TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

JONNE PETTERI PIHLANEN

SUOSITTELIJAJÄRJESTELMÄN RAKENTAMINEN APACHE SPARKILLA

Diplomityö

Examiner: ????
Examiner and topic approved by the
Faculty Council of the Faculty of
xxxx
on 1st September 2014

ABSTRACT

JONNE PETTERI PIHLANEN: Building a Recommendation Engine with Apache Spark
Tampere University of Technology
Diplomityö, xx pages
September 2016
Master's Degree Program in Signal Processing
Major: Data Engineering
Examiner: ????
Keywords:

The amount of recommendation engines around the Internet is constantly growing.

This paper studies the usage of Apache Spark when building a recommendation engine.

TIIVISTELMÄ

JONNE PETTERI PIHLANEN: Building a Recommendation Engine with Apache Spark
Tampereen teknillinen yliopisto
Diplomityö, xx sivua
syyskuu 2016
Signaalinkäsittelyn koulutusohjelma
Pääaine: Data Engineering
Tarkastajat: ????
Avainsanat:

PREFACE

Tampere,

Jonne Pihlanen

SISÄLLYS

LYHENTEET JA MERKINNÄT

Spark	Nopea ja yleinen kehys suuren mittakaavan dataprosessointiin

1. JOHDANTO

Suosittelujärjestelmät ovat nykyisin jatkuvasti läsnä jokapäiväisessä elämässämme. Ne auttavat päätöksenteossa verkko-ostoksissa, suoratoistopalveluissa, sosiaalisessa mediassa tai yksinkertaisti uutisten lukemisessa. Yksinkertaisin ja luonnollisin suosittelun muoto on ihmiseltä ihmiselle suosittelu. Ihmiset voivat kuitenkin tehokkaasti suositella vain niitä asioita, jotka ovat itse henkilökohtaisesti kokeneet. Tällöin suosittelijajärjestelmistä tulee hyödyllisiä, sillä ne voivat mahdollisesti tarjota suosituksia tuhansista tai jopa miljoonista erilaisista tuotteista.

Suosittelu voidaan jakaa kahteen pääkategoriaan: tuotepohjaiseen ja käyttäjäpohjaiseen. [2] Tuotepohjaisessa suosittelussa tarkoituksena on etsiä samankaltaisia tuotteita, sillä käyttäjän ajatellaan olevan mahdollisesti kiinnostunut samankaltaisista tuotteista myös tulevaisuudessa. Käyttäjäpohjaisessa suosittelussa käyttäjän ajatellaan olevan kiinnostunut tuotteista, joita samankaltaiset käyttäjät ovat ostaneet. Käyttäjäpohjainen suosittelu yrittää siis etsiä samankaltaisia käyttäjiä, jotta voidaan suositella näiden ostamia tuotteita.

Apache Spark on sovelluskehys, joka mahdollistaa hajautettujen ohjelmien rakentamisen. [7] Hajautetussa ohjelmassa suoritus voidaan jakaa useiden käsittelysolmujen kesken. Jotkin suositteluongelmat voidaan mallintaa hajautettuna ohjelmana, jossa kaksi matriisia, käyttäjät ja tuotteet, prosessoidaan iteratiivisella algoritmilla, joka mahdollistaa ohjelman suorittamisen rinnakkain. [7]

Apache Spark on rakennettu Scala ohjelmointikielellä. [7] Scala on monikäyttöinen, moniparadigmainen ohjelmointikieli, joka tarjoaa tuen funktionaaliselle ohjelmoinnille sekä vahvan tyypityksen. Työn käytännön osuus on toteutettu Scalaa käyttäen, joten lyhyt johdanto ohjelmointikieleen tarjotaan lukijalle.

Työn päämääränä on tutustua Apache Spark sovelluskehykseen sekä Scala ohjelmointikieleen ja toteuttaa näiden tekniikoiden avulla suosittelujärjestelmä.

Tämä työ on rakentuu seuraavista osista. Luku kaksi kuvailee suosittelujärjestelmiä. Luvussa kolme keskutellaan Apache Sparkista, avoimen lähdekoodin järjestelmästä, joka mahdollistaa hajautettujen ohjelmien rakentamisen. Luku neljä esittää toteutuksen suosittelijajärjestelmälle. Luvussa viisi käydään läpi tulokset. Lopuksi luvussa kuusi esitellään johtopäätökset.

2. SUOSITTELIJAJÄRJESTELMÄT

Suosittelulla tarkoitetaan tehtävää, jossa tuotteita suositellaan käyttäjille. Yksinkertaisin suosittelu tapahtuu ihmiseltä toiselle, ilman tietokoneita. Ihmiset voivat kuitenkin tehokkaasti suositella vain niitä asioita, jotka ovat itse henkilökohtaisesti kokeneet. Tällöin suosittelijajärjestelmistä tulee hyödyllisiä, sillä ne voivat mahdollisesti tarjota suosituksia sadoista tai jopa tuhansista erilaisista tuotteista. Suosittelijajärjestelmät ovat joukko tekniikoita ja ohjelmistoja, jotka tarjovat suosituksia mahdollisesti hyödyllisistä tuotteista. Tuotteella tarkoitetaan tässä yhteydessä yleistä asiaa, jota järjestelmä suosittelee henkilölle. Suosittelujärjestelmät on yleensä rakennettu suosittelemaan vain tietyn tyyppisiä tuotteita kuten esimerkiksi kirjoja tai elokuvia. [14]

Suosittelijajärjestelmien tarkoitus on auttaa asiakkaita päätöksenteossa, tuotteiden määrän ollessa valtava. Tavallisesti suositukset ovat räätälöityjä, millä tarkoitetaan että suositukset eroavat käyttäjien tai käyttäjäryhmien välillä. Suositukset voivat olla myös räätälöimättömiä ja niiden tuottaminen onkin usein yksinkertaisempaa. Lista, joka sisältää 10 suosituinta tuotetta, on esimerkki räätälöimättömästä suosittelusta. Järjestäminen tehdään ennustamalla kaikista sopivimmat tuotteet käyttäjän mieltymysten tai vaatimusten perusteella. Tämän suorittamiseksi suosittelijajärjestelmän on kerättävä käyttäjältä tämän mieltymykset. Mieltymykset voivat olla suoraan käyttäjältä kysyttyjä tai käyttäjän antamia tuotearvioita tai ne voidaan tulkita käyttäjän toiminnasta kuten klikkauksista, sivun katselukerroista tai ajasta jonka käyttäjä on viipynyt tietyllä tuotesivulla. Suosittelijajärjestelmä voisi esimerkiksi tulkita tuotesivulle päätymisen todisteeksi mieltymyksestä sivun tuotteista. [14]

Suosittelijajärjestelmien kehitys alkoi melko yksinkertaisesta havainnosta: ihmiset tapaavat luottaa toisten ihmisten suosituksiin tehdessään rutiininomaisia päätöksiä. On esimerkiksi yleistä luottaa vertaispalautteeseen valittaessa kirjaa luettavaksi tai luottaa elokuvakriitikoiden kirjoittamiin arvioihin. Ensimmäinen suosittelija-

järjestelmä yritti matkia tätä käytöstä etsimällä yhteisöstä suosituksia aktiiviselle käyttäjälle. Suositukset haettiin käyttämällä algoritmeja. Tämä lähestymistapa on tyypiltään yhteisösuodattamista. Yhteisösuodattamisessa ideana on että, jos käyttäjät pitivät samankaltaisista tuotteista aikaisemmin, he luultavasti pitävät samoja tuotteita ostaneiden henkilöiden suosituksia merkityksellisinä. [14]

Verkkokauppojen kehityksen myötä syntyi tarve suosittelulle vaihtoehtojen rajoittamiseksi. Käyttäjät kokivat aina vain vaikeammaksi löytää oikeat tuotteet sivustojen suurista valikoimista. Tiedon määrän räjähdysmäinen kasvaminen internetissä on ajanut käyttäjät tekemään huonoja päätöksiä. Vaihtoehdot ovat hyväksi, mutta vaihtoehtojen lisääntyminen on alkanut hyödyn tuottamisen sijaan heikentää kuluttajien hyvinvointia. [14]

Viimeaikoina suosittelijajärjestelmät ovat osoittautuneet tehokkaaksi lääkkeeksi tiedon *ylimääräongelmaa* vastaan. Suosittelijajärjestelmät käsittelevät tätä ilmiötä tarjoamalla uusia, aiemmin tuntemattomia, tuotteita jotka ovat todennäköisesti merkityksellisiä käyttäjälle tämän nykyisessä tehtävässä. Kun käyttäjä pyytää suosituksia, suosittelujärjestelmä tuottaa suosituksia käyttämällä tietoa ja tuntemusta käyttäjistä, saatavilla olevista tuotteista ja aiemmista *tapahtumista* (transactions). Tutkittuaan tarjotut suositukset, käyttäjä voi hyväksyä tai hylätä ne tarjoten epäsuoraa ja täsmällistä palautetta suosittelijalle. Tätä uutta tietoa voidaan myöhemmin käyttää hyödyksi tuotettaessa uusia suosituksia seuraaviin käyttäjän ja järjestelmän vuorovaikutuksiin. [14]

Verrattuna klassisten tietojärjestelmien, kuten tietokantojen ja hakukoneiden, tutkimukseen, suosittelijajärjestelmien tutkimus on verrattain tuoretta. Suositteljiajärjestelmistä tuli itsenäisiä tutkimusalueita 90-luvun puolivälissä. Viimeaikoina mielenkiinto suosittelujärjestelmiä kohtaan on kasvanut merkittävästi. Esimerkkinä suuren profiilin verkkosivustot kuten Amazon.com, YouTube, Netflix sekä IMDB, joissa suosittelujärjestelmillä on iso rooli. Oma lukunsa ovat myös vain suosittelujärjestelmien tutkimiseen ja kehittämiseen tarkoitetut konferenssit kuten RecSys ja AI Communications (2008). [14]

Suosittelujärjestelmällä voidaan ajatella olevan kaksi päätarkoitusta. Ensimmäinen on avustaa palveluntarjoajaa ja toinen on tuottaa arvoa palvelun käyttäjälle. Suosittelujärjestelmän on siis tasapainoteltava sekä palveluntarjoajan että käyttäjän tarpeiden välillä. [14] Palveluntarjoaja voi esimerkiksi ottaa suosittelujärjestelmän avuksi parantamaan tai monipuolistamaan myyntiä, lisäämään käyttäjien tyytyväi-

syyttä, lisäämään käyttäjien uskollisuutta tai ymmärtämään paremmin mitä käyttäjä haluaa [14]. Käyttäjä puolestaan saattaa haluta suosituksena tuotesarjan, apua selaamiseen tai mahdollistaa muihin vaikuttamisen. [14]

GroupLens, BookLens ja MovieLens olivat uranuurtajia suosittelujärjestelmissä. GroupLens on tutkimuslaboratorio tietojenkäsittelytieteen- ja tekniikan laitoksella Minnesotan Yliopistossa, joka on erikoistunut muun muassa suosittelujärjestelmiin ja verkkoyhteisöihin [2]. BookLens on on GroupLensin rakentama kirjojen suosittelujärjestelmä [3]. MovieLens on GroupLensin ylläpitämä elokuvien suosittelujärjestelmä [11]. Uranuurtavan tutkimuksen lisäksi nämä sivustot julkaisivat aineistoja, joka ei ollut yleistä tuohon aikaan. [2]

2.1 Suositustekniikat

Suosittelujärjestelmällä täytyy olla ymmärrys tuotteista, jotta se pystyy tekemään suosituksia. Tämän mahdollistamiseksi järjestelmän täytyy pystyä ennustamaan tuotteen käytännöllisyys tai ainakin verrata tuotteiden hyödyllisyyttä ja tämän perusteella päättää suositeltavat tuotteet. Ennustamista voidaan luonnostella yksinkertaisella personoimattomalla suosittelualgoritmilla, joka suosittelee vain suosituimpia elokuvia. Tätä lähestymistapaa voidaan perustella sillä, että tarkemman tiedon puuttuessa käyttäjän mieltymyksistä, elokuva josta muutkin ovat pitäneet on todennäköisesti myös keskivertokäyttäjän mieleen, ainakin enemmän kuin satunaisesti valikoitu elokuva. Suosittujen elokuvien voidaan siis katsoa olevan kohtuullisen osuvia suosituksia keskivertokäyttäjälle. [14]

Tuotteen i hyödyllisyyttä käyttäjälle u voidaan mallintaa reaaliarvoisella funktiolla $R(u, i)$, kuten yleensä tehdään *yhteisösuodatuksessa* ottamalla huomioon käyttäjien antamat arviot tuotteista. Yhteisösuodatuksessa suosittelijan perustehtävä on ennustaa R:n arvoa käyttäjä-tuote pareille ja laskea arvio todelliselle funktiolle R. Laskiessaan tätä ennustetta käyttäjälle u ja tuotejoukolle, järjestelmä suosittelee tuotteita, joilla on suurin ennustettu hyödyllisyys. Ennustettujen tuotteiden määrä on usein paljon pienempi kuin tuotteiden koko määrä, joten voidaan sanoa, että suosittelijajärjestelmä suodattaa käyttäjälle suositeltavat tuotteet. [14]

Suosittelijajärjestelmät eroavat toisistaan kohdistetun toimialan, käytetyn tiedon ja erityisesti siinä kuinka suositukset tehdään, jolla tarkoitetaan suosittelualgoritmia [14]. Tässä työssä keskitytään vain yhteen suositteluteknikoiden luokkkaan, yhtei-

sösuodatukseen, sillä tätä menetelmää käytetään Apache Sparkin MLlib kirjastossa.

Yhteisöllistä suodatusta käyttävät suosittelijajärjestelmät perustuvat käyttäjien yhteistyöhön. Niiden tavoitteena on tunnistaa malleja käyttäjän mielenkiinnoista voidakseen tehdä suunnattuja suosituksia [1]. Tämän lähestymistavan alkuperäisessä toteutuksessa suositellaan aktiiviselle käyttäjälle niitä tuotteita, joita muut samankaltaiset käyttäjät ovat pitäneet aiemmin [14]. Käyttäjä arvostelee tuotteita. Seuraavaksi algoritmi etsii suosituksia perustuen käyttäjiin, jotka ovat ostaneet samanlaisia tuotteita tai perustuen tuotteisiin, jotka ovat eniten samanlaisia käyttäjän ostoshistoriaan verrattuna. Yhteisösuodatus voidaan jakaa kahteen kategoriaan, jotka ovat *tuotepohjainen- ja käyttäjäpohjainen yhteisösuodatus*. Yhteisösuodatus on eniten käytetty ja toteutettu tekniikka suositusjärjestelmissä [6] [14] [4].

Yhteisöllinen suodatus analysoi käyttäjien välisiä suhteita ja tuotteiden välisiä riippuvuuksia tunnistaakseen uusia käyttäjä-tuote -assosiaatioita [8]. Päätelmä siitä, että käyttäjät voisivat pitää samasta laulusta, koska molemmat kuuntelevat muita samankaltaisia lauluja on esimerkki yhteisöllisestä suodatuksesta [15].

Koska yhteisöllisessä suodatuksessa suosittelu perustuu pelkästään käyttäjän arvosteluihin tuotteesta, yhteisöllinen suodatus kärsii ongelmista jotka tunnetaan nimillä *uusi käyttäjäongelma* ja *uusi tuoteongelma* [6]. Ellei käyttäjä ole arvostellut yhtään tuotetta, algoritmi ei kykene tuottamaan myöskään yhtään suositusta. Muita yhteisöllisen suodatuksen haasteita ovat *kylmä aloitus* sekä *niukkuus* (sparsity). Kylmällä aloituksella tarkoitetaan sitä, että tarkkojen suositusten tuottamiseen tarvitaan tyypillisesti suuri määrä dataa. Niukkuudella tarkoitetaan sitä, että tuotteiden määrä ylittää usein käyttäjien määrän. Tästä johtuen suhteiden määrä on todella niukka, sillä useat käyttäjät ovat arvostelleet tai ostaneet vain murto-osan tuotteiden koko määrästä. [1]

2.1.1 Muistiperustainen yhteisöllinen suodatus

Muistiperustaisissa menetelmissä käyttäjä-tuote -suosituksia käytetään suoraan uusien tuotteiden ennustamiseksi. Tämä voidaan toteuttaa kahdella tavalla, käyttäjäpohjaisena suositteluna tai tuotepohjaisena suositteluna.

Seuraavat kappaleet kuvaavat käyttäjäpohjaista yhteisösuodatusta ja tuotepohjaista yhteisösuodatusta.

Tuotepohjainen yhteisösuodatus

Tuotepohjaisessa yhteisösuodatuksessa (Item-based collaborative filtering, IBCF) algoritmi aloittaa etsimällä samankaltaisia tuotteita käyttäjän ostoshistoriasta [6]. Seuraavaksi mallinnetaan käyttäjän mieltymykset tuotteelle perustuen saman käyttäjän tekemiin arvosteluihin [14]. Alla oleva koodinpätkä esittelee tuotepohjaisen yhteisösuodatuksen idean jokaiselle uudelle käyttäjälle.

Program 2.1 *Tuotepohjaisen yhteisösuodatuksen algoritmi [6]*

1. Jokaiselle kahdelle tuotteelle, mittaa kuinka samankaltaisia ne ovat sen suhteen, kuinka samankaltaisia arvioita ne ovat saaneet samankaltaisilta käyttäjiltä. Samankaltaisuutta voidaan arvioida esimerkiksi kosinimitan avulla.

```
case class Item(id: Int, feature1: String, feature2: String)
val items: Seq[Item] = Seq(Item(), Item(), Item())
case class Similarity(item1: Item, item2: Item, similarity: Double)

val similarItems: Seq[Similarity] = items.map { item1 =>
  items.map { item2 =>
    Similarity (item1, item2, cosineSimilarity(item1, item2))
  }
}
```

2. Tunnista k samankaltaisinta tuotetta jokaiselle tuotteelle

```
case class Similarities (item: Item, kMostSimilarItems: Seq[Similarity])

val similarities = findKMostSimilarItems(similarItems)
```

3. Jokaiselle käyttäjälle, tunnista tuotteet jotka ovat eniten samankaltaisia käyttäjän ostoshistorian kanssa.

```
users.foreach { user =>
  user.purchases.foreach { purchase =>
    val mostSimilar = findSimilarItem(purchase, similarities)
  }
}
```

Amerikan suurimman verkkokaupan, Amazon.com:in, on aiemmin tiedetty käyttäneen tuote-tuotteeseen yhteisösuodatusta. Tässä toteutuksessa algoritmi rakentaa samankaltaisten tuotteiden taulun etsimällä tuotteita joita käyttäjät tapaavat ostaa yhdessä. Seuraavaksi algoritmi etsii käyttäjän ostoshistoriaa ja arvosteluita vastaavat tuotteet, yhdistää nämä tuotteet ja palauttaa suosituimmat tai eniten korreloivat tuotteet. [9]

Käyttäjäpohjainen yhteisösuodatus

Tuotepohjaisessa yhteisösuodatuksessa (User-based collaborative filtering, UBCF) algoritmi aloittaa etsimällä samankaltaisimmat käyttäjät. Seuraava askel on arvostella samankaltaisten käyttäjien ostamat tuotteet. Lopuksi valitaan parhaan arvosanan saaneet tuotteet. Samankaltaisuus saadaan laskettua vertaamalla käyttäjien ostoshistorioiden samankaltaisuutta. [14]

Askeleet jokaiselle uudelle käyttäjälle käyttäjäpohjaisessa yhteisösuodatuksessa ovat:

Program 2.2 *Käyttäjäpohjainen yhteisösuodatus algoritmi [6]*

1. Mittaa jokaisen käyttäjän samankaltaisuus uuteen käyttäjään. Kuten IBCF:ssä , suosittuja samankaltaisuusarvioita ovat korrelaatio sekä kosinimitta.

```
case class Similarity(userId1: Int, userId2: Int, score: Int)

val newUser: User = User("Adam", 31, purchases)
val similarities = users.map { user =>
  Similarity(newUser.id, user.id, cosineSimilarity(user, newUser)
}
```

2. Tunnista samankaltaisimmat käyttäjät. Vaihtoehtoja on kaksi: Voidaan valita joko parhaat k käyttäjää (k−nearest neighbors) tai voidaan valita käyttäjät, joiden samankaltaisuus ylittää tietyn kynnysarvon.

```
val mostSimilarUsers = similarities. filter (_.score > 0.8)
```

3. Arvostele samankaltaisimpien käyttäjien ostamat tuotteet. Arvostelu saadaan joko keskiarvona kaikista tai painotettuna keskiarvona, käyttäen samankaltaisuuksia painoina. TODO

```
val ratedItems = mostSimilarUsers.map { user =>
  user.purchases.map { purchase =>
    val purchases = mostSimilarUsers.map { usr =>
      usr.purchases. filter (_.id === purchase.id)
    }
    purchases.sum() / purchases.size
  }
}
```

4. Valitse parhaiten arvostellut tuotteet.

```
val topRatedItems = ratedItems.take(10)
```

2.1.2 Mallipohjainen yhteisösuodatus

Muistipohjaisen yhteisösuodatuksen käyttäessä tallennettuja suosituksia suoraan ennustamisen apuna, mallipohjaisissa lähestymistavoissa näitä arvosteluita käytetään ennustavan mallin oppimiseen. Perusajatus on mallintaa käyttäjä-tuote vuorovaikutuksia tekijöillä jotka edustavat käyttäjien ja tuotteiden piileviä ominaisuuksia (latent factors) järjestelmässä. Piileviä ominaisuuksia ovat esimerkiksi käyttäjän mieltymykset ja tuotteiden kategoriat. Tämä malli opetetaan käyttämällä saatavilla olevaa dataa ja myöhemmin käytetään ennustamaan käyttäjien arvioita uusille tuotteille. [14]

Vuorottelevat pienimmät neliöt (Alternating Least Squares, ALS) algoritmi on esimerkki mallipohjaisesta yhteisösuodatusalgoritmista ja se esitetään seuraavassa luvussa.

3. APACHE SPARK

Apache Spark on avoimen lähdekoodin sovelluskehys, joka yhdistää hajautettujen ohjelmien kirjoittamiseen tarkoitetun järjestelmän sekä elegantin mallin ohjelmien kirjoittamiseen [15]. Spark tarjoaa korkean tason rajapinnat Java, Scala, Python sekä R ohjelmointikielille.

Jokainen Spark sovelus koostuu driver-ohjelmasta sekä yhdestä tai useammasta executor-ohjelmasta. Driver on ohjelma, joka ajaa käyttäjän pääohjelmaa ja suorittaa erilaisia rinnakkaisia operaatioita klusterissa. Executor on yksi kone klusterissa. [15]

Spark voidaan esitellä kuvailemalla sen edeltäjää, MapReduce:a, ja sen tarjoamia etuja. MapReduce tarjosi yksinkertaisen mallin ohjelmien kirjoittamiseen ja pystyi suorittamaan kirjoitettua ohjelmaa rinnakkain sadoilla tietokoneilla. MapReduce skaalautuu lähes lineaarisesti datan koon kasvaessa. Suoritusaikaa hallitaan lisäämällä lisää tietokoneita suorittamaan tehtävää. [15]

Apache Spark säilyttää MapReduce:n lineaarisen skaalautuvuuden ja vikasietokyvyn mutta laajentaa sitä kolmella merkittävällä tavalla. Ensiksi, MapReducessa map- ja reduce-tehtävien väliset tulokset täytyy kirjoittaa levylle kun taas Spark kykenee välittämään tulokset suoraan putkiston (pipeline) seuraavalle vaiheelle. Toiseksi, Apache Spark:in voidaan ajatella kohtelevan kehittäjiä paremmin tarjoamalla rikkaan joukon muunnoksia (transformations) joiden avulla voidaan muutamalla koodirivillä ilmaista monimutkaisia putkistoja. (ESIMERKKI?) Kolmanneksi, Spark esittelee muistissa tapahtuvan prosessoinnin tarjoamalla abstraktion nimeltä Resilient Distributed Dataset (RDD). RDD tarjoaa kehittäjälle mahdollisuuden materialisoida minkä tahansa askeleen putkistossa ja tallentaa sen muistiin. Tämä tarkoittaa sitä, että tulevien askelien ei tarvitse laskea aiempia tuloksia uudelleen ja tällöin on mahdollista jatkaa juuri käyttäjän haluamasta askeleesta. Aiemmin tämänkaltaista ominaisuutta ei ole ollut saatavilla hajautetun laskennan järjestelmissä. [15]

Spark ohjelmia voidaan kirjottaa Java, Scala, Python tai R-ohjelmointikielellä. Scalan käyttämisellä saavutetaan kuitenkin muutamia etuja, joita muut kielet eivät tarjoa. Tehokkuus saatta parantua, sillä datan siirtäminen eri kerrosten välillä tai muunnosten suorittaminen datalle voi johtaa heikompaan tehokkuuteen. Spark on kirjoitettu Scala-ohjelmointikielellä, joten viimeisimmät ja parhaimmat ominaisuudet ovat aina käytössä, eikä niiden käännöstä tarvitse odotella. Spark ohjelmoinnin filosofia on helpompi ymmärtää kun Sparkia käytetään kielellä, jolla se on rakennettu. Suurin hyöty, jonka Scalan käyttäminen tarjoaa, on kuitenkin kehittäjäkokemus joka tulee saman ohjelmointikielen käyttämisestä kaikkeen. Datan tuonti, manipulointi ja koodin lähettäminen klustereihin hoituvat samalla kielellä. [15]

Spark-jakelun mukana toimitetaan luku-evaluointi-tulostus-silmukka, komentorivityökalu, (Read eval print loop, REPL), joka mahdollistaa uusien asioiden nopean testailun konsolissa, eikä sovelluksista tarvitse rakentaa itsenäisiä (self-contained) alusta asti. Kun REPLissä kehitetyn sovelluksen tai sovelluksen osan voidaan katsoa olevan tarpeeksi valmis, on järkevää tehdä siitä koottu kirjasto (JAR). Näin varmistutaan ettei ohjelmakoodia tai tuloksia pääse katoamaan, vaikkakin REPL tarjoaa samantapaisen muistin komentohistoriasta kuin perinteinen komentorivikin.

JAR eli Java ARchive on suosittuun ZIP tiedostoformaattiin perustuva alustariippumaton tiedostoformaatti, jota käytetään kokoamaan monta tiedostoa yhdeksi tiedostoksi. [12]

JVM (Java Virtual Machine, Java-virtuaalikone) on abstrakti laskentakone (computing machine). Kuten oikea laskentakone, se omaa käskykannan ja muokkaa useita muistialueita ajon aikana. JVM ei tiedä mitään ohjelmointikielistä, kuten Scala tai Java, vaan se operoi ainoastaan class-tiedostoilla, jotka ovat tietynlaisia binääritiedostoja. Class-tiedosto sisältää JVM käskyt sekä symbolitaulun. [13]

3.1 Scala

Scala on moniparadigmainen ohjelmointikieli, joka tukee sekä olio- että funktionaalista ohjelmointia. Funktionaalista ohjelmointia varten Scalasta löytyy tuki funktionaalisen ohjelmoinnin konsepteille kuten muuttumattomat tietorakenteet ja funktiot ensimmäisen luokan kansalaisina. Olio-ohjelmointia varten Scalasta löytyy tuki konsepteille kuten luokat, oliot ja piirre (trait). Scala tukee myös kapselointia, perintää, moniperintää ja muita tärkeitä olio-ohjelmoinnin konsepteja. Scala on staattisesti

tyypitetty kieli ja sillä kirjoitetut ohjelmat käännetään Scala kääntäjää käyttäen. Scala on JVM-perustainen (Java Virtual Machine, Java-virtuaalikone) kieli, joten Scala kääntäjä kääntää sovelluksen Java-tavukoodiksi, joka voidaan ajaa missä tahansa Java-virtuaalikoneessa. Tavukooditasolla Scala ohjelmaa ei voida erottaa Java sovelluksesta. Scalan ollessa JVM-perustainen, Scala on täysin yhteensopiva Javan kanssa ja näin ollen Java-kirjastoja voidaan käyttää suoraan Scala-koodissa. Tästä syystä Scala-sovellukset hyötyvät suuresta Java-koodin määrästä. Vaikka Scala tukee sekä olio- että funktionaalista ohjelmointia, funktionaalista ohjelmointia suositaan. [7]

3.1.1 Perustyypit

Scalan perustyypit numeroiden esittämiseen ovat Byte, Short, Int, Long, Float ja Double. Lisäksi Scalassa on perustyypit Char, String ja Boolean. Char on 16 bittinen etumerkitön Unicode merkki. String on jono Char:eja. Boolean esittää totuusarvoa tosi (true) tai epätosi (false). [7]

Javasta poiketen Scalassa ei ole ollenkaan primitiivisiä tyyppejä vaan jokainen tyyppi on toteutettu luokkana. Käännöksen aikana kääntäjä tarvittaessa automaattisesti muuntaa Scala tyypit Javan primitiivisiksi tyypeiksi. [7]

3.1.2 Muuttujat

Scalassa on kahdentyyppisiä muuttujia: muuttuvia ja vakioita. Muuttuva muuttuja määritellään avainsanan *var* avulla. Muuttuvaa muuttujaa ei voida asettaa uudelleen luomisen jälkeen. Var:ien käyttöä ei suositella, mutta joskus niiden käyttämisellä saadaan aikaan yksinkertaisempaa ohjelmakoodia ja tästä syystä Scala tukee myös muuttuvia muuttujia. [7]

Syntaksi *var*:in luomiseksi on

Program 3.1 *Muuttuvan muuttujan luominen ja uudelleen asettaminen*

```
var x = 10
x = 20
```

Muuttumatonta muuttujaa, *val*, ei sen sijaan voida antaa uudelleen luomisen jälkeen. Syntaksi *val*:in luomiseksi on

Program 3.2 *Muuttumattoman muuttujan luominen*

```
val y = 10
```

Mikäli muuttumatonta muuttujaa koitetaan antaa uudelleen myöhemmin ohjelmassa, kääntäjä antaa virheen. Huomionarvoista ylläolevassa syntaksissa on se, että Scala kääntäjä ei pakota määrittelemään muuttujan tyyppiä sillon kuin kääntäjä pystyy päättelemään sen.

Program 3.3 *Muuttujan luominen tyyppimäärittelyn avulla*

```
var x: Int = 10
val y: Int = 10
```

3.1.3 Funktiot

Funktio on lohko suoritettavaa koodia joka palauttaa arvon. Se on konseptuaalisesti samankaltainen kuin matematiikassa: funktio ottaa sisääntulon ja palauttaa ulostulon. [7]

Scalan funktiot ovat ensimmäisen luokan kansalaisia, jolla tarkoitetaan että funktiota voidaan:

- käyttää kuten muuttujaa
- antaa syötteenä toiselle funktiolle
- määritellä nimettömänä funktioliteraalina
- asettaa muuttujaan
- määritellä toisen funktion sisällä
- palauttaa toisen funktion ulostulona

[7]

Scalassa funktio määritellään avainsanalla def. Funktion määrittely aloitetaan funktion nimellä, jota seuraa sulkeissa olevat, pilkulla erotetut, parametrit tyyppimäärittelyineen. Parametrien jälkeen funktiomäärittelyyn tulee kaksoispiste, funktion

ulostulon tyyppi, yhtäsuuruusmerkki sekä funktion runko joko aaltosulkeissa tai ilman. [7]

Program 3.4 *Funktio*

```
def add(first: Int, second: Int): Int = {
  val sum = first + second
  return sum
}
```

Ylläolevassa esimerkissä funktion nimi on *add* ja se ottaa kaksi *Int* tyyppistä sisääntuloa. Funktio palauttaa *Int* tyyppisen arvon jonka se muodostaa lisäämällä annetut sisääntulot yhteen ja palauttamalla tuloksen.

Scala sallii myös lyhyemmän version samasta funktiosta:

Program 3.5 *Funktio*

```
def add(first: Int, second: Int): Int = first + second
```

Toinen versio tekee täsmälleen saman asian kuin ensimmäinenkin, mutta se on vain kirjoitetty käyttäen lyhyempää syntaksia. Paluuarvon tyyppi on jätetty antamatta, sillä kääntäjä pystyy päättelemään sen koodista. Paluuarvo suositellaan kuitenkin annettavan aina. Aaltosulkeet on myöskin jätetty pois, sillä ne ovat pakolliset vain kun funktion runko sisältää useamman kuin yhden käskyn. Lisäksi, *return* avainsana on ohitettu, sillä se on vapaaehtoinen. Scalassa kaikki lausekkeet ovat arvon palauttavia lausekkeita, joten funktion rungon viimeisen lausekkeen arvosta tulee funktion paluuarvo. [7]

3.2 Resilient Distributed Dataset (RDD)

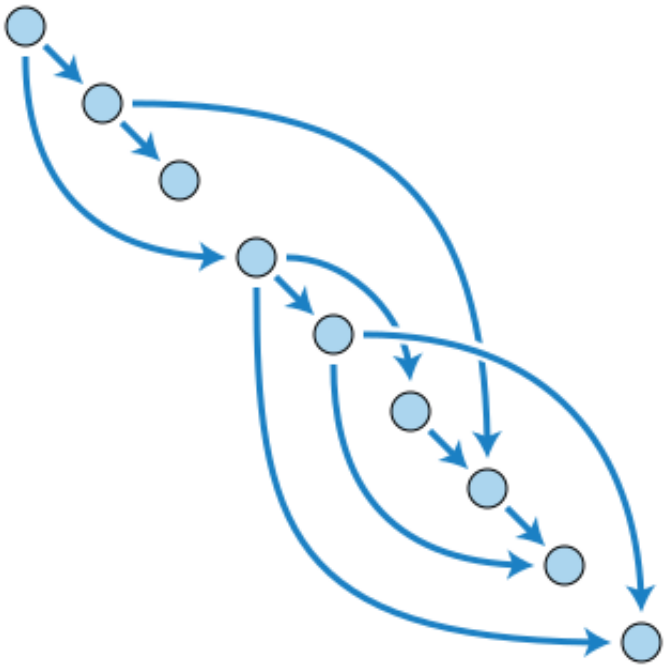
Resilient Distributed Dataset (RDD) on Sparkin tarjoama pääabstraktio. RDD on muuttumaton, partitioitu elementtikokoelma, joka voidaan hajauttaa klusterin useiden koneiden välillä. [20]

RDD:t ovat laiskasti evaluoituvia, jolla tarkoitetaan sitä, että lausekkeen evaluointia viivytetään siihen asti kun sen arvoa tarvitaan. Kun uusi RDD luodaan, mitään laskentaa ei oikeasti vielä tapahdu, vaan Spark tietää missä data sijaitsee tai miten data saadaan laskettua kun tulee aika tehdä sille jotain.

RDD voidaan luoda kahdella tavalla, rinnakkaistamalla (parallelize) tai viittaamalla ulkoiseen aineistoon. Rinnakkaistamisessa olemassaoleva Scala kokoelma voidaan rinnakkaistaa RDD:ksi. Ulkoiseen ainestoon viittaamisella tarkoitetaan viittaamista aineistoon ulkoisessa varastointijärjestelmässä kuten HDFS:sä, HBase:ssa tai missä tahansa Hadoopin tuntemassa tiedostojärjestelmässä. [18]

RDD:t voidaan tallentaa muistiin, jolloin ohjelmistokehittäjä voi uudelleenkäyttää niitä tehokkaasti rinnakkaisissa operaatioissa. RDD:t voivat palautua solmuvirheistä automaattisesti käyttäen Directed Acyclic Graph (DAG) moottoria. DAG tukee asyklistä datavirtaa, jolla tarkoitetaan sitä, että jokainen graafin kaari kulkee topologisessa järjestyksessä aiemmasta myöhempään. Jokaista Spark-työtä kohti luodaan DAG klusterissa suoritettavan tehtävän tasoista. Verrattuna MapReduceen, joka luo DAGin kahdesta ennaltamäärätystä tilasta (Map ja Reduce), Sparkin luomat DAGit voivat sisältää minkä tahansa määrän tasoja. Tästä syystä jotkin työt voivat valmistua nopeammin kuin ne valmistuisivat MapReducessa. TODO Yksinkertaisimmat työt voivat valmistua vain yhden tason jälkeen ja monimutkaisemmat tehtävät valmistuvat yhden monitasoisen ajon jälkeen, ilman että niitä täytyy pilkkoa useampiin töihin. [22]

Kuva 3.1 *Directed Acyclic Graph [5]*



3.3 Dataset API

Dataset (DS) on vahvasti tyypitetty kokoelma aluespesifisiä objekteja, jotka voidaan muuntaa rinnakkain käyttäen funktionaalisia tai relaatio-operaatioita. DS on RDD:n korvaaja Sparkissa. Dataset:ille olemassa olevat operaatiot on jaettu *muunnoksiin* (transformations) ja *toimiin* (actions). Muunnokset ovat operaatioita, jotka luovat uusia Dataset objekteja, kuten map, filter, select, aggregate. Toimet ovat operaatioita jotka suorittavat laskentaa ja palauttavat tuloksia. Toimia ovat esimerkiksi count, show tai datan kirjoittaminen tiedostojärjestelmään. [19]

Dataset-instanssit ovat laiskasti evaluoituvia, jolla tarkoitetaan sitä, että laskenta aloitetaan vasta kun toimintoa kutsutaan tai instanssin arvoa tarvitaan. Dataset on pohjimmiltaan looginen suunnitelma, jolla kuvataan datan tuottamiseen tarvittava laskenta. Toimea kutsuttaessa, Sparkin kyselyoptimoija (query optimizer) optimoi loogisen suunnitelman ja generoi fyysisen suunnitelman. Fyysinen suunnitelma takaa rinnakkaisesti ja hajautetusti tapahtuvan tehokkaan suorituksen. Loogista suunnitelmaa, kuten myös optimoitua fyysistä suunnitelmaa, voidaan tutkia käyttämällä DS:n *explain* funktiota. [19]

Domain-spesifisten olioiden tehokkaaseen tukemiseen tarvitaan enkooderia. Enkooderilla tarkoitetaan ohjelmaa, joka muuntaa tietoa jonkin algoritmin mukaisesti ja tässä tapauksessa sitä käytetään yhdistämään domain-spesifinen tyyppi T Sparkin sisäiseen tyyppijärjestelmään. Esimerkiksi luokan $Person$ tapauksessa, joka sisältää kentät nimi (merkkijono) ja ikä (kokonaisluku), enkooderia voidaan käyttää käskemään Sparkia luomaan koodia ajon aikana joka serialisoi $Person$ olion binäärirakenteeksi. Generoidulla binäärirakenteella on usein pienempi muistijalanjälki ja se on myös optimoitu tehokkaaseen dataprosessointiin. Datan binääriesitys voidaan tarkistaa käyttämällä DS:n tarjoamaa *schema* funktiota. [19]

Dataset voidaan luoda tyypillisesti kahdella eri tavalla. Yleisin tapa on käyttää *SparkSession*:in tarjoamaa *read* funktiota ja osoittaa Spark joihinkin tiedostoihin tiedostojärjestelmässä, kuten seuraavaan *json* tiedostoon.

Program 3.6 *Esimerkki JSON tiedosto*

```
[{
  "name": "Matt",
  "salary": 5400
```

```
}, {
  "name": "George",
  "salary": 6000
}]
```

Dataset voidaan luoda myös tekemällä muutoksia olemassaoleville Dataset olioille:

Program 3.7 *Creating a new Dataset through a transformation*

```
val names = people.map(_.name)
```

Program 3.8 *Uuden Dataset olion luominen käyttäen read funktiota*

```
val people = spark.read.json("./people.json").as[Person]
```

jossa *Person* olisi Scala case-luokka, esimerkiksi:

Program 3.9 *case class Person*

```
case class Person(id: BigInt, firstName: String, lastName:
    String)
```

Case-luokat ovat tavallisia Scala-luokkia jotka ovat:

- Oletustarvoisesti muuttumattomia (immutable)
- Hajoitettavia (decomposable) hahmonsovitusta hyväksikäyttäen
- Vertailtavissa viitteiden sijasta rakenteellisen samankaltaisuuden mukaan
- Lyhyitä luoda (instantiate) ja käyttää

Mikäli tyyppimuunnos (casting) jätettäisiin tekemättä, päädyttäisiin luomaan DataFrame olio, jonka sisäinen mallin (schema) Spark pyrkisi arvaamaan. Tyyppimuunnos tehdään käyttämällä *as* avainsanaa.

Program 3.10 *SparkSession kontekstin luominen*

```
val spark = SparkSession
```

```
.builder
.appName("MovieLensALS")
.config("spark.executor.memory", "2g")
.getOrCreate()
```

SparkSession on Spark ohjelmoinnin aloituspiste, kun halutaan käyttää Dataset ja Dataframe rajapintoja. Ylläolevassa koodinpätkässä luodaan *SparkSession* ketjuttamalla rakentajan kutsuja.

[19]

Dataset oliot ovat samankaltaisia kuin RDD:t, sillä nekin tarjoavat vahvan tyypityksen ja mahdollisuuden käyttää voimakkaita lambda-funktioita [21]. Lambda-funktioita avustaa Spark SQL:n optimoitu suoritusmoottori [21]. Perinteisen serialisoinnin, kuten Java serialisoinnin, sijaan käytetään erikoistunutta enkooderia olioiden serialisointiin. Serialisaatiolla tarkoitetaan olion muuntamista tavuiksi, jolloin olion muistijalanjälki pienenee. Yleisesti serialisointia tarvitaan datan prosessointiin tai verkon yli lähettämiseen. Molempia, sekä enkoodereita ja serialisointia käytetään olioiden muuntamiseen tavuiksi, mutta koodi luo enkooderit dynaamisesti. Enkooderit käyttävät sellaista muotoa, että Spark kykenee suorittamaan monenlaisia operaatioita, kuten suodattamista, järjestämistä ja hajautusta (hashing), ilman että tavuja tarvitsee deserialisoida takaisin objektiksi. [18]

Seuraavassa koodilistauksessa luodaan uusi Datase lukemalla *json* tiedosto tiedostojärjestelmästä. Seuraavaksi luodaan uusi Dataset muunnoksen kautta. Objektin kloonaamiseksi käytetään case luokan *copy* metodia, koska *people* Dataset oli määritelty muuttumattomaksi. Lopuksi fyysinen suunnitelma tulostetaan konsoliin käyttämällä *explain* funktiota uudelle Dataset objektille.

Program 3.11 *Dataset olion loogisen ja fyysisen suunnitelman näyttäminen*

```
val people = spark.read.json("./people.json").as[Person]

val peopleWithDoubleSalary = people.map { person =>
  person.copy(salary = person.salary * 2)
}

peopleWithDoubleSalary.explain(true)
```

Program 3.12 *Dataset olion looginen suunnitelma*

```
== Optimized Logical Plan ==
SerializeFromObject [staticinvoke(class org.apache.spark.
    unsafe.types.UTF8String, StringType, fromString,
    assertnotnull(input[0, $line32.$read$$iw$$iw$Person, true
    ], top level Product input object).name, true) AS name#
    67, staticinvoke(class org.apache.spark.sql.types.
    Decimal$, DecimalType(38,0), apply, assertnotnull(input
    [0, $line32.$read$$iw$$iw$Person, true], top level
    Product input object).salary, true) AS salary#68]
+- MapElements <function1>, class $line32.
    $read$$iw$$iw$Person, [StructField(name,StringType,true),
     StructField(salary,DecimalType(38,0),true)], obj#66:
    $line32.$read$$iw$$iw$Person
   +- DeserializeToObject newInstance(class $line32.
       $read$$iw$$iw$Person), obj#65: $line32.
       $read$$iw$$iw$Person
      +- Relation[name#55,salary#56L] json
```

Program 3.13 *Dataset olion fyysinen suunnitelma*

```
== Physical Plan ==
*SerializeFromObject [staticinvoke(class org.apache.spark.
    unsafe.types.UTF8String, StringType, fromString,
    assertnotnull(input[0, $line32.$read$$iw$$iw$Person, true
    ], top level Product input object).name, true) AS name#
    67, staticinvoke(class org.apache.spark.sql.types.
    Decimal$, DecimalType(38,0), apply, assertnotnull(input
    [0, $line32.$read$$iw$$iw$Person, true], top level
    Product input object).salary, true) AS salary#68]
+- *MapElements <function1>, obj#66: $line32.
    $read$$iw$$iw$Person
   +- *DeserializeToObject newInstance(class $line32.
       $read$$iw$$iw$Person), obj#65: $line32.
       $read$$iw$$iw$Person
```

```
+- *FileScan json [name#55,salary#56L] Batched: false,
   Format: JSON, Location: InMemoryFileIndex[file:/home/
   joonne/Documents/GitHub/thesis-code/people.json],
   PartitionFilters: [], PushedFilters: [], ReadSchema:
   struct<name:string,salary:bigint>
```

3.4 DataFrame API

DataFrame on pohjimmiltaan nimettyihin sarakkeisiin järjestetty Dataset. Se on käsitteellisesti yhtenevä relaatiotietokannan taulun tai R/Python kielten tietokehyksen (data frame) kanssa, mutta DataFrame omaa rikkaammat optimoinnit konepellin alla. DataFrame voidaan rakentaa useammalla tavalla, kuten esimerkiksi jäsennellyistä tiedostoista, Hive tauluista, ulkoisista tietokannoista tai olemassaolevista RDD olioista. DataFrame rajapinta on saatavilla Scala, Java, Python ja R -ohjelmointikielille. Scala toteutuksessa DataFrame on riveistä rakentuva Dataset, se on siis yksinkertaisesti tyyppialias Dataset[Row]. [18]

Program 3.14 *DataFrame luominen käyttäen read funktiota*

```
val people = spark.read.json("./people.json")
```

DataFrame objektia luotaessa, Spark arvaa luodun objektin sisäisen mallin.

3.5 Matriisin tekijöihinjako

Matriisin tekijöihinjako on toimi, jossa matriisi hajoitetaan matriisien tuloksi. Matriisi voidaan hajoittaa tekijöihinsä usealla eri tavalla. Seuraava kappale kuvailee matriisin tekijöihinjakoa yleisellä tasolla sekä vuorottelevien pienenmpien neliöiden (Alternating Least Squares, ALS) algoritmia. ALS on Sparkin toteuttama matriisin tekijöihinjako algoritmi ja se perustuu samalle ajatukselle Netflix prize kilpailun voittajan, matriisin tekijöihinjako mallin kanssa. [15]

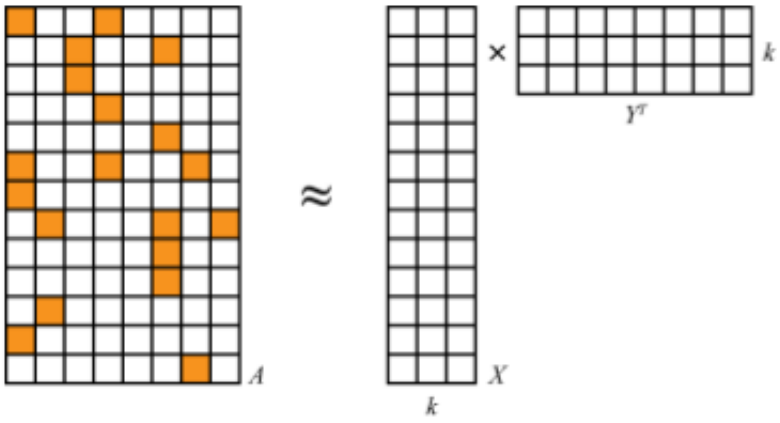
Matriisin tekijöihinjako kuuluu suureen algoritmien luokkaan nimeltä piilevien tekijöiden mallit (Latent-factor models). Piilevien tekijöiden mallit yrittävät selittää usean käyttäjän ja tuotteen välillä havaittuja vuorovaikutuksia käyttämällä suhteellisen pientä määrää havaitsemattomia, piileviä tekijöitä. Voidaan esimerkiksi yrittää selittää miksi ihminen ostaisi tietyn albumin lukemattomien mahdolisuuksien

Kuva 3.2 *DataFrame*

Name	Age	Weight
String	Int	Double
String	Int	Double
String	Int	Double

joukosta kuvailemalla käyttäjiä ja tuotteita mieltymysten perusteella, joista ei ole mahdollista saada tietoa. [15] Piilevää tekijää ei ole mahdollista tarkastella sellaisenaan. Ihmisen terveys on esimerkki piilevästä tekijästä, sillä sitä ei ole mahdollista mitata kuten esimerkiksi verenpainetta.

Kuva 3.3 *Matrix factorization [15]*



Matriisin tekijöihinjako algoritmit käsittelevät käyttäjä- ja tuotetietoja suurena matriisina A. Jokainen rivissä i sekä sarakkeessa j sijaitseva alkio esittää arvostelua jonka käyttäjä on antanut tietylle tuotteelle. [15]

Yleensä A on harva (sparse), jolla tarkoitetaan että useimmat A:n alkiot sisältävät

arvon nolla. Tämä johtuu siitä, että kaikista mahdollisuuksista usein vain muutama käyttäjä-tuote kombinaatio on olemassa. [15]

Matriisin tekijöihinjako mallintaa A:n kahden pienemmän matriisin X ja Y tulona, jotka ovat varsin pieniä. Koska A:ssa on monta riviä ja saraketta, X ja Y sisältävät paljon rivejä mutta vain muutaman (k) sarakkeen. Nämä k saraketta vastaavat piileviä tekijöitä joita käytetään kuvailemaan tiedossa sijaitsevia vuorovaikutuksia. Hajotelma (factorization) on ainoastaan arvio, sillä k on pieni. [15]

Tavanomainen lähestymistapa matriisin tekijöihinjakoon perustuvassa yhteisöllisessä suodatuksessa on kohdella käyttäjä-tuote matriisin alkioita käyttäjien antamina täsmällisinä arvosteluina. Eksplisiittistä tietoa on esimerkiksi käyttäjän antama arvio tuotteelle. Spark ALS kykenee käsittelemään sekä implisiittistä että eksplisiittistä tietoa. Implisiittistä tietoa on esimerkiksi sivujen katselukerrat tai tieto siitä, onko käyttäjä kuunnellut tiettyä artistia. [17] [15]

Usein monissa tosielämän käyttötapauksissa on käytettävissä ainoastaan implisiittistä tietoa kuten katselukerrat, klikkaukset, ostokset, tykkäykset tai jakamiset. Spark MLlib kohtelee tietoa numeroina jotka esittävät havaintojen vahvuutta kuten klikkausten määrä tai kumulatiivinen aika joka käytetään elokuvan katseluun, sen sijaan että mallinnettaisiin arviomatriisia suoraan. Ekplisiittisten arvioioiden sijaan, nämä numerot liittyvät havaittujen käyttäjämieltymysten varmuuteen. Tämän tiedon perusteella malli koettaa etsiä piileviä tekijöitä joiden avulla voidaan ennustaa käyttäjän odotettu arvio tuotteelle. [17]

Näihin algoritmeihin viitataan joskus matriisin täyttö algoritmeina. Tämä johtuu siitä, että alkuperäinen matriisi A saattaa olla harva vaikka matriisitulo XY^T on tiheä. Vaikka tulosmatriisi sisältää arvon kaikille alkioille, se on kuitenkin vain arvio A:sta. [15]

3.5.1 Alternating Least Squares (ALS)

Yhteisöllistä suodatusta käytetään usein suosittelijajärjestelmissä. Nämä tekniikat pyrkivät täyttämään käyttäjä-tuote assosiaatiomatriisin puuttuvat kohdat. Spark MLlib tukee mallipohjaista yhteisösuodatusta, jossa käyttäjiä ja tuotteita kuvaillaan pienellä määrällä piileviä tekijöitä, joita voidaan käyttää puuttuvien kohtien ennustamiseen. Spark MLlib käyttää vaihtelevien pienimpien neliöiden (Alternating Least Squares, ALS) algoritmia näiden piilevien tekijöiden oppimiseen. [17]

Spark ALS yrittää arvata arvostelumatriisin A kahden alemman arvon matriisin, X ja Y, tulona. [16]

$$A = XY^T \tag{3.1}$$

Tyypillisesti näihin arvioihin viitataan tekijämatriiseina. Perinteinen lähestymistapa on iteratiivinen. Jokaisen iteraation aikana, toista tekijämatriisia pidetään vakiona ja toinen ratkaistaan käyttäen pienimpien summien algoritmia. Juuri ratkaistua tekijämatriisia pidetään vuorostaan vakiona kun ratkaistaan toista tekijämatriisia. [16] Spark ALS mahdollistaa massiivisen rinnakkaistamisen sillä algoritmia voidaan suorittaa erikseen. Tämä on erinomainen ominaisuus laajamittaiselle (large-scale) laskenta-algoritmille. [15]

Spark ALS on lohkotettu versio ALS tekijöihinjako algoritmista. Ajatuksena on ryhmittää kaksi tekijäryhmää, *käyttäjät* ja *tuotteet*, lohkoihin. Ryhmittämistä seuraa kommunikaation vähentäminen lähettämällä jokaiseen tuotelohkoon vain yksi kopio jokaisesta käyttäjävektorista iteraation aikana. Vain ne käyttäjä vektorit lähetetään, joita tarvitaan tuotelohkoissa. Vähennetty kommunikaatio saavutetaan valmiiksi laskemalla joitain tietoja suositusmatriisista jotta voidaan päätellä jokaisen käyttäjän ulostulot ja jokaisen tuotteen sisääntulot. Ulostulolla tarkoitetaan niitä tuotelohkoja, joihin käyttäjä tulee myötävaikuttamaan. Sisääntulolla tarkoitetaan niitä ominaisuusvektoreita jotka jokainen tuote ottaa vastaan niiltä käyttäjälohkoilta joista ne ovat riippuvaisia. Tämä mahdollistaa sen, että voidaan lähettää vain taulukollinen ominaisuusvektoreita jokaisen käyttäjä- ja tuotelohkon välillä. Vastaavasti tuotelohko löytää käyttäjän arviot ja päivittää tuotteita näiden viestien perusteella. [16]

Sen sijaan että etsittäisiin alemman tason arviot suositusmatriisille A, etsitäänkin arviot mieltymysmatriisi P:lle, jossa P:n alkiot saavat arvon 1 kun $r > 0$ ja arvon 0 kun $r <= 0$. Eksplisiittisen tuotearvion sijaan arvostelut kuvaavat käyttäjän mieltymyksen vahvuuden luottamusarvoa. [16]

$$A_i Y (Y^T Y)^{-1} = X_i \tag{3.2}$$

ALS operoi kiinnittämällä yhden tuntemattomista u_i ja v_j ja vaihtelemalla tätä

kiinnittämistä. Kun toinen on kiinnitetty, toinen voidaan laskea ratkaisemalla pienimpien neliöiden ongelma. Tämä lähestymistapa on hyödyllinen, koska se muuttaa aiemman, ei-konveksin, ongelman neliömäiseksi, jolloin se voidaan ratkaista optimaalisesti. [1] Alla on [1] mukainen yleinen kuvaus ALS algoritmista:

Program 3.15 *Vaihtelevien pienimpien neliöiden algoritmi (ALS) [1]*

1. Alusta matriisi V asettamalla ensimmäiseksi riviksi elokuvan keskimääräinen arvio ja pieni satunnaisluku jäljelläoleviin alkioihin.

2. Kiinnitä V, ratkaise U minimoimalla RMSE funktio.

3. Kiinnitä U, ratkaise V minimoimalla RMSE funktio.

4. Toista askeleita 2 ja 3 konvergenssiin asti.

RMSE (Root Mean Square Error) on kenties suosituin ennustettujen arvosteluiden tarkkuuden evaluointiin käytetty metriikka. Sitä käytetään yleisesti regressioalgoritmien avulla luotujen mallien evaluointiin. Regressioalgoritmien yhteydessä virheellä tarkoitetaan havainnon todellisen sekä ennustetun numeroarvon välistä eroa. RMSE:n tuntemiseksi tulee tuntea ensin MSE (Mean Square Error). Kuten nimi viittaa, MSE on virheiden neliöiden keskiarvo ja se voidaan laskea neliöimällä jokaisen havainnon virhe ja laskemalla virheiden neliöiden keskiarvo. RMSE voidaan puolestaan laskea ottamalla neliöjuuri MSE:stä. Sekä RMSE että MSE edustavat opetusvirhettä ja ne ilmoittavat kuinka hyvin malli sovittuu opetusdataan. Niiden avulla saadaan selville havaintojen sekä ennustettujen arvojen välinen poikkeavuus. Alhaisemman MSE:n tai RMSE:n omaavan mallin sanotaan sovittuvan paremmin opetusdataan kuin korkeammat virhearvot omaavan mallin. [7]

Suosittelujärjestelmä luo ennustettuja arvosteluita $\hat{r}_{ui}$ testiaineistolle τ käyttäjätuote pareja (u, i) joille todelliset arviot r tunnetaan. [7] Ennustettujen ja todellisten arvioiden välinen RMSE saadaan laskettua seuraavasti:

$$RMSE = \sqrt{\frac{1}{|\tau|} \sum_{(u,i)\in\tau} (\hat{r}_{ui} - r_{ui})^2} \tag{3.3}$$

Konvergenssilla tarkoitetaan jonkin ilmiön lähestymistä ajan kuluessa jotain tiettyä arvoa, tässä tapauksessa sitä, että RMSE ei enään pienene tarpeeksi.

4. TOTEUTUS

GroupLens Research on kerännyt ja laittanut saataville arvioaineistoja MovieLens sivustolta. Aineistot on kerätty useiden aikajaksojen aikana, riippuen aineston koosta. MovieLens ml-latest-small aineisto sisältää 100 000 arviota, jotka ovat antaneet 700 käyttäjää 9000 elokuvalle. Näiden aineistojen haittapuolena on, että ne muuttuvat ajan myötä, eivätkä näin ollen ole sopivia tutkimustulosten raportointiin. Nykyinen, lokakuussa 2016 julkistettu versio on saatavilla projektin versionhallinnassa. Tämä aineisto valittiin, jotta voitaisiin antaa arvioita elokuville, jotka on oikeasti nähty ja myös laitteiston vuoksi. MovieLens ml-latest-small aineisto koostuu *movies.csv* and *ratings.csv* tiedostoista.

Taulukko 4.1 *movies.csv*

movieId	title	genres
1	Toy Story (1995)	Adventure|Animation|Children
2	Jumanji (1995)	Adventure|Children|Fantasy
3	Grumpier Old Men (1995)	Comedy|Romance
4	Waiting to Exhale (1995)	Comedy|Drama|Romance
5	Father of the Bride Part II (1995)	Comedy
6	Heat (1995)	Action|Crime|Thriller
7	Sabrina (1995)	Comedy|Romance
8	Tom and Huck (1995)	Adventure|Children
9	Sudden Death (1995)	Action
10	GoldenEye (1995)	Action|Adventure|Thriller

Taulukko 4.2 *ratings.csv*

userId	movieId	rating	timestamp
1	31	2.5	1260759144
1	1029	3.0	1260759179
1	1061	3.0	1260759182
1	1129	2.0	1260759185
1	1172	4.0	1260759205
1	1263	2.0	1260759151
1	1287	2.0	1260759187
1	1293	2.0	1260759148
1	1339	3.5	1260759125

Toteutuksessa käytettiin RDD-pohjaista rajapintaa, sillä dataset-pohjainen rajapinta ei ole vielä täysin toiminnallinen yhteisöllisen suodatuksen tehtävissä. Aineiston lataaminen voidaan tehdä dataset rajapintaa hyödyntäen, mutta varsinaisen suositus täytyy tehdä RDD rajapintaa käyttäen. Dataset rajapinta tarjoaa useita parannuksia, kuten esimerkiksi yksinkertaisemman tiedon lataamisen.

4.1 MovieLensRecommendation.scala

Ensimmäinen askel itsenäisen spark sovelluksen rakentamisessa on tehdä oikeanlainen kansiorakenne ja luoda $< PROJEKTI >$ *.sbt* niminen tiedosto, jossa kuvaillaan sovelluksen riippuvuudet. Itsenäinen spark sovellus tarkoittaa käyttövalmista *jar* tiedostoa (Java ARchive), joka voidaan jakaa spark klusterille ja se sisältää sekä koodin että kaikki riippuvuudet.
Sovelluksia voidaan ottaa käyttöön klusterissa spark-submit työkalun avulla, joka mahdollistaa Sparkin kaikkien tuettujen klusterinhoitajien käyttämisen yhteinäisen käyttöliittymän kautta. Tällöin käyttäjän ei tarvitse määrittää sovellusta toimimaan erikseen kaikkien kanssa.

Program 4.1 *Sovelluksen paketointi sbt työkalulla*

```
sbt package
```

Program 4.2 *Sovelluksen käyttöönotto klusterissa*

```
spark-submit --class "MovieLensALS" \
        --master local[4] \
        movielens-recommendations_2.11-1.0.jar
```

Alla olevassa esimerkissä 4.3 ladataan työssä käytetyt suositukset RDD rajapintaa käyttäen.

Program 4.3 *Suositusten lataaminen RDD rajapintaa käyttäen*

```
1  val ratings = sc.textFile("ml-latest-small/ratings.csv")
2    .filter(arr => arr(0) != "userId")
3    .map { line =>
4      val fields = line.split(",")
5      val timestamp = fields(3).toLong % 10
6      val userId = fields(0).toInt
7      val movieId = fields(1).toInt
8      val rating = fields(2).toDouble
9
10     (timestamp, Rating(userId, movieId, rating))
11   }
```

Alla olevassa esimerkissä 4.4 ladataan työssä käytetyt suositukset Dataset rajapintaa käyttäen.

Program 4.4 *Suositusten lataaminen Dataset rajapintaa käyttäen*

```
1  val ratings = spark.read.csv("ml-latest-small/ratings.csv")
2    .filter(arr => arr(0) != "userId")
3    .map { fields =>
4      val userId = fields(0).asInstanceOf[String].toInt
5      val movieId = fields(1).asInstanceOf[String].toInt
6      val rating = fields(2).asInstanceOf[String].toFloat
7      val timestamp = fields(3).asInstanceOf[String].toDouble % 10
8
9      Rating(userId, movieId, rating, timestamp)
10   }
```

Alla olevassa listauksessa on esitetty toteutetun suosittelujärjestelmän implementaatio.

Program 4.5 *MovieLensALS.scala*

```
1  import org.apache.spark.mllib.recommendation._
2
3  object MovieLensALS {
4    def main(args: Array[String]) {
5
6      val conf = new SparkConf()
7        .setAppName("MovieLensALS")
8        .set("spark.executor.memory", "4g")
9      val sc = new SparkContext(conf)
10
```

```
11      /* load personal ratings */
12      val personalRatings = Source.fromFile("personalRatings.txt")
13        .getLines()
14        .map { line =>
15            val fields = line.split(",")
16            Rating(fields(0).toInt, fields(1).toInt, fields(2).toDouble)
17        }.toSeq
18
19      val personalRatingsRDD = sc.parallelize(personalRatings, 1)
20
21      /* load ratings and movie titles */
22      val ratings = sc.textFile("ml-latest-small/ratings.csv")
23        .filter(x => !isHeader("userId", x))
24        .map { line =>
25            val fields = line.split(",")
26            val timestamp = fields(3).toLong % 10
27            val userId = fields(0).toInt
28            val movieId = fields(1).toInt
29            val rating = fields(2).toDouble)
30
31            (timestamp, Rating(userId, movieId, rating)
32        }
33
34      val movies = sc.textFile("ml-latest-small/movies.csv")
35        .filter(x => !isHeader("movieId", x))
36        .map { line =>
37            val fields = line.split(",")
38            (fields(0).toInt, fields(1))
39        }.collect().toMap
40
41      val numRatings = ratings.count
42      val numUsers = ratings.map(_._2.user).distinct.count
43      val numMovies = ratings.map(_._2.product).distinct.count
44
45      val numPartitions = 4
46      val training = ratings.filter(x => x._1 < 6)
47        .values
48        .union(personalRatingsRDD)
49        .repartition(numPartitions)
50        .cache()
51      val validation = ratings.filter(x => x._1 >= 6 && x._1 < 8)
52        .values
53        .repartition(numPartitions)
54        .cache()
```

```
55      val test = ratings.filter(x => x._1 >= 8).values.cache()
56
57      val numTraining = training.count()
58      val numValidation = validation.count()
59      val numTest = test.count()
60
61      /* training */
62
63      val ranks = List(8, 12)
64      val lambdas = List(1.0, 10.0)
65      val numIters = List(10, 20)
66      var bestModel: Option[MatrixFactorizationModel] = None
67      var bestValidationRmse = Double.MaxValue
68      var bestRank = 0
69      var bestLambda = -1.0
70      var bestNumIter = -1
71      for (rank <- ranks; lambda <- lambdas; numIter <- numIters) {
72        val model = ALS.train(training, rank, numIter, lambda)
73        val validationRmse =
74            computeRmse(model, validation, numValidation)
75
76        if (validationRmse < bestValidationRmse) {
77            bestModel = Some(model)
78            bestValidationRmse = validationRmse
79            bestRank = rank
80            bestLambda = lambda
81            bestNumIter = numIter
82        }
83      }
84
85      val testRmse = computeRmse(bestModel.get, test, numTest)
86
87      val myRatedMovieIds = personalRatings.map(_.product).toSet
88      val candidates = sc.parallelize(
89          movies.keys.filter(!myRatedMovieIds.contains(_)).toSeq
90      )
91      val recommendations = bestModel.get
92        .predict(candidates.map((0, _)))
93        .collect()
94        .sortBy(- _.rating)
95        .take(10)
96
97      var i = 1
98      println("Movies recommended for you:")
```

```
99      recommendations.foreach { r =>
100       println("%2d".format(i) + ": " + movies(r.product))
101       i += 1
102     }
103
104     // clean up
105     sc.stop()
106   }
107
108   def isHeader(id: String, line: String): Boolean = line.contains(id)
109
110   /** Compute RMSE */
111   def computeRmse(
112       model: MatrixFactorizationModel,
113       data: RDD[Rating],
114       n: Long
115   ): Double = {
116     val predictions: RDD[Rating] =
117       model.predict(data.map(x => (x.user, x.product)))
118     val predictionsAndRatings =
119       predictions.map(x => ((x.user, x.product), x.rating))
120         .join(data.map(x => ((x.user, x.product), x.rating)))
121         .values
122
123     math.sqrt(
124       predictionsAndRatings
125         .map(x => (x._1 - x._2) * (x._1 - x._2))
126         .reduce(_ + _) / n
127     )
128   }
129 }
```

Rivillä 1 tuodaan saataville kaikki recommendation paketin sisältämät kentät tai metodit käyttäen *import* avainsanaa. Rivillä 3 määritellään MovieLensALS niminen objekti. Objekti on nimetty instanssi joka sisältää jäseniä kuten kenttiä (field) sekä metodeita (method). Rivillä 4 on määritelty *main* funktio tarkoittaa sitä, että määritelty objekti *MovieLensALS* on ohjelman aloituspiste (entry point) sillä *main* funktio sisältää tietynlaisen allekirjoituksen eli tietynlaiset parametrit. Riveillä 6-9 luodaan *SparkConf* objekti, jonka avulla luodaan ohjelman käyttöön uusi *SparkContext* objekti. *SparkContext* objektin avulla päästään käsiksi Sparkin sisäisiin toiminnallisuuksiin. Riveillä 11-17 ladataan henkilökohtaiset suositukset tekstitiedostosta nimeltä *personalRatings.txt*, pilkotaan tiedoston rivit pilkun

kohdalta ja luodaan uusia *Rating* objekteja yhtä monta, kuin tiedostossa on rivejä. Rivillä 19 ladatut suositukset muutetaan vielä RDD (Resilient Distributed Dataset) muotoiseksi käyttäen *sc.parallelize* funktiota. Funktiolle annettava toinen parametri tarkoittaa hajautuksen määrää, eli kuinka monelle solmulle klusterissa tiedosto halutaan hajauttaa. Riveillä 22-36 luodaan RDD oliot *ratings* ja *movies* lataamalla kaksi erillistä csv tiedostoa. Tiedostoista suodatetaan ensin pois otsikkorivit käyttäen $isHeader$ apufunktiota. Tämän jälkeen tiedosto käydään läpi rivi kerrallaan ja pätkitään pilkulla erotetut arvot taulukkoon käyttäen Scalan String luokan sisäänrakennettua *split* funktiota. Tämän jälkeen taulukossa olevista arvoista muodostetaan Tupleja. Riveillä 44-54 valmistellaan opetus, validaatio sekä testidatat. Rivillä 47 opetusdataan lisätään omat henkilökohtaiset arvostelut käyttäen RDD:n union funktiota. Riveillä 64-83 suoritetaan varsinainen mallin opetus. Opetus suoritetaan niin, että opetetaan muutama versio mallista, ja lopuksi valitaan opetetuista malleista paras käyttäen RMSE-metriikkaa mittarina. Varsinainen mallin opetus tehdään käyttäen ALS kirjaston funktiota *train* ja tarkemmin sanottuna *train* funktion ylikuormitettua versiota, joka ottaa sisääntulonaan *ratings*, *rank*, *iterations* sekä *lambda* parametrit. Ratings on RDD Rating olioita, jotka sisältävät käyttäjän id:n, elokuvan id:n ja suosituksen. Rank tarkoittaa piilevien ominaisuuksien sisällytettävää määrää. Iterations tarkoittaa ALS algoritmin iteraatioiden määrää. Lambda tarkoittaa regularisaatio parametria, jolla yritetään ehkäistä mallin ylioppimista. Riveillä 89-102 haetaan henkilökohtaiset suositukset käyttämällä mallin *predict* metodia, joka ottaa parametrinaan mahdollisten elokuvien joukon. Mahdollisilla elokuvilla tarkoitetaan elokuvia joita käyttäjä ei ole vielä nähnyt, eli ne eivät sisälly *personalRatings* muuttujan sisältämiin elokuviin. Rivillä 105 kutsutaan lopuksi *sparkContext* objektin *stop* funktiota, jolla kerrotaan että laskenta on suoritettu loppuun. Rivillä 108 määritellään apufunktio $isHeader$, jota käytetään apuna suodattamaan lähtöaineistosta ei halutut rivit pois. Riveillä 111-128 määritellään apufunktio $computeRMSE$, jonka avulla evaluoidaan opetetun mallin virhettä.

5. TULOKSET

Tässä kappaleessa käsitellään työn tärkeimpiä tuloksia.

5.1 Sisääntulot

Tässä osassa esitetään suosittelujärjestelmän sisääntulot.

Taulukko 5.1 *Arvostellut elokuvat*

Tunniste	Nimi	Arvostelu
112897	The Expendables 3 (2014)	4.0
116887	Exodus: Gods and Kings (2014)	4.0
117529	Jurassic World (2015)	4.0
118696	The Hobbit: The Battle of the Five Armies (2014)	4.5
128520	The Wedding Ringer (2015)	4.5
122882	Mad Max: Fury Road (2015)	4.0
122886	Star Wars: Episode VII - The Force Awakens (2015)	4.5
131013	Get Hard (2015)	4.0
132796	San Andreas (2015)	3.0
136305	Sharknado 3: Oh Hell No! (2015)	1.0
136598	Vacation (2015)	4.0
137595	Magic Mike XXL (2015)	1.0
138208	The Walk (2015)	2.0
140523	The Visit (2015)	3.5
146656	Creed (2015)	4.0
148626	The Big Short (2015)	4.5
149532	Marco Polo: One Hundred Eyes (2015)	4.5
150548	Sherlock: The Abominable Bride (2016)	4.5
156609	Neighbors 2: Sorority Rising (2016)	3.5
159093	Now You See Me 2 (2016)	4.0
160271	Central Intelligence (2016)	4.0

Taulukossa 5.1 on esitetty suosittelujärjestelmän sisääntulona annetut jo nähdyt elokuvat. Sisääntulon rakenne on seuraava: sarakkeessa yksi sijaitsee elokuvan tunniste, sarakkeeseen kaksi on sijoitettu elokuvan nimi ja sarakkeessa kolme sijaitsee elokuvalle annettu arvio asteikolla 0-5. Taulukossa olevat arvot ovat vain pieni osa kaikesta opetukseen käytetystä datasta.
Jonkun työkalun avulla arvosteluiden antaminen olisi ollut paljon mielekkäämpää.

5.2 Suositukset

Tässä osassa käsitellään suosittelujärjestelmän tarjoamat suositukset, eli työn varsinaiset tulokset.

Taulukko 5.2 *Toteutetun järjestelmän suosittelemat elokuvat*

Numero	Nimi	Tyylilajit
1	Death of a superhero (2011)	Animation, Drama
2	Prisoner of the Mountains (1996)	War
3	Funeral in Berlin (1966)	Action, Drama, Thriller
4	Caveman (1981)	Comedy
5	Dream With the Fishes (1997)	Drama
6	Erik the Viking (1989)	Adventure, Comedy, Fantasy
7	Dead Man's Shoes (2004)	Crime, Thriller
8	Excision (2012)	Crime, Drama, Horror, Thriller
9	Mifune's Last Song (1999)	Comedy, Drama, Romance
10	Maelström (2000)	Drama, Romance

Taulukossa 5.2 on suosittelujärjestelmältä saaduista suosituksista 10 ensimmäistä, mukaan on lisätty myös elokuvien tyylilajit, jotta on helpompi arvioida suosituksien paikkansapitävyyttä.

Taulukko 5.3 *Vertailtavan järjestelmän suosittelemat elokuvat*

Tunniste	Korrelaatio	Eniten korreloivat elokuvat
112897	-	-
116887	-	-
117529	1.0	159858, 142997, 140267
118696	1.0	122890, 130490, 135436
128520	1.0	131013
122882	1.0	127136, 127202, 128520
122886	1.0	122890, 128520, 129428
131013	1.0	132796, 134368, 135569
132796	1.0	133419, 134368, 134853
136305	-	-
136598	1.0	139385, 139642, 138036 (0.867)
137595	-	-
138208	1.0	142448
140523	-	-
146656	1.0	152077 (0.999), 152081, 156609
148626	1.0	157296, 152077 (0.499)
149532	-	-
150548	-	-
156609	-	-
159093	-	-
160271	-	-

Itse toteutetun järjestelmän suositukset saadaan opetetun mallin ennustamina kun taas vertailtavan järjestelmän suositukset on luotu hieman yksinkertaisemmin. Itse toteutetun järjestelmän tuloksissa puutoksia ei ole havaittavissa, sillä järjestelmältä voi kysyä tuloksia niin monta kuin elokuvia sisääntuloaineistossa on. Tuloksien paikaansapitävyyttä voidaan arvioida esimerkiksi tyylilajien perusteella. Elokuvan hyvyys on hyvinkin henkilökohtainen kokemus, eikä siihen oteta kantaa tässä työssä.

6. YHTEENVETO

Tässä kappaleessa esitetään yhteenveto.

6.1 Johtopäätökset

Suosittelujärjestelmän rakentamiseen on olemassa monia mahdollisia toteutusvaihtoehtoja, kuten SQL ja ElasticSearch. Apache Spark vaikutti mielenkiintoiselta opiskelukohteelta ja tulevaisuuden kannalta hyödylliseltä. Scala ohjelmoinnin oppiminen vaikutti myöskin teknologian valintaan.

Olemassaolevien suosittelujärjestelmien tai analytiikkajärjestelmien evaluointi tulisi suorittaa ennen suosittelujärjestelmän valintaa.

Lopulta kaikista vaikein asia oli löytää oikea lähestymistapa tähän kyseiseen tehtävään. Yrittämisen ja lukuisten epäonnistumisien jälkeen oikea teknologioiden joukko sekä varsinainen toimiva esimerkki löydettiin.

Suuremman datasetin käyttäminen, sekä isomman arvostelumäärän tarjoaminen järjestelmälle voisi parantaa tuloksia.

6.2 Tulevaa työtä

Mikäli käytettävissä oleva laitteisto sallisi, olisi mahdollista rinnakkaistaa suoritusta sekä samalla kasvattaa käytettävän datasetin kokoa.

MLlib kirjastoa voitaisiin tutkia uudestaan siinä vaiheessa, kun Dataset rajapintaa voidaan käyttää MLlib:n kanssa. Toteutusta yritettiin myös Dataset rajapintaa hyväksikäyttäen, mutta kaikki toiminnallisuudet eivät olleet vielä käytössä.

Viimeaikoina useat palveluntarjoajat, kuten Telegram ja Microsoft, ovat esitelleet bot sovelluskehyksiä palveluihinsa. Botti on web-palvelu, joka keskustelee käyttäjien

kanssa pikaviesti kanavalla. Käyttäjät voivat aloittaa keskustelun botin kanssa missä tahansa kanavalla, jota bot on määritety kuuntelemaan. Keskustelut voivat olla vapaamuotoisia tai ne voivat koostua tietyistä, ennaltamäärätyistä valinnoista. [10]

Ajatus apureista tai boteista ei ole uusi, sillä esimerkiksi IRC (Internet Relay Chat) -kanavilla botteja on ollut olemassa jo pitkän aikaa, mutta nyt boteille on olemassa suositumpia alustoja. Kuten mikä tahansa ajatus, myös suosittelujärjestelmä voitaisiin toteuttaa botin avulla käytettäväksi. Esimerkiksi ElasticSearch muodostaa RESTful rajapinnan jota vasten botti voisi ajaa kyselyitä.

BIBLIOGRAPHY

[1] C. Aberger, "Recommender: An analysis of collaborative filtering techniques," 2014. [Online]. Available: http://cs229.stanford.edu/proj2014/Christopher%20Aberger,%20Recommender.pdf

[2] C. C. Aggarwal, *Recommender Systems*. Springer International Publishing, 2016.

[3] BookLens. [Online]. Available: https://booklens.umn.edu/

[4] R. Burke, "Hybrid recommender systems: Survey and experiments," *User Modeling and User-Adapted Interaction*, vol. 12, no. 4, pp. 331–370. [Online]. Available: http://dx.doi.org/10.1023/A:1021240730564

[5] D. Eppstein. Directed acyclic graph. [Online]. Available: https://en.wikipedia.org/wiki/Directed_acyclic_graph#/media/File:Topological_Ordering.svg

[6] S. K. Gorakala and M. Usuelli, *Building a Recommendation Engine with R*, 1st ed. Packt Publishing, 2015.

[7] M. Guller, *Big Data Analytics with Spark*, 1st ed. Apress, 2015.

[8] Y. Koren, R. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," 2009. [Online]. Available: https://datajobs.com/data-science-repo/Recommender-Systems-[Netflix].pdf

[9] G. Linden, B. Smith, and J. York, "Amazon.com recommendations," *IEEE INTERNET COMPUTING*, pp. 76–79, 2003. [Online]. Available: http://www.cin.ufpe.br/~idal/rs/Amazon-Recommendations.pdf

[10] Microsoft, "Bots." [Online]. Available: https://docs.botframework.com/en-us/

[11] MovieLens. [Online]. Available: https://movielens.org/info/about

[12] Oracle. Jar file overview. [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/jar/jarGuide.html

[13] ——. The java virtual machine specification. [Online]. Available: https://docs.oracle.com/javase/specs/jvms/se10/html/index.html

[14] F. Ricci, L. Rokach, B. Shapira, and P. B. Kanto, *Recommender Systems Handbook*, 1st ed. Springer, 2011.

[15] S. Ryza, U. Laserson, S. Owen, and J. Wills, *Advanced Analytics with Spark*. O'Reilly Media, Inc., 2015.

[16] Spark. (2014) ALS. [Online]. Available: http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.mllib.recommendation.ALS

[17] ——. (2014) Collaborative filtering - rdd-based api. [Online]. Available: http://spark.apache.org/docs/latest/mllib-collaborative-filtering.html

[18] ——. (2014) Spark programming guide. [Online]. Available: http://spark.apache.org/docs/latest/programming-guide.html

[19] ——. (2016) Dataset. [Online]. Available: https://spark.apache.org/docs/2.1.0/api/java/org/apache/spark/sql/Dataset.html

[20] ——. (2016) Rdd. [Online]. Available: https://spark.apache.org/docs/2.1.0/api/scala/index.html#org.apache.spark.rdd.RDD

[21] ——. (2016) Spark sql programming guide. [Online]. Available: http://spark.apache.org/docs/latest/sql-programming-guide.html

[22] M. Technologies. Apache spark. [Online]. Available: https://mapr.com/products/product-overview/apache-spark/