UNIVERZITA KOMENSKÉHO V BRATISLAVE

FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

Detekcia anomálií

Bakalárska práca

2016

Michal Kubica

UNIVERZITA KOMENSKÉHO V BRATISLAVE

FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

Detekcia anomálií

Bakalárska práca

Študijný program: Informatika

Študijný odbor: 2508 Informatika

Školiace pracovisko: Katedra informatiky

Školiteľ: Mgr. Lukáš Polesňák

Bratislava 2016

Michal Kubica

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Michal Kubica
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Detekcia anomálií
Anomaly detection

Cieľ: Cieľom práce je ukázať, ako detegovať anomálie v dátach využitím metódy k-means v distribuovanom prostredí Apache Spark. Následne sa pokúsiť upraviť a zefektívniť tento detektor anomálií.

Vedúci: Mgr. Lukáš Polesňák
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: doc. RNDr. Daniel Olejár, PhD.

Dátum zadania: 27.10.2015

Dátum schválenia: 29.10.2015

doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

POĎAKOVANIE

Veľmi rád by som sa chcel touto cestou poďakovať všetkým, ktorí mi pomáhali pri písaní tejto práce. Moja vďaka patrí najmä školiteľovi Mgr. Lukášovi Polesňákovi za vedenie a odbornú pomoc. Takisto by som sa chcel poďakovať mojej rodine a priateľom za poskytnutie cenných rád pri písaní tejto práce.

Abstrakt

Cieľom práce je popísať pojem anomália, ako aj problémy spojené s ich detegovaním. Zamerali sme sa hlavne na problémy detekcie anomálií pomocou klastrovacej metódy k-means a neustále zväčšujúcim sa množstvom dát, ktoré je nutné analyzovať. Pre tieto účely sme využili distribuované prostredie na prácu s dátami veľkých rozmerov Apache Spark. V závere sme implementovali modifikovanú metódu na detegovanie anomálií, ktorá dosahovala lepšie výsledky v porovnaní s klasickým detektorom založenom na metóde k-means a jednoduchej hranici.

Kľúčové slová: anomália, Hadoop, Cloudera, k-means, Apache Spark

Abstract

The objective of the thesis is to describe the term anomaly and the problems connected with their detection. The main focus lies on the problems connected to the anomaly detection by clustering method k-means and the ever increasing amount of data needed to be analysed. To fulfil stated objectives, Apache Spark, the big data processing framework, was used. In the final part of the thesis, we implemented a modified method for anomaly detection. The use of the modified method achieved better results than the use of the detector, which is based on the k-means method and simple threshold, did.

Key words: anomaly, Hadoop, Cloudera, k-means, Apache Spark

Obsah

Úvod

V súčasnom svete sa začína zberať a uchovávať čoraz viac dát pre potreby analýz, či už marketingového zamerania, manažérskeho rozhodovania, ochrany a iných dôležitých zameraní každej spoločnosti. Pre správnu analýzu je treba poznať charakter dát. Niekedy sa stáva, že sa v dátach vyskytnú hodnoty, ktoré vznikli neprirodzene. Takéto dáta, takzvané anomálie je dôležité odhaliť, aby nedošlo k škodám a k skresleniu analýz. Napríklad výchylky zo senzorov pri skúškach leteckých motorov môžu signalizovať možné fyzické chyby, ktoré môžu viesť ku katastrofickým následkom. Práve preto sme sa rozhodli zaoberať sa touto problematikou a implementovať špeciálnu metódu na detegovanie anomálií, ktorá by skvalitnila doposiaľ využívané algoritmy.

V prvej časti práce popisujeme, čo sa skrýva za pojmom anomália a ako sa anomálie rozlišujú napríklad od šumu v dátach. Ďalej si priblížime možné typy premenných s ktorými sa pri práci s dátami môžeme stretnúť, ako aj problémy s nimi spojené. Uvedieme si, akými spôsobmi sa dá merať vzdialenosť medzi jednotlivými vzormi dát.

Druhá časť práce je zameraná na niektoré typy algoritmov, ktoré sa využívajú pri detekcií anomálií. Pre každý algoritmus si bližšie špecifikujeme, ako sa implementujú jednotlivé detektory a aké problémy nás môžu stretnúť, ak si daný algoritmus vyberieme.

V ďalšej časti si popíšeme problémy a výhody spojené so spracúvaním a uchovávaním veľkých dát, ako aj technológie, ktoré sa využívajú pre prácu spojenú s dátami veľkých rozmerov.

Posledná časť je zameraná na implementáciu k-means algoritmu pre potreby detekcie anomálií, pričom využijeme aj implementáciu z knižnice tvorenej jadro distribuovaného nástroja na prácu s dátami Apache Spark. V tejto časti si ukážeme aj vlastnú distribuovanú implementáciu k-means algoritmu. Nakoniec sa budeme venovať vylepšeniu detektora založeného na metóde k-means.

1 Detekcia anomálií

Detekcia anomálií súvisí s problémom hľadania vzorov v dátach, ktoré nezodpovedajú očakávanému správaniu. Detekcia anomálii nachádza uplatnenie v mnohých odvetviach. Od bankovníctva, kde sa snaží odhaliť možné bankové podvody, cez zdravotníctvo, až po priemysel, kde detekcia anomálií pomáha odhaľovať chyby v produktoch. Anomália v MRI obrázku môže indikovať zhubný tumor. Ak sa na takéto anomálie príde skoro, môžu sa nasadiť postupy, ktoré minimalizujú možné škody spôsobené týmito anomáliami. Predstavme si situáciu, že sme majitelia firmy. a dozvedeli sme sa, že sa z našej firmy pred mesiacom odoslali po sieti citlivé informácie, ktoré môžu zničiť našu firmu. Ak by sme mali systém na detekciu anomálií, tak by sme mohli takémuto scenáru zabrániť, poprípade obmedziť škody, ktoré nastali.

Schopnosť detegovať anomálie môže byť nevyčísliteľne cenná. Nasledujúce príklady poukazujú na možné využitie detekcie anomálií:

- Výchylky v tlkote srdca, ktoré môžu indikovať možné zdravotné problémy.
- Pozorovať malé výchylky v behu leteckého motora, ktoré môžu zapríčiniť zlyhanie motora.
- Identifikovanie zmeny správania zamestnanca, ktoré môžu signalizovať bezpečnostný problém.

Detekcia anomálií je úzko spätá s redukciou šumu, kde sa snažíme dáta, ktoré sú určené na analyzovanie, najskôr očistiť od šumu a až následne po čistení dát sa začne proces analýzy. Šum môže byt definovaný ako jav v dátach, ktorý nie je v záujme analytika, ale pôsobí ako prekážka pre analýzu dát.

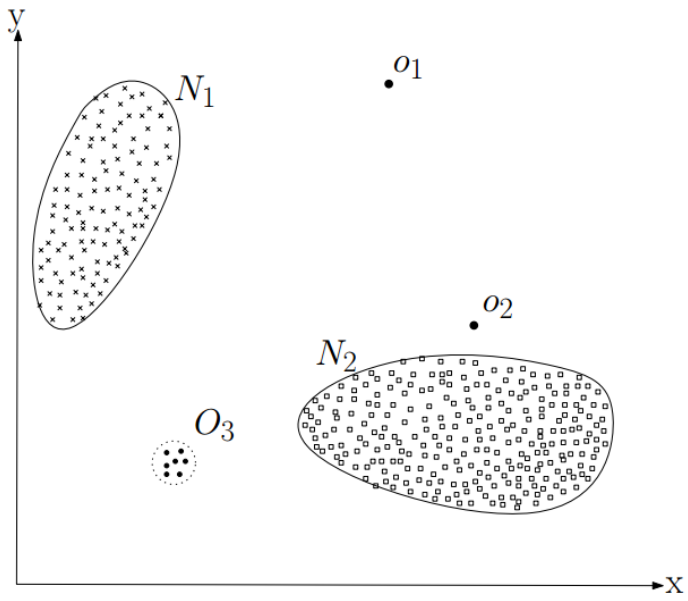
Ďalší pojem úzko spätý s detekciou anomálií je detekcia noviniek (novelty detection) [1]. Tu sa snažíme identifikovať dosiaľ nevidené vzory v dátach. Príklad detekcia noviniek je identifikovanie nových tém v diskusných fórach, spracovanie signálu, kde sa snažíme detegovať nové signály, ktoré sú odlišné od signálov doposiaľ skúmaných. Rozdiel medzi detekciou anomálií a detekciou noviniek je hlavne v tom, že vzory, ktoré objaví detekcia noviniek sa zvyčajne stanú súčasťou normálnych vzorov dát.

Metódy, ktoré sa používajú pre detekciou noviniek a redukciu šumu sa zvyčajne dajú využiť aj pri metódach detekcie anomálií a naopak.

1.1 Anomálie

Anomálie sú vzory v dátach, ktoré nezodpovedajú normálnemu správaniu, sú zjavne odlišné od ostatných dát [2], a s touto vlastnosťou pracujú mnohé metódy na detekciu anomálií. Obrázok 1 ukazuje anomálie v 2-rozmernom priestore. Na obrázku môžeme pozorovať dva normálne regióny N1 a N2, ďalej môžeme pozorovať anomálie o1,o2 a množinu bodov o3, ktoré sú v našom priestore ďaleko od normálnych regiónov N1 a N2.

Obrázok 1 : Jednoduchý príklad anomálie v 2-rozmernom priestore, Zdroj : [2]



Anomálie môžu byť v dátach zahrnuté z rôznych dôvodov, napríklad podvodná aktivita, chyby v produktoch, prieniky do bezpečnostných systémov. Všetky tieto dáta sa dajú nejakým spôsobom identifikovať, čím nám dávajú možnosť ich pozorovať a predikovať.
Anomálne udalosti sa vyskytujú relatívne zriedka, no pri ich výskyte vznikajú zväčša najväčšie škody. No existujú aj anomálie, ktorých identifikovanie a zvýšenie ich počtu vplýva pozitívne pre oblasť, ktorú skúmame. Niekedy sa môže stať, že anomálie neprinesú len škody, ale môžu priniesť aj výhody. Ak je takýchto anomálií viac, snažíme sa detegovať pôvod, alebo príčinu vzniku daných anomálií. Ak dokážeme nájsť dôvod vzniku anomálie a odhalíme vzor dát, ktorý je zdanlivo rozdielny a zdá sa, že ide o anomáliu, ale v konečnom dôsledku sa správa podobne na základne neskôr odhalených kritérií, môžeme to využiť na obmedzenie alebo na zvýšenie počtu takýchto vzorov. Predstavme si situáciu,

kde sme v roli tvorcu hry, ktorá generuje rôzne udalosti (*Hráč sa dostal do deviateho levelu*, *Hráč zabil obludu 32* ...). Jedna z takýchto udalostí by bola *„predplatenie prémium účtu"*. Našim cieľom je mať takýchto udalostí čo najviac, keďže práve tieto udalosti generujú zisk. Takáto udalosť je vcelku unikátna a náš systém na detekciu anomálií by sa snažiť naučiť pravidlá na detekciu hráčov s danou udalosťou. Keď zistíme, čo mali spoločné, môžeme to využiť pri kampaniach na získanie väčšieho počtu predplatiteľov.

Rozdelenie anomálií podľa vzniku:

- Podvodné hodnoty
- Náhodné dáta, ktoré sa líšia od zvyšku
- Poškodené dáta

Náhodné dáta skresľujú výsledky analýz. Odľahlé údaje môžu spôsobiť, že celková štatistická analýza bude skreslená, čo sa prejaví na kvalite výsledku štatistickej analýzy. Poškodené dáta znemožňujú správnu analýzu.

Z príkladov môžeme vidieť, že detegovať anomálie nie je ľahký problém, no ak sa mu nebudeme dostatočne venovať a na anomálie prídeme neskoro, môžeme prísť k značným škodám.

1.2 Typy premenných

"Premenné, alebo inak, štatistické znaky sú hodnoty, ktoré meriame, sledujeme alebo s nimi manipulujeme počas výskumu. Líšia sa v tom, akú rolu zohrávajú v našom výskume a v spôsobe ich merateľnosti" [1].

Základné rozdelenie premenných:

- **Numerické premenné**

  Sú to číselné premenné, ktoré dokážeme zaradzovať, pripočítavať, odčítať, deliť a násobiť a výsledok bude zrozumiteľný. Dokážeme sa pýtať otázky typu „koľko" alebo „ako veľa". Numerické premenné sa ďalej delia na:

  - **Diskrétne** - premenná dokáže nadobudnúť len konečný alebo spočítateľný počet možných hodnôt. Diskrétna premenná však nemôže nadobudnúť

[1] StatSoft, Inc. (1999). Electronic Statistics Textbook. Tulsa, OK: StatSoft

hodnoty medzi jednou a nasledujúcou izolovanou hodnotou. Príklad diskrétnej premennej je „počet_detí“ táto premenná môže nadobudnúť hodnoty z množiny {0,1,2,3,...,x}, pričom nie je prípustná hodnota *2,5*.

- **Spojité** – premenná dokáže nadobudnúť všetky hodnoty z konečného alebo nekonečného intervalu. To znamená, že premenná spojitého typu môže nadobudnúť nekonečne veľa hodnôt. V praxi spojité premenné nadobúdajú hodnoty také malé, ako to dovoľuje merací systém, ktorý premenné meria. Príklady spojitých premenných sú napríklad vek, čas, výška, teplota, hmotnosť. Napríklad výška múra môže byť 210cm, 320cm ale aj hodnota medzi nimi 225,3cm.

- **Kategorické premenné**

  Ich hodnoty popisujú kvalitu alebo charakteristiku premennej, ako napríklad „aký typ“ alebo „ktorá kategória“. Tieto premenné môžu nadobudnúť len jednu hodnotu z danej množiny prípustných hodnôt. Kategorické premenné môžu byť ďalej rozdelené na:

  - **Ordinálne** – premennú takéhoto typu dokážeme logicky zoradiť. O kategóriách ordinálnych premenných dokážeme povedať či je jedná „lepšia“, „horšia“ ako druhá, ale nemusíme vedieť povedať presnú číselnú hodnotu ich rozdielu ako je to pri numerických premenných. Príklady ordinálnych premenných sú napríklad známkovanie na vysokej škole {A,B,C, D, E, Fx}, veľkosť oblečenia {L,M,S,XL} ....
  - **Nominálne** - premenné takéhoto typu nedokážeme zoradiť z hľadiska logickej sekvencie ako je to pri kategorických premenných ordinálneho typu. Príklady nominálnych premenných sú farba očí, vierovyznanie, pohlavie ...

1.3 Metrický priestor

Metrický priestor je matematická štruktúra, ktorá na danej neprázdnej množine umožňuje zadefinovať pojem vzdialenosti. Pozostáva z neprázdnej základnej množiny X a funkcie d, nazývanej **metrika** na X, ktorá každej dvojici bodov zo základnej množiny X priraďuje ich vzdialenosť, pričom sú pre ňu splnené isté podmienky.

Metrický priestor [2] môžeme definovať ako usporiadanú dvojicu (X, d), kde X je neprázdna množina a d je zobrazenie $d : X^2 \rightarrow R$ na usporiadaných dvojiciach prvkov X, pre ktoré sú splnené nasledujúce podmienky :

1. $Pre\ \forall x, y \in X\ je\ d(x, y) \geq 0\ a\ d(x, y) = 0 <=> x = y$
2. $Pre\ \forall x, y \in X\ je\ d(x, y) = d(y, x)$ (symetria)
3. $Pre\ \forall x, y, z \in X\ je\ d(x, y) \leq d(x, z) + d(z, y)$ (trojuholníková nerovnosť)

Funkciu d nazývame metrika na množine X a dvojicu (X, d) nazývame metrickým priestorom.

Na jednej základnej množine môže byť definovaných aj viacero metrík, preto metrický priestor nie je svojou základnou množinou jednoznačne určený.

Pri hľadaní anomálií potrebujeme vedieť, aká vzdialenosť je medzi jednotlivými vzormi. Výber správnej funkcie, podľa ktorej budeme merať vzdialenosť medzi bodmi je ťažký problém a závisí to hlavne od typu dát s ktorými pracujeme. V euklidovskom n-rozmernom priestore E_n, ktorý je definovaný ako množina všetkých usporiadaných n-tíc $X = (x_1, x_2, x_3, \dots, x_n)$ reálnych čísel, pričom vzdialenosť d ľubovoľných dvojíc $X = (x_1, x_2, x_3, \dots, x_n)$, $Y = (y_1, y_2, y_3, \dots, y_n)$ je vzdialenosť definovaná vzťahom $d(X, Y)$, kde n-ticu $X = (x_1, x_2, x_3, \dots, x_n)$ reálnych čísel nazývame bodom priestoru a čísla $x_1, x_2, x_3, \dots, x_n$ súradnice tohto bodu.

Základné typy vzdialeností:

- Euklidovská vzdialenosť : $d(X, Y) = \sqrt{\sum_{i=0}^{n} |x_i - y_i|^2}$, kde x_i, y_i sú i-té súradnice bodov X, Y

  Najväčšia nevýhoda euklidovskej vzdialeností je tá, že je citlivá na škálovanie jednotlivých premenných. Ak môže jedna premenná nadobudnúť hodnoty v intervale <-100,100>, a druhá <-1,1>, tak v konečnom výsledku bude druhá premenná skoro nepodstatná. Preto je nevyhnutné dáta normovať.

[2] *Simmons, G. F.: Introduction to Topology and Modern Analysis. McGraw-Hill, 1963.*

- Manhattanská vzdialenosť L_1: $d(X,Y) = \sum_{i=0}^{n} |x_i - y_i|$ kde x_i, y_i sú i-té súradnice bodov X, Y

  Taktiež známa pod menom *newyorská vzdialenosť* podľa pravouhlého systému ulíc v Manhattene. Táto norma je taktiež náchylná na rôzne škálovanie premenných.
- Maximova vzdialenosť L_∞ : $d(X,Y) = \max |x_i - y_i|$ kde x_i, y_i sú i-té súradnice bodov X, Y

  Pri maximovej vzdialenosť je výsledná vzdialenosť daná ako maximálna vzdialenosť medzi atribútmi.

1.4 Fiktívne premenné

Situáciu s výberom správnej funkcie na výpočet vzdialenosti neuľahčuje ani fakt, že premenné, ktoré spracúvame nemusia byť nutne všetky numerického charakteru. Ak máme dáta, ktoré obsahujú aj kategorické premenné, napríklad *pohlavie*, ktoré môže nadobudnúť hodnoty {muž, žena}, musíme najskôr takéto premenné preložiť do priestoru, kde môžeme aspoň trochu merať jeho vzdialenosť. Väčšina algoritmov nedokáže pracovať s kategorickými premennými. Pri takýchto typoch premenných budeme využívať metódu na prevod atribútov do binárnej podoby. Ak sa daná kategória vyskytuje nadobudne premenná hodnotu *1* ak sa kategória nevyskytuje, tak premenná nadobudne hodnotu *0*.

Príklad prevodu kategorickej premennej do fiktívnych premenných(dummy variable).

Majme kategorickú premennú *pohlavie*, ktorá môže nadobudnúť tri hodnoty {muž, žena, nešpecifikované}. Majme záznam o mužskom užívateľovi s vekom *23* a identifikačným číslom *10*. Takýto užívateľ by z pohľadu našich dát vyzeral nasledovne.

Tabuľka 1 : Užívateľ

id	vek	pohlavie
10	23	muž

Ak prevedieme kategorickú premennú *pohlavie* do novej formy, tak nové dáta budú vyzerať nasledovne:

Tabuľka 2 : Užívateľ, po prevode na fiktívne premenné

id	vek	d_muz	d_zena	d_nespecifikovane
10	23	1	0	0

Ak by bol záznam o žene, jej premenná *d_zena* by nadobudla hodnotu „1“ a hodnota premennej *d_muz* by sa zmenila na „0“. Takýmto spôsobom dokážeme previesť kategorické premenné do nových premenných, s ktorými už väčšina algoritmov dokáže pracovať. Treba si uvedomiť, že prevod na nové premenné nie je ideálny. Po prevode už dokážeme merať vzdialenosti medzi premennými, no vzdialenosti, ktoré dostaneme nemusia zodpovedať realite. Ako príklad môžeme uviesť kategorické premenné z množiny {muž, žena, dievča, chlapec} ak by sme merali euklidovskú vzdialenosti medzi dvoma vzorkami dát, kde jedna vzorka by bola kategórie muž, a druhá žena, tak by sme dostali vzdialenosť $\sqrt{(1-0)^2+(0-1)^2+(0-0)^2+(0-0)^2} \pm 1{,}41$, čo by zodpovedalo vzdialenosti muža a ženy. No vzdialenosť medzi ženou a dievčaťom by vyšla taká istá, čo nezodpovedá skutočnej realite.

Táto metóda nie je dokonalá ako sme mohli vidieť na príklade vyššie, no treba si uvedomiť, že pri takomto prevode predpokladáme, že jednotlivé kategórie sú štatisticky úplne odlišné. Netreba zabúdať ani na to, že ak môže naša premenná nadobudnúť *x* rôznych kategórií, tak sa veľkosť našej množiny parametrov zväčší o *x*. Pre viac informácií [14].

2 Algoritmy

V tejto kapitole sa zaoberáme vybranými algoritmami, ktoré sa využívajú pre detekciu anomálií. Podrobnejšie si popíšeme len algoritmus k-means, ktorý budeme neskôr aj implementovať.

2.1 K-means

K-means[4] klastrovanie je jeden z najpoužívanejších algoritmov na klastrovanie dát, ktorý sa dá využíva aj pri problémoch zameraných na detekciu anomálií. Najväčšou nevýhodou je nutnosť určiť hodnotu parametra *k*, ktorý nám hovorí o počte klastrov, ktoré sa po dobehnutí algoritmu vytvoria. Pre potreby tejto kapitoly si musíme zadefinovať pojem *centroid*.

Pri algoritmoch k-means a iných klastrovacích algoritmoch je pojem centroid spájaný so stredom jedného klastra dát. Jednotlivé atribúty centroidu sa často počítajú ako aritmetický priemer zo všetkých hodnôt dát pre danú dimenziu, ktoré spadajú pod daný centroid, alebo pomocou iných odhadov stredných hodnôt.

Algoritmu k-means sa skladá zo štyroch fáz :

1. Inicializácia vstupných centroidov
2. Pre všetky vzory dát nájsť najbližší centroid
3. Prepočítanie nových *k* centroidov.
4. Opakovanie kroku 2. 3. pokiaľ sa pozície centroidov nemenia.

Vysvetlenie jednotlivých krokov.

Prvý krok je inicializovanie vstupných centroidov. Inicializácia je výsledkom buď určenia *k* náhodných centier, alebo vybratia z dát *k* vzoriek, o ktorých si myslíme, že by mohli patriť do odlišných tried dát. *K* náhodných centroidov sa vyberá priamo z dát. Tento krok je veľmi dôležitý, keďže od počiatočnej inicializácie centroidov sa odvíja celková pozícia výsledných centroidov. Pre tento fakt sa *k-means* algoritmus zvykne opakovať niekoľkokrát s tým, že počiatočne centroidy sa vždy zvolia ako nová náhodná množina. Na konci takéhoto opakovania sa vyberie pokus, ktorý mal veľkosť výslednej chyby

najmenšiu. Veľkosť chyby sa väčšinou ráta ako súčet vzdialenosti všetkých dát od ich najbližšieho centroidu. Pri meraní vzdialeností dát od centroidov sa využívajú rôzne funkcie na meranie vzdialeností, ako boli popísane v 1.3.

Druhý krok je priradiť každému vzoru dát jeho výsledný centroid. Víťazný centroid je taký centroid, ktorý ma najmenšiu vzdialenosť od daného vzoru dát. Opäť je možné určiť, aká funkcia bude použitá pre výpočet vzdialenosti medzi centroidmi a vzormi dát. Pri tomto kroku si môžeme všimnúť, že jeho priebeh sa dá paralelizovať. Ak si uvedomíme, že pre každý vzor postupne spočítame vzdialenosti medzi každým z *k* centroidov, tak nám stačí rozdeliť dáta na *x* skupín, ktorá bude každá zvlášť počítaná na samostatnom výpočtovom zariadení. Keďže výpočet jednotlivých centroidov k vzorom dát nie je zvislý na predchádzajúcich výpočtoch, ale len na parametroch jednotlivých centroidov. Stačí nám zabezpečiť len to, aby každý server, ktorý počíta druhý krok obsahoval *k* aktuálnych centroidov s ich parametrami. Ak zabezpečíme túto podmienku, môžeme napísať map-reduce program, ktorý túto myšlienku premení na algoritmus vykonávajúci sa paralérne. Viac o implementácií paralérneho k-mean algoritmu v 5.4.

Tretí krok súvisí s výpočtom nových centroidov, ktoré budú použite pri behu druhého kroku ďalšej epochy algoritmu. V tomto kroku má už každý vzor dát pridelený jeho najbližší centroid. Ako ďalší krok je spočítanie priemerov pre každý atribút všetkých centroidov. Takto spriemerované atribúty sa stanú novými centroidmi ďalšej epochy *k-means* algoritmu. Týmto vlastne dosiahneme to, že každý centroid sa presunie do stredu všetkých vzorov dát, ktoré k danému cetroidu patria.

Štvrtý krok súvisí s opakovaním druhého a tretieho kroku. V praxi sa používajú dve podmienky, ktoré zabezpečia, aby k-means algoritmus nebežal donekonečna. Prvá z nich je nastavenie počtu epoch, po ktorých budeme považovať naposledy vypočítané centroidy ako výsledne. Ďalším je celková vzdialenosť, o ktorú sa nové centroidy posunú od centroidov, ktoré boli použité v predchádzajúcej epoche. Ak súčet týchto vzdialeností nepresiahne stanovenú hranicu, ktorú sme určili, tak naposledy vypočítane centroidy budeme považovať za výsledné.

V tomto momente už máme vypočítané výsledné centroidy, no ako sme spomínali v sekcií, kde bol popisovaný priebeh výpočtu prvého kroku, tak parametre výsledných

centroidov súvisia s počiatočnou inicializáciou centroidov. Výpočet k-means sa zvykne opakovať niekoľkokrát pričom po každom dobehnutí si uchováme výsledné centroidy a ich chybu. Po rozumnom počte opakovaní si za výsledné centroidy zvolíme také, ktoré patrili do pokusu s najmenšou výslednou chybou. Výsledná chyba sa zvykne počítať ako súčet vzdialeností vzorov dát od ich najbližších centroidov. Dáta, ktoré sa použijú pre rátanie chyby by mali byť odlišné od dát, ktoré sa použili na trénovanie centroidov. Preto sa zvyknú dáta rozdeľovať na dve skupiny. Trénovaciu a testovaciu množninu.Tieto množiny bývajú väčšinou v pomere 70:30, kde na sedemdesiatich percentách dát náš model natrénujeme a na tridsiatich percentách ho otestujeme. Chyba sa dá opäť rátať za použitia rôznych parametrov. V praxi sa používajú zložitejšie techniky na trénovanie a testovanie modelu, napríklad k-fold metóda [13], ktorá rozdelí dáta na k množín, pričom *k*-1 množín je použitých na trénovanie a jedna na testovanie.

Keď už máme výsledné centroidy, môžeme začať detegovať anomálie. Detekcia anomálií založená na *k-means* sa skladá z dvoch častí. Prvou je vypočítanie výsledných centroidov, ktorá bola popísaná vyššie, druhou časťou je celková detekcia anomálií. Pri detekcií anomálií za pomoci centroidov sa vychádza z pozorovania, že normálne dáta ležia v blízkosti ich centroidov a anomálne dáta sa nachádzajú ďalej od centroidov. Toto pozorovanie nemusí byť vždy správne, ako bude popísane v 5.5. Detektor anomálií, ktorý bude použitý na detegovanie anomálií pre nové vzory dát spraví nasledujúce akcie:

- Zmeria vzdialenosť medzi novým vzorom dát a výslednými centroidmi
- Vyberie centroid s najmenšou vzdialenosťou
- Ak výsledná vzdialenosť presiahne stanovenú hranicu, tak daný vzor dát budeme považovať za anomáliu

Hranicu, ktorá je použitá na oddelenie normálnych dát a anomálií je veľmi ťažké určiť. Zvykne sa používať *tri sigma pravidlo*, ktoré sa počíta nasledovne. Zoberieme si všetky trénovacie dáta a vypočítame vzdialenosti od ich najbližších výsledných centroidov. Pre vypočítané vzdialenosti spočítame priemer a štandardnú odchylku. Keď máme vypočítané tieto hodnoty, môžeme na nový vzor dát aplikovať *tri-sigma pravidlo*. Existujú aj iné algoritmy na určovanie vychyľujúcich sa hodnôt, ale vybraním najlepšej takejto metódy sa v tejto práci zaoberať nebudeme a zobrali sme pre našu potrebu už spomínane *tri sigma pravidlo.*

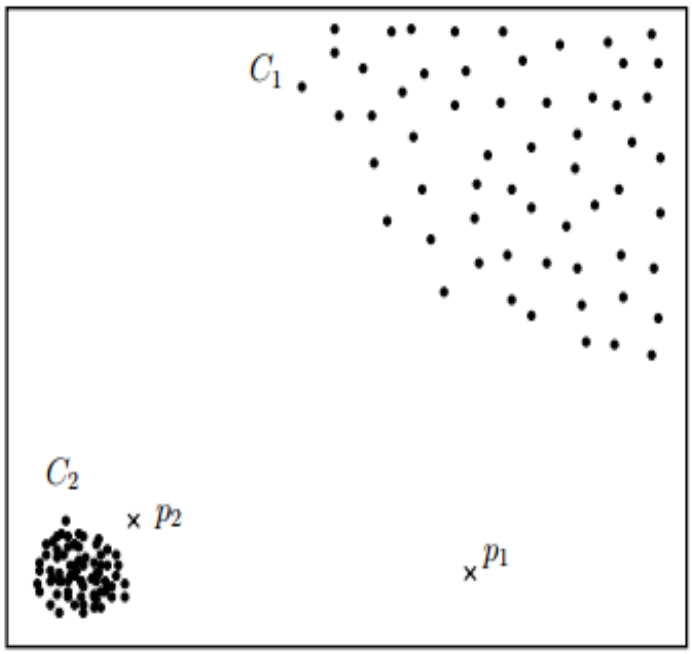
```
mu  = priemer dát
std = štandardná odchylka dát
IF abs(x-mu) > 3*std  THEN  x je anomália
```

Takýmto spôsobom môžeme detegovať anomálie za pomoci *k-means* algoritmu. Je to veľmi jednoduchý prístup, ktorý má svoje nedostatky. Ďalej sa zvykne využívať striktne stanovená hranica, ktorá býva určená ako *x*-tý najvzdialenejší bod všetkých dát od prislúchajúcich centroidov. Aj tu ide o celkom jednoduchý pohľad na problematiku. Vylepšovaniu týchto metód sa budeme venovať v časti 5.5.

2.2 Relative Density

Do tejto množiny algoritmov patria algoritmy založené na meraní hustoty dát. Na základe hustoty okolia daného vzoru dát sa určí „skóre hustoty“ (density score), ktoré vyjadruje, aká veľká hustota dát je v okolí daného bodu. Tieto algoritmy vychádzajú z pozorovania, že normálne dáta majú skóre veľké a anomálne dáta majú skóre hustoty malé. Toto skóre sa dá počítať viacerými spôsobmi. Najčastejšie sa počíta zo vzdialenosti od *k*-teho najbližšieho vzoru dát od pozorovaného vzoru dát. Opäť si môžeme všimnúť problém spojený s určením hranice, ktorá nám oddelí normálne dáta od anomálnych. Tento problém sa dá riešiť podobne, ako pri algoritme k-means popísaný v 2.1.

Obrázok 2 : Detekcia anomálií založená na relatívnej hustote, Zdroj : [2]



Ak bližšie preskúmame obrázku číslo 2, môžeme si všimnúť ďalší problém. Tentokrát spojený s rozdielnou hustotou normálnych klastrov C1 a C2. Algoritmy založené len na

celkovej hustote by s najväčšou pravdepodobnosťou určilo celý klaster C2 za anomálny. Preto sa zvyknú využívať aj metódy, ktoré berú do úvahy aj lokálnu hustotu *k* najbližších vzorov dát. Viac o týchto metódach v [6].

2.3 Rozhodovacie stromy

Ďalšou metódou, ktorá sa využíva na detekciu anomálií sú rozhodovacie stromy. Pri tejto metóde si je treba uvedomiť, že ide o metódu založenú na učení s učiteľom, čo v praxi znamená, že je nutné disponovať trénovacou množinou, o ktorej dátach vieme povedať, do ktorej triedy patria. Najväčšou výhodou rozhodovacích stromov je ich dobrá schopnosť pracovať aj s nenormalizovanými dátami, čím sa vyhneme provodom kategorických atribútov 1.4. Detekcia je založená na vytvorení rozhodovacích pravidiel, podľa ktorých sa nové dáta rozdeľujú na *anomálie*, alebo *normálne*.

3 Distribuované prostredie

Dáta sú nový olej digitálnej ekonomiky a ich veľkosť sa neustále zväčšuje. Každých osemnásť mesiacov sa ich veľkosť zdvojnásobuje, čo má za dôsledok nutnosť vytvárať nové technológie na ich uchovávanie a spracúvanie. Facebook denne vyprodukuje cez 60TB logov, Large Synoptic Survey Telescope vyprodukuje až 30TB dát každú noc. Takéto množstvá dát je nemožné uchovávať a spracúvať bežnými spôsobmi. Dáta takéhoto množstva musíme spracúvať paralélne, pričom nesmieme zabúdať aj na poruchy hardvéru, ktoré môžu nastať pri ich spracúvaní a uchovávaní. Práve na dáta takýchto rozmerov sa hodia technológie, založené na technológií Hadoop a jeho ekosystéme, ktorá sa snaží prácu s veľkými dátami uľahčiť.

3.1 Apache Hadoop

Apache Hadoop je distribuovaný framework na prácu s veľkými dátami. Hadoop projekt vznikol v roku 2006. Jeho jadro je tvorené distribuovaným súborovým systémom a programovým modelom pre spracovanie dát v distribuovanom prostredí. Obom technológiám je venovaná samostatná podkapitola, kde sú popísané jednotlivé časti detailnejšie.

Slovo „hadoop“ nesúvisí len s distribuovaným framevorkom, ale taktiež s Hadoop ekosystémom, ktorý združuje technológie zaoberajúce sa efektívnym spracúvaním a uchovávaním veľkých objemov dát. Hadoop ekosystém sa skladá z množstva technológií, niektoré slúžiace na uchovávanie dát, ako napríklad Hive alebo Impala, iné na uľahčenie práce písania zložitých map-reduce programov, ako napríklad Pig alebo Spark. Všetky tieto technológie spája spoločná vlastnosť a to uľahčiť prácu ľudom s veľkými dátami. Pokiaľ ste chceli v začiatkoch hadoopu napísať jednoduchý map-reduce program na počítanie výskytu jednotlivých slov v texte, tak tento program mohol obsahovať aj niekoľko desiatok až stoviek riadkov kódu. S príchodom aplikácií ako je napríklad *Pig* sa množstvo kódu niekoľkonásobne zredukovalo. Hadoop ekosystém má za úlohu sprístupniť pre spracovanie a uchovávanie obrovské množstvá dát, ktoré by nebolo možné spracúvať bežnými technológiami založenými na výpočtovom výkone jedného stroja. Výhodou hadoopu je jeho horizontálna škálovateľnosť a možnosť využitia komoditného hardvéru pre potreby spracovania a uchovávania dát. To v praxi znamená, že nemusíte

platiť za prémiový hardvér pri nutnosti zvýšiť výkon systému. Horizontálna škálovateľnosť znamená, že ak potrebujete zvýšiť výkon alebo úložnú kapacitu clustra pridáte len ďalší server. Takýmto spôsobom môžeme začať s hadoop clustrom o veľkosti pár jednotiek serverov a skončiť s inštanciou o počte niekoľko tisíc serverov.

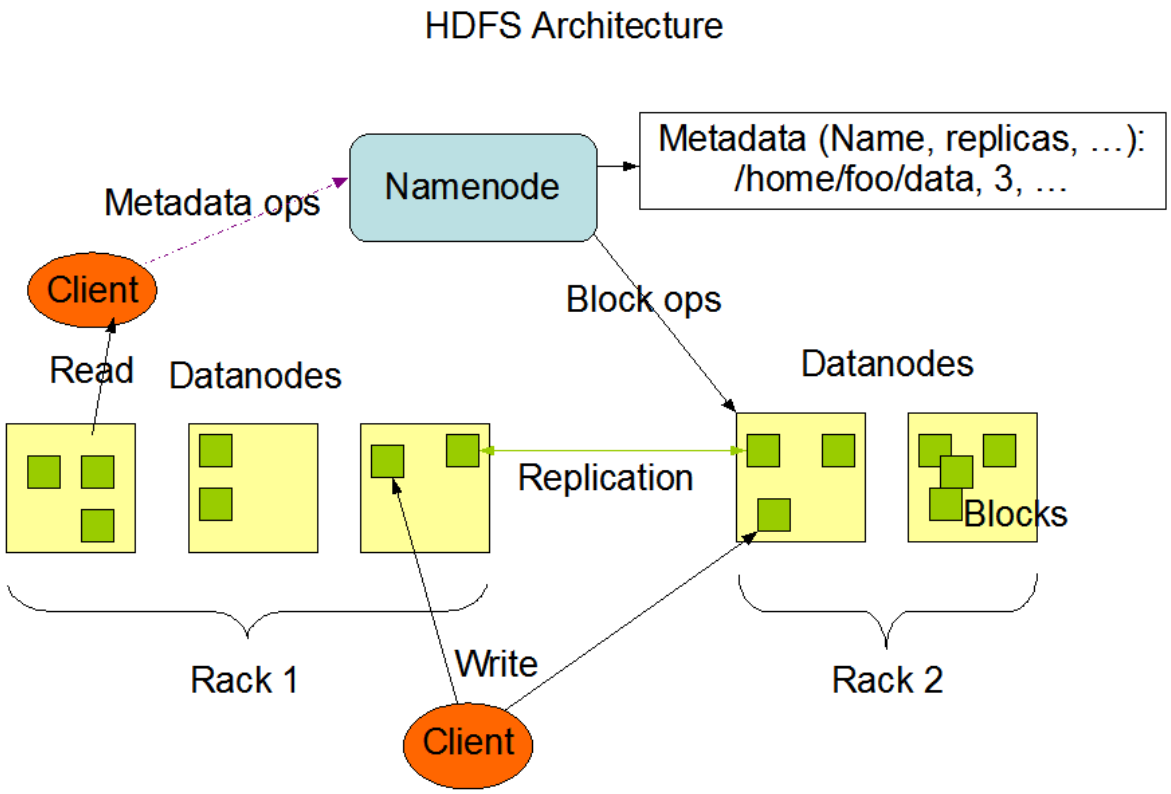
3.1.1 HDFS

HDFS [7] (Hadoop distributed file system) je distribuovaný súborový systém, ktorý tvorí základnú stavebnú jednotku Hadoop ekosystému. Všetky časti Hadoop ekosystému sú nejakým spôsobom prepojené s týmto distribuovaným súborovým systémom. Napríklad Hive databáza ukladá všetky dáta a metadáta práve na HDFS. Ak pracujeme so Sparkom, tak zdroj dát, ktoré budeme analyzovať budú pochádzať pravdepodobne z tohto súborového systému. Na rozdiel od klasických súborových systémov, ako poznáme z prostredia Linuxu a Windowsu spadá HDFS do množinu distribuovaných súborových systémov. Ako aj iné súborové systémy, tak aj HDFS podporuje bežné operácie so súbormi a priečinkami, ako je napríklad vytváranie priečinkov a súborov, mazanie, kopírovanie, ... Väčšina distribuovaných súborových systémov potrebuje na svoju prevádzku prémiový hardware, aby zabezpečila bezchybné fungovanie aj pri zlyhaní niektorého z diskov. Možné riešenie problému so zlyhaním diskov je napríklad použitie RAIDu(redundant array of independent disks) v nastavení, kde sú všetky dáta redundantne uložene na dvoch alebo viacerých diskoch. HDFS bol zo začiatku vytváraný s myšlienkou, že na svoju prevádzku nebude využívať prémiový hardvér ale vystačí si s komoditným hardvérom. HDFS podporuje redundantnosť dát, čím zabezpečuje dostupnosť všetkých dát aj v prípade zlyhania niektorých z diskov, poprípade celých serverov. HDFS ukladá všetky dáta do blokov dát zväčša o veľkosti 64MB, čim je umožnená ľahšia manipulácia so súbormi, napríklad ak treba vyvážiť dáta na jednotlivých DataNodoch. Takéto bloky sa ďalej replikujú a ukladajú na rôzne disky a lokality. Ak náš HDFS obsahuje viac ako jeden rack, je možné dáta fyzicky rozdeliť tak, aby sme sa vyhli situácií, keď budú všetky repliky jedného bloku dát fyzicky uložené v jednom racku.

HDFS sa skladá z dvoch častí. Prvá z nich je NameNode a druhá DataNode. DataNode je komponent, na ktorom sa ukladajú jednotlivé bloky dát. Keďže o redundanciu dát sa stará samotný HDFS bývajú disky v DataNodoch zväčša nakonfigurovaných pomocou architektúry JBOD (just a bunch of disk) [15]. NameNode je centrálny bod HDFS. Je zodpovedný za orchestráciu a replikovanie jednotlivých blokov dát, ako aj za

riadenie DataNodov. NameNode rozhoduje, na ktorom z DataNodov budú uložené jednotlivé bloky dát. Viac informácií o HDFS [7].

Obrázok 3 : HDFS architektúra, Zdroj : http://hadoop.apache.org/docs/r2.4.1/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html



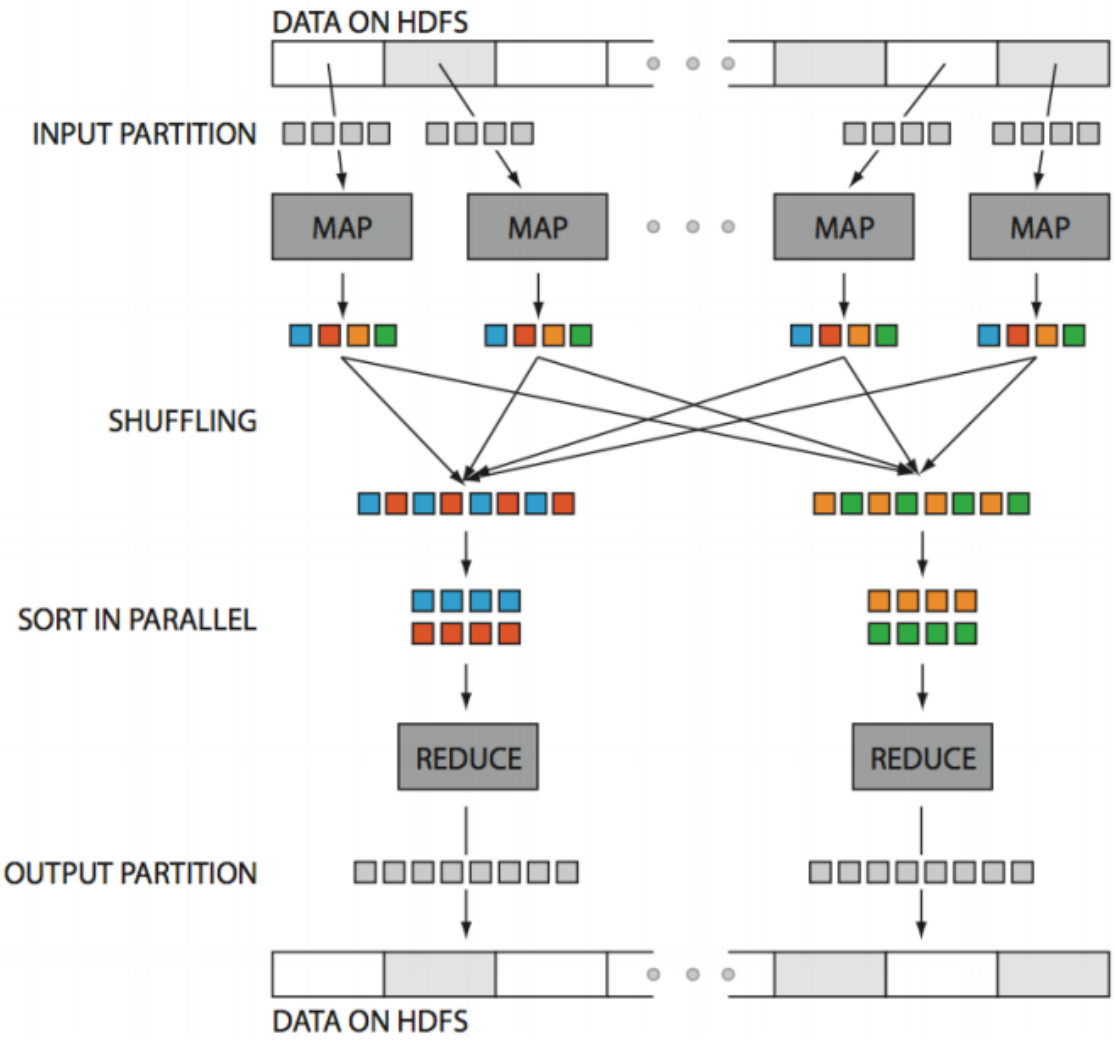
3.1.2 MapReduce

Postupné znižovanie cien diskov vedie k možnostiam ukladať neustále viac dát. Problém nastáva, keď chceme zozbierané dáta spracovať. Prečítanie 1TB uložených dát na klasickom hard disku trvá tri hodiny. Ak si zoberieme len objem dát, ktoré vyprodukuje Large Synoptic Survey Telescope za jednu noc, tak ich prečítanie by trvalo deväťdesiat hodín. Už pri práci s dátami o veľkosti niekoľko GB nám klasické spôsoby na spracovanie dát nestačia. Práve tu sa dá využiť myšlienka programového modelu MapReduce.

MapReduce je programový model a framework pre paralérne spracovanie veľkých objemov dát v distribuovanom prostredí. Oficiálna história MapReducu sa začala písať v roku 2004 [5]. Vývoj bol inšpirovaný funkciami *map* a *reduce* z prostredia

funkcionálnych programovacích jazykov. MapReduce sa skladá z troch hlavných častí: a.)Map, b.)Shuffle, c.) Reduce

Obrázok 4 : MapReduce diagram, Zdroj : Huy Vo, NYU Poly



V prvej fáze distribuovaného spracovania dát sa dáta rozdelia na *x* častí. Každá z *x* častí sa pošle na spracovanie do jedného z *„mapprov“*. To znamená, že ak dáta rozdelíme na *x* časí, tak každá časť dát bude spracovaná samostatne. Každý *mapper* ďalej spracúva svoju časť dát pomocou dopredu definovanej funkcie *f*. Funkcia *f* berie na vstup vstupné dáta, ktoré sú určené pre daný *mapper* a transformuje ich do <kľúč, hodnota> párov, ktoré sa následne posielajú do fáze shuffle. Napríklad pri počítaní výskytu jednotlivých slov by boli vstupné dáta pre jeden *mapper* časť dát z celkového textu a výstup by bol tvorený postupnosťou <kľúč, hodnota> párov. Kľúče by boli tvorené všetkými slovami, ktoré daný

mapper spracoval. Následné hodnoty pre každý kľúč by boli celkový počet výskytov slova tvoreného kľúč v spracovanom sete dát pre *mapper*.

Ďalšia fáza je *shuffle*. Táto fáza zabezpečuje aby sa dáta, ktoré vygenerujú jednotlivé *mappre* dostali na spracovanie do *reducerov*, ktoré sú zodpovedné za spracovanie daných kľúčov.

Posledná fáza je *reduce*, ktorá má za úlohu zredukovať dáta, ktoré vyprodukovali jednotlivé *mappre*. Operácia *shuffle* sa postarala o to, aby každý *reducer* dostal na spracovanie len dáta ku kľúču, ktorý spracúva. Na hodnoty, ktoré dostane na vstup daný reducer sa aplikuje funkcia *f'*. Výsledkom reduce operácie býva väčšinou <kľúč, hodnota'> pár. V prípade počítania slov by funkcia *f'* spočítala súčet všetkých hodnôt k danému kľúču a výsledok by uložila. MapReduce sa dá využiť aj na násobenie veľkých matíc alebo spájanie veľkých tabuliek. Pri spájaní tabuliek *A* a *B* by boli jednotlivé kľúče tvorené hodnotami stĺpca, podľa ktorého budú tabuľky prepojené a hodnota v pároch by obsahovali všetky hodnoty daného záznamu. Pri hodnote si musíme ešte pamätať, aj z ktorej tabuľky pochádza. *Map* fáza by potom generovala kľúče vo formáte <kľúč,(hodnota, tabuľka)>. *Reduce* operácia by následne dostala na vstup všetky riadky z tabuľky A aj B, pre kľúč_1. Následne by tieto dáta spojila pričom by využila poznatok o zdrojovej tabuľke(nespájala by hodnoty z tej istej tabuľky).

Častým a zároveň prirodzeným problémom spracovania dát v distribuovanom prostredí, tvorenom desiatkami až tisíckami serverov, sú výpadky jedného alebo viacerých serverov z clustra. V Hadoop ekosystéme, kde zastávajú jednotlivé uzly v clustry dôležité úlohy nielen v procese spracovania MapReduce úloh, ale aj ako uložisko dát, je implementácia tolerancie výpadkov nevyhnutná. Ak dôjde počas akejkoľvek výpočtovej operácie k výpadku servera, Hadoop ekosystém automaticky zareaguje na tento problém. Spadnuté úlohy a potrebné dáta rovnomerne rozdistribuuje medzi funkčné uzly v clustry. Väčší problém ako výpadok servera je dlhá latencia. V tomto prípade je ťažké určiť, či pomalá odozva uzla je spôsobená objektívnymi faktormi distribuovaného prostredia, alebo náročnosťou spracovanej úlohy. Platforma Hadoop vie však odhaliť i detegovať o aký typ problému sa jedna a následne zareagovať.

3.2 Cloudera

Hadoop cluster sa skladá z desiatok aplikácií a serverov, ktoré treba riadiť a monitorovať. Každá z týchto aplikácií ma konfiguračné súbory a pri každej zmene konfigurácie clustra treba na všetkých nodoch prekonfigurovať a reštartovať ovplyvnené aplikácie. Napríklad ak pridávame alebo premiestňujeme aplikáciu Hive Metastore na novú lokalitu, tak treba zmeniť konfigurácie a následne reštartovať aplikácie Hive, Hue, a Impala. Pri malých clustroch o velkosti pár serverov je možné takého zmeny robiť aj ručne, no pri clustorch o väčších rozmeroch by bolo manuálne riešenie zmien konfigurácií nereálne. Tu sa uplatňujú Hadoop distribúcie, ktoré prinášajú pohodlnú konfiguráciu Hadoop clustrov aj o veľkosti stoviek serverov. V praxi sú používané najmä tieto tri distribúcie :

1. Cloudera
2. MapR
3. Hortonworks

Všetky tieto distribúcie sa nejakým spôsobom odlišujú. Napríklad Cloudera distribúcia ponúka projekt Impala, ktorý slúži na distribuované spracovanie a dotazovanie nad štruktúrovanými dátami pomocou klasického SQL jazyka. Impala sa snaží nahradiť do značnej miery projekt Hive, keďže najväčšou slabinou Hivu je práve jeho rýchlosť, z dôvodu, že všetky SQL dotazy prekladá do map-reduce úloh. Impala tu do značnej miery ťaží z využívania in-memory technológií pre ukladanie dát. MapR zas prichádza s vlastným distribuovaným súborovým systémom MapR-FS [10].

V našej bakalárske práci sme pracovali s Cloudera distribúciou, ktorá ponúka prehľadné monitorovanie a riadenie Hadoop clustra.

Jednotlivé distribúcie majú aj nevýhody, napríklad nutnosť čakať na integráciu s najnovšími aplikáciami. Ako príklad môžeme uviesť projekt Apache Spark, ktorého momentálna verzia je 1.6.1, ale Cloudera distribúcia zatiaľ podporuje len verziu 1.5.0.

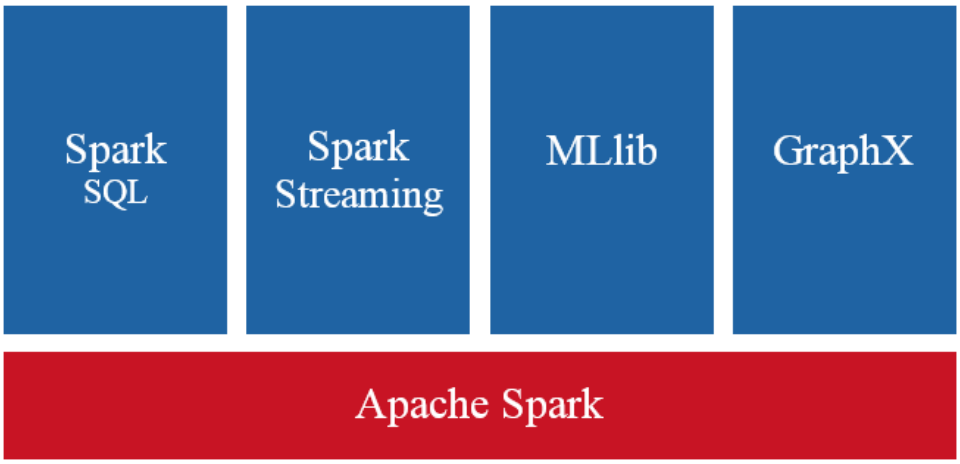
3.3 Apache Spark

Apache Spark [16] je distribuovaný nástroj pre prácu s veľkým objemom dát. Za jeho vytvorením stojí Matei Zaharia, ktorý Spark vytvoril v roku 2009 na UC Berkeley's. V roku 2013 prešiel pod Apache licenciu a od tej doby sa stal jedným z najaktívnejších

Apache projektov. Najväčšou výhodou Sparku je jeho horizontálna škálovateľnosť a možnosť pracovať v in-memory režime, čo v praxi znamená, že jednotlivé výsledky map-reduce operácií nebudú ukladané na disk, ale priamo do operačnej pamäte serverov, čím dokáže dosahovať až 100 násobné zvýšenie výkonu oproti klasickej implementácií MapReduce. Túto výhodu využívajú najmä iteratívne algoritmy z prostredia strojového učenia, pri ktorej sa tie isté dáta spracúvajú niekoľkokrát. No aj pri použití klasických diskov sa dosahuje 10 násobné zvýšenie rýchlosti výpočtu. Spark obdržal aj rekord v zotriedení 100TB dát v čase 23 minút za použitia 206 serverov bez použitia in-memory technológie [8]. Pre porovnanie, predchádzajúci rekord držal Hadoop MapReduce cluster o veľkosti 2100 serverov, ktorý dáta zotriedil za 72 minút.

Jadro Sparku je tvorené štyrmi základnými knižnicami : Spar SQL, Spark Streaming, MLlib, GraphX.

Obrázok 5 : Jadro Apache Spark



Spark SQL je knižnica na prácu so štrukturovanými dátami. Jej súčasťou sú konektory na rôzne databázy, či už z relačného sveta alebo sveta NoSQL databáz. Spark SQL prináša do Sparku možnosť písania dotazov nad dátami v SQL jazyku. Vo výsledku sa SQL dotazy prepíšu do map-reduce programov. O ich výpočet sa už postará Spark. Tu je možné kombinovať a prepájať rôzne zdrojové systémy. Môžeme teda spojiť dve tabuľky, pričom každá bude pochádzať z úplne iného systému.

Spark Streaming je knižnica, ktorá dovoľuje spracovať a analyzovať dáta v reálnom čase. Vstupné dáta, ktoré vchádzajú na spracovanie do Spark Streamingu sa rozdelia na snímky zvané DStreami. Každý snímok obsahuje dáta za určené časové obdobie. Nad

DStreamami môžeme následne robiť analýzy a agregácie, pričom je možné využiť *akcie* a *transformacie* popísané v 3.3.1. Jednotlivé výsledky môžeme ukladať na rôzne úložiská, alebo ich ďalej spracúvať. Ďalším príkladom použitia je detekcia podvodov a anomálií. Kde si na začiatku Spark Streaming aplikácie natrénujeme model, na základe ktorého budeme prichádzajúce dáta filtrovať. Momentálne má knižnica MLlib implementovaný aj *Streaming k-means* algoritmus, ktorý sa dá trénovať aj dynamicky. S touto knižnicou sme nepracovali pri vytváraní tejto bakalárskej práce. Typické zdroje dát pre Spark Streaming sú napríklad Apache Kafka, Apache Flume, a TCP.

MLlib je knižnica, ktorá združuje distribuované algoritmy pre strojové učenie a štatistiku. Vzhľadom na to, že väčšina algoritmy pre strojové učenie je založená na princípe iteratívneho prechádzania tej istej dátovej štruktúry, môžu tieto algoritmy ťažiť zo schopnosti Sparku uloženia všetkých dát do operačnej pamäte serverov. Čím sa vyhýbajú nutnosti dáta znovu načítavať z diskov. Knižnica obsahuje implementácie mnohých algoritmy pre účely klasifikácie, regresie a klastrovania. Má zakomponované algoritmy pre delenie dát na trénovacie a testovacie množiny, ako aj implementáciu cross-validácie. Túto knižnicu sme využili pri písaní tejto bakalárskej práce.

GraphX je knižnica obsahujúca množstvo distribuovaných algoritmov pre prácu s grafmi.

Spark programy môžu byť napísanie v štyroch programovacích jazykoch. Patria medzi ne Python, Java, Scala a obľúbený programovací jazyk štatistikov R. Spark je napísaný v Scale, čo je objektovo orientovaný funkcionálny jazyk založený na Jave. Z našej skúsenosti odporúčame písať Spark aplikácie v Scale, keďže sa Vám môže stať, že funkcie a knižnice, ktoré sú naimplementované pre Scalu nebudú ešte dostupné pre iné programovacie jazyky. Keďže Scala vychádza z Javy, je možné v Scala programoch využívať funkcie a knižnice naprogramované v Jave. Pri účely vytvorenia tejto bakalárskej práce sme zvolili programovací jazyk Scala. Hlavným dôvodom bola veľká expresívna sila jazyku, ako aj možnosť využitia Spark Shellu 3.3.2.

3.3.1 RDD

Hlavnou abstrakciou ktorú Spark ponúka je RDD (*resilient distributed dataset*), čo si vlastne môžeme predstaviť ako distribuovanú množinu dát, nad ktorou sa vykonávajú map-reduce operácie. Paralelne operácie ktoré nad ním pracujú sú odolné voči výpadkom

serverov. Keďže RDD je nemeniteľná štruktúra (immutable) je možne túto štruktúru len transformovať na iné RDD alebo na danej štruktúre vykonať nejakú *akciu*. Operácie nad RDD sa rozdeľujú do dvoch skupín. Prvá z nich sú *transformácie*, ktorej výsledkom je nove RDD. To je generované za použitia funkcie *f* , ktorá sa aplikuje na každý element štruktúry. Medzi *transformácie* patria funkcie ako *map, filter, flatmap, ...* . *Akcie* tvoria funkcie, ktoré vracajú nejaký napočítaný výsledok z RDD na ktorom bola akcia vykonaná. Ide napríklad o funkcie *count, collect, take, avg, min,...* . Jeden z najväčším rozdielov medzi akciami a transformáciami je takzvane lenivé vyhodnocovanie (lazy evaluation) transformácií. Čo znamená, že transformácia sa nezačne vykonávať hneď po zavolaní, ako sme na to zvyknutý z iných programovacích jazykov, ale až po zavolaní *akcie* na dané RDD.

3.3.2 Spark Shell

Pre nás je jednou z najväčších výhodou Sparku oproti iným technológiám zaoberajúcich sa spracovaním veľkých dát Spark Shell. Spark Shell je interaktívna konzola, ktorá Vám dovoľuje preskúmavať dáta pomocou Spark API bez nutnosti kompilácie kódu. Spark Shell pripomína napríklad Python REPL(read, eval, print, loop), až na fakt, že všetky funkcie a volania, ktoré zavoláte v Spark Shell budú vykonávané na všetkých serveroch distribuovane. Spark Shell je možné využiť s programovacími jazykmi Python, Scala a R.

4 KDD cup 1999

KDD cup [11] je súťaž organizovaná každoročne skupinou ACM. Každý rok je zverejnený problém z oblasti strojového učenia spolu s dátami, ktorý je riešený odborníkmi a skupinami z oblasti strojového učenia. V roku 1999 bola KDD súťaž zameraná na detekciu narušenia počítačovej siete. Spolu so zadaním boli zverejnené aj dáta, ktoré sú k dispozícií dodnes. Tieto dáta sme použili pri trénovaní a testovaní algoritmov v našej práci.

Dáta sú už predspracované organizátormi. Ich veľkosť je 709MB a obsahuje záznamy približne o 4,9 miliónov spojení. Každé spojenie je napočítané zo sieťových dát. Výsledné záznamy obsahujúce 41 atribútov, napríklad veľkosť odoslaných dát, počet pokusov o prihlásenie, počet TCP chýb a ďalšie. Celkový prehľaď atribútov je možne nájsť na stránke datasetu [12]. Ďalšou výhodou týchto dát je vlastnosť, že o každom zázname dokážeme povedať, do akej triedy dát patrí. Či ide o normálne sieťové spojenie, alebo o spojenie spojené s úlohou narušiť počítačovú sieť. Táto vlastnosť sa nám zíde pri vyhodnocovaní modelov, keďže budeme schopní povedať, ako si vedie model, ktorý sme natrénovali. Dohromady dáta obsahujú 23 rôznych tried.

Príklad jedného záznamu :

„0,tcp,http,SF,181,5450,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,8,8,0.00,0.00,0.00,0.00,1.00,0.00,0.00,9,9,1.00,0.00,0.11,0.00,0.00,0.00,0.00,0.00,normal“

Tento záznam nám hovorí, že ide o *tcp* spojenie na *http* službu. Bolo odoslaných 181 bytov a 5450 bytov bolo prijatých. Tento záznam je označený ako „normálny“. Takže nešlo o žiadnu nežiadúcu aktivitu.

Mnoho z atribútov nadobúda hodnoty z množiny {0,1}, „0“ ak daný atribút nie je prezentovaný „1“ ak je. Napríklad veľmi zaujímavý je atribút „su_attempted“, ktorý nadobúda hodnotu „1“ ak v spojení bol pokus o príkaz ‚su root‘. Atribúty, ktoré nadobúdajú hodnoty z množiny {0,1} sú kategorické. Ako sme spomínali v 1.2 a 1.4, každú kategorické premenne by sme mali preto zakódovať. Napríklad z atribútu „su_attempted“ by sa stali dve nové premenné. Jedna z nich by vždy nadobúdala hodnotu ‘1‘, a druhá ‘0‘ v závislosti od hodnoty vstupnej premennej. No v prípade binárnych

kategorických premenných môžeme tento krok preskočiť. Ak by ale naša kategorická premenná mohla nadobúdať viac hodnôt, pre väčšinu algoritmov by sme ju museli zakódovať. KDD dataset obsahuje tri kategorické atribúty. Toto je aj prípad atribútu ‚service', ktorá môže nadobúdať až 70 odlišných hodnôt, napríklad , *private*','*http*', '*ecr_i*' .. takýto atribút by sme museli zakódovať podľa postupu z 1.4, aby sme ho dokázali použiť pri algoritmoch, ktoré nedokážu pracovať s kategorickými premennými.

5 Implementácia

Obsah tejto kapitoly je zameraný na popis vývojového prostredia, ako aj na celkovú implementáciu algoritmov k-means pre účely detekcie anomálií v distribuovanom prostredí Apache Spark. Ďalej sa v tejto kapitole venujeme porovnaniu vlastnej implementácie k-means algoritmu a implementácie k-means z MLlib knižnice 3.3. Jedna z podkapitol je venovaná výberu hodnoty parametra *k* pre algoritmy k-means.

Na potreby bakalárskej práce sme využívali Hadoop cluster, ktorý sme vlastnoručne nainštalovali a nakonfigurovali. Jeho konfigurácia pozostáva z jedného hlavného servra (MasterNodu) a troch serverov, ktoré sa starajú o výpočtový výkon a úložnú kapacitu (WorkerNodov). MasterNode disponuje 32GB operačnej pamäte a procesorom Intel i5-4460. Každý WorkerNode disponuje 24GB operačnej pamäte a taktiež procesorom Intel i5-4460. Jednotlivé Nody sú prepojené 1Gb/s sieťou. Ako Hadoop distribúciu sme zvolili Clouderu vo verzii Express 5.5.3, ako bolo spomínané v kapitole 3.2, hlavne kvôli predchádzajúcim skúsenostiam s týmto prostredím. V momente písania práce bola najnovšia verzia Sparku 1.6.1. Pre účely práce sme pracovali s verziu 1.5.0, ktorá je súčasťou CDH-5.5.2-1.cdh [9]. Ako programovací jazyk sme zvolili Scalu, hlavne kvôli jej expresívnej sile a podpore Spark Shellu 3.3.2 . Knižnice, ktoré sme využili sú *org.apache.spark.mllib.linalg, org.apache.spark.mllib.clustering*.

Výsledný dataset sme pred spracovaním ešte upravili. Keďže detekcia anomálií založená na algoritme k-means vychádza z pozorovania, že anomálie sú také dáta, ktoré neležia v blízkosti žiadneho z výsledných centroidov a náš dataset obsahoval dve veľké triedy dát, ktoré ovplyvňovali výsledné algoritmy takým spôsobom, že by detekcia anomálií založená na metóde k-means nedávala správne výsledky. Množstvo dát, ktoré obsahovali tieto dve triedy bolo také signifikantné, že naše metódy ich považovali za normálne triedy. Konkrétne išlo o triedy “smurf.“ s 2807886 záznamami v dáta sete a “neptune.“ s 1072017 záznamami. Celkový počet záznamov, s ktorým sme pracovali bol 1018528. Z datasetu sme odstránili aj kategorické premenné, na ktorých prevod sme sa nezamerali. Každé spustenie algoritmov si rozdelilo vstupné dáta na trénovaciu a testoviaciu množinu v pomere70:30. Všetky algoritmy sme trénovali pomocou trénovacej množiny a testovanie prebiehalo na množine testovacej.

5.1 Normalizácia dát

K-means klastrovací algoritmus je náchylný na nenormalizované dáta, keďže počíta vzdialenosť medzi centroidmi a jednotlivými dátami v dáta sete. Aj jeden nenormalizovaný atribút nám môže spôsobiť situáciu, keď len jeho hodnota bude taká signifikantná, že na výpočet celkovej vzdialenosti od centroidov budú ostatné atribúty pôsobiť len minimálne. Niekedy je takýto scenár vítaný, no v našom prípade by napríklad atribút, ktorý hovorí o množstve prenesených bytov, ktorý nadobúda veľké hodnoty pôsobil príznačnejšie na výslednú vzdialenosť v porovnaním s atribútmi, ktoré nadobúdajú hodnoty len z množiny {0,1}. Nasledujúci skript transformuje RDD premennú `data`, ktorá obsahuje dáta typu `Array[Double]` na normalizovaný dáta set, ktorý bude uložený v premennej `normalizedData`. Dáta sme normalizovali podľa štandardného skóre [17] , ktorý ma vzorec $z = \frac{x-\mu}{\sigma}$, kde x zodpovedá normalizovanej premennej, μ je priemer a σ je smerodajná odchylka. V skripte si môžeme všimnúť, akú expresívnu silu má programovací jazyk Scala v spojení so Sparkom. Na niekoľko pár riadkov dokážeme normalizovať dáta, pričom je jedno, či pracujete s dátami o veľkosťou 1GB alebo 10TB. Výsledný skript sa vôbec nemení.

Skript 1 : Skript na normalizáciu dát

```
val dataArr = data.map(x => x.toArray)
val numCol = data.first().toArray.length
val n = data.count()
val attrSum = dataArr.reduce( (a,b) => a.zip(b).map(x => x._1 + x._2)
val attrSumSqua = dataArr.aggregate(
  new Array[Double](numCol)
)(
  (a,b) => a.zip(b).map( x => x._1 + x._2 * x._2),
  (a,b) => a.zip(b).map( x => x._1 + x._2)
)
val  stdev = attrSumSqua.zip(attrSum).map({
  case (sumSq,sum) => math.sqrt(n * sumSq - sum*sum)/n
})
val means = attrSum.map( x => x / n)
```

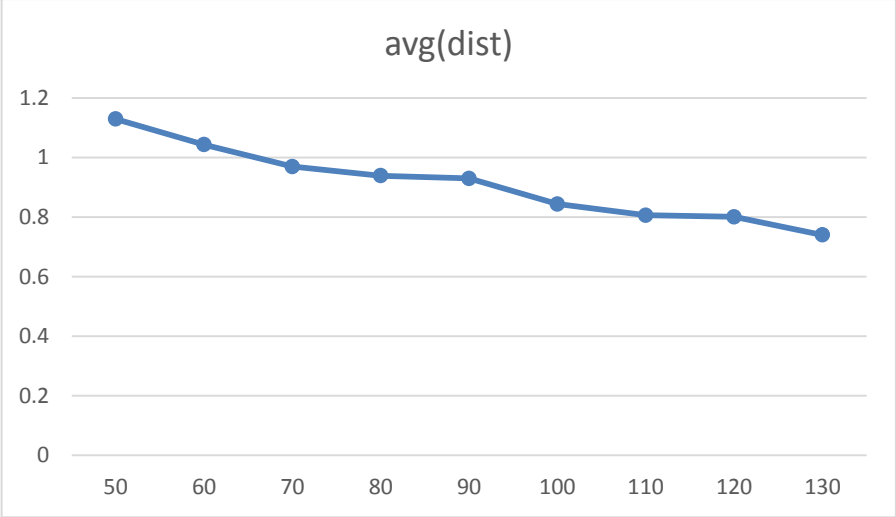
```
def normalizeArr(row : Vector) ={
  val norm = (row.toArray, means, stdev).zipped.map(
  (x, mean, stdev) =>
    if(stdev == 0) (x - mean) else (x - mean) / stdev
  )
  Vectors.dense(norm)
}
val normalizedData = data.map(x => normalizeArr(x)).cache
```

5.2 Výber hodnoty parametra *k*

Výber správnej hodnoty pre parameter *k* je ťažký problém. Problém spočíva vtom, že hodnota parametra *k* nemôže byť ani príliš malá, čo by malo za dôsledok to, že v jednom klastry by sa nám miešali dáta z rôznych tried, no nechceme mať hodnotu parametra *k* ani príliš veľkú, čo by malo za dôsledok, že každá malá skupinka dát by mala vlastný klaster. Kedže vieme, že náše dáta obsahujú jednu normálnu triedu a 20 anomálnych tried po odstránení tried „*smurf*„ a „*neptune*“, tak náš výsledný model by mal obsahovať minimálne 21 klastrov. Túto informáciu pri klastrovacích algoritmoch väčšinou nepoznáme, takže ju budeme ignorovať.

Pre účely výberu hodnoty parametra *k* sme využili implementáciu k-means z knižnice MLlib. Nebola využitá vlastná implementáciu z kapitoly 5.2. Postupovali sme nasledovne. V cykle sme skúšali hodnoty *k* od 50 do 130 s postupným zvyšovaním po 10. Pre každú hodnotu sme nechali vypočítať model so zmeneným parametrom *k*, následne sme pre všetky dáta spočítali priemernú vzdialenosť k najbližším centroidom. Vzdialenosť medzi bodom a najbližším centroidom bola meraná Euklidovou vzdialenosťou. Pri tomto pokuse sme pracovali s normalizovaným dátami.

Graf 1 : Priemerná vzdialenosť dát v závislosti od hodnoty parametra *k*



Graf 1 nám ukazuje, že priemerná vzdialenosť sa s narastajúcim počtom centroidov zmenšuje. Tento fakt je celkom zrejmý, keďže čím viac centroidov pridáme, tým by sa mala zmenšiť vzdialenosť medzi dátami a ich výherným centroidom. Ak by sme za hodnotu parametra *k* zvolili *n* (celkový počet dát) mali by sme dostať priemernú vzdialenosť *0*, keďže každý bod by bol aj svoj vlastný centroid. Z grafu môžeme vidieť, že niekde pri hodnote *k* = 80 sa priemerná vzdialenosť ustálila. Z tohto pozorovania sme usúdili, že pri našej ďalšej práci budeme voliť hodnotu parametra *k* číslo *80*. Po stanovení *k* = *80* sme zneužili fakt, že k dátam poznám aj ich výsledné triedy a vytvorili sme skript, ktorý ku každému centroidu určí, koľko dát z akej výslednej triedy do neho spadá. Nasledujúca tabuľka ukazuje, že výber parametra *k* je uspokojujúci. Tabuľka obsahuje stĺpce *klasterID, trieda a počet* pre šesť výsledných centroidov. Dáta nám hovoria, aké triedy a v akých počtoch boli priradené k jednotlivým klastrom.

Tabuľka 3 : K-means priradené triedy k centroidom

klasterID	trieda	počet
44	normal.	31042
44	portsweep.	1
45	normal.	2137
46	normal.	6613
46	warezclient.	2
47	portsweep.	27
48	imap.	1
48	normal.	4704
48	portsweep.	1
48	rootkit.	1
48	satan.	2
49	ipsweep.	2
49	normal.	1436
49	satan.	2
49	warezclient.	1
50	teardrop.	29
51	back.	124
51	normal.	6153

5.3 K-means detektor

Nasledujúca kapitola popisuje implementáciu detektora anomálií využitím k-means algoritmu implementovaného v MLlib a *pravidla tri sigma* na odhalenie anomálií.

Prvá časť spočíva v normalizovaní vstupného datasetu. Použili sme postup z kapitoly 5.1. Ďalším bodom bolo natrénovať model, ktorý sa použil pri odhaľovaní anomálií. Hodnotu parametra *k* sme nastavili na *80*, čo je hodnota, ktorá vyšla z predchádzajúceho pozorovania 5.2. Ďalším krokom bolo spočítať jednotlivé vzdialenosti dát od ich najbližších centroidov. Ďalej sme museli vypočítať *priemer* a *smerodajnú odchylku* z celkových vzdialeností, aby sme mohli aplikovať *pravidlo tri sigma*. Po

vypočítaní priemeru a smerodajnej odchylky sme dáta transformovali na podobu, ktorá nám hovorila, či dáta prešli *pravidlom tri sigma* alebo nie. Ak dáta neprešli označili sme ich za "anomálie", ak dáta prešli boli označené za "normálne". Na záver sme výstupné dáta vyhodnotili, a tým sme určili kvalitu daného detektora.

Tabuľka 4 : Výsledky k-means - jednoduchá hranica

K-means jednoduchá hranica		
Dáta / **Detektor**	norm.	anom.
norm	289574	12982
anom	2260	741

Tabuľka výsledkov 3 nám hovorí, že nás detektor určil správne 289574 normálnych dát a 741 anomálií. Ďalej náš algoritmus určil 12982 anomálnych dát za dáta normále. Nakoniec nám ukazuje, že detektor zle určili 2260 normálnych dát. Z dát môžeme vidieť, že tento spôsob odhaľovania anomálií sa pokúsil vyhodnotiť za anomálie len 3001 testovacích dát, pričom správne určil iba 741 z celkového počtu anomálií v testovacej množine 13723. Kapitole 5.5 je venovaná zlepšeniu detektora, pričom sa môžeme odraziť od dát, ktoré sme ziskali.

5.4 Vlastná implementácia algoritmu k-means

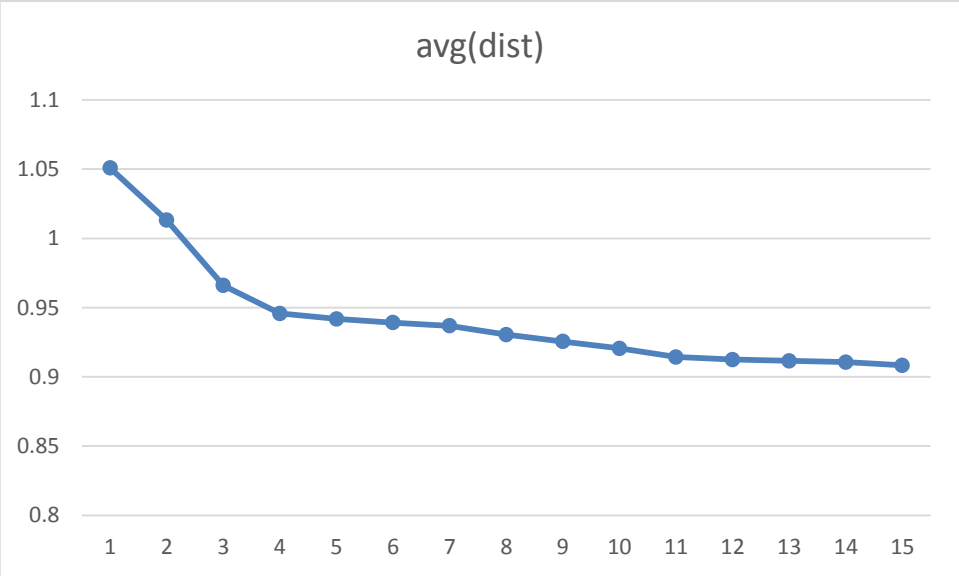
Pre vlastnú implementáciu algoritmu k-means v distribuovanom prostredí Apache Spark sme sa rozhodli hlavne kvôli limitácií voľby vlastnej funkcie pre počítanie vzdialeností medzi dátami a jednotlivými centroidmi v implementácií k-means algoritmu z MLlib knižnice. Implementovať k-means v distribuovanom prostredí sa zdal jednoduchý problém. Veľkosť výsledného kódu by sa dala počítať na desiatky, no najväčší problém s ktorým sme sa stretli bola práve funkcionálna Scala a odhodlanie nepoužiť pre tento výpočet viac ako jeden klasický cyklus. Problém s ktorým sme sa dlhšiu dobu trápili bola inicializácia nových centroidov po zbehnutí jednej epochy algoritmu. V bežnom programovacom jazyku by sme vytvorili kód s pár vnorenými cyklami, a postupne by sme pre každý centroid a jeho všetky atribúty spočítali nové hodnoty. Vo funkcionálnej Scale a distribuovanom Sparku sme nakoniec na tento krok nemusel použiť ani jeden klasický cyklus. Stačili „dva" riadky kódu.

```
+1 var resClusters = data.map( x => ( findCluster( x, centroids ), ( x.toArray,1 ) ) );

+2 centroids = resClusters.reduceByKey( { case ( ( value1, count1 ), ( value2, count2 ) ) => ( value1.zip( value2 ).map( x => x._1 + x._2 ), count1 + count2 ) } ).mapValues( {case ( value, count ) =>  value.map( x => x / count.toDouble ) } ).map( x => Vectors.dense( x._2 ) ).collect()
```

Prvý riadok vypočíta pre všetký dáta ich výsledné centroidy a tak ich priradí do jedného z *x* klastrov. Ďalší riadok slúži na vypočítanie nových centroidov. Výhoda Scaly je jej obrovská expresívna sila, no na druhej strane si je treba uvedomiť, že obyčajné komentáre pri takomto kuse kódu nepomôžu.

Graf 2 : Priebeh chyby v porovnaní s počtom epoch



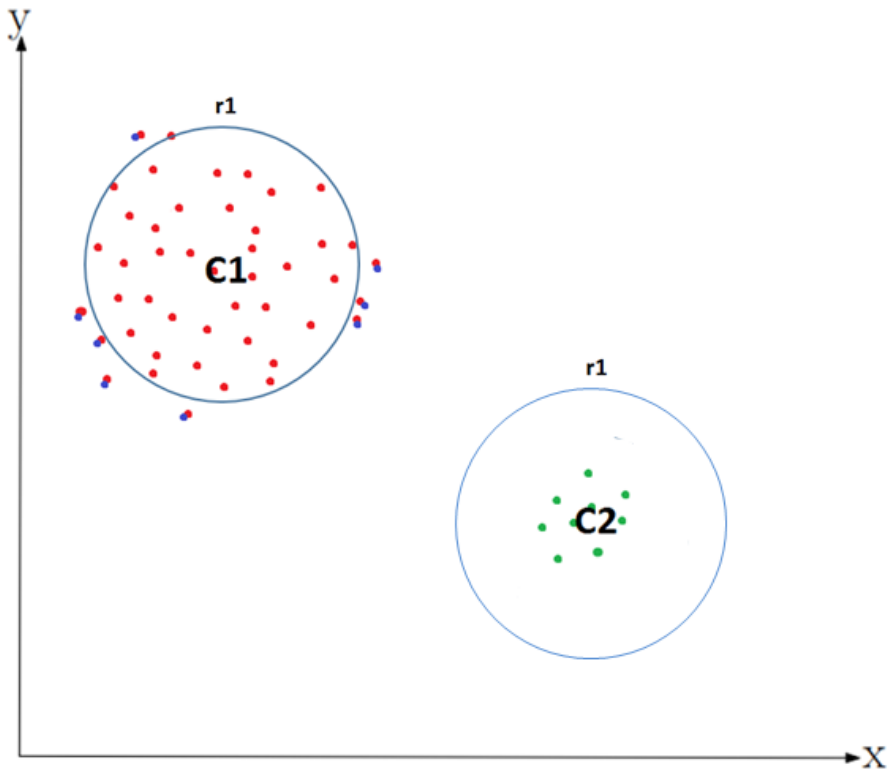
V grafe 2 môžeme pozorovať, postupné znižovanie priemernej vzdialenosti dát od ich výsledných centroidov po každej epoche algoritmu. Z toho môžeme usúdiť, že vlastná implementácia algoritmu *k-means* v distribuovanom prostredí Apache Spark je funkčná.

Pre účely výsledného algoritmu pre detekciu anomálií sme uprednostnili implementáciu *k-means* z knižnice MLlib, hlavne kvôli jej rýchlosti. Nami naimplementovaný k-means bol v priemere 4 krát pomalší ako implementácia z knižnice MLlib. Pri vyhodnocovaní sme oba algoritmy nechali vypočítať výsledné centroidy na normalizovaných dátach. Výsledný čas implementácie z knižnice MLlib bol v priemere 10 minút a naša implementácia dosahovala priemerný čas 40 minút.

5.5 Modifikovaný k-means detektor

Ak si uvedomíme ako fungujú najčastejšie implementácie detekcie anomálií za požitia k-means algoritmu, tak dospejeme k záveru, že určovať si jednoduchú hranicu a podľa nej sa rozhodovať, či dané dáta sú anomálie je dosť naivné.

Obrázok 6 Ukážka problému s využití jednoduchej hranice pri algoritme k-means



Obrázok 8 nám ukazuje možnú situáciu, ako by sa zachovala väčšina algoritmov na detekciu anomálií založených na metóde *k-means* a jednoduchej hranici. V prvom kroku by sme našli dva centroidy *C1* a C2. V ďalšom kroku by sme si určili hranicu, ktorá je znázornená na obrázku ako kruhy *r1*. Následne by sme určili všetky dáta, ktoré sú mimo kruhov za anomálie. Tento princíp je dosť jednoduchý a v mnohých prípadoch sa nám môže stať, že takýto spôsob určovania anomálií bude za anomálie určovať aj dáta, ktoré vôbec nie sú anomálie a naopak, ako sme mohli pozorovať z výsledkov 5.3. Práve preto sme sa rozhodli implementovať aj metódu, ktorá by mala potláčať takéto scenáre.

Hlavnou myšlienkou je nebrať do úvahy vzdialenosť len od jedného výsledného centroidu, ale zobrať do úvahy vzdialeností od *x* centroidov v závislosti od počtu dát, ktoré dané centroidy pokrývajú.

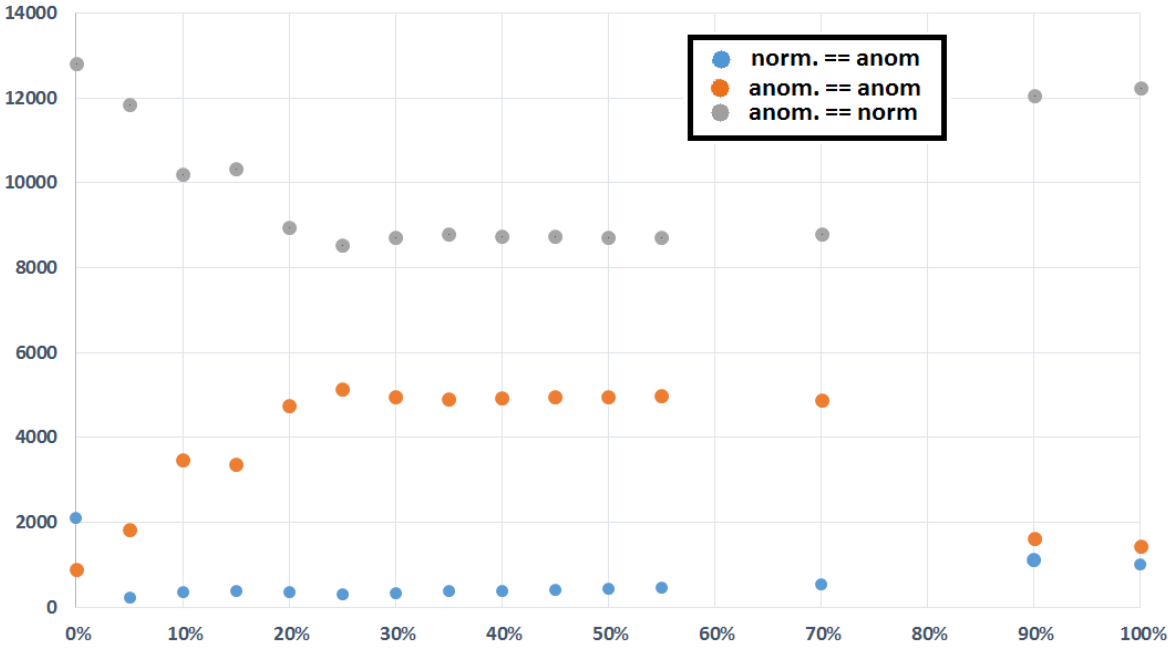
Postup, ktorým sme postupovali :

1. Rozdelili sme množinu dát na trénovaciu a testovaciu v pomere 70:30. Následne sme normalizovali dáta v trénovacej množine a na základe vypočítaných parametrov sme normalizovali aj množinu testovaciu.
2. V prvom kroku modifikovaného algoritmu *k-means* na detekciu anomálií sme natrénovali centroidy klasickým spôsobom s tým rozdielom, že ku každému výslednému centroidu sme si zapamätali aj počet všetkých trénovacích dát, ktoré pod daný centroid patria. V tomto bode už máme konečné hodnoty atribútov pre všetky výsledné centroidy a počty dát, ktoré spadajú pod centroidy.
3. Ďalším krokom bolo pre všetky trénovacie dáta vypočítať vzdialenosti od všetkých výsledných centroidov. Vypočítané vzdialenosti aj spolu s jednotlivými centroidmi sme si usporiadali podľa vzdialenosti vzostupne. Následne sme prešli cez zotriedené pole centroidov pre každý prvok trénovacích dát. Pri každom prechode sme kontrolovali podmienku, či súčet dát, ktoré patria pod doteraz prejdené centroidy nepresiahol hranicu *x* % celkového počtu trénovacích dát. Ak áno, vrátili sme súčet vzdialeností od vzoru dát z doteraz prejdených centroidov. Ak nie, pokračovali sme ďalším prechodom.
4. V tomto kroku sme už mali ku všetkým dátam ich výsledné vzdialenosti k centroidom, ktoré pokrývajú *x* % dát. Z týchto vzdialeností sme vypočítali smerodajnú odchylku a priemer, ktoré sme použili pri *pravidle tri sigma* v kroku 5.
5. Aplikovali sme pravidlo tri sigma na testovacie dáta, čím sme určili, či ide o anomáliu alebo nie.

Tento postup sme zopakovali trikrát. Pre každý pokus sme zopakovali kroky 3., 4., 5. niekoľkokrát, pričom sme menili hodnotu parametra *x*. Graf 3 nám ukazuje ako si náša modifikovaná metóda viedla. V grafe môžeme pozorovať tri typy hodnôt. Prvá z nich je „**norm. == anom“**, ktorá nám hovorí, koľko normálnych dát sme označili za anomálie. Čím je táto hodnota nižšia, tým je metóda presnejšia. Ďalší typ je „**anom. == norm“,** ktorý popisuje aké množstvo anomálnych dát sme určili ako dáta normálne. Čím je táto hodnota nižšia, tým je metóda presnejšia. Posledná trieda je „**anom. == anom“,** ktorá popisuje ako

presne sme si viedli v určovaní anomálnych dát za anomálie. Čím je táto hodnota vyššia, tým je náš detektor presnejší. Z grafu môžeme pozorovať, že okolo hranice **25**% dosahujeme najlepšie výsledky. Od tejto hranice začína detektor určovať čoraz viac normálnych dát za anomálie.

Graf 3 : Úspešnosť modifikovaného detektora v závislosti od parametra *x*



Tabuľka 5 : Výsledky k-means – modifikovaný 25%

Modifikovaný k-means - 25%		
Dáta / **Detektor**	norm.	anom.
norm	291635	8527
anom	310	5145

Tabuľka 6 : Výsledky k-means – jednoduchá hranica

k-means – jednoduchá hranica		
Dáta / **Detektor**	norm.	anom.
norm	289836	12803
anom	2112	883

Pri porovnaní hodnôt z tabuľky 4 a z tabuľky 5 môžeme usúdiť, že upravená metóda na detekciu anomálií dosahuje lepšie výsledky ako metóda popisovaná v 5.3. Pri dôslednom preskúmaní oboch tabuliek si môžeme všimnúť, že náš modifikovaný algoritmus, dosahuje lepšie výsledky vo všetkých kategóriách.

Záver

V práci sme sa venovali implementácií algoritmov na detegovanie anomálií v dátach, ako aj technológiám zaoberajúcim sa spracovaním a uchovávaním dát väčších rozmerov.

V prvej časti sme si definovali pojem anomália a prečo je dôležité ich skúmať. Následne sme ukázali, ako sa dajú anomálie odhaľovať za pomoci rôznych metód. Stručnejšie sme si popísali metódu k-means, ktorá sa dá využiť pre lepšie detegovanie anomálií v dátach. Ďalej sme popísali rôzne typy metód pre hľadanie špeciálnych vzorov dát ako *tri sigma pravidlo,* analýzu hustoty dát.

Aby dané algoritmy dokázali spracúvať aj dáta väčších rozmerov priblížili sme si technológiu Hadoop a jeho ekosystém. Spomenuli sme aj najčastejšie distribúcie Hadoopu a ich výhody. Ďalšia časť bola venovaná Apachu Spark a jeho knižniciam ako nástroji pre distribuovanú prácu s veľkými dátami. Pre účelu oddemonštrovania funkčnosti detektorov sme použili dáta zo súťaže KDD cup 1999.

Následne sme využili algoritmus k-means z kniznice MLlib, aby sme dostali prvé výsledky, od ktorých sme sa odrazili pri vyhodnocovaní nášho modifikovaného algoritmu. Následný krok bola vlastná implementácia algoritmu k-means v distribuovanom prostredí Apache Spark. Naša implementácia algoritmus nedosahovala dostačujúce časové požiadavky a tak sme sa rozhodli využívať naďalej implementáciu z knižnice MLlib. Posledná časť bola venovaná navrhnutí a implementovaní modifikovaného detektora na odhaľovanie anomálnych vzorov dát. Tu sa nám podarilo zlepšiť metódu, ktorá sa využíva na detekciu pri použití k-means algoritmu. Zlepšenie sme dosiahli nebratím do úvahy len vzdialenosť od najbližšieho centroidu od pozorovaného vzoru dát, ale zakomponovaním väčšieho množstva centroidov do výslednej vzdialenosti. Počet zakomponovaných centroidov sme určovali v závislosti od pokrytého množstva dát jednotlivými centroidmi.

Navrhnutá metóda, ktorá je výsledkom tejto práce sa dá použiť na lepšie detegovanie anomálnych vzorov dát v porovnaní s detektorom založenom na algoritme k-means pracujúcim len s jednoduchou hranicou. Metóda, ktorú sme navrhli môže byť ďalej modifikovaná, pričom je možné otestovať iné spôsoby rozhodovania, či výsledná vzdialenosť od centoridov spadá do množiny anomálií alebo nie. My sme využívali

pravidlo tri sigma, no existujú aj iné metódy, ktoré by mohli dosahovať lepšie výsledky. Pri spracúvaní tejto bakalárskej práce sme sa nevenovali knižnici Spark Streaming, ktorá by sa dala využiť na odhaľovanie anomálií v reálnom čase.

Zoznam použitej literatúry

[1] M. Markou, S. Singh - *Novelty detection: A review*

[2] Varun Chandola, Arindam Banerjee, and Vipin Kumar - *Anomaly Detection : A Survey, ACM Computing Surveys, Vol. 41(3), Article 15, July 2009.*

[3] Weizhong Zhao, Huifang Ma, and Qing He - *Parallel K-Means Clustering Based on MapReduce*

[4] J. B. MacQueen (1967) - *Some Methods for classification and Analysis of Multivariate Observations*

[5] Jeffrey Dean and Sanjay Ghemawat (2004) - *MapReduce: Simplified Data Processing on Large Clusters*

[6] Markus M. Breunig, Hans-Peter Kriegel, Raymond T. Ng, Jörg Sander - *LOF: Identifying Density-Based Local Outliers*

[7] *HdfsUserGuide*, [online] Dostupné na internete 15.5.2016 : *https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html*

[8] *Spark benchmark,* [online] Dostupné na internete 15.5.2016 : *http://spark.apache.org/news/spark-wins-daytona-gray-sort-100tb-benchmark.html*

[9] *CDH components,* [online] Dostupné na internete 15.5.2016 : *https://www.cloudera.com/products/apache-hadoop/key-cdh-components.html*

[10] *MapR-fs*, [online] Dostupné na internete 15.5.2016 : *https://www.mapr.com/products/mapr-fs*

[11] *KDD*, [online] Dostupné na internete 15.5.2016 : *http://www.kdd.org/kdd-cup*

[12] *KDD dataset*, [online] Dostupné na internete 15.5.2016 : *http://kdd.ics.uci.edu/databases/kddcup99/task.html*

[13] *KFold metóda*, [online] Dostupné na internete 15.5.2016 *: http://scikit-learn.org/stable/modules/generated/sklearn.cross_validation.*KFold

[14] Ivy Liu, Alan Agesti - *The analysis of ordered categorical data: An overview and a survey of recent developments*

[15] *JBOD* , [online] Dostupné na internete 15.5.2016 :*https://en.wikipedia.org/wiki/Non-RAID_drive_architectures*

[16] *Apache Spark*, [online] Dostupné na internete 15.5.2016 : *http://spark.apache.org/*

[17] *Standard Score*, [online] Dostupné na internete 15.5.2016 : *https://en.wikipedia.org/wiki/Standard_score*

Dodatok A

Elektronická verzia práce na CD

Priložený disk CD obsahuje zdrojové kódy práce, ktoré sa dajú spustiť pomocou aplikácie Spark Shell.