výpočtů Sparku. Každý z těchto uzlů zároveň slouží jako datový uzel pro HDFS. Počet pracovních uzlů (3) odpovídá výchozímu replikačnímu faktoru HDFS. Reálně to tedy znamená že jsou všechna data uložena na všech třech pracovních uzlech.

Tab. 2. Topologie klastru

Jméno	IP adresa	RAM	Počet jader	Role
c7201	192.168.72.101	6GB	3	Master, Edge
c7202	192.168.72.102	6GB	2	Worker
c7203	192.168.72.103	6GB	2	Worker
c7204	192.168.72.104	6GB	2	Worker

Soubor Vagrantfile je uložen v podadresáři vagrant v kořenovém adresáři projektu. V tomto souboru jsou nadefinovány jednotlivé virtuální stroje a jejich parametry.

```
1     config.vm.box = "centos/7"
```

Atribut config.vm.box obsahuje definici boxu z kterého budou vytvořeny všechny nadefinované virtuální stroje. Tyto boxy jsou uloženy ve veřejném repositáři nástroje Vagrant. Při vytvoření prvního virtuálního počítače se tento box automaticky stáhne z příslušného repozitáře a následně se používá lokálně uložena kopie.

```
1     config.vm.synced_folder '../.',       '/opt/dev',           type: 'nfs'
2     config.vm.synced_folder '/yum/repo',  '/var/www/html',      type: 'nfs'
```

Atributy config.vm.synced_folder obsahují definice sdílených adresářů. Dle této definice se adresáře ze souborového systému automaticky připojí na souborový systém virtuálních strojů. První definice připojí kořenový adresář projektu do adresáře

/opt/dev virtuálních strojů. Tato definice požívá relativní cestu vzhledem k Vagrantfile souboru. Druhá definice připojí lokalní Yum repozitař do adresáře /var/www/html virtuálních strojů. Soubory uložené v tomto adresáři jsou následně dostupné přes HTTP protokol a příslušné URL mohou být použité k definici jednotlivých Yum repozitářů na virtuálních počítačích.

```
1     config.vm.provider :virtualbox do |vb|
2         vb.customize ["modifyvm", :id, "--memory", 6120]
3         vb.customize ["modifyvm", :id, "--cpus", 2]
4     end
```

Sekce config.vm.provider obsahuje konfigurace specifické pro virtualizační platformu. V tomto případě instruujeme VirtualBox kolik operační paměti a jader procesoru má dát k dispozici jednotlivým virtuálním počítačům.

```
1     config.vm.define :c7202 do |c7202|
2         c7202.vm.hostname = "c7202.barenode.org"
3         c7202.vm.network :private_network, ip: "192.168.72.102"
4         c7202.vm.synced_folder '/data/2', '/hadoop/hdfs',        type: 'nfs'
5     end
```

Sekce config.vm.define obsahuje definici virtuálního stroje c7202 viz Tab. 2. Pro tento virtuální počítač definujeme pouze specifickou IP adresu a jméno hosta. Všechny ostatní parametry jsou nadefinovány globálně, společně pro všechny virtuálním počítače. Inicializaci clustru spustíme z příkazové řádky relativně k deskriptoru Vagrantfile:

```
> vagrant up
```

5.3. Struktura projektu

V kořenovém adresáři projektu vytvoříme základní adresářovou strukturu tak jak je vyžadována nástrojem Apache Maven. Tato struktura je následující:

```
src/
  main/
    scala/ <zdrojové soubory aplikace>
  test/
    scala/ <zdrojové soubory testu aplikace>
```

Projekty řízené nástrojem Apache Maven dále vyžadují základní deskriptor projektu. Tento deskriptor je uložený v kořenovém adresáři projektu pod fixním jménem pom.xml. Deskriptor obsahuje několik základních částí, které ovlivňují průběh sestavení

a výslednou podobu aplikace.

```
<groupId>org.barenode</groupId>
<artifactId>ml-on-spark</artifactId>
<version>1.0</version>
<type>jar</type>
<name>Strojove uceni na paltforme Spark</name>
```

Každý deskriptor projektu musí obsahovat základní informace o aplikaci. Jedná se o identifikátor skupiny, identifikátor aplikace a verzi aplikace. Tyto informace řídí jak bude cílová aplikace pojmenována a jakého bude typu. V případě naší aplikace bude tedy výsledkem standartní java Java archiv (jar) s přiřazeným jménem ml-on-spark-1.0.jar.

```
<properties>
  <spark.version>2.3.1</spark.version>
  <java.version>1.8</java.version>
  <scala.version>2.11.12</scala.version>
</properties>
```

V sekci properties definujeme proměnné použité dále v deskriptoru. V našem případě obsahují tyto proměnné zejména verze jednotlivých závislostí projektu. Verzím je třeba věnovat zvýšenou pozornost, je žádoucí aby tyto verze přesně odpovídaly verzím, které jsou nainstalovány v našem testovacím Spark clustru. Pokud by byly použity nekompatibilní verze, mohlo by docházet k chybám při běhu programu. Tedy ačkoliv byla v době implementace aplikace posední vydaná verze Sparku 2.4.4 v projektu jsme použili verzi 2.3.1, která odpovídá verzi Sparku z Hadoop distribuce našeho testovacího clustru. Použijeme také stejné verze Javy a Scaly, které jsou použity v této verzi. Pro tuto verzi Sparku byla použita Java 8 a Scala 2.11.12.

```
<dependencies>
  <dependency>
    <groupId>org.scala-lang</groupId>
    <artifactId>scala-library</artifactId>
    <version>${scala.version}</version>    ①
  </dependency>
  <dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-core_${scala.binary.version}</artifactId>
    <version>${spark.version}</version>    ②
  </dependency>
  ...
<dependencies>
```

① Verze Scaly je nadefinována v sekci properties

② Verze Sparku je nadefinována v sekci properties

Sekce dependencies obsahuje takzvané závislosti projektu. Reálně se jedná o standartní Java archivy (jary) ze kterých budeme importovat třídy do našeho zdrojového kódu. Apache Maven pro nás tyto knihovny automaticky stáhne a nalinkuje včetně tranzitivních závislostí těchto knihoven. Seznam závislostí je poměrně rozsáhlý, uvádíme zde tedy pouze zkrácenou verzi s knihovnami pro Spark a Scalu. Dále deskriptor obsahuje v sekci plugins definici rozšířeni pro Scalu. Ve výchozím stavu podporuje Apache Maven pouze zdrojový kód v jazyce Java, podporu Scaly je nutné expilictně definovat. Rozšíření scala-maven-plugin zavádí podporu kompilaci Scala zdrojového kódu. Rozšíření scalatest-maven-plugin aktivuje podporu unit testů pro Scalu. Tato definice obsahuje dodatečné atributy potřebné pro vytváření lokálních instancí Sparku v rámci jednotlivých testů.

Korektní sestavení aplikace ověříme z příkazové radky příkazem:

```
> mvn clean install
```

Do konzole se nám následně vypíší informace o uspěšném zkompletovaní aplikace:

```
[INFO] BUILD SUCCESS
[INFO] ------------------------
[INFO] Total time: 6.714 s
[INFO] Finished at: 2019-10-15T07:49:24+02:00
[INFO] Final Memory: 29M/421M
```

5.4. Alternating Least Square (ALS)

Algoritmus ALS je rozdělen na dvě základní komponenty **Model** a **Estimator** viz. 4.3. Estimator na základě datasetu s hodnoceními spočítá uživatelské a produktové faktory a inicializuje s nimi Model. Model následně na základě těchto faktorů bude poskytovat doporučení.

5.4.1. Model

Model je inicializován s referencemi na uživatelské a produktové faktorové datasety. Oba tyto datasety mají analogickou strukturu, kde řádky jsou dvojice klíč, hodnota. Klíč obsahuje celočíselný identifikátor uživatele respektive produktu. Jako hodnota je příslušný faktorový vektor v podobě pole reálných numerických hodnot. Vektorový součin uživatelského a produktového faktoru predikuje hodnocení produktu uživatelem. Maticový součin obou datasetů je kompletní matice obsahující predikce hodnocení všech produktů pro všechny uživatele. Model musí povinně implementovat metodu **transform**

se vstupním parametrem typu Dataset viz. 4.3. Pro interní manipulaci s tímto datasetem Model vyžaduje následující parametry:

userCol

Jméno sloupce ve vstupním datasetu, který obsahuje numerický identifikátor uživatele.

itemCol

Jméno sloupce ve vstupním datasetu, který obsahuje numerický identifikátor produktu.

predictionCol

Jméno sloupce pod kterým Model přidá predikce hodnocení do vstupního datasetu.

Metoda transform provede spojení mezi vstupním datasetem a faktorovými datasety, na úrovni řádku provede vektorový součin faktorů a výsledek uloží do sloupce s predikcemi:

```
1       dataset
2       .join(userFactors, dataset($(userCol)) === userFactors("id"), "left")
3       .join(itemFactors, dataset($(itemCol)) === itemFactors("id"), "left")
4       .select(
5         dataset("*"),
6         predict(userFactors("features"), itemFactors("features")).as($(
  predictionCol)))
```

Pokud tedy použijeme testovací dataset jako vstup do modelu, budeme mít v každém řádku reálné hodnocení produktu uživatelem a zároveň i predikci hodnocení. Na základě těchto dvou hodnot bude možné následně hodnotit přesnost našeho modelu.

Model dále obsahuje metody pro doporučení konkrétního počtu produktů uživatelům a naopak.

5.4.2. Estimator

Hyper paramtery mají vliv na výsledné predikce hodnocení, jejich optimální hodnoty jsou závislé na datasetu, pro který vyvářime model. Přesné hodnoty odvodíme až při experimentování, typicky pomocí křížové validace viz. 2.3.1. Algoritmus bude podporovat následující vstupní hyper parametry:

rank

Rank určuje dimenzi faktorových vektorů. Tato dimenze je vždy stejná jak pro uživatele tak i pro produkty. Pokud je velikost faktorových vektorů příliš nízká, model bude také příliš zjednodušený a nebude podávat optimální predikce. Na druhou stranu pokud bude tento počet příliš velký, může dojít k takzvanému přeučení, kdy model bude příliš spjatý s trénovacími daty. Výchozí hodnota je 10.

alpha

Alpha nastavuje poměr nárůstu důvěry v hodnocení, viz viz. 3.3. Výchozí hodnota je 0.1.

regParam

Regularizační parametr zabraňuje přeučení modelu. Hodnota 0 má za následek, že se při výpočtu regularizace neaplikuje. Výchozí hodnota je 0.1.

Tréninkové parametry nemají přímo vliv na nastavení výpočtu ale můžeme pomocí nich kontrolovat jak budou data distribuována v rámci výpočetního clustru. Data jsou před výpočtem samotným rozdělena do bloků. Tyto bloky jsou vlastně partition a jsou následně zpracovávány paralelně. Zvolený počet bloků spolu s kapacitou výpočetního clustru mají zásadní dopad na to, jak dlouho zabere vytvoření modelu. Dle [8] je optimalní velikost bloku mezi jedním až pěti miliony hodnocení. Náš algoritmus bude podporovat následující trénovací parametry:

numUserBlocks

Počet bloků do kterých budou rozděleni uživatelé, výchozí hodnota je 10.

numItemBlocks

Počet bloků do kterých budou rozdělené produkty, výchozí hodnota je 10.

maxIter

Počet iterací při výpočtu. V každé iteraci se provede výpočet uživatelských a produktových faktorů. Po určitém počtu iterací se již výsledné faktory nemění a nemá smysl ve výpočtu pokračovat. Optimální počet iterací zjistíme například pomocí RMSE. Výchozí hodnota je 10.

Další, pomocné parametry identifikují sloupce požadované pro výpočet v rámci vstupního datasetu:

userCol

Jméno sloupce ve vstupním datasetu, který obsahuje numerický identifikátor uživatele.

itemCol

Jméno sloupce ve vstupním datasetu, který obsahuje numerický identifikátor produktu.

rating

Jméno sloupce ve vstupním datasetu, který obsahuje hodnocení. Hodnocení mohou nabývat reálných, nezáporných hodnot.

Struktura bloků

Pro práci s bloky zvolíme rozdílnou terminologii než rozdělení na uživatele a produkty. Vzhledem k povaze algoritmu, kdy se střídavě provádí výpočet faktorů zvlášť pro uživatele a zvlášť pro produkty, použijeme pro oba stejný algoritmus a vytvoříme pro ně analogické struktury. Jednou budou jako cíl výpočtu uživatelské faktory, které použijí jako zdroj pro výpočet faktory produktové. Podruhé to bude naopak. Dále tedy budeme pro vysvětlení algoritmu používat termíny zdroj a cíl.

Pro vyhodnocení cílového bloku vytvoříme jednoduchou hašovací funkci. Tato rozdělí hodnocení rovnoměrně dle zvoleného celkového počtu bloků. Každému uživateli resp. produktu přidělí cílový blok na základě jejich celočíselného identifikátoru. Výsledné id bloku bude také celé číslo v řadě začínající nulou, bude tedy také zároveň jeho indexem v sadě příslušných bloků.

Obr. 7. Diagram bloku hodnoceni

V diagramu Obr. 7 je zobrazena struktura bloku hodnocení. Pro každý blok vzniknou tři nezávislé struktury:

Blok

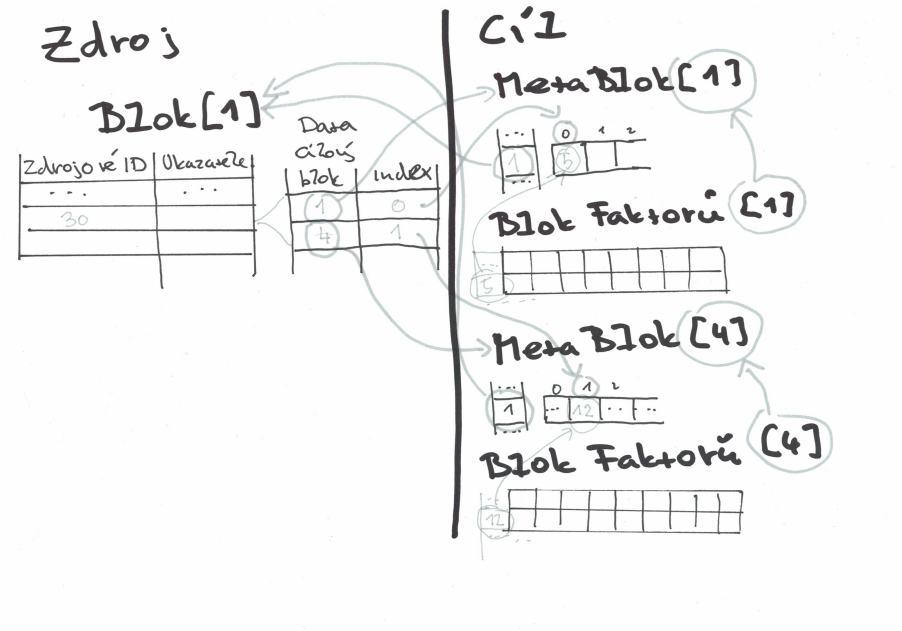
Základní struktura, obsahující příslušná hodnocení. Tyto bloky se inicializují na začátku výpočtu na základě vstupních dat. Data v nich zůstávají konstantní přes všechny iterace výpočtu. Výpočet se provádí lokálně na počítači kde jsou tyto data uložena, není tedy třeba tyto bloky v průběhu výpočtu opakovaně přenášet po síti v rámci výpočetního clustru. V těchto blocích je jeden z identifikátorů zvolen jak zdrojový a druhý jako cílový. Hodnocení jsou seřazena dle zdrojového identifikátoru (vzestupně). Data v bloku jsou zkomprimována do takzvané CSC (compressed sparse column) struktury. Vzhledem k tomu, že jeden zdrojový identifikátor může být v datech zastoupený několikrát, vybereme nejprve unikátní zdrojové identifikátory viz. (Zdrojové ID) Obr. 7. K těmto identifikátorům přiřadíme ukazatel do bloku hodnocení. Tento ukazatel určuje ke kterým hodnocením příslušné zdrojové ID patří. Cílový identifikátor nahradíme v datech identifikátorem cílového bloku a lokálním indexem. Lokální index určuje index seřazených unikátních cílových id pro daný zdrojový a cílový blok. Tyto identifikátory provazují zdrojový blok a cílový meta blok.

Blok faktorů

Pro každé unikátní zdrojové id v rámci propojeného bloku bude v tomto bloku právě jeden faktorový vektor. Dimenze těchto vektorů je dána zvoleným rankem, typicky se jedná o relativně nízkou hodnotu, viz. výchozí hodnota 10. Tyto faktory se přepočítávají v rámci každé iterace a jsou následně použity jako zdroj pro výpočet cílových faktorů. Tyto bloky je tedy nutné v průběhu výpočtu distribuovat v rámci clustru. Blok faktorů inicializujeme na začátku výpočtu a jednotlivým faktorům přiřadíme náhodná reálná čísla.

Meta blok

Na základě dat v bloku vytvoříme takzvaný metablok. Metabloky mezi sebou provazují zdrojové a cílové bloky. Reálně se jedná o dvourozměrné pole kde je záznam pro každý cílový blok. V těchto záznamech jsou zdrojová id, pro které existuje hodnocení v daném zdrojovém a cílovém bloku.



Obr. 8. Diagram propojení bloků

V diagramu Obr. 8 je zobrazené propojené mezi zdrojovými a cílovými bloky. Pokud chceme spočítat zdrojový faktor pro jedno zdrojové id (30) musíme do lineárního systému zahrnout všechny cílové faktory (cílové id 5, 12) pro které existuje hodnocení pro dané zdrojové a cílové id. Tyto mohou být typicky v rozdílných cílových blocích (1, 4) uložených na rozdílných počítačích v rámci clustru.

Rozdělení hodnocení do bloků

Jako top level vstup do algoritmu použijeme strukturované API jako preferovaný způsob od druhé generace Sparku. Dataset musí obsahovat sloupce identifikované vstupními

parametry userCol, itemCol a rating. V rámci algoritmu samotného ale pro práci s hodnoceními použijeme RDD API. Toto API nižší úrovně nám, narozdíl od doporučovaného, strukturovaného API, dává možnost řídit rozložení dat v rámci clustru. Jako vstup do části algoritmu, který nám rozdělí hodnocení do příslušných bloků, nejprve převedeme vstupní dataset do RDD uspořádaných trojic:

```
1     val ratings = dataset
2       .select(col($(userCol)), col($(itemCol)), col($(ratingCol))).rdd
3       .map { row =>
4         (row.getInt(0), row.getInt(1), row.getFloat(2))}
```

Tato trojice se skládá ze zdrojového id (srcId), cílového id (dstId) a hodnocení. Nejprve vytvoříme uživatelské bloky, tj. jako zdrojové id použijeme id uživatele a jako cílové id použijeme id produktu. Nejprve pro každou trojici určíme příslušný zdrojový a cílový blok. Toto je nutné z důvodu určení vazeb mezi všemi bloky. Pro každou vstupní trojici tedy emitujeme klíč, dvojici hodnot obsahující identifikátory zdrojového a cílového bloku:

```
1     val blocks = rdd.map { case (srcId, dstId, rating) =>
2       val srcBlockId = srcPartitioner.getPartition(srcId)
3       val dstBlockId = dstPartitioner.getPartition(dstId)
4       ((srcBlockId, dstBlockId), (srcId, dstId, rating))
```

Dále provedeme seskupovací operaci. Tato nám seskupí všechny hodnocení pro danou kombinaci zdrojového a cílového bloku. Výstupem je tedy příslušný klíč a kolekce odpovídajících hodnocení. Metoda RDD mapValues, nám dovolí transformovat pouze hodnoty (druhou položku ze vstupní dvojice), výstupem je tedy klíč ze vstupu spolu s novou hodnotou. Zde je vstupní hodnotou kolekce hodnocení kterou převedeme do pomocné struktury pro snazší manipulaci s daty bloku. Tato struktura je v takzvané COO (Coordinate Format) formě. Obsahuje tři pole, kde hodnoty se stejným indexem určují trojici zdrojového resp. cílového id a příslušného hodnocení:

```
1     }.groupByKey().mapValues{ratings =>
2       val builder = new RatingBlockBuilder
3       ratings.foreach{case(srcId, dstId, rating) =>
4         builder.add(srcId, dstId, rating)}
5       builder.build()
```

Z pole cílových id pro daný zdrojový a cílový blok vybereme unikátní hodnoty, tyto seřadíme a přiřadíme jim index určující jejich pořadí. Hodnoty cílových id namapujeme na odpovidající index. Tyto indexy slouží k propojení cílových a zdrojových bloků viz. Obr. 8. Vzhledem k tomu, že vytváříme cílové bloky, transformujeme klíč pouze na

zdrojové id:

```
    }.map{case((srcBlockId, dstBlockId), RatingBlock(srcIds, dstIds, ratings))=>
      val dstIdToLocalIndex = dstIds.toSet.toSeq.sorted.zipWithIndex.toMap
      val dstLocalIndices = dstIds.map(dstIdToLocalIndex.apply)
      (srcBlockId, (dstBlockId, srcIds, dstIds, dstLocalIndices, ratings))
```

V této chvíli máme tedy RDD kde je klíčem id zdrojového bloku a hodnotou blok hodnocení. Pro jedno id zdrojového bloku může být v datasetu více záznamů, vzhledem k tomu, že příslušný blok obsahuje hodnocení pouze pro jediný cílový blok. Dateset tedy seskupíme dle klíče, id zdrojového bloku. Jednotlivé bloky hodnocení spojíme do jediného bloku, obsahujícího všechny hodnocení pro daný zdrojový blok. Tento výsledný blok převedeme do cílové formy tak jak je zobrazena v diagramu Obr. 7. Dále na základě tohoto bloku vytvoříme příslušný meta blok a blok faktorů, inicializovaný na náhodné hodnoty.

Tento postup následně opakujeme, pouze s tím rozdílem, že je jako zdrojové id použito id produktu a jako cílové id, id uživatele. V této chvíli máme tedy vstupní data rozdělena do příslušných bloků, připravených pro výpočet samotný.

Výpočet faktorů

Po rozdělení hodnocení do bloků je možné provést výpočet samotných faktorů. Tento výpočet probíhá iterativně, celkový počet iterací je určen vstupním parametrem maxIter. V každé iteraci nejprve proběhne výpočet uživatelských a následně produktových faktorů, pro oba výpočty se použije identický algoritmus vysvětlený viz. 3.3. Do lineárního systému nejprve postupně vložíme všechny vektory z matice zdrojových faktorů Y, ten tedy bude následně obsahovat matici odpovídající výrazu $Y^T Y$. Tato operace vyžaduje přeskládání v rámci výpočetního clustru, kdy je do lineárního systému nutné zahrnout všechny zdrojové faktorové bloky. Výslednou čtvercovou matici, obsahující pouze $r x r$ elementů, kde r odpovídá zadanému ranku, následně použijeme pro výpočet cílových faktorů. Tento výpočet již provádíme paralelně na úrovni jednotlivých cílových bloků, kde v cyklu spočítáme jednotlivé cílové faktory. Zde je nutné do lineárního systému přidat všechny zdrojové faktory a příslušné hodnocení, které existují pro daný cílový faktor. Řešení lineárního systému odpovídá hledanému cílovému faktoru.

5.4.3. Lineární systém

V rámci algoritmu, jak už sám název napovídá, je třeba opakovaně řešit lineární systém, kdy vektorový součin uživatelských a produktových faktorů odpovídá příslušnému hodnocení. Pro výpočet faktorového vektoru x pro jednoho uživatele tedy platí, že součin

matice produktových faktorů A a vektoru x odpovídá vektoru hodnocení daného uživatele pro všechny produkty:

$$Ax = b$$

Počet rovnic v lineárním systému odpovídá počtu uživatelů resp. počtu produktů a počet neznámých odpovídá zvolenému ranku faktoru. Vzhledem k tomu, že je počet rovnic mnohokrát vyšší než počet neznámých nebude mít tento lineární systém typicky jediné přesné řešení. Pro výpočet tedy použijeme lineární regresi, kdy se snažíme nalézt řešení s nejmenší možnou chybou. Lineární systém budeme udržovat v takzvaném normálním stavu:

$$A^T Ax = A^T b$$

Konkrétně si budeme udržovat dvě komponenty. První z nich odpovídající výrazu $A^T A$ je čtvercová a symetrická matice. Vzhledem k tomu, že jako matici A použijeme střídavě uživatelské resp. produktové faktory bude mít tato matice relativně malé rozměry $r x r$ kde r odpovídá zvolenému ranku. Druhá komponenta bude vektor odpovídající výrazu $A^T b$, kde vektor b odpovídá vektoru hodnocení, tato komponenta bude mít dimenzi r.

Do lineárního systému bude možné přidávat komponenty jednotlivě, kdy budou jako argumenty faktorový vektor a a skalár b_i odpovídající konkrétnímu hodnocení. Následně přičteme k první komponentě výraz $a^T a$ a ke druhé vyraz $a^T b_i$.

Spark interně používá pro výpočty lineární algebry knihovnu JBLAS. Tato knihovna je pouze tenký Java klient nad nativními knihovnami LAPACK a BLAS. Tyto knihovny, vytvořené ve Fortranu 77, jsou optimalizované na vysoký výkon a použité v nástrojích jako jsou MATLAB nebo RStudio. Pro řešení lineárního systému použijeme rutinu DPPSV, založenou na Choleského faktorizaci.

6. Vytvoření modelu

6.1. Získání dat

Pro test algoritmu byl vybrán implicitní dataset volně dostupný přes API společnosti Last.fm. Tato společnost provozuje hudební web, založený ve Velké Británii v roce 2002. Pomocí systému hudebních doporučení s názvem „Audioscrobbler“ vytváří Last.fm podrobný profil hudebního vkusu každého uživatele zaznamenáním podrobností o skladbách, které uživatel poslouchá, a to buď z Internetové rozhlasové stanice, počítače uživatele nebo pomocí přenosného hudebního zařízení. Tyto informace jsou přenášeny do databáze Last.fm buď prostřednictvím samotného hudebního přehrávače (mimo jiné včetně Spotify, Deezer, Tidal a MusicBee) nebo prostřednictvím plug-in nainstalovaného do hudebního přehrávače uživatele. Data se poté zobrazí na stránce profilu uživatele.

6.2. Příprava dat

Dataset je distribuován v archivu obsahujícím několik souborů. Relevantní data jsou uložena v jediném textovém souboru usersha1-artmbid-artname-plays.tsv. Data nejprve zkopírujeme z lokálního souborového systému do HDFS. V dalším kroku namapujeme soubor do datasetu, pomocí metody **show** zobrazíme první dva řádky :

```
1 rawData = spark.read.format('text').load("usersha1-artmbid-artname-plays.tsv")
2 rawData.show(2, False)
```

Dataset obsahuje jediný sloupec nazvaný **value**, obsah buněk odpovídá jednotlivým řádkům ze zdrojového souboru. Každý řádek obsahuje čtyři údaje. Identifikátor uživatele, identifikátor interpreta, název interpreta a počet přehrání některé skladby tohoto interpreta:

```
+----------------------------------------------------------------------------------------------------+
|value                                                                                               |
+----------------------------------------------------------------------------------------------------+
|00000c289a1829a808ac09c00daf10bc3c4e223b   3bd73256-3905-4f3a-97e2-8b341527f805    betty blowtorch 2137  |
|00000c289a1829a808ac09c00daf10bc3c4e223b   f2fb0ff0-5679-42ec-a55c-15109ce6e320    die Ärzte   1099      |
+----------------------------------------------------------------------------------------------------+
```

Dalším krokem je převést jednotlivé řádky do struktury vhodnější pro další výpočty. Jednou z možností by bylo nadefinovat vlastní funkci, která by rozdělila vstupní řádek dle tabulátoru. Jednodušší postup ale je interpretovat vstupní soubor jako CSV formát s tabulátorem jako separátorem buněk. Spark obsahuje zabudovanou podporu pro tento formát spolu s řadou řídících voleb. V našem případě použijeme volbu **mode** s hodnotou

DROPMALFORMED tak, že se při zpracování ignorují řádky, které nemají vyhovující formát. Dále aktivujeme volbu **inferSchema** aby se Spark pokusil přiřadit vhodné datové typy jednotlivým sloupcům. Nakonec přiřadíme sloupcům vhodná jména pomocí metody **withColumnRenamed**:

```
df = spark.read.format('csv')\
    .option("header", "false")\
    .option("sep", "\t")\
    .option("mode", "DROPMALFORMED")\
    .option("inferSchema", "true")\
    .load("usersha1-artmbid-artname-plays.tsv")\
    .withColumnRenamed("_c0", "userHash")\
    .withColumnRenamed("_c1", "artistMBID")\
    .withColumnRenamed("_c2", "artistName")\
    .withColumnRenamed("_c3", "listenCount")
```

Pokud si nyní necháme vypsat schéma výsledného datasetu spolu s prvními třemi řádky:

```
df.printSchema()
df.show(3)
```

Dataset obsahuje čtyři sloupce s tím, že sloupec **listenCount** má přiřazený numerický datový typ:

```
root
 |-- userHash: string (nullable = true)
 |-- artistMBID: string (nullable = true)
 |-- artistName: string (nullable = true)
 |-- listenCount: double (nullable = true)

+--------------------+--------------------+----------------+-----------+
|            userHash|          artistMBID|      artistName|listenCount|
+--------------------+--------------------+----------------+-----------+
|00000c289a1829a80...|3bd73256-3905-4f3...|  betty blowtorch|     2137.0|
|00000c289a1829a80...|f2fb0ff0-5679-42e...|        die Ärzte|     1099.0|
|00000c289a1829a80...|b3ae82c2-e60b-455...|melissa etheridge|      897.0|
+--------------------+--------------------+----------------+-----------+
only showing top 3 rows
```

Dalším problémem jsou datové typy sloupců userHash a artistName. ALS algoritmus vyžaduje aby identifikátory uživatelů resp. produktů byla celá čísla. Tyto jsou ale ve zdrojovém datasetu identifikovány řetězci, které se nedají automaticky převést na numerické hodnoty. Ze zdrojového datasetu tedy vytvoříme nový dataset obsahující pouze unikátní identifikátory uživatelů. Každému uživateli přiřadíme pomocí funkce **zipWithIndex** unikátní celočíselný identifikátor odpovídající indexu příslušného řádku v datasetu:

```
users = df.select(df.userHash).distinct()
users = users.rdd.zipWithIndex().toDF()
users = users.select(\
    users._1.userHash.alias("userHash"),\
    users._2.alias("userId").cast("integer"))
```

Analogickou operaci jako s uživateli opakujeme i s interprety a vytvoříme další dataset artists obsahující unikátní interprety spolu s jejich celočíselnými identifikátory. Tyto dva datasety propojíme se zdrojovým datasetem a každému hodnocení přiřadíme celočíselné identifikátory uživatele a intepreta:

```
df = df\
    .join(users, df.userHash==users.userHash, 'inner')\
    .join(artists, df.artistName==artists.artistName, 'inner')\
    .select(users.userId, artists.artistId, df.listenCount.cast("float"))
```

Pokud necháme Spark vypsat schéma výsledného datasetu, ověříme, že se výsledný dataset ratings skládá ze tří sloupců. Sloupec userId pro celočíslený identifikátor uživatele, artistId pro celočíselný identifikátor interpreta a listenCount obsahující reálné číslo odpovídající počtu poslechů:

```
root
 |-- userId: integer (nullable = true)
 |-- artistId: integer (nullable = true)
 |-- listenCount: float (nullable = true)
```

6.3. Průzkum dat

Další fází je získat povědomí o datech ze kterých se následně pokusíme vytrénovat příslušný model. Nejprve zjistíme krajní hodnoty pro počet poslechů:

```
df.select(min("listenCount"), max("listenCount"), avg("listenCount")).show()
```

```
+---------------+---------------+------------------+
|min(listenCount)|max(listenCount)|  avg(listenCount)|
+---------------+---------------+------------------+
|            1.0|       419157.0|215.18530888924704|
+---------------+---------------+------------------+
```

Z výstupu je patrné, že hodnoty nejsou v datasetu rovnoměrně distribuované. Minimální hodnota, kdy je pro jediného uživatele 400 tisíc poslechů jediného interpreta, mnohonásobně převyšuje průměrnou hodnotu 215. Z Obr. 9 je zřejmé, že se většina

hodnot pohybuje kolem průměru ale dataset obsahuje malý počet extrémních hodnot.

Obr. 9. Rozložení hodnot v datasetu hodnocení

6.4. Vytvoření tréningového a testovacího datasetu

Před vytvořením samotného modelu je třeba vyčlenit část dat tak, aby nebyla zahrnuta v tréninkové fázi. Tento takzvaný testovací dataset bude použit pro vyhodnocení přesnosti modelu viz. 2.3.1. Zdrojový dataset rozdělíme do dvou částí v poměru 70% pro trénovací dataset a zbylých 30% pro dataset testovací:

```
train, test = ratings.randomSplit([0.7, 0.3])
```

6.5. Vytvoření iniciálního modelu

Iniciální model bude obsahovat výchozí hodnoty pro všechny parametry algoritmu tak jak jsou uvedeny viz. 5.4.2. Model bude vytvořen pouze na základě tréninkových dat:

```
from mlonspark.alternating_least_square import AlternatingLeastSquare
alg = AlternatingLeastSquare()\
    .setUserCol("userId")\
    .setItemCol("artistId")\
    .setRatingCol("listenCount")

model = alg.fit(train)
```

Po vytvoření modelu vyhodnotíme jeho přesnost. Nejprve model pomocí metody **transform** aplikujeme na testovací i trénovací data. Tato operace do testovacího a trénovacího datasetu přidá nový sloupec prediction, který obsahuje predikci počtu poslechů:

```
trainPredictions = model.transform(train)
testPredictions = model.transform(test)
```

Následně pomocí vestavěné Spark třídy RegressionEvaluator spočítáme RMSE na základě naměřených poslechů ve sloupci listenCount a těch predikovaných ve sloupci prediction:

```
evaluator = RegressionEvaluator()\
    .setMetricName("rmse")\
    .setLabelCol("listenCount")\
    .setPredictionCol("prediction")

trainRmse = evaluator.evaluate(trainPredictions)
testRmse = evaluator.evaluate(testPredictions)
```

Výsledek jak pro trénovací RMSE tak i pro testovací obsahuje extrémně vysoké hodnoty. Průměrná chyba odpovídá trojnásobku průměru počtu poslechů. Také RMSE pro testovací data, která nebyla součástí učícího procesu, je paradoxně nižší než pro data trénovací:

```
train RMSE = 655.374786
test RMSE = 641.610511
```

Tento, extrémně nepřesný výsledek, je pravděpodobně způsobený nerovnoměrnou distribucí hodnot ve zdrojovém datasetu viz. Obr. 9. Zejména malý počet extrémních hodnot bude mít s velkou pravděpodobností za následek pokřivení výsledného modelu.

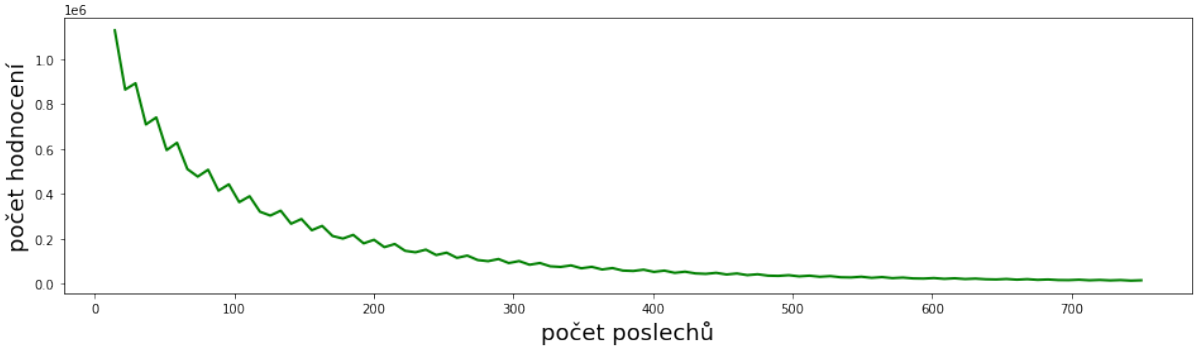
6.6. Úprava vstupních dat

Pro úpravu vstupních dat zvolíme jednoduchou metodu kdy z datasetu odstraníme nejnižších a nejvyšších 5% hodnot. Nejprve spočítáme 5% resp. 95% kvantil ze zdrojových dat:

```
from pyspark.sql import DataFrameStatFunctions as statFunc
quantiles = statFunc(df).approxQuantile("listenCount", [0.05, 0.95], 0.001)
```

```
[6.0, 751.0]
```

Z výsledku je patrné, že zejména hodnota 95% kvantilu 751 je výrazně nižší než maximální hodnota 419157. Po odstranění nejnižších a nejvyšších 5% hodnot je distribuce hodnot v datasetu výrazně rovnoměrnější viz. Obr. 10.



Obr. 10. Rozložení hodnot v upraveném datasetu

Pro model vytvořený na základě upraveného datasetu jsme naměřili RMSE 212.78 pro tréninkový dataset, resp. 213.15 pro dataset testovací. Tyto hodnoty se již blíží průměrné hodnotě 149.60. Také RMSE pro testovací dataset je vyšší než pro dataset tréninkový.

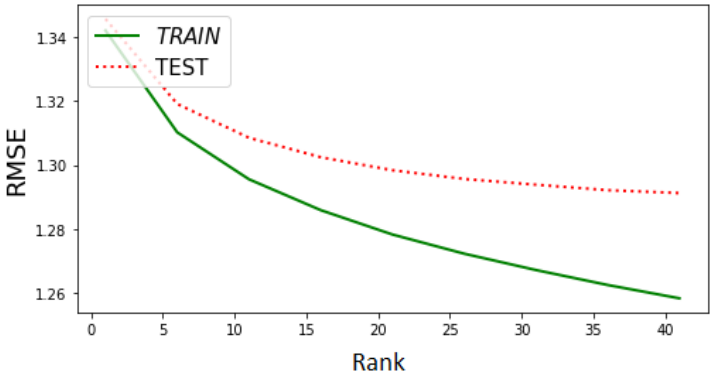
Další úpravou zdrojových dat je standardizace hodnot viz. 2.1. Tato úprava převede hodnoty tak aby měly nulový průměr a směrodatnou odchylku rovnou 1. Problém je, že ALS algoritmus očekává pouze kladné hodnoty, proto všechny hodnoty posuneme tak, aby minimální hodnota byla rovná nule. Pro model vytvořený na základě standardizovaného datasetu jsme naměřili RMSE 1.297 pro tréninkový datset, resp. 1.310 pro dataset testovací.

6.7. Ladění hyper parametrů

V další fázi se pokusíme nalézt optimální hodnoty pro jednotlivé hyper parametry viz. 5.4.2. Optimální postup by byl použít křížovou validaci spolu se Spark třídou **ParamGridBuilder**, která umožní nalézt optimální hodnoty pro kombinace více hyper parametrů. Bohužel tato technika dalece přesahuje dostupný výkon použitého testovacího clustru. V našem případě budeme hledat optimální hodnotu pro každý hyper parametr zvlášť pro trénovací i testovací dataset.

6.7.1. Rank

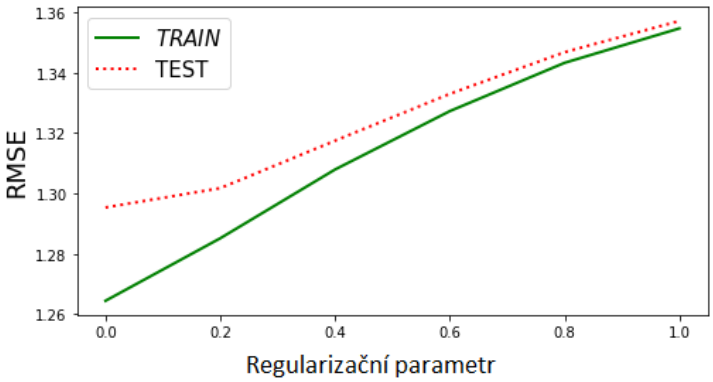
Z Obr. 11 je patrné, že RMSE klesá jak pro trénovací i testovací sadu pro rank menší než 25. Dále již klesá RMSE pouze pro sadu trénovací, což znamená že se model stává postupně přeučeným na tuto sadu. Pro další výpočty použijeme hodnotu hyper parametru rank 25.



Obr. 11. Rank

6.7.2. Regularizační parametr

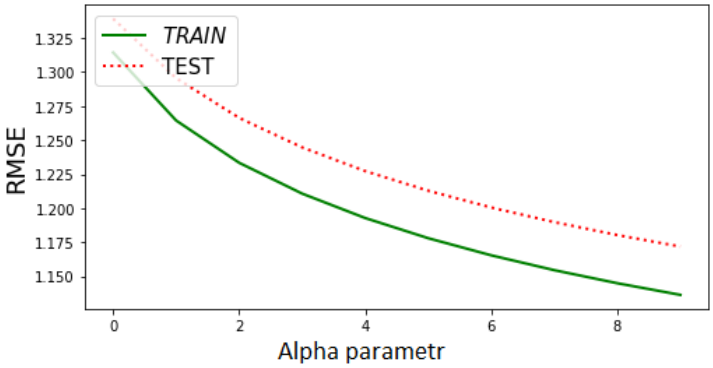
Z Obr. 12 je patrné, že i minimální regularizace má negativní vliv na výsledný model. Typicky by se měla regularizace aplikovat pokud by docházelo k přeučení modelu s velkým rozdílem mezi trénovací a testovací sadou. V našem případě se obejdeme bez regularizace a použijeme nulový regularizační parametr.



Obr. 12. Regulariční parametr

6.7.3. Alpha parametr

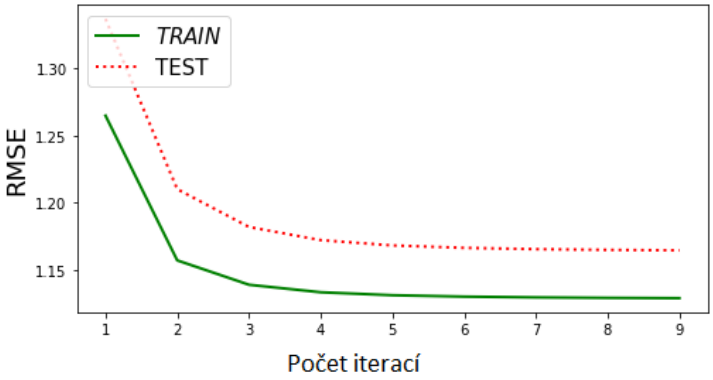
Z Obr. 13 je patrné, že pro alpha parametr poskytují lepší výsledky vyšší hodnoty než výchozí hodnota 1. Pro další výpočty použijeme hodnotu hyper parametru alpha 10.



Obr. 13. Alpha parametr

6.7.4. Počet iterací

Počet iterací by se měl nastavit dostatečně vysoký aby nadále nedocházelo ke snížení výsledného RMSE, tj. tato hodnota by měla uváznout ve svém minimu. Z Obr. 14 je zřejmé že výchozí počet iterací 10 je dostatečný, hodnota RMSE se ustálí již po šesté iteraci.



Obr. 14. Počet iterací

6.7.5. Optimální hodnoty hyper parametrů

V Tab. 3 je zobrazen přehled hyper parametrů a nalezených optimálních hodnot. Vzhledem k použité technice není zaručené, že nedošlo k uváznutí na lokálním minimu a neexistuje kombinace parametrů, která by dosáhla lepších výsledků.

Tab. 3. Optimální hodnoty hyper parametrů

Parametr	**Hodnota**
Rank	25
Regularizační parametr	0.0
Alpha parametr	10

Parametr	Hodnota
Počet iterací	10

7. Zhodnocení vytvořeného modelu

Pro vyhodnocení konkrétních doporučení byl vybrán jeden z uživatelů zaměřený téměř výhradně na poslech elektronické hudby. V následujícím přehledu je zobrazeno 20 interpretů s nejvíce poslechy pro tohoto uživatele. Nejvíce poslouchaný interpret je *infected mushroom* spadající do podžánru *psy trance*, ostatní interpreti, až na *stereo mcs* a *underworld*) spadají do podžánru *drum&bass*:

```
+--------------------+-----------+
|          artistName|listenCount|
+--------------------+-----------+
|   infected mushroom|      330.0|
|      chase & status|      323.0|
|       high contrast|      298.0|
|            syncopix|      217.0|
|           aphrodite|      214.0|
|          stereo mcs|      206.0|
|          underworld|      193.0|
|bachelors of science|      168.0|
|  london elektricity|      166.0|
|                  tc|      153.0|
|               cosma|      151.0|
|             nu:tone|      137.0|
|             dj zinc|      123.0|
|             calibre|      115.0|
|              makoto|       99.0|
|              commix|       91.0|
|              bungle|       86.0|
|            pendulum|       86.0|
|           frou frou|       82.0|
|           sub focus|       80.0|
+--------------------+-----------+
```

Nejprve si necháme sestavit doporučení nad kompletními vstupními daty a modelem s výchozími hodnotami pro jednotlivé hyper parametry. Z výstupu je patrné, že nám model poskytl relativně uspokojivá doporučení. Všichni interpreti spadají do elektronické hudby. *Drum&bass* je v doporučení zastoupen ve 40%, ve výběru není žádný interpret z podžánru *psy trance*:

```
+------------------+
|        artistName|
+------------------+
|     high contrast|
|              krec|
|london elektricity|
|           deadmau5|
|          aphrodite|
|          junkie xl|
|             hybrid|
|          logistics|
|     groove armada|
| nightmares on wax|
+------------------+
```

Další doporučení si necháme sestavit od modelu, který byl vytvořen nad daty, ze kterých byl odebrán dolní 5% kvantil resp. horní 95% kvantil. Tímto jsme z datasetu odebrali extrémní hodnoty a dosáhli tím vyrovnanější distribuce hodnot viz. Obr. 10. K vytvoření modelu použijeme optimální hodnoty hyper parametrů viz. Tab. 3. Z výstupu je patrné, že nám model poskytl perfektní doporučení. Všichni intepreti jsou z podžánru *drum&bass* až na *1200 micrograms*, který patří do žánru *psy trance*. Model doporučuje interprety u kterých tento uživatel nemá žádné poslechny. Oproti modelu s použitými výchozími hodnotami došlo k výraznému kvalitativnímu posunu:

```
+------------------+
|        artistName|
+------------------+
|          logistics|
|   1200 micrograms|
|     high contrast|
|             john b|
|      concord dawn|
|london elektricity|
|              klute|
|          aphrodite|
|           dj fresh|
|          ltj bukem|
+------------------+
```

Poslední doporučení si necháme sestavit od modelu, u kterého jsme provedli standardizaci hodnot. K vytvoření modelu byla také použita data bez extrémních hodnot viz. Obr. 10 a optimální hodnoty hyper parametrů viz. Tab. 3. Z výstupu je patrné, že tento model poskytuje zatím nejhorší doporučení. Ačkoliv všichni interpreti spadají do elektronické hudby, pouze jediný, *pendulum*, spadá do preferovaných podžánrů. Standardizace hodnot pravděpodobně zapříčinila pokřivení výsledného modelu:

```
+--------------------+
|          artistName|
+--------------------+
|         the prodigy|
|the chemical brot...|
|            pendulum|
|                moby|
|         fatboy slim|
|thievery corporation|
|           daft punk|
|            röyksopp|
|   infected mushroom|
|           faithless|
+--------------------+
```

8. Závěr

Účelem této práce byla experimentální aplikace strojového učení na platformě Apache Spark. Vybraný algoritmus měl být implementován s důrazem na paralelní výpočet a škálovatelnost. Pro algoritmus měla být navržena metodika ohodnocení přesnosti jednotlivých výpočtů.

Těchto vytyčených cílů se podařilo dosáhnout. Pro test byl zvolen dataset obsahující záznamy o počtu poslechů hudebních interpretů jednotlivými uživateli. Na základě těchto implicitních dat se podařilo pomocí nástroje Apache Spark vytrénovat prakticky použitelný model. Tento model byl schopen doporučovat uživatelům hudební interprety dle jejich preferencí. Implementovaný algoritmus dokázal provádět výpočty paralelně nad částmi datasetu, distribuovaného v rámci výpočetního clustru. Při dostatečném výpočetním výkonu by bylo možné tento systém aplikovat i na řádově větší objemy dat.

Apache Spark je užitečný nástroj s jednoduchým a srozumitelným API. Komplexní, distribuovaný algoritmus pro strojové učení se na této platformě podařilo implementovat i bez předchozích zkušeností. Apache Spark má kvalitní dokumentaci a širokou vývojářskou základnu. Tyto dva prvky výrazně usnadňovaly řešení problémů v průběhu implementace. V projektu byl použit programovací jazyk Scala, nativní jazyk Sparku. Ačkoliv se pro Java vývojáře mohla zdát syntaxe Scaly poněkud složitá, během implementace projektu vynikla síla tohoto jazyka. Zejména podpora funkcionálního programování činí z tohoto jazyka ideální volbu pro Spark. Integrované vývojové prostředí IntelliJ IDEA se ukázalo jako správná volba. Kvalitní podpora programovaciho jazyka Scala a jednotkových testů umožnila efektivní vývoj a rychlé testování aplikace.

Apache Hadoop ma pozvolnou křivku učení. Porozumět účelu jednotlivých částí systému potřebných pro běh Spark aplikace a poskládat tyto části do funkčního celku se ukázalo jako časové náročný proces. Při spoustění aplikace docházelo zprvu k haváriím, které si vyžádaly prověřování specifických systémových logů a následnou rekonfiguraci problematických prvků.

Pro aplikaci vyvinutého algoritmu na konkrétní dataset byl použit programovací jazyk Python v rámci interaktivních zápisníků Jupyter. Tato kombinace se ukázala jako praktický nástroj pro ladění jednotlivých hyper parametrů a vizualizaci výsledků.

Velkým překvapením je algoritmus ALS. Doporučení sestavená na základě tohoto algoritmu se ukázala jako velice kvalitní. Také nalezení optimálních hodnot pro jednotlivé hyper parametry mělo za následek zvýšení přesnosti modelu i kvalitativní posun výsledných doporučení.

Závěrem lze konstatovat, že na zvolených technologiích je možné realizovat systém pro produkční použití. Takový systém, odolný vůči softwarovým i hardwarovým haváriím, by bylo možné škálovat pro milióny zákazníků a desítky tisíc produktů.

9. Seznam použité literatury

[1] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 29–43, Oct. 2003, doi: 10.1145/1165389.945450. [Online]. Available: https://doi.org/10.1145/1165389.945450

[2] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, San Francisco, CA, 2004, pp. 137–150.

[3] "Apache Hadoop." 2020 [Online]. Available: http://hadoop.apache.org/

[4] "Apache Spark." 2020 [Online]. Available: https://spark.apache.org/

[5] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An Introduction to Statistical Learning: with Applications in R*. Springer New York, 2013 [Online]. Available: https://books.google.cz/books?id=qcI_AAAAQBAJ

[6] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer New York, 2013 [Online]. Available: https://books.google.cz/books?id=yPfZBwAAQBAJ

[7] A. Géron, *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, 2017 [Online]. Available: https://books.google.cz/books?id=khpYDgAAQBAJ

[8] B. Chambers and M. Zaharia, *Spark: The Definitive Guide: Big Data Processing Made Simple*. O'Reilly Media, 2018 [Online]. Available: https://books.google.cz/books?id=oitLDwAAQBAJ

[9] Y. Hu, Y. Koren, and C. Volinsky, "Collaborative Filtering for Implicit Feedback Datasets," in *Proceedings of the 2008 Eighth IEEE International Conference on Data Mining*, USA, 2008, pp. 263–272, doi: 10.1109/ICDM.2008.22 [Online]. Available: https://doi.org/10.1109/ICDM.2008.22

[10] S. Ryza, U. Laserson, S. Owen, and J. Wills, *Advanced Analytics with Spark: Patterns for Learning from Data at Scale*. O'Reilly Media, 2015 [Online]. Available: https://books.google.cz/books?id=WE_GBwAAQBAJ

[11] D. Skillicorn, *Understanding Complex Datasets: Data Mining with Matrix Decompositions*. CRC Press, 2007 [Online]. Available: https://books.google.cz/books?id=9SzLDi8jUnAC

[12] L. Elden, *Matrix Methods in Data Mining and Pattern Recognition, Second Edition*. Society for Industrial and Applied Mathematics, 2019 [Online]. Available: https://books.google.cz/books?id=2M2sDwAAQBAJ

[13] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan, "Large-Scale Parallel Collaborative Filtering for the Netflix Prize," in *Proceedings of the 4th International Conference on Algorithmic Aspects in Information and Management*, Berlin, Heidelberg, 2008, pp. 337–348, doi: 10.1007/978-3-540-68880-8_32 [Online]. Available: https://doi.org/10.1007/978-3-540-68880-8_32

[14] K. Liao, "Prototyping a Recommender System Step by Step Part 1: KNN Item-Based Collaborative Filtering." 2018 [Online]. Available: http://towardsdatascience.com/prototyping-a-recommender-system-step-by-step-part-1-knn-item-based-collaborative-filtering-637969614ea

[15] J. Kunigk, I. Buss, L. George, and P. Wilkinson, *Architecting Modern Data Platforms: A Guide to Enterprise Hadoop at Scale*. O'Reilly Media, Incorporated, 2019 [Online]. Available: https://books.google.cz/books?id=ErAFMQAACAAJ

Příloha A: Zdrojový kód projektu

Přílohou práce je exportovaný git repozitář obsahující kompletní zdrojový kód projektu. Tento repozitář je také dostupný na domovské stránce projektu https://github.com/barenode/bp nebo přes lokální klon repozitáře:

```
git clone https://github.com/barenode/bp.git
```

UNIVERZITA HRADEC KRÁLOVÉ
Fakulta informatiky a managementu
Akademický rok: 2018/2019

Studijní program: Aplikovaná informatika
Forma studia: Kombinovaná
Obor/kombinace: Aplikovaná informatika (ai3-k)

Podklad pro zadání BAKALÁŘSKÉ práce studenta

Jméno a příjmení: **Frantisek Hylmar**
Osobní číslo: **I1700643**
Adresa: J. Grafka 1452, Nová Paka, 50901 Nová Paka, Česká republika

Téma práce: Strojové učení na platformě Apache Spark
Téma práce anglicky: Machine learning with Apache Spark

Vedoucí práce: prof. RNDr. PhDr. Antonín Slabý, CSc.
Katedra informatiky a kvantitativních metod

Zásady pro vypracování:

Cíl: Prozkoumat metody strojového učení na platformě Spark v oblasti doporučovacích systémů.
Osnova

1. Úvod do problematiky doporučovacích systémů
2. Prozkoumání souvisejících technologií a algoritmů
3. Představení platformy Spark
4. Implementace vybraných algoritmů v řešené oblasti
5. Optimalizace algoritmických parametrů v oblasti doporučovacích systémů nad testovacím vzorkem dat

Seznam doporučené literatury:

- Yifan Hu, Yehuda Koren, Chris Volinsky. Collaborative Filtering for Implicit Feedback Datasets. IEEE. 2008 Eighth IEEE International Conference on Data Mining. 15-19 Prosinec 2008. http://yifanhu.net/PUB/cf.pdf
- Zhou Y., Wilkinson D., Schreiber R., Pan R. (2008) Large-Scale Parallel Collaborative Filtering for the Netflix Prize. In: Fleischer R., Xu J. (eds) Algorithmic Aspects in Information and Management. AAIM 2008. Lecture Notes in Computer Science, vol 5034. Springer, Berlin, Heidelberg
- Uri Laserson, Sandy Ryza, Sean Owen, Josh Wills. Advanced Analytics with Spark, 2nd Edition. O'Reilly Media, Inc. Cerven 2017. ISBN: 9781491972946
- Chambers, Bill, and Matei Zaharia. Spark: the Definitive Guide: Big Data Processing Made Simple. O'Reilly Media, 2018.
- James, Gareth. An Introduction to Statistical Learning: with Applications in R. Springer, 2017.

Podpis studenta: Datum:

Podpis vedoucího práce: Datum: