Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Grundseminararbeit

Timo Lange

Big Data Systeme und Recommendations

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Timo Lange

Big Data Systeme und Recommendations

Grundseminararbeit eingereicht im Rahmen der Grundseminarprüfung

im Studiengang Master of Science Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Kai von Luck
Zweitgutachter: Prof. Dr. Tim Tiedemann

Eingereicht am: 31. August 2017

Timo Lange

Thema der Arbeit

Big Data Systeme und Recommendations

Stichworte

Big Data, Recommendation, Flink, Spark, ALS, eALS

Kurzzusammenfassung

Diese Arbeit befasst sich mit den beiden Big Data Systeme Apache Flink und Apache Spark, sowie Recommender Systemen. Es werden Grundkonzepte der Big Data Systeme, sowie deren Gemeinsamkeiten und Besonderheiten beschrieben. Es folgt ein Überblick über Recommender Systeme und den dabei eingesetzte Verfahren. Des Weiteren wird auf Forschungsarbeiten zum Thema Recommender Systeme eingegangen. Diese setzen sich vornehmlich mit der Matrixfaktorisierung als *collaborative filtering* Methode auseinander. Schlussendlich wird ein Ausblick auf weitere Schritte gegeben.

Timo Lange

Title of the paper

Big Data systems and Recommendations

Keywords

Big Data, Recommendation, Flink, Spark, ALS, eALS

Abstract

This work deals with the two Big Data systems Apache Flink and Apache Spark, as well as Recommender systems. The basic concepts of the Big Data systems as well as their similarities and peculiarities are described. This will be followed by an overview of recommender systems and the used techniques. In addition, some papers on recommender systems will be discussed. These are mainly concerned with the matrix factorization as a *collaborative filtering* method. In the end, an outlook on further steps is given.

Inhaltsverzeichnis

1 Einleitung und Motivation **1**

2 Big Data Systeme **2**

2.1 Apache Flink 2

2.2 Apache Spark 3

2.3 Ähnliche Konzepte in Flink und Spark 4

2.4 Besonderheiten von Flink und Spark 5

2.4.1 Flink 5

2.4.2 Spark 5

3 Recommender Systeme **6**

3.1 Verfahren 6

3.1.1 Collaborative Filtering (CF) 6

3.1.2 Content-Based 7

3.1.3 Knowledge-Based 7

3.1.4 Hybrid Systems 7

4 Arbeiten zum Thema **8**

4.1 Matrix Factorization Techniques for Recommender Systems 8

4.1.1 Recommender System Strategien 8

4.1.2 Matrix Factorization Methoden 8

4.1.3 Ein basis Matrix Factorization Modell 8

4.1.4 Learning Algorithmen 9

4.1.5 Verfahren zur Verbesserung der Vorhersagegenauigkeit 9

4.2 Large-scale parallel collaborative filtering for the netflix prize 10

4.3 Fast Matrix Factorization for Online Recommendation with Implicit Feedback 10

4.3.1 MF Method for Implicit Feedback 11

4.3.2 The authors’ IMPLICIT MF METHOD 11

4.3.3 Optimization by ALS 12

4.3.4 Generic Element-wise ALS Learner 13

4.3.5 Fast eALS Learning Algorithm 13

4.3.6 Online Update 13

4.3.7 Probleme 14

5 Ausblick **15**

1 Einleitung und Motivation

Auf Grund der großen Datenmengen, die bei einem realen Anwendungsfall, wie etwa bei der Empfehlung von Produkten bei z.B. Netflix, Amazon oder Spotify, anfallen ist es nötig Systeme zu verwenden, welche hoch verteilt arbeiten. Ein gutes Beispiel für ein Anwendungsfeld von Big Data Systemen sind damit Recommendation Systeme. Zunächst werden die zwei Big Data Systeme Apache Flink und Apache Spark vorgestellt, mit denen Batch- als auch Streamdaten verarbeitet werden können. Beide Systeme finden Anwendung in der Industrie, wobei Apache Spark das verbreitetere von beiden ist. Nachfolgend werden Konzepte vorgestellt, die sowohl von Flink als auch von Spark genutzt werden. Beide verwenden unterschiedliche Ansätze, auf die hier nicht im Detail eingegangen wird, die aber das gleiche Ziel verfolgen. Anschließend werden Konzepte vorgestellt, mit denen sich Flink bzw. Spark voneinander abheben. Darauf folgend werden in Kürze verschiedene Recommendation Verfahren nach Ricci u. a. und Jannach u. a. vorgestellt, um einen Überblick über die gängigsten Verfahren zu vermitteln. Zuletzt werden drei verschiedene Paper zum Thema Recommendation vorgestellt, wobei der Fokus auf dem dritten Paper von He u. a. liegt, welches von diesen das aktuellste ist. Die erste Arbeit von Koren u. a. (2009–08) behandelt verschiedene Techniken um die Vorhersagegenauigkeit bei Matrixfaktorisierung zu verbessern. Für die Matrixfaktorisierung wird hierbei vornehmlich der *Alternating Least Squares* (ALS) Algorithmus behandelt. Die zweite Forschungsarbeit von Zhou u. a. (2008) setzt ebenfalls auf den ALS Algorithmus, allerdings in der Ausprägung mit Weighted-λ -Regularization (ALS-WR). Beide Paper sind im Rahmen des Netflix-Preis-Kontests Netlfix Inc. entstanden. Die dritte Arbeit von He u. a. befasst sich ebenfalls mit der Matrixfaktorisierung mittels ALS und fokussiert sich dabei auf Online-Updates des Modells sowie implizites Feedback. Der genutzte Algorithmus ist hier *element-wise Alternating Least Squares* (eALS). Abschließend wird ein Ausblick gegeben, welche weiteren Schritte nach dieser Arbeit folgen sollen.

2 Big Data Systeme

2.1 Apache Flink

Apache Flink ist ein Framework für verteilte Stream- und Batch-Datenverarbeitung und basiert auf einer Streaming-Dataflow-Engine, mit der sowohl Stream- als auch Batch-Daten verarbeitet werden. Dabei werden Batch-Daten als ein Speziallfall von Streams betrachtet, welche als endliche Streams verabeitet werden. Foundation (2016c,b)

Abbildung 2.1 gibt eine Übersicht über die Architektur und Komponenten von Flink. Es werden APIs für Java und Scala angeboten, sowie eine *Table* genannte API um deklarativ, ähnlich wie SQL, Anfragen an das System zu stellen. Darüber hinaus gibt es die integrierten Bibliotheken FlinkML für Maschinelles Lernen, Gelly für die Graph Verarbeitung und FlinkCEP für Complex Event Processing.

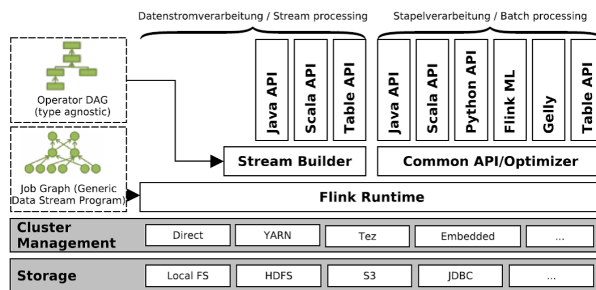

Abbildung 2.1: Architektur und Komponentenübersicht der Apache Flink Plattform Traub u. a. (2015)

Die Hauptabstraktion von Flink sind DataSets und DataStreams.

DataSets sind verteilte immutable Collections, welche Duplikate enthalten können und endlich sind. Erzeugt werden sie aus Quellen wie etwa Dateien, Java Collections, Apache Kafka und HDFS. Um mit den Daten zu Arbeiten, werden auf den DataSets Transformationen ausgeführt, die wiederum ein neues DataSet liefern. Die Ausgabe der Daten erfolgt über Datensenken, die u.a. Dateien, Apache Kafka StdOut oder Datenbanken sein können. Die interne Verarbeitung erfolgt mittels eines so genannten Job Graphen.

DataStreams haben die gleichen Eigenschaften wie DataSets und werden intern gleich verarbeitet. Der Unterschied von DataStreams ist, dass diese unendliche Collections darstellen.

JobGraph Der JobGraph ist die interne Datenstruktur für die Verarbeitung der Daten. Dieser ist dargestellt in Abbildung 2.2. Er ist ein generisches, paralleles Datenflussprogramm und besteht

aus Operatoren (Job-Vertex) und Zwischenergebnissen (IntermediateDataSet). Der ExecutionGraph ist eine parallele Version des JobGraph, bei dem es pro Job-Vertex im JobGraph einen Execution-Vertex pro parallelem Subtask gibt und die Zwischenergebnisse partitioniert sind. Lange (2017)

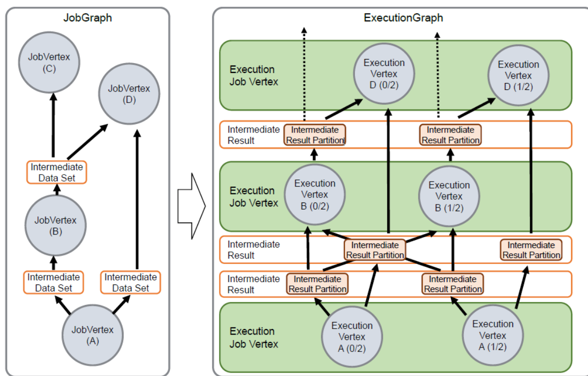

Abbildung 2.2: JobGraph und ExecutionGraph Foundation (2016a)

2.2 Apache Spark

Apache Spark ist ebenso wie Apache Flink ein Framework für verteilte Stream- und Batch- Datenverarbeitung, basiert allerdings nicht auf Streams, sondern verarbeitet alle Daten als Batches. Abbildung 2.3 gibt eine Übersicht über den Komponentenstack von Spark. Neben dem Spark Core, welcher APIs für Java und Scala bereitstellt, bringt Spark auch integrierte Bibliotheken mit. Diese umfassen SparkSQL für deklarative Abfragen mittels SQL, Spark Streaming für die Verarbeitung von Datenströmen, MLlib für Maschinelles Lernen und GraphX für die Arbeit mit Graphen. Lange (2017)

Die Hauptabstraktion von Spark sind Resilient Distributed Datasets (RDD) und Discretized Streams (DStreams).

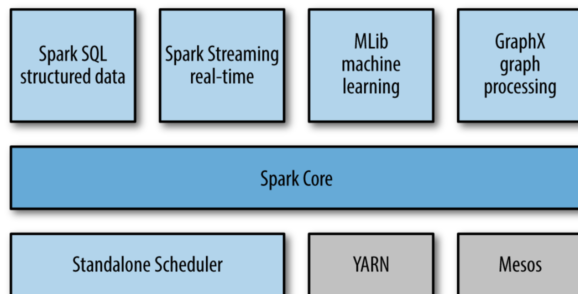

Abbildung 2.3: Der Komponenten-Stack von Apache Spark

RDD sind verteilte immutable Collections und stellen eine Erweiterung des MapReduce Models dar. RDDs erlauben zwei Typen von Operationen: *Transformations* und *Actions*, wobei *Transformations* neue RDDs zurückliefern und *Actions* genutzt werden, um Werte zurück zu liefern. Intern wird ein so genannter Lineage Graph genutzt um die Daten zu verarbeiten. Zaharia (2016); Zaharia u. a. (2012a)

DStream bestehen aus vielen so genannten MicroBatches, also aus vielen kleinen RDDs. Somit werden in Spark die Streamdaten in kleine RDDs aufgeteilt, welche anschließend nacheinander abgearbeitet werden. Die API ist größtenteils mit der von RDDs identisch und es ist möglich beide APIs gemischt

zu nutzen. Zaharia u. a. (2012b); Tathagata Das (2015–01–15); Tathagata Das u. a. (2015–07–30); The Apache Software Foundation (2016)

Lineage Graph Der Lineage Graph ähnelt Flinks JobGraph sehr. Jedes RDD merkt sich den Graph von Operationen, aus denen es erzeugt wurde. So entsteht ein Graph aus Operationen und Zwischenergebnissen, dargestellt in Abbildung 2.4. So wird durch jede Transformation auf ein RDD ein neues RDD erzeugt. Die blaue Umrandung fasst eine RDD zusammen, die aus mehreren Partitionen, hier als blaue Kästen dargestellt, besteht. Die grau gestrichelten Stages fassen RDDs und Operationen darauf zusammen, die direkt auf einem Knoten des Clusters ohne Kommunikation mit anderen Knoten durchgeführt werden können.

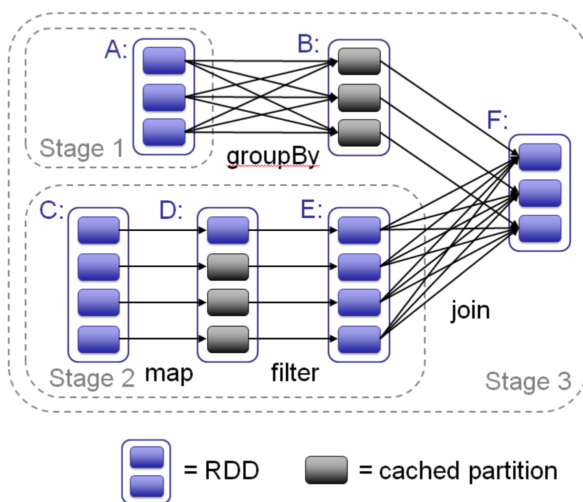

Abbildung 2.4: RDD Operationen als Lineage Graph, aufgeteilt in Stages Matei Zaharia

2.3 Ähnliche Konzepte in Flink und Spark

Speicherverwaltung Beide Systeme verwenden eine eigene Speicherverwaltung innerhalb der JVM. So werden Zeitverluste durch Garbage Collection vermieden und es ermöglicht die Ausführung von Out-of-Core Algorithmen.

Off-heap Memory Als Alternative zur Speicherverwaltung innerhalb der JVM, kann der Off-heap Memory genutzt werden, also der Speicherbereich außerhalb der JVM. Hierdurch ist der Garbage Collector komplett ausgegrenzt und es kann beliebig viel Speicher genutzt werden, ohne dass die JVM sehr groß wird.

Typen Serialisierung Die Daten werden in Speichersegmente serialisiert, was etwa durch den Java build-in Serialisierer, Kryo oder eigene Serialisierer von Spark und Flink geschieht. Hierdurch kann Speicherplatz gespart werden und durch direktes Arbeiten auf den serialisierten Daten bzw. auf den nur teilweise deserialisierten Daten können die Operationen sehr effizient ausgeführt werden. Zudem wird ein einfaches Verschieben der serialisierten Daten zwischen Arbeitsspeicher und Festplatte möglich.

Integrierte Bibliotheken Beide Systeme beinhalten Bibliotheken, um schnell und einfach für Maschinelles Lernen, Graphverarbeitung und deklarative Abfragen eingesetzt werden zu können. Lange (2017)

2.4 Besonderheiten von Flink und Spark

2.4.1 Flink

Streaming Windows Es gibt zeitgesteuerte und datengesteuerte Fenster, die über Streams definiert werden können. So ist es etwa möglich, für alle Events, die in ein gleitendes Fenster von 5 Minuten fallen, Operationen auszuführen wie z.B. permanent einen Durchschnitt der letzten 5 Minuten zu berechnen. Gleiches kann auch für ein gleitendes Fenster von einer vorgegebenen Menge an eingehenden Records durchgeführt werden.

Iterationen (Bulk & Delta) Iterationen sind mit Hilfe spezieller Operatoren direkt in den Dataflow eingebettet und können so effizient systemintern durchgeführt werden. Dies bedeutet, dass keine Schleifen im User-Code ausgeführt werden, wodurch sonst jedes Iterationsergebnis zum ausführenden Programm übertragen und von dort wieder zurück zu Flink für die nächste Iteration übertragen werden müsste. Flink unterscheidet zwischen Bulk- und Delta-Iteration, wobei Bulk-Iteration bei jedem Teilschritt ein komplettes Teilergebnis liefern und Delt-Iteration nur mit den geänderten Daten des Teilergebnisses rechnen, wodurch viel Rechenzeit eingespart werden kann.

FlinkCEP - Complex Event Processing FlinkCEP ist eine Library um komplexe Event Muster innerhalb eines Streams zu erkennen. Dabei können komplexe Event Muster aus *matching sequences* erstellt werden, wobei Muster aus States aufgebaut sind und der Übergang zwischen den States durch Conditions festgelegt wird.

2.4.2 Spark

Persistenz Spark bietet die Möglichkeit RDDs im RAM oder auf der Festplatte zu persistieren. Dies kann genutzt werden um ein häufig genutztes Zwischenergebnis wieder zu verwenden und eine erneute Berechnung des RDD zu umgehen.

RDD in DStream Nutzbar Spark bietet die Möglichkeit RDDs in DStreams zu verwenden und so die APIs für Batch- und Streamverarbeitung gemischt zu verwenden. Lange (2017)

3 Recommender Systeme

Recommender Systeme (RS) sind Softwaretools und Techniken, um Vorschläge für Items mit einem Nutzen für einen User zu liefern.Ricci u. a. Die Konstruktion von Systemen, die Nutzer in ihrer (online) Entscheidungsfindung unterstützen, ist das Hauptziel im Bereich der Empfehlungssysteme. Im Besonderen ist das Ziel von RS einfach zugängliche, hoch qualitative Empfehlungen für eine große Gemeinschaft zu liefern.Jannach u. a. Somit helfen RS Nutzern mit dem zunehmenden Information Overload James E. Short fertig zu werden.

Oder: "The Web, they say, is leaving the era of search and entering one of discovery. What's the difference? Search is what you do when you're looking for something. Discovery is when something wonderful that you didn't know existed, or didn't know how to ask for, finds you." Jeffrey M. O'Brien

Weiterhin können RS genutzt werden, um Nutzer beispielsweise zum Kauf beeinflussen. RS helfen, die Nutzerpräferenzen einzufangen und es hat sich gezeigt, dass diese die Nutzerzufriedenheit und die Einnahmen von Content-Anbietern steigern können. He u. a.

3.1 Verfahren

3.1.1 Collaborative Filtering (CF)

Beim CF wird angenommen, dass User mit ähnlichen Interessen in der Vergangenheit auch in Zukunft ähnliche Interessen haben werden. Wenn zwei User eine stark überlappende Kaufhistorie aufweisen und ein User ein neues Item kauft, so ist dies evtl. auch für den anderen interessant. Durch eine hohe Anzahl an Bewertungen kollaborieren die Nutzer miteinander, wodurch pure CF-Verfahren kein Wissen über die Items selbst brauchen. Für eine gute Vorhersagegenauigkeit wird eine möglichst große Anzahl an Nutzern benötigt. Zudem muss eine Nutzerhistorie vorhanden sein und neue Nutzer und Items müssen „neu gelernt" werden, was ein Update oder eine Neuberechnung des Empfehlungsmodells bedeutet. Des Weiteren wird beim CF primär nach zwei Methoden unterschieden. Den Neighborhood Methoden und den Latent Factor Models. Die Neighborhood Methoden befassen sich mit der Berechnung von Beziehungen zwischen Produkten oder Nutzern. Bei Latent Factor Models wird versucht die Bewertungen zu erklären, indem Nutzer und Produkte anhand von einer bestimmten Anzahl von Faktoren charakterisiert werden, welche aus den Bewertungsmustern abgeleitet werden. Um diese Faktoren zu berechnen ist die Matrixfaktorisierung eine populäre Methode. Auf die Matrixfaktorisie-

rung, insbesondere den unter diese Kategorie fallenden Alternating Least Squares (ALS) Algorithmus, wird im Abschnitt 4 mit Forschungsarbeiten zum Thema eingegangen. Ricci u. a.; Jannach u. a.

3.1.2 Content-Based

Diese Methoden basieren auf Item-Beschreibungen, die entweder Manuell, durch den Nutzer oder automatisch erfasst werden. Als Beispiel sei ein Buch genannt mit einem Genre, einem bestimmten Thema und dem Autor. Die Empfehlung wird gegeben, je nachdem wie stark ein Nutzerprofil der Item-Beschreibung ähnelt. Dabei werden die Profile durch Analyse von Nutzerverhalten, Nutzer-Feedback oder Fragen des Nutzers nach dessen Interessen erstellt. Für eine hohe Genauigkeit wird keine große Nutzergruppe benötigt, allerdings muss eine Nutzerhistorie vorhanden sein. Zudem können neue Items ohne „neues lernen" sofort empfohlen werden. Ricci u. a.; Jannach u. a.

3.1.3 Knowledge-Based

Knowledge-Based Recommender basierend auf vorhandenen, detaillierteren Inhalten, wie technische oder qualitative Eigenschaften. Diese müssen meist manuell bereitgestellt werden und für Nutzer sowie Items vorliegen. Dabei ist häufig eine Nutzerinteraktion nötig, um Nutzerprofile zu erstellen. So fragt das System den Nutzer etwa nach relativer Relevanz von Eigenschaften und dem Preis. Ein Beispiel sind Constraint-based Recommender, die eine Kamera nach Auflösung, Gewicht und Preis empfehlen. Diese Systeme sind besonders nützlich, wenn auf keine Nutzerhistorie zurückgegriffen werden kann, wie dies häufig der Fall bei teuren Elektronikartikeln ist. Ricci u. a.; Jannach u. a.

3.1.4 Hybrid Systems

Es werden verschiedene Verfahren kombiniert, um Schwächen eines Verfahrens auszugleichen oder noch bessere Empfehlungen zu erhalten, indem mehr Informationen genutzt werden. Ricci u. a.; Jannach u. a.

4 Arbeiten zum Thema

4.1 Matrix Factorization Techniques for Recommender Systems

4.1.1 Recommender System Strategien

Nach den Autoren gibt es hauptsächlich zwei Strategien für Recommender Systeme: *Content filtering* und *collaborative filtering*. Beide Strategien wurden bereits im vorherigen Abschnitt 3.1 beschrieben.

4.1.2 Matrix Factorization Methoden

Einige der erfolgreichsten Realisierungen von *latent factor models* basieren auf der Matrixfaktorisierung. Bei der Matrixfaktorisierung werden Nutzer und Items als Vektoren $p_u \in \mathbb{R}^f$ und $q_i \in \mathbb{R}^f$ von *latent factors* f charakterisiert, welche aus den Bewertungsmustern gewonnen werden. Eine hohe Übereinstimmung von Nutzer- und Item-Vektor führt zu einer Empfehlung. Bei fehlendem explizitem Feedback ist es bei dieser Methode auch möglich, implizites Feedback zu verwenden. Das implizite Feedback kann durch die Beobachtung des Nutzerverhaltens gewonnen werden, wie Kaufhistorie, Browser-Historie, Suchanfragen oder Mausbewegungen.

4.1.3 Ein basis Matrix Factorization Modell

Die User-Item Interaktionen werden als inneres Produkt $\hat{r}_{ui} = q_i^T p_u$ der Item- und User-Vektoren modelliert, wobei hier die Herausforderung in der Abbildung von den Items und Usern auf die *factor* Vektoren $p_u, q_i \in \mathbb{R}^f$ besteht. Um die *factor* Vektoren zu erlernen, minimiert das System den *regularized squared error* über die Menge der bekannten Ratings (4.1).

$$\min_{q^*,p^*} \sum_{(u,i)\in\mathcal{K}} \left(r_{ui} - q_i^T p_u\right)^2 + \lambda \left(||q_i||^2 + ||p_u||^2\right) \tag{4.1}$$

Hierbei steht $\mathcal{K}$ für die Menge der (u, i) Tupel für welche r_{ui} bekannt ist und der λ Term wird für eine Regulierung um *Overfitting* zu vermeiden genutzt.

4.1.4 Learning Algorithmen

Um die Gleichung (4.1) zu lösen, sind *stochastic gradient descent* (SGD) und *alternating least squares* (ALS) geeignete Algorithmen.

Stochastic gradient descent Hierbei durchläuft der Algorithmus alle Ratings und berechnet für jedes r_{ui} den zugehörigen *prediction error* $e_{ui} = r_{ui} - q_i^T p_u$. Anschließend werden die Parameter proportional zu λ in die entgegengesetzte Richtung des Gradienten modifiziert, woraus sich (4.2) ergibt.

$$\begin{aligned} q_i &\leftarrow q_i + \lambda \cdot e_{ui} \cdot p_u - \lambda \cdot q_i \\ p_u &\leftarrow p_u + \lambda \cdot e_{ui} \cdot q_i - \lambda \cdot p_u \end{aligned} \tag{4.2}$$

Alternating least squares Da q_i und p_u beides Unbekannte sind, ist Gleichung (4.1) nicht konvex. Wenn nun eine der Unbekannten als fix angenommen wird, wird das Optimierungsproblem quadratisch und kann optimal gelöst werden. ALS fixiert nun jeweils alle q_i und berechnet die p_u Vektoren. Im nächsten Schritt werden alle p_u fixiert und alle q_i berechnet. Dies stellt sicher, dass Gleichung (4.1) in jedem Schritt verringert wird, bis ein Grenzwert erreicht ist. ALS hat gegenüber SGD den Vorteil, dass eine massive Parallelisierung möglich ist, da alle p_u und q_i unabhängig voneinander parallel berechnet werden können und das ALS effizient mit implizitem Feedback umgehen kann.

4.1.5 Verfahren zur Verbesserung der Vorhersagegenauigkeit

Adding Biases Ein relativ großer Anteil der Variation in den Rating-Werten stammt von *biases*, also der Befangenheit von Nutzern. So gibt ein bestimmter Nutzer generell niedrigere oder höhere Bewertungen als alle anderen Nutzer oder ein Item wird immer höher bewertet. Dabei sind diese Variationen unabhängig von der eigentlichen User-Item Interaktion. Um dies zu berücksichtigen kann die Gleichung (4.1) zu Gleichung (4.3) erweitert werden. Hierbei gibt μ das durchschnittliche Rating über alle Items an, b_i die Abweichung des Items vom Durchschnitt und b_u die Abweichung des Nutzers vom Durchschnitt.

$$\min_{q^*,p^*,b^*} \sum_{(u,i)\in\mathcal{K}} \left(r_{ui} - \mu - b_u - b_i - q_i^T p_u\right)^2 + \lambda \left(||q_i||^2 + ||p_u||^2 + b_u^2 + b_i^2\right) \tag{4.3}$$

Additional Input Sources Um die Vorhersagegenauigkeit zu verbessern bzw. das *cold start* Problem abzumildern können weitere Quellen einbezogen werden. So kann es praktikabel sein, einen neuen *factor* Vektor einzuführen, welcher durch implizite Feedbackdaten berechnet wird.

Temporal Dynamics Da sich die Präferenzen von Nutzern über die Zeit ändern können und neuere Ratings die aktuellen Vorlieben besser widerspiegeln, ist es sinnvoll den *bias* b_i sowie b_u und den *user factor* Vektor p_u als eine Funktion über die Zeit zu modellieren (4.4).

$$\hat{r}_{ui}(t) = \mu - b_u(t) - b_i(t) - q_i^T p_u(t) \tag{4.4}$$

Inputs with varying confidence levels Um zu berücksichtigen, dass ein Rating nicht immer die gleiche Gewichtung im Vergleich zu anderen Ratings haben muss, bzw. bei implizitem negativem Feedback dies auch tatsächlich ein negatives ist, kann eine *confidence* c_{ui} eingeführt werden (4.5).

$$\min_{q^*,p^*,b^*} \sum_{(u,i)\in\mathcal{K}} c_{ui} \left(r_{ui} - \mu - b_u - b_i - q_i^T p_u\right)^2 + \lambda \left(||q_i||^2 + ||p_u||^2 + b_u^2 + b_i^2\right) \tag{4.5}$$

4.2 Large-scale parallel collaborative filtering for the netflix prize

In diesem Paper wird der Alternating-Least-Squares with Weighted-λ-Regularization (ALS-WR) Algorithmus vorgestellt, bei welchem die Vorhersage-Performance monoton mit der Anzahl an *features* und ALS Iterationen steigen soll. Um ein Stopp-Kriterium für die ALS-WR Iterationen festzulegen, verwenden die Autoren einen Testdatensatz, mit dem nach jeder ALS-WR Iteration geprüft wird, ob der Root Mean Square Error (RMSE) den Wert von 0.0001 unterschreitet. Der ALS-WR Algorithmus ist dem Standard-ALS sehr ähnlich und unterscheidet sich im Grunde nur durch den λ Term der Kostenfunktion, dargestellt in Gleichung (4.6).

$$f(P,Q) = \sum_{(u,i)\in\mathcal{K}} \left(r_{ui} - p_u^T q_i\right)^2 + \lambda \left(\sum_i n_{qi}||q_i||^2 + \sum_u n_{pu}||p_u||^2\right) \tag{4.6}$$

Hierbei kennzeichnen n_{pu} und n_{qi} die Zahl aller Ratings von User u bzw. Item i. Nach empirischen Ergebnissen führt diese λ-Regulierung dazu, dass, wenn die Anzahl der *features* oder die der Iterationen angehoben wird, keine Überanpassung stattfindet.

Die Autoren verwenden eine parallele Implementierung von ALS-WR, in welcher die Updates von P bzw. Q parallel erfolgen. Um die Berechnung verteilt über mehrere Rechner hinweg durchzuführen, wird eine Matlab Version eingesetzt, welche parallele Matlab Berechnungen erlaubt.

4.3 Fast Matrix Factorization for Online Recommendation with Implicit Feedback

In diesem Paper werden Techniken vorgestellt, die es ermöglichen sollen, durch implizites Feedback effektiv mit unvollständigen Daten umzugehen und zugleich online Updates des Empfehlungsmodells

durchzuführen. Hierfür wird das fehlende Feedback aufgrund der Item-Popularität gewichtet, anstatt auf eine uniforme Gewichtung zu setzen. Weiterhin wird ein auf *element-wise Alternating Least Squares* (eALS), eine Abwandlung des bereits vorgestellten ALS, basierender Algorithmus eingesetzt. Der vorgestellte Algorithmus soll hierbei um den Faktor k schneller sein als der eingestzte ALS Algorithmus in der Arbeit von Hu u. a. und gleich schnell wie der bei Devooght u. a. eingesetzte *randomized block coordinate descent* (RCD) Algorithmus. k beschreibt hierbei die Anzahl der latenten Faktoren. Die Element-weise Berechnung der latenten Faktoren ermöglicht zudem die Online Update Fähigkeit des Modells.

4.3.1 MF Method for Implicit Feedback

Als Beispiel für eine einfache gewichtete Regressionsfunktion um die Parameter des MF Modells zu erlernen, wird in dem Paper die Funktion (4.7) nach Hu u. a. als Beispiel verwendet.

Folgende Bezeichnungen sind für das Verständnis wichtig und gelten für alle nachfolgenden Formeln. M und N beschreiben die Anzahl der Nutzer bzw. Items, K die Anzahl der *feature* Vektoren, $R \in \mathbb{R}^{M \times N}$ ist die Nutzer/Item Interaktionsmatrix, $\mathcal{R}$ beschreibt die Menge der User-Item Paare die nicht null sind und die Indizes u und i sind für User und Items reserviert. Des Weiteren sind p_u und q_i die *latent feature* Vektoren von Nutzer u bzw. Item i und $\mathcal{R}_u$ ist die Menge der Items, die mit Nutzer u interagieren. $P \in \mathbb{R}^{M \times K}$ und $Q \in \mathbb{R}^{N \times K}$ sind die *latent factor* Matrizen für Nutzer und Items und $\hat{r}_{ui} = p_u^T q_i$.

$$J = \sum_{u=1}^{M} \sum_{i=1}^{N} w_{ui} \left(r_{ui} - \hat{r}_{ui}\right)^2 + \lambda \left(\sum_{u=1}^{M} \|p_u\|^2 + \sum_{u=1}^{N} \|q_i\|^2 \right) \tag{4.7}$$

Hierbei wird w_{ui} genutzt, um für den Eintrag r_{ui} eine Gewichtung einzuführen, wobei die Gewichtungsmatrix durch $W = [w_{ui} M \times N]$ repräsentiert wird. Zu erwähnen sei noch, dass für fehlende Einträge in der Regel r_{ui} ein Nullwert zugewiesen wird, aber w_{ui} einen Nicht-Nullwert erhält.

4.3.2 The authors' IMPLICIT MF METHOD

Die Autoren führen eine Item-orientierte Gewichtung für die fehlenden Daten ein, die dann um die Kenntnis der Item Popularität erweitert wird. Die große Anzahl an fehlendem Feedback ist eine Mischung aus negativem Feedback und unbekanntem Feedback. Dabei ist es wünschenswert, dem negativen Feedback eine höhere Gewichtung zuzuteilen.

Item-OrientedWeighting on Missing Data

Um die Gewichtung des negativen Feedback mit einzubeziehen, führen die Autoren die Formel (4.8) ein.

$$L = \sum_{(u,i)\in R} w_{ui} \left(r_{ui} - \hat{r}_{ui}\right)^2 + \sum_{u=1}^{M} \sum_{i \notin R_u} c_i \hat{r}_{ui}^2 + \lambda \left(\sum_{u=1}^{M} \|p_u\|^2 + \sum_{u=1}^{N} \|q_i\|^2 \right) \quad (4.8)$$

Hierbei beschreibt c_i die *confidence*, dass für Item i keine Bewertung vorliegt, ein wirklich negatives Feedback darstellt. Hierdurch kann Domänenwissen vom jeweiligen Nutzer des Algorithmus mit einbezogen werden. Somit beschreibt der erste Term den Vorhersagefehler der bekannten Einträge und der zweite Term bezieht sich auf das fehlende Feedback. Der dritte Term ist unverändert die λ-Regulierung.

Popularity-aware Weighting Strategy

Da es bei populären Items wahrscheinlicher ist, dass sie dem Nutzer bekannt sind, so ist es auch wahrscheinlicher, dass ein fehlendes Feedback zu diesen Items ein echtes negatives Feedback ist. Um diesen Effekt zu berücksichtigen, haben die Autoren für c_i die Parameter in (4.9) eingeführt.

$$c_i = c0 \frac{f_i^{\alpha}}{\sum_{j=1}^{N} f_j^{\alpha}} \quad (4.9)$$

Dabei ist f_i die Popularität von Item i nach der Häufigkeit des Auftretens in den impliziten Feedback Daten im Verhältnis zum Auftreten aller anderen Items zusammen. Der Exponent α reguliert die Signifikanz von populären gegenüber unpopulären Items und $c0$ bestimmt die gesamte Gewichtung der fehlenden Daten.

4.3.3 Optimization by ALS

Der ALS Algorithmus berechnet den kompletten *latent feature* Vektor für einen Nutzer respektive Item als ganzes. Um den Vektor für einen Nutzer zu berechnen muss schlussendlich die Formel (4.10) berechnet werden.

$$p_u = \left(Q^T W^u Q + \lambda I\right)^{-1} Q^T W^u r_u \quad (4.10)$$

W^u ist hierbei eine $N \times N$ Matrix mit $W_{ii}^u = w_{ui}$, also der Gewichtung und I ist die Identitätsmatrix. Wie man erkennen kann, ist hier eine teure Invertierung einer $K \times K$ Matrix nötig, die eine Komplexität von $O(K^3)$ aufweist. Damit hat das Update eines User-Vektors eine Zeitkomplexität von $O(K^3 + NK^2)$ und das Update des gesamten Modells $O((M+N)K^3 + MNK^2)$. Durch uniforme Gewichtung kann nach (Hu u. a.) mit Memoisierung eine Verbesserung auf $O((M+N)K^3 + |R|K^2)$ erreicht werden.

4.3.4 Generic Element-wise ALS Learner

Der Flaschenhals von ALS liegt bei der Matrixinvertierung aufgrund des Vektor-Updates als ganzes. Die Element-weise Berechnung bei eAls wird durch folgende Formel (4.11) ausgedrückt:

$$p_{uf} = \frac{\sum_{i=1}^{N} \left(r_{ui} - \hat{r}_{ui}^{f}\right) w_{ui} q_{if}}{\sum_{i=1}^{N} w_{ui} q_{if}^{2} + \lambda} \tag{4.11}$$

Hierbei ist $\hat{r}_{ui}^{f} = \hat{r}_{ui} - p_{uf} q_{if}$, wobei f der Index für das jeweilige *feature* im *feature* Vektor ist. Jedes Element des *feature* Vektor wird einzeln optimiert, während die anderen fix bleiben. Eine naive Implementierung liefert hier bereits eine Komplexität von $O(MNK^2)$ für eine Iteration. Durch eine Vorberechnung von $\hat{r}_{ui}$ lässt sich eine Komplexität von $O(MNK)$ erreichen.

4.3.5 Fast eALS Learning Algorithm

Der von den Autoren erweiterte eALS Algorithmus reduziert den Rechenaufwand weiter, indem wiederholte Berechnungen der gewichteten fehlenden Daten durch Caches vermieden werden. Zunächst wird die Gleichung (4.11) um p_{uf} zu berechnen nach (4.12) umgeschrieben, sodass der Teil für die Berechnung für die vorhandenen und die fehlenden Daten separat betrachtet werden.

$$p_{uf} = \frac{\sum_{i \in \mathcal{R}_u} \left(r_{ui} - \hat{r}_{ui}^{f}\right) w_{ui} q_{if} - \sum_{i \notin \mathcal{R}_u} \hat{r}_{ui}^{f} c_i q_{if}}{\sum_{i \in \mathcal{R}_u} w_{ui} q_{if}^{2} + \sum_{i \notin \mathcal{R}_u} c_i q_{if}^{2} + \lambda} \tag{4.12}$$

Durch mehrere Umformungsschritte, auf die hier nicht weiter eingegangen wird, kann die Gleichung (4.12) nach (4.13) umgestellt werden, in welcher der als $S^q = \sum_{i=1}^{N} c_i q_i q_i^T$ definierte Cache verwendet werden kann. Auf die gleich Weise kann q_{if} mit dem Cache $S^p = P^T P$ berechnet werden.

$$p_{uf} = \frac{\sum_{i \in \mathcal{R}_u} \left[w_{ui} r_{ui} - (w_{ui} - c_i)\, \hat{r}_{ui}^{f}\right] q_{if} - \sum_{k \neq f} p_{uk} S_{kf}^{q}}{\sum_{i \in \mathcal{R}_u} (w_{ui} - c_i)\, q_{if}^{2} + S_{ff}^{q} + \lambda} \tag{4.13}$$

Durch die Verwendung der Caches kann die Komplexität für ein Update des User *latent factor* auf $O(K + |\mathcal{R}|)$ und für eine komplette Iteration von eALS auf $O(M + N)K^2 + |\mathcal{R}|K)$ reduziert werden.

4.3.6 Online Update

Beim Online Update werden die Modellparameter des offline trainierten Modells mit einer neuen User-Item Interaktion aktualisiert. Es erfolgt also ein inkrementelles Update bei jeder neuen Interaktion. Dabei werden die Parameter einzig für p_u und q_i aktualisiert, wobei angenommen wird, dass P und Q aus globaler Perspektive nicht zu sehr geändert werden aber die lokalen *features* von u und i eine signifikante Änderung erfahren. Neue Nutzer bzw. Items führen dazu, dass p_u und q_i sehr ähnlich

sind. Nachdem das Update für den *latent* Vektor erfolgt ist, muss der *S Cache* ebenfalls aktualisiert werden. Da neue Interaktionen das aktuelle Interesse eines Nutzers widerspiegeln, wird ebenfalls die Gewichtung für die neue Interaktion aktualisiert. Die Zeitkomplexität für das Update beträgt insgesamt $O(K^2 + (|\mathcal{R}_u| + |\mathcal{R}_i|)K)$ und ist unabhängig von der Gesamtzahl an Interaktionen, Nutzern und Items, womit es auch bei großen Datenmengen praktikabel anwendbar ist.

4.3.7 Probleme

Bei dem beschriebenen fast-eALS Algorithmus ergeben sich auch mögliche Probleme, die u.a. auftreten könnten, wenn der Algorithmus verteilt implementiert werden soll. So müssen die verwendeten Caches S^q und S^p bei jeder Iteration neu berechnet und auch über den Cluster verteilt werden. Die Verteilung über den Cluster und der damit entstehende Netzwerkverkehr könnte die Effizienz des Algorithmus verschlechtern. Zudem werden von den Caches zusätzliche Speicherressourcen belegt, was bei weiteren Messungen untersucht werden sollte. Weiterhin wurden die Messungen in dem Paper auf einem einzelnen Rechner in einem einzigen Thread ausgeführt. Hier muss untersucht werden, welche Auswirkungen die Verteilung, abgesehen von den erwähnten Caches, auf die Geschwindigkeit und die Effizienz hat. Untersuchungspunkte sind etwa der Overhead durch den Netzwerktransfer und die Aufteilung der Daten auf die einzelnen Konten im Cluster

Betreffend der Online-Updates erscheint eine weitere Messung bezüglich der Vorhersage-Performance angebracht. Da die Updates nur lokal durchgeführt werden und die zunächst minimal erscheinenden globalen Auswirkungen über viele Updates hinweg zunehmend bedeutende Konsequenzen für die Vorhersagegenauigkeit bedeuten könnten, sollte dies genauer untersucht werden.

5 Ausblick

Im Anschluss an diese Arbeit sollen einige der vorgetellten Verfahren, vor allem die von He u. a., mittels der Big Data Systeme Apache Flink und Apache Spark implementiert und die Implementierungen miteinander verglichen werden. Hierbei soll insbesondere auf die Möglichkeiten der Parallelisierung und effizienten Verteilung der Daten eingegangen werden. Weiterhin sollen die in Abschnitt 4.3.7 beschriebenen Probleme genauer untersucht werden. So ist ein aussagekräftiger Vergleich zwischen der Laufzeit von Als und eALS erst möglich, wenn beide Algorithmen verteilt implementiert sind und nicht nur auf einem einzelnen Rechner in nur einem Thread laufen, sowie mit einer großen Datenmenge getestet wird, die nicht mehr in den Speicher eines einzelnen Rechners passt.

Literaturverzeichnis

[Devooght u. a.] DEVOOGHT, Robin ; KOURTELLIS, Nicolas ; MANTRACH, Amin: Dynamic Matrix Factorization with Priors on Unknown Values, ACM Press, S. 189–198. – URL http://dl.acm.org/citation.cfm?doid=2783258.2783346. – Zugriffsdatum: 2017-08-26. – ISBN 978-1-4503-3664-2

[Foundation 2016a] FOUNDATION, The Apache S.: *Apache Flink 1.1-SNAPSHOT Documentation: Jobs and Scheduling*. 2016. – URL https://ci.apache.org/projects/flink/flink-docs-master/internals/job_scheduling.html. – abgerufen am 2016-05-15

[Foundation 2016b] FOUNDATION, The Apache S.: *Apache Flink: Features*. 2016. – URL http://flink.apache.org/features.html. – abgerufen am 2016-04-09

[Foundation 2016c] FOUNDATION, The Apache S.: *Apache Flink: Scalable Batch and Stream Data Processing*. 2016. – URL http://flink.apache.org/. – abgerufen am 2016-04-09

[He u. a.] HE, Xiangnan ; ZHANG, Hanwang ; KAN, Min-Yen ; CHUA, Tat-Seng: Fast Matrix Factorization for Online Recommendation with Implicit Feedback. In: *Proceedings of the 39th International ACM SIGIR Conference on Research and Development in Information Retrieval*, ACM Press, S. 549–558. – URL http://dl.acm.org/citation.cfm?doid=2911451.2911489. – Zugriffsdatum: 2017-05-22. – ISBN 978-1-4503-4069-4

[Hu u. a.] HU, Y. ; KOREN, Y. ; VOLINSKY, C.: Collaborative Filtering for Implicit Feedback Datasets. In: *2008 Eighth IEEE International Conference on Data Mining*, S. 263–272

[James E. Short] JAMES E. SHORT: *How Much Media? 2013 Report on American Consumers*. – URL https://business.tivo.com/content/dam/tivo/resources/tivo-HMM-Consumer-Report-2013_Release.pdf. – Zugriffsdatum: 2017-06-04

[Jannach u. a.] JANNACH, Dietmar ; ZANKER, markus ; FELFERNIG, alexander ; FRIEDRICH, gerhard: *Recommender systems: an introduction*. Cambridge University Press. – OCLC: ocn645789647. – ISBN 978-0-521-49336-9

[Jeffrey M. O'Brien] Jeffrey M. O'Brien: *The race to create a 'smart' Google.* – URL http://archive.fortune.com/magazines/fortune/fortune_archive/2006/11/27/8394347/index.htm. – Zugriffsdatum: 2017-08-30

[Koren u. a. 2009–08] Koren, Y. ; Bell, R. ; Volinsky, C.: Matrix Factorization Techniques for Recommender Systems. 42 (2009-08), Nr. 8, S. 30–37. – ISSN 0018-9162

[Lange 2017] Lange, Timo: *Big Data Systeme: Konzeptioneller und experimenteller Vergleich von Apache Flink mit Apache Spark anhand eines Anwendungsszenarios.* 2017

[Matei Zaharia] Matei Zaharia: *Parallel Programming With Spark.* – URL strataconf.com. – Strata Conference

[Netlfix Inc.] Netlfix Inc.: *Netflix Prize: Home.* – URL http://www.netflixprize.com/. – Zugriffsdatum: 2017-08-31

[Ricci u. a.] Ricci, Francesco ; Rokach, Lior ; Shapira, Bracha ; Kantor, Paul B.: *Recommender Systems Handbook.* Springer US. – URL http://link.springer.com/10.1007/978-0-387-85820-3. – Zugriffsdatum: 2017-05-21. – DOI: 10.1007/978-0-387-85820-3. – ISBN 978-0-387-85819-7 978-0-387-85820-3

[Tathagata Das 2015–01–15] Tathagata Das: *Improved Fault-tolerance and Zero Data Loss in Apache Spark Streaming.* 2015-01-15. – URL https://databricks.com/blog/2015/01/15/improved-driver-fault-tolerance-and-zero-data-loss-in-spark-streaming.html. – abgerufen am 2016-08-08

[Tathagata Das u. a. 2015–07–30] Tathagata Das ; Matei Zaharia ; Patrick Wendell: *Diving into Apache Spark Streaming's Execution Model.* 2015-07-30. – URL https://databricks.com/blog/2015/07/30/diving-into-apache-spark-streamings-execution-model.html. – abgerufen am 2016-08-08

[The Apache Software Foundation 2016] The Apache Software Foundation: *Spark Streaming - Spark 2.0.0 Documentation.* 2016. – URL http://spark.apache.org/docs/latest/streaming-programming-guide.html. – abgerufen am 2016-08-08

[Traub u. a. 2015] Traub, Jonas ; Rabl, Tilmann ; Hueske, Fabian ; Rohrmann, Till ; Markl, Volker: *Die Apache Flink Plattform zur parallelen Analyse von Datenströmen und Stapeldaten.* 2015. – URL https://pdfs.semanticscholar.org/6558/ba3ae95bf29147e03741ff2df57afd735c16.pdf. – abgerufen am 2016-04-11

[Zaharia 2016] ZAHARIA, Matei: *An architecture for fast and general data processing on large clusters*. Morgan & Claypool, 2016. – URL https://books.google.com/books?hl=en&lr=&id=TNUlDAAAQBAJ&oi=fnd&pg=PP2&dq=%22However,+the+processing+capabilities+of+single+machines+have+not+kept%22+%22processing,+streaming+analysis+of+new+real-time+data+sources+is+required+to%22+%22systems+only+support+simple+one-pass+computations+(e.g.,%22+&ots=jSkwv4yQ_H&sig=1L7JInZHtLX1blxqmXu_Ho8tm88. – abgerufen am 2016-07-19

[Zaharia u. a. 2012a] ZAHARIA, Matei ; CHOWDHURY, Mosharaf ; DAS, Tathagata ; DAVE, Ankur ; MA, Justin ; MCCAULEY, Murphy ; FRANKLIN, Michael J. ; SHENKER, Scott ; STOICA, Ion: Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, USENIX Association, 2012, S. 2–2. – URL http://dl.acm.org/citation.cfm?id=2228301. – abgerufen am 2016-07-17

[Zaharia u. a. 2012b] ZAHARIA, Matei ; DAS, Tathagata ; LI, Haoyuan ; SHENKER, Scott ; STOICA, Ion: Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In: *Presented as part of the*, URL https://www.usenix.org/conference/hotcloud12/workshop-program/presentation/zaharia, 2012. – abgerufen am 2016-07-19

[Zhou u. a. 2008] ZHOU, Yunhong ; WILKINSON, Dennis ; SCHREIBER, Robert ; PAN, Rong: Large-scale parallel collaborative filtering for the netflix prize. In: *International Conference on Algorithmic Applications in Management*, Springer, 2008, S. 337–348. – URL http://link.springer.com/chapter/10.1007/978-3-540-68880-8_32. – abgerufen am 2016-09-13