Echtzeit Recommender System mit Collaborative Filtering

Eine speichereffiziente Implementierung

Diplomarbeit von Clemens Dörrhöfer aus Wiesbaden

April 2012

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik
Knowledge Engineering Group

Echtzeit Recommender System mit Collaborative Filtering
Eine speichereffiziente Implementierung

Vorgelegte Diplomarbeit von Clemens Dörrhöfer aus Wiesbaden

1. Gutachten: Prof Dr. Johannes Fürnkranz
2. Gutachten: Eneldo Loza Mencía

Tag der Einreichung:

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Diplomarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mainz, der 14. April 2012

Unterschrift - Clemens Dörrhöfer

Zusammenfassung

Im Rahmen dieser Diplomarbeit wird ein Recommender System für den Buchhandel implementiert. Der Fokus der Arbeit liegt in der Echtzeitfähigkeit des Systems. Zusätzlich sind die benötigten Ressourcen so gering gehalten, dass das System auf einem handelsüblichen Server lauffähig ist.

Zunächst wird ein Überblick über die verschiedenen Methoden von Recommender Systemen gegeben. Das System selbst wird als *Item* basiertes Collaborative Filtering realisiert. Bei der Umsetzung sind drei Parameter relevant. Zum einen die benötigt Rechenzeit, die essenziell ist für den Einsatz als Echtzeit System, der benötigte Speicher und die Qualität der Empfehlung. Dabei hat sich der Speicherverbrauch als das kritischste Element erwiesen. Diese Arbeit stellt verschiedene Vorgehensweisen vor, den Speicherverbrauch so gering wie möglich zu halten und vergleicht deren Leistungsfähigkeit anhand der relevanten Parameter.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Motivation

Ziel eines jeden Händlers ist es, Ware zu verkaufen. Dabei bedient er sich diverser Methoden. Eine davon besteht darin, die Kaufentscheidungskriterien von Kunden zu evaluieren, indem er Verhaltensmuster ganzer Benutzergruppen auswertet, um auf die Interessen Einzelner zu schließen. Diese Technik wird als Collaborative Filtering bezeichnet. Sie kommt aus dem *Maschinellen Lernen* und macht es möglich, personalisierte Empfehlungen im großen Maßstab zu tätigen.

Nehmen wir uns ganz konkret den Bereich der Buchhändler vor. Mit zunehmender Konkurrenz ist es weit wichtiger als früher, sich von der Konkurrenz abzuheben. Dies gilt insbesondere für den Bereich des Onlinehandels. In der Vergangenheit konnte sich ein Buchhändler in der Hauptsache durch persönliche Kundenberatung positiv hervorheben. Dies ist im Internetzeitalter immer schwieriger umzusetzen, da der Kunde zunehmend Einkäufe tätigt ohne direkten persönlichen Kontakt zum Händler. Doch individuelle Buchempfehlungen auf Basis bisheriger Vorlieben sind für die Kaufentscheidung enorm wichtig, auf die auch der elektronische Handel nicht gerne verzichtet.

Abbildung 1.1 illustriert die Kernidee des Kollaborativen Filterns und basiert sowohl auf Büchern, die der Händler im Sortiment führt als auch Büchern, die der Kunde in der Vergangenheit positiv bewertet hat. Aus den positiv bewerteten Büchern erstellt das System eine Liste aus dem bestehenden Sortiment zusammen, die anhand ihrer Ähnlichkeit den Kunden interessieren könnten. Bekannt sind diese Techniken bereits von den „Global Players“ im Internetgeschäft. Webshops kleinerer und mittlerer Buchhandlungen können diesen Dienst bisher nicht bieten, obwohl gerade für sie im Hinblick auf die immer größer werdende Produktauswahl diese Technik sehr hilfreich wäre, um das Gesamtangebot besser zu überblicken. Bei der Vielzahl verschiedener zur Verfügung stehender Techniken sollte es aber selbst dem kleinen Buchhändler möglich sein, eine solche Dienstleistung anzubieten.

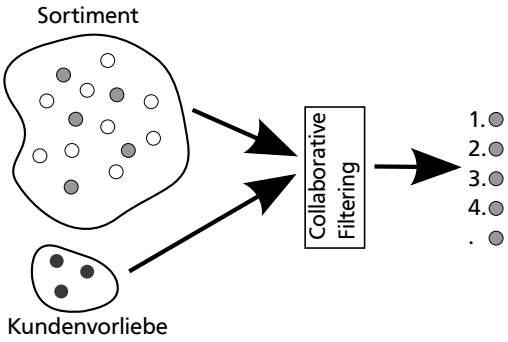


Abbildung 1.1.: Empfehlungen, als eine individuell sortierte Liste von Elementen

1.2 Ziele und Anforderungen

Ziel ist es, für das Kunden Informations System *KIS*, einem Produkt der Firma indis Kommunikationssysteme GmbH (Mainz), ein Empfehlungssystem zu realisieren. *KIS* ist eine EDV-Lösung, die den Geschäftsablauf eines Buchhändlers, angefangen bei der Bestellung bis hin zur Rechnungsstellung, abwickelt. Darüber hinaus bietet es viele Zusatzleistungen wie Statistiken, Verwaltung von Kostenstellen aber auch Anbindung über einen Online-Shop sowie die Möglichkeit, den Status einer Bestellung abzufragen. Kunden, die ihre Bestellungen über *KIS* verwalten oder den Online-Shop verwenden, sollen in Zukunft mit passenden Empfehlungen versorgt werden. Als Ausgangpunkt dienen die gekauften Bücher der im System angemeldeten Kunden.

Die Empfehlungssoftware selbst wird kein Bestandteil von *KIS* sein, welches zum größten Teil auf einem CMS namens Typo3 aufbaut. Vielmehr soll die Empfehlung unabhängig vom System über einen Webservice realisiert werden, der die Berechnungen aufgrund der Datenbasis von *KIS* tätigt. Das Berechnen der Empfehlungen muss dabei in Echtzeit geschehen, um die Akzeptanz des Kunden zu gewährleisten. Die Anforderungen an die Hardware müssen so gering wie möglich gehalten werden, um den Kostenrahmen möglichst klein zu halten. Die gängige Hardware, die in diesem Segment eingesetzt wird, sind virtualisierte Server mit zwei bis vier Prozessoren zu je 2,4 GHz und Arbeitsspeicher bis zu vier *GiB*. Bei der Umsetzung ist darauf zu achten, dass diese Anforderungen nicht überschritten werden. Die bestehenden Systeme zeigen vermehrt einen Engpass bei der *I/O* Last. Das äußert sich vor allem bei Datenbankzugriffen und Im- und Export von Datenbeständen. Auf starken *I/O* Zugriff sollte deshalb verzichtet werden. Gleichzeitig ist aber in der Umsetzbarkeit auf Skalierbarkeit zu achten, um das System an wachsende Anforderungen anpassen zu können. Dabei muss das System sowohl die wachsende Anzahl von Kunden, als auch die wachsende Anzahl von Produkten im Sortiment bewältigen.

1.3 Bestehende Produkte

Für die Umsetzung der Software verschaffen wir uns zunächst einen Überblick über bereits bestehende Systeme und deren Funktionsumfang. Der Verzicht auf eine vollständige Neuentwicklung hilft, Produktionszeit und Kosten zu minimieren. Eine fertige Software, die dem konkreten Szenario entspricht, gibt es nicht und kann es auch nicht geben, wie im Laufe dieser Ausarbeitung deutlich wird. Es existieren aber eine Reihe von Bibliotheken und Frameworks, die sich mit dieser Materie beschäftigen und für die Lösung des Problems herangezogen werden können.

Folgende Projekte wurden für die Umsetzung in Betracht gezogen: Dabei hat diese Liste nicht den Anspruch vollständig zu sein.

Cofi

Cofi [12] ist ein Projekt der Universität von Quebec in Montreal (UQAM) unter der Leitung von Daniel Lemire. Es bietet eine kleine Auswahl an Algorithmen unter anderem den *Slope One* Algorithmus [13] und eine *Pearson* Implementierung. Der Anspruch von Cofi besteht nicht darin, ein umfassendes Framework für alle Probleme des *Collaborative Filtering* zu bieten. Zitat Lemire: „So the focus is on delivering very few lines of code, and to rely on the programmer for providing the necessary glue.“ Der wesentliche Nachteil von Cofi ist, dass es auf keine Implementierung für spärlich besetzte Matrizen und Vektoren zurückgreift. Das bedeutet, dass diese Lösung mit wachsender Problemstellung nicht skalieren kann. Insbesondere kann es bei der Speicherverwaltung zu Problemen führen.

Weka

Weka [23] ist eine *Data Mining* Bibliothek der Universität von Waikato (Neuseeland). Es bietet eine breite Sammlung von Algorithmen aus dem Bereich des maschinellen Lernens wie etwa Algorithmen für das *automatische Klassifizieren*, *Data Preprocessing* und vieles mehr. Weka bietet keine direkte Unterstützung für *Collaborative Filtering* an. Die Vielzahl an Werkzeugen, die es bietet, machen es dennoch zu einem brauchbaren Werkzeug für dieses Problem. Weka wird unter der GPLv2 veröffentlicht und ermöglicht so den Einblick in den gesamten Quellcode.

DBLens

DBLens [1] ist ein auf Oracle basiertes Toolkit, um *Collaborative Filtering* zu realisieren. Es ist im Wesentlichen eine Sammlung von PL/SQL und Shell Skripten, die einen flexiblen und effizienten Einsatz von *Collaborative Filtering* möglich machen soll. Entwickelt wurde es von Jon Herlocker und Hannu Huhdanpaa. Nach deren Angaben wurde DBLens primär für den

Einsatz von offline Berechnungen entwickelt. Die Einschränkung auf PL/SQL Skripte und die Tatsache, dass der Fokus dieses Projektes auf offline Berechnungen liegt, schließt diese Lösung für das vorliegende Problem aus.

Mahout™

Mahout [3] ist ein Framework unter der *Apache*™Lizenz aus dem Bereich des *Maschinellen Lernens*. Es bietet mitunter Algorithmen aus der *automatischen Klassifikation*, dem *Clustering* und dem *Pattern Mining*. Der Bereich Recommenders / Collaborative Filtering ist aus dem Projekt *Taste* hervorgegangen. Es bietet einige Referenzimplementierungen verschiedener *Collaborative Filtering* Techniken an, sowie umfangreiche Schnittstellen, die Eigenimplementierungen und Erweiterungen ermöglichen. Zudem setzt Mahout auf dem Hadoop Projekt auf und eröffnet die Möglichkeit, Berechnungen zu parallelisieren und auf einem Rechner Cluster bearbeiten zu lassen.

Die Recherche legt nahe, dass es kein fertiges Produkt gibt, das die gegebenen Anforderungen erfüllt und daher eine eigene Implementierung notwendig ist. Die vorgestellten Produkte bieten jedoch allesamt Werkzeuge, die für die Umsetzung eines solchen Produktes nützlich sind. Am vielversprechendsten sind dabei die beiden Frameworks Weka und Mahout.

Die Entscheidung fällt auf Mahout, weil es zum einen bereits Schnittstellen für Recommender Systeme bereitstellt und zum anderen die Möglichkeit bietet, das Programm später in einem Hadoop™Cluster, laufen zu lassen. Dies ist zwar als Anforderung nicht direkt vorgegeben, kann sich jedoch als nützlich erweisen, wenn die Anforderungen an das System steigen. Das kann sich in Form von umfangreicheren Eingabedaten äußern, wenn beispielsweise die Anzahl der Kunden zunimmt. Zusätzliche Funktionalitäten, wie etwa eine schwarze Liste oder Neubewertungen von Empfehlungen durch z.B. Werbekampagnen können auch dazu beitragen. Darüber hinaus wird Mahout unter der Apache Lizenz angeboten, die für den kommerziellen Gebrauch vorteilhafter ist, als die GNU General Public License.

2 Einführung in die Theorie der Recommender Systeme

Im folgenden Kapitel wird kurz auf die theoretischen Grundlagen von Recommender Systemen eingegangen. Es werden die grundlegenden Konzepte geklärt. In dem Zusammenhang wird kurz auf die gängigsten Metriken eingegangen, die in diesem Bereich Verwendung finden. Danach folgt eine grobe Gliederung über die verschiedenen Herangehensweisen, abgeschlossen mit einem kurzen Überblick über die Problematik der Evaluation von Recommender Systemen.

2.1 Konzepte

Bei Recommender Systemen unterscheidet man traditionell zwei Entitäten. Im Allgemeinen gibt es eine Menge $\mathscr{U} = \{u_1, u_2, \ldots, u_k\}$ an Benutzern bezeichnet als *User* mit $|\mathscr{U}| = k$. Der *User* ist ein Subjekt, welches mit dem System interagieren und Vorlieben äußern kann.

Neben den Subjekten gibt es eine Menge an Objekten $\mathscr{I} = \{i_1, i_2, \ldots, i_n\}$ auch *Item* genannt mit $|\mathscr{I}| = n$. *Items* sind Objekte mit denen interagiert werden kann. Objekte können Eigenschaften besitzen, die abhängig vom Kontext sind.

Zusätzlich zu *User* und *Item* gibt es die Menge der Präferenzen $\mathscr{P} = \{u, i, v\}$. Präferenzen ordnen einem *User-Item* Paar einen numerischen Wert v zu, der den Grad der Vorliebe des Subjektes u zu dem Objekt i widerspiegelt.

Die Präferenzliste eines *Users* u_k bezeichnet also die Teilmenge $\mathscr{P}_{u_k} = \{(u_k, i_1, v_1), \ \ldots, \ (u_k, i_n, v_n)\}$, wobei nicht für jedes Paar (u, i) eine Wertung v vorliegen muss.

Su and Khoshgoftaar[22] führen mehrere Techniken für die Realisierung eines *Recommender* Systems auf. Sie unterscheiden im Wesentlichen zwei Herangehensweisen. Da gibt es zum einen den *Content basierten* Ansatz und zum anderen den *Collaborative basierten* Ansatz. Dazu kommen noch hybride Verfahren, die beide Techniken miteinander kombinieren.

Die *Content basierten* Techniken setzen auf den Vergleich von Objekten. Diese Objekte werden dabei durch eine Anzahl von Eigenschaften charakterisiert. Die Eigenschaften können vielseitig in ihrer Gestaltung sein. Das können Autoren von Büchern, Preise, Kategorien, Gewichte, Abmessungen aber auch Farben und vieles mehr sein. Objekte können anhand ihrer Eigenschaften kategorisiert werden. Aufgrund dieser Kategorien können zwischen Objekten Empfehlungen ausgesprochen werden. Ein solches System ist kontextabhängig. Es müssen zunächst die Eigenschaften identifiziert und gegebenenfalls bewertet werden.

Im Gegensatz dazu arbeiten *Collaborative Filtering* Methoden auf den aggregierten Bewertungen der *User* und sind dadurch Domain unabhängig. Vorteil dieser Herangehensweise ist es, dass die *Items* nicht mit Eigenschaften besetzt sein müssen. Oft sind Informationen unvollständig oder fehlerhaft, sodass die *Content basierten* Ansätze meist nicht anwendbar sind. Der *CF* Ansatz löst dazu noch ein weiteres Problem der Performance. In der Regel ist die Anzahl der Features wesentlich größer als die Anzahl der Bewertungen, die ein *User* abgegeben hat.

Die grundsätzliche Annahme, die dem *Collaborative Filtering* zugrunde liegt ist, dass zwei Personen u_1 und $u_2 \in \mathscr{U}$, die ein Element $i_1 \in \mathscr{I}$ ähnlich bewerten, auch ein anderes Element $i_2 \in \mathscr{I}$ ähnlich bewerten, aufgrund ihrer persönlichen Präferenzen. Hinter der Vorgehensweise von Collaborative Filtering (*CF*) steht folgende Annahme: Wenn zwei Personen eine große Übereinstimmungen in ihrer Präferenzliste haben, ist es möglich daraus, zu schließen, dass auch alle anderen Gegenstände in der Präferenzliste, bei denen keine Überschneidung existiert, den jeweiligen Geschmack der anderen Person treffen. Um eine solche Ähnlichkeit quantitativ erfassbar zu machen, sind Metriken notwendig. Sobald die Metriken berechnet wurden, können alle Elemente, die einen festgelegten Schwellwert überschreiten, als Empfehlung abgegeben werden. Abbildung 2.1 zeigt ein illustriertes Beispiel, wie so eine Präferenz aussehen kann. Das Beispiel wurde in leicht modifizierter Form aus [16] entnommen. Die Daten für dieses Beispiel sind in Abbildung 2.4a im Detail aufgelistet. Die *User* werden dargestellt durch Zahlen, die *Items* durch Buchstaben. Durchgezogene Linien stehen für eine positive Bewertung, gepunktete Linien stellen eine negative Bewertung dar. In der Präferenzliste von *User* 3 befinden sich danach die *Items* D und E mit einer positiven Wertung und *Item* A mit einer negativen Wertung. Exakte numerische Bewertungen lassen sich dieser Darstellung nicht entnehmen. Jedoch sind qualitative Aussagen möglich. So haben beide *User* 1 und 5 die Vorlieben für die Bücher A, B und C gemeinsam. Eine Empfehlung für *User* 1 würde demnach wahrscheinlich über die Präfernzen von *User* 5 gehen.

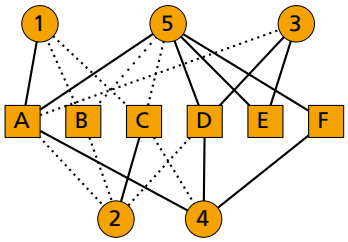


Abbildung 2.1.: User - Item Relationen

2.1.1 Prediction & Recommendation

Das *CF* erfüllt zwei Aufgaben, die Vorhersage und die Empfehlung. Wie es in Abbildung 2.2, übernommen aus [20] von Sarwar et al., zu entnehmen ist.

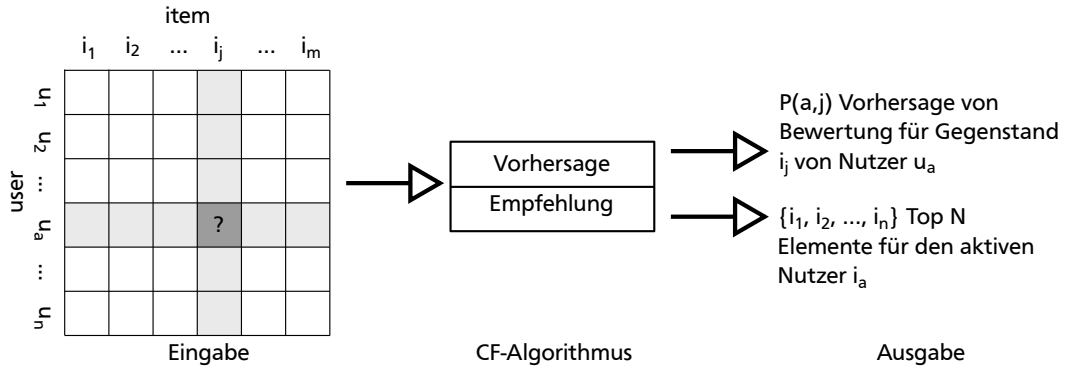


Abbildung 2.2.: Vorhersage und Empfehlung

Vorhersage (Prediction)

Die Prediction ist ein numerischer Wert, $P_{a,j}$, der abschätzt, wie der aktive Benutzer u_a das Element $i_j \notin \mathscr{P}_{u_a}$ bewerten würde. Dabei kann $P_{a,j}$ den gleichen Wertebereich annehmen, den auch alle anderen Bewertungen bekommen.

Empfehlung (Recommendation)

Die Recommendation ist eine Liste von Elementen $R \subset \mathscr{I}$, die der aktive Benutzer mit hoher Wahrscheinlichkeit als gut bewerten wird. Dabei gilt, dass kein Element aus R in $\mathscr{P}_{u_a}$ enthalten sein darf.

2.1.2 Metriken

Im Folgenden werden gängige Metriken vorgestellt und deren Eigenschaften werden kurz erläutert. All diese Metriken operieren auf numerischen Vektoren. Einträge können auch leer sein und werden als `NaN` bezeichnet. Im Folgenden werden diese Vektoren als $p = [p_1, p_2, \ldots, p_n]$ und $q = [q_1, q_2, \ldots, q_n]$ bezeichnet. Mit $\overline{p}$ bzw. $\overline{q}$ wird das arithmetische Mittel der Vektoreinträge bezeichnet. Sei $\mathscr{S}$ die Menge aller Indizes, für die gilt $\forall i \in \mathscr{S}.p_i \neq \texttt{NaN} \wedge q_i \neq \texttt{NaN}$

Pearson Metrik

Die Pearsons Metrik [6](Bravais-Pearson-Korrelationskoeffizient) gibt an, in welchem Maß zwei Vektoren zueinander linear korrelieren. Der Wertebereich der Pearson Metrik liegt in der

Menge $[-1,1]$. Sie misst, wie wahrscheinlich es ist, dass eine relativ extreme Bewertung in einer Menge, auch in der anderen Menge extrem bewertet wird. Sind die Tendenzen gleich, liegt der Wert nahe 1. Sind die Extremwerte hingegen gegenläufig, so ist der Wert nahe -1. Bei einem Wert von 0 besteht keine lineare Abhängigkeit. Diese Metrik erfasst nur lineare Abhängigkeiten. Andere Zusammenhänge werden nicht erkannt.

$$Pearson(p,q) = \frac{Cov(p,q)}{\sqrt{Var(p)}\sqrt{Var(q)}} = \frac{\sum_{i \in \mathscr{S}} (p_i - \overline{p})(q_i - \overline{q})}{\sqrt{\sum_{i \in \mathscr{S}} (p_i - \overline{p})^2}\sqrt{\sum_{i \in \mathscr{S}} (q_i - \overline{q})^2}} \tag{2.1}$$

Spearman Metrik

Die Spearman Metrik [6] entspricht der Person Metrik, mit dem Unterschied, dass nicht die Rohdaten für die Bewertung herangezogen werden. Stattdessen wird über alle Bewertungen ein Ranking ermittelt. Die Pearson Distanz wird danach über die Rankings berechnet. Für die Spearman Metrik gilt, dass sie nicht die Stärke des linearen Zusammenhangs misst, sondern die des monotonen Zusammenhangs.

$$\rho(p,q) = Pearson(Rang(p), Rang(q)) \tag{2.2}$$

Jaccard Metrik

Die Jaccard Metrik arbeitet auf binären Vektoren. Als Schnittmenge ($\cap$) wird die Anzahl der Vektor indizes verstanden, bei denen sowohl p, als auch q den Wert eins haben. Unter der Vereinigungsmene ($\cup$) verseht man die Anzahl der Vektor indizes, bei denen entweder p oder q den Wert eins aufweisen. Die Metrik selbst gibt den Quotient aus Schnitt- und Vereinigungsmenge an. Abbildung 2.3 illustriert anschaulich die Funktionsweise der Metrik. Diese Metrik bewegt sich im Wertebereich $[0,1] \subset \mathbb{Q}$. Bei Gleichheit der Bewertungen ergibt sich ein Wert von 1. Haben beide Benutzer keine Übereinstimmungen, ermittelt sich ein Wert von 0.

$$Jaccard(p,q) = \frac{\left|p \cap q\right|}{\left|p \cup q\right|} \tag{2.3}$$

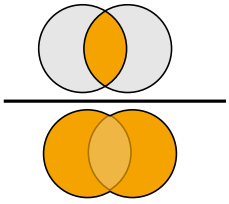


Abbildung 2.3.: Jaccard Distanz

Euklidische Metrik

Abstand zweier Punkte in einem n-dimensionalen Vektorraum über dem Körper $\mathbb{R}^n$. Der Nachteil dieser Metrik liegt darin, dass wenn ein Bewertungsvektor $\vec{p} = x \cdot \vec{q}$ ein Vielfaches eines anderen Vektors ist, diese Metrik dann einen großen Abstand der beiden Vektoren p und q ermittelt, obwohl die Tendenzen eigentlich identisch sind.

$$Euklid(p,q) = \sqrt{\sum_{i=1}^{n}(p_i - q_i)^2} \tag{2.4}$$

Kosinus Metrik

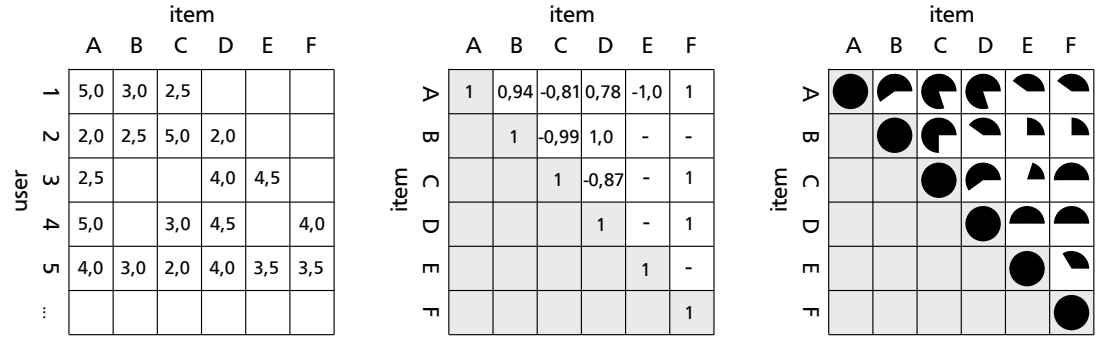
Die Kosinus Metrik interpretiert den Winkel zwischen den Vektoren als Ähnlichkeitsmaß. Bewertungen sollten für diese Metrik normiert werden. Bei einer binären Bewertung in Form von Verkaufsprotokollen stellt dies aber keinen Nachteil dar.

$$cos(p,q) = \frac{p \cdot q}{||p|| \cdot ||q||} \tag{2.5}$$

2.1.3 Memory basiert

Memory basierte Ansätze arbeiten direkt auf der *User-Item* Präferenzmatrix bzw. *Item-Item* Matrix (Abbildung 2.4a). Die *User-Item* Matrix ist eine Schreibweise, um alle Präferenzvektoren $\mathscr{P}_{u_i}$ darzustellen. Das Beispiel aus Abbildung 2.1 ist hier numerisch aufgelistet. Wir hatten vorher schon feststellen können, dass *User* u_3 Präferenzen für *Items* i_A, i_D und i_E geäußert hat. In diesem Fall ist das auf einer Skala von ein bis fünf geschehen, wobei höhere Werte eine bessere Bewertung bedeuten. Die Präferenzen von u_3 sind laut Abbildung 2.4a $\mathscr{P}_{u_3} = \{(u_3; i_A; 5{,}0), (u_3; i_D; 4{,}0), (u_3; i_E; 4{,}5)\}$.

In der Regel halten Memory basierte Verfahren den gesamten Datensatz im Speicher vor. Daher auch der Name. Die Daten müssen jedoch nicht komplett im Speicher vorgehalten werden, sondern können auch auf Caching Technologien oder Datenbanken zurückgreifen. Mit dem Einsatz solcher Techniken erhöht sich allerdings auch die Zugriffszeit und kann somit ein Problem für den Echtzeitansatz sein. Eine weitere Lösungsmöglichkeit ist es, das Problem in Teilprobleme zu zerlegen und auf einem parallelen Cluster berechnen zu lassen, wie es im Rahmen des Netflix Preises von Yunhong et al. [24] untersucht wurde. Nicht jedes Problem kann jedoch parallelisiert werden. Diese Verfahren verwenden statistische Verfahren, um eine Teilmenge der Nutzer zu finden, auch als Nachbarschaft bezeichnet, die in der Vergangenheit mit dem aktiven Benutzer Übereinstimmungen hatten. Sobald die Nachbarschaft eines aktiven Nutzers erstellt wurde, werden verschiedene Algorithmen angewandt, um eine Empfehlung zu ermitteln. Dabei kommen *nearest-neighbour* zum Einsatz.



(a) User-Item Korrelation **(b)** Item-Item: Pearson **(c)** Item-Item: Jaccard

Abbildung 2.4.: Utility Matrix

User basiert

	Alexander (u_3)
Constantin (u_1)	0,30
Clemens (u_2)	0,32
Alexander (u_3)	1,00
Markus (u_4)	0,56
Michael (u_5)	0,77

Tabelle 2.1.: Beispiel Kosinus Distanz

Beim *User* basierten Ansatz steht die Ähnlichkeit von Benutzern im Vordergrund der Betrachtung. Als Grundlage dienen die Reihenvektoren in der *User-Item* Matrix. Nehmen wir also an, *Items* A bis F sind Musikalben und *User* 1 bis 5 sind Schulfreunde. Die Freunde von Alexander (u_3) wollen ihm zum Geburtstag ein Musikalbum schenken. Sie wissen, dass er die Alben A,D und E bereits besitzt und kennen seine Vorlieben für diese Alben. Also vergleichen sie, welche der Freunde ähnliche Vorlieben haben wie Alexander. In Tabelle 2.1 sind die Kosinus Distanzen zwischen den Bewertungsvektor von Alexander und allen anderen Freunden aufgelistet. Der Freund mit den ähnlichsten Musikvorlieben ist Michael (u_5) mit einer Wertung von 0,77. Als Geschenk wird Album F gekauft, da Alexander dieses noch nicht besitzt und Michael es als bestes Album gewählt hat unter denen, die Alexander noch nicht besitzt.

Mit der Anzahl der registrierten Benutzer wächst die Komplexität des Problems. Je mehr Benutzer in einem solchen Verfahren teilnehmen, umso mehr Vergleiche müssen gemacht werden.

Item basiert

Beim *Item* basierten Ansatz von Sarwar et al.[20] wird die Ähnlichkeit der *Items* als Bezugspunkt herangezogen. Zunächst einmal wird die Ähnlichkeit der Items untereinander berechnet. Dies wird aus der *User-Item* Matrix abgeleitet und als *Item-Item* Matrix abgespeichert. Abbildungen 2.4b und 2.4c zeigen zwei verschiedene Ableitungen aus der *User-Item* Matrix aus Abbildung 2.4a. Das Vorgehen dabei ist identisch, nur die gewählte Metrik ändert sich. Verglichen wird jeder Spaltenvektor i_x mit jedem anderen Spaltenvektor i_y wobei $x \neq y$. Im Fall der Pearson Metrik besteht der Vektor aus numerischen Werten, auf die die Pearson Metrik angewandt wird. Die Person Distanz von $i_A = [5; 2; 2{,}5; 5; 4]$ und $i_B = [3; 2{,}5; \texttt{NaN}, \texttt{NaN}, 3]$ steht dabei in der *Item-Item* Matrix an Position (A,B) mit einem Wert von 0,94.

Die gleiche Matrix mit Jaccard Distanz lässt sich leichter nachvollziehen. Die Werte in der *User-Item* Matrix lassen sich dabei binär interpretieren. Alle Werte ungleich `NaN` werden als `true` interpretiert und andernfalls `false`. Vergleicht man die *Items* B und C miteinander, zeigt sich, dass vier *User* (1,2,4,5) vertreten sind. Überschneidungen gibt es lediglich bei Dreien (1,2,5). Das ergibt eine Jaccard Distanz von $3/4$. Diese taucht als Tortendiagramm in unserer *Item-Item* Jaccard Matrix an der Position(B,C) auf.

Das Resultat einer solchen *Item-Item* Matrix ist, dass man direkt ablesen kann, welche *Items* am besten zueinander passen. Auf das Beispiel der Freunde, die ein Geburtstagsgeschenk für Alexander kaufen wollen, wirkt sich das wie folgt aus: Alexander gefallen die Alben A, D und E. Aus Abbildung 2.4c lässt sich ablesen, dass die betragsmäßig größte Übereinstimmung zweier Alben bei $4/5$ liegt, nämlich zwischen den Alben C und A. Da Alexander das Album C noch nicht besitzt, werden seine Freunde ihm dieses schenken. Album F aus dem User basierten Ansatz hat auch hier eine hohe Bewertung von $1/2$.

Aus Vorlieben von *Usern* errechnen sich also Ähnlichkeiten zwischen *Items*. An dieser Stelle wird auch der Vorteil von *CF* sehr deutlich. Die Art und Weise, wie Ähnlichkeiten ermittelt werden, ist kontextunabhängig. Es ist irrelevant, ob hier Bücher, Musikstücke oder Autos bewertet werden. Das Resultat ist das gleiche.

Ermitteln der Präferenzen und Empfehlung

Das Ermitteln der Präferenzen und Empfehlungen aus den berechneten Ähnlichkeiten ist wohl der wichtigste Schritt in einem *CF* System.

Top-N Empfehlungen

Die Idee von Top-N Recommendation, wie sie auch von Su and Khoshgoftaar[22] verfolgt wird, ist es, die jeweils N ähnlichsten Gegenstände als Basis für die Empfehlung zu ver-

wenden. Die Ähnlichkeit ist dabei, wie bereits an den Beispielen gesehen, abhängig von der Metrik.

User basierte Top-N: Der *User* basierte Top-N Algorithmus identifiziert zunächst die k, zum aktiven *User* u_a, ähnlichsten *User*, mittels einer Ähnlichkeitsmetrik und der User-Item Matrix. In dieser Matrix (Abbildung 2.4b) wird jeder *User* als ein Vektor von Präferenzen charakterisiert. Nachdem die κ ähnlichsten *User* ermittelt wurden, werden ihre Einträge aggregiert, zusammen mit der Häufigkeit ihres Auftretens. Empfohlen werden dann die N häufigsten *Items*, die nicht bereits vom aktiven *User* u_a bewertet wurden. Die Komplexität dieses Verfahrens steigt mit der Anzahl der *User*

Item basierte Top-N: Item basierte Top-N Algorithmen gehen das Problem der Skalierbarkeit der traditionellen Top-N *User* basierten Algorithmen an. Dort ist es erforderlich, dass für jeden aktiven *User* erneut eine Nachbarschaft errechnet wird. Das gilt insbesondere dann, wenn die Daten in der User-Item Matrix erneuert wurden. Item basierte Verfahren gehen gemäß [20, 22] einen anderen Weg. Diese Algorithmen berechnen zunächst die κ ähnlichsten Gegenstände für jeden Gegenstand im Sortiment. Diese Berechnung ist sehr rechen- und speicheraufwendig, muss aber nicht zur Laufzeit passieren, sondern kann vorher ermittelt werden. Zur Laufzeit fällt dann nur noch die Vereinigungsmenge aller κ-ähnlichen Gegenstände über alle von u_a präferierten Gegenstände an. In einem letzten Schritt werden alle bereit bekannten Elemente entfernt. Die verbleibende Menge wird sortiert nach Ähnlichkeit und die ersten N Elemente werden für die Empfehlung herangezogen.
Betrachtet man beispielsweise die Abbildung 2.4c, lassen sich für eine 2-Nachbarschaft für das *Item* A schnell die beiden *Items* C und D ausmachen. Möchte man nun für *User* 3 eine Empfehlung auf dieser Basis abgeben, dann stünde *Item* C an vorderster Stelle. *Item* D würde aussortiert, weil dieses bereits von *User* 3 gekauft wurde. Auffällig ist auch, dass bei n *Items* nicht der Speicherplatz für n^2 Wertungen benötigt wird, sondern nur $\sum_{i=1}^{n}(n-i) = n \cdot (n-1)/2$, da die Werte zur Nebendiagonale spiegelsymmetrisch sind und die Werte auf der Diagonalen selber alle identisch sind.

2.1.4 Modell basiert

Der *Modell* basierte Ansatz unterscheidet sich von dem Memory basierten dahin gehend, dass er von den Benutzerbewertungen zunächst ein Modell erstellt. Das bedeutet, sie abstrahieren zunächst die vorhandenen Daten [20]. Hierbei kommen Algorithmen aus dem *Maschinellen Lernen* zum Einsatz wie z.B. *Bay'sche Netze*, *Clustering* und regelbasierte Herangehensweisen. Dabei wird das Problem des *CF* auf ein Klassifizierungsproblem abgebildet. Gesucht wird die Klasse von Gegenständen, die dem aktiven Benutzer gefallen.

Clustering

Bei *Memory* basierten Verfahren besteht ein Problem darin, dass die Komplexität der Berechnung sowohl mit der Anzahl der *User*, als auch der Anzahl der *Items* steigt. Dies schlägt sich in der Rechenzeit auch im Speicherverbrauch nieder. Gong [8] geht auf die Möglichkeit des Clusterings ein. Dabei wird die Komplexität des Problems dadurch verringert, dass in einer der beiden oder auch in beiden Dimensionen, Elemente zusammengefasst werden. Abbilung 2.5 zeigt die Reduktion der *Item* Ebene. Dabei werden ähnliche *Items*, wie beispielsweise C und D, zu einem neuen, virtuellem *Item* zusammengefasst. Alle weiteren Verfahren arbeiten dann auf den in ihrer Anzahl reduzierten virtuellen *Items*.

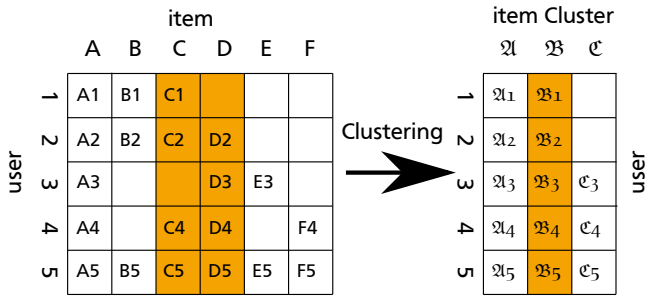


Abbildung 2.5.: Item clustering

Singular Value Decomposition

Singular Value Decomposition (SVD) ist eine Technik aus der linearen Algebra, die wie in [5] dazu verwendet werden kann, um die Komplexität eines Recommender Systems zu reduzieren. Ausgangspunkt ist die *User-Item* Matrix A, die so bearbeitet wird, dass einem *Item* mindestens zwei Wertungen zugewiesen sind. Eine Singulärwertzerlegung von $A \in \mathbb{R}^{m \times n}$ mit Rang r hat die Form:

$$A = U \sum V^T \tag{2.6}$$

Dabei ist U eine unitäre $m \times m$ Matrix, V^T die Adjungierte einer unitären $n \times n$ Matrix V und Σ eine $m \times n$ Diagonalmatrix, auf deren Diagonalen die Singulärwerte stehen. Diese Singulärwerte repräsentieren die Höhe der Varianz der Originaldaten. Die kleinsten Singulärwerte werden in Σ^* auf Null gesetzt. Als Bewertungsgrundlage dient danach $A' = A^T \cdot U \cdot \Sigma^*$, die weniger Einträge enthält, als die Originaldaten. Die Problemdimension wird dabei durch Abstrahieren der Daten verringert. Ein zusätzlicher Rechenaufwand in der Vorverarbeitung der Daten muss jedoch einkalkuliert werden.

2.1.5 Hybrid recommenders

Bisher wurde das *cold start* Problem nicht weiter betrachtet. Das *cold start* Problem tritt immer dann auf, wenn nicht genügend Informationen vorliegen, um Empfehlungen zu tätigen. So kann es vorkommen, dass nicht zwischen allen *Items* eine Ähnlichkeit errechnet werden kann, weil die vorliegenden Daten dies nicht hergeben. In diesem Fall ist es erforderlich, alternative Informationsquellen zu bemühen, um die Lücken zu schließen. Andernfalls können in diesen Fällen einfach keine Aussagen getroffen werden.

Kontextangereicherte CF

Bei kontextangereicherten CF Verfahren [22] kommt diese zusätzliche Information aus den Eigenschaften der Gegenstände. So wird beim kontextangereichertem CF beispielsweise ein *naïve Bayes* als Content Klassifizierer eingesetzt. Der Klassifizierer erstellt Vorhersagen für alle Gegenstände, für die noch keine Bewertung vorliegt, und füllt die fehlenden Felder auf. Der Empfehlungsprozess wird dann über die Matrix der Pseudobewertungen berechnet. Dabei kommen in der Regel gewichtete Metriken zum Einsatz. Das bedeutet, dass *User* gegebene Bewertungen mehr ins Gewicht fallen als geschätzte Bewertungen.

Diese Techniken machen es auch bei spärlich besetzten Matrizen möglich, in jedem Fall Empfehlungen abzugeben. Zudem können diese hybriden Systeme im Einzelfall qualitativ bessere Vorhersagen tätigen. Für den Betrieb von Echtzeitsystemen bedeutet dies jedoch einen erheblich höheren Rechenaufwand. Aus Abschnitt 3.1 ist beispielsweise erkennbar, dass ein Käufer im Durchschnitt 12 Bücher bewertet hat. Das gesamte Sortiment umfasst jedoch fast 40.000 Bücher. Das bedeutet in der Berechnung eine Steigerung in der Größenordnung von 10^3. Der Einsatz solcher Techniken ist demnach in einer Echtzeitumgebung fragwürdig.

2.2 Evaluation

2.2.1 Performanzmaß

Herlocker et al. geht in [10] auf viele verschiedendste Aspekte der Bewertung von Recommender Systemen ein. In den folgenden Abschnitten sollen die wesentlichen Aussagen zusammengefasst werden.

Die Evaluation von CF Techniken ist aus verschiedensten Gründen komplex. Zum einen sind Algorithmen besser oder schlechter auf verschiedene Datensätze anwendbar. Einige Algorithmen wurden beispielsweise auf Daten optimiert, die wesentlich mehr *User* als *Items* haben. (So hat der MovieLens Datensatz 65.000 *User* und um die 5.000 Filme.) Diese Algorithmen sind in Umgebungen, wo es wesentlich mehr *Items* als *User* gibt, ungeeignet.

Ähnliche Phänomene treten auf, wenn man die Dichte der Bewertungen, die Bewertungsskala und andere Eigenschaften des Datensatzes heranzieht. Zum anderen ist es schwierig, Evaluationen verschiedener Algorithmen miteinander zu vergleichen wenn die Ziele der Algorithmen unterschiedliche sind. So kann je nach Szenario die Reduzierung der False Positive Rate oder aber auch der Erhalt der Ratingreihenfolge im Vordergrund stehen.

Ein nicht zu unterschätzendes Problem bei *CF* Verfahren ist die Variabilität der Nutzerwertungen. Legt man einem Benutzer einen Gegenstand mehrmals zur Bewertung vor, werden seine Bewertungen variieren. Diese Varianz kann man als natürliche Barriere der Vorhersagegenauigkeit sehen. Kein Algorithmus kann eine bessere Vorhersage treffen, als die Varianz, die in den Bewertungen des Users selbst vorhanden ist. Eine bloße Betrachtung der Accuracy wird der Bewertung von Recommender Systemen also nicht gerecht. Im Gegenteil, es führt dazu, sich einseitig auf dieses Maß festzulegen und riskiert die Vernachlässigung anderer wichtiger Eigenschaften von Recommendern. In letzter Konsequenz ist nur die Zufriedenheit der Personen ausschlaggebend, die ein solches System verwenden, denn nur dann wird es aktiv verwendet und es kommt zu spontanen Kaufentscheidungen.

2.2.2 Zielsetzungen

Herlocker et al. definiert eine Vielzahl an Zielsetzungen (User Tasks), die ein Recommender System verfolgen kann. Im Folgenden werden die Zielsetzungen näher erläutert, die dem *KIS* System entgegen kommen.

Find Good Items

Dieses Task fordert, dass dem Benutzer eine sortierten Liste von *Items* ausgegeben wird. Zusätzlich zu der Liste wird eine *Prediction* (2.1.1) für jedes Item ausgegeben. Eine Forderung, die in dieser Implementierung nicht nachgegangen wird.

Im Gegensatz zu *Find Good Items* fordert der Task *Find All Good Items*, dass alle passenden Empfehlungen gefunden werden. Das ist in diesem Szenario nicht nur nicht gefordert, sondern gar nicht erwünscht. Empfehlungen in einem Webshop sollen anregen, unbekannte Bücher zu erkunden. Eine Auflistung tausender, wenn auch passender, Empfehlungen schreckt eher davor zurück.

Just Browsing

Umfragen von MovieLens und Amazon.com Kunden haben ergeben, dass Artikelempfehlungen oft nur als Hilfe verwendet werden, um durch das Produktangebot zu browsen. Und dies, unabhängig von Kaufabsichten. In diesem Fall ist die Genauigkeit der Vorhersage weniger entscheidend als die Bedienbarkeit der Oberfläche. Eine intuitive Bedienung lädt dazu

ein, unbekannte Artikel zu erkunden und führt in letzter Konsequenz zu Kaufentscheidungen, die ohne ein solches Feature wohl nicht getroffen würden.

2.2.3 Accuracy als Kriterium

Gängige Accuracy Metriken sind Mean Absolute Error (*MAE*), *Precision* und *Recall* und verwandte Metriken wie z.B. *F1* Metrik und *ROC* Kurven. Im Allgemeinen unterscheidet man *predictive accuracy*, *classification accuracy* und *rank accuracy* Metriken. In den folgenden Abschnitten werden einige Vertreter dieser Metriken zusammen mit deren Vor- und Nachteilen genannt.

Mean Absolute Error (*MAE*)

Predictive accuracy Metriken messen, wie nahe die vom Recommender System ermittelten Vorhersagen an den realen Bewertungen sind. Ein Vertreter dieser Kategorie ist *MAE*. Es misst die absolute Abweichung zwischen einer Vorhersage p_i und der tatsächlichen Bewertung r_i über die Menge aller *Items* $\mathscr{I}$ mit der Mächtigkeit $|\mathscr{I}|$.

$$MAE = \frac{\sum_{i=1}^{N} |p_i - r_i|}{|\mathscr{I}|} \tag{2.7}$$

Pro

- Misst die *accuracy* der Vorhersage an jedem Rang
- *MAE* ist eine weit verbreitete und gut verstandene Metrik

Kontra

- Für das Ziel *Find Good Items* weniger geeignet
- Weniger geeignet, wenn die Anzahl der Untergliederung der Präferenzen gering sind.

Precision & Recall

Classification accuracy misst die Häufigkeit, mit der ein Recommender System korrekte oder falsche Entscheidungen über gute bzw schlechte *Items* macht. Die wohl bekanntesten Vertreter dieser Kategorie sind die beiden Maßzahlen *precision* (Gleichung 2.8) und *recall* (Gleichung 2.9).

$$\texttt{precision} = \frac{\{\texttt{relevante Items}\} \cap \{\texttt{gefundene Items}\}}{\{\texttt{gefundene Items}\}} \tag{2.8}$$

$$\texttt{recall} = \frac{\{\texttt{relevante Items}\} \cap \{\texttt{gefundene Items}\}}{\{\texttt{relevante Items}\}} \tag{2.9}$$

Pro

- Sehr gut geeignet, wenn echte binäre Präferenzen vorliegen.

Kontra

- Normierung von Nutzerwertung wird notwendig.
- Gegenstände müssen in relevant und irrelevant eingeteilt werden können.

Rank accuracy

Rank accuracy Metriken messen die Fähigkeit eines Recommender Systems, eine geordnete Liste von Empfehlungen zu tätigen, die in ihrer Präferenzreihenfolge dem des wahren Rankings entspricht. Ein Vertreter dieser Metriken ist der *NDPM* „normalized distance-based performance measure“ (Gleichung 2.10). Sie ist dazu geeignet, zwei schwach geordnete Ranglisten miteinander zu vergleichen. Wobei C^- die Anzahl sich widersprechender Präferenzen ist, C^u die Anzahl der übereinstimmenden Präferenzen und C^i die Gesamtzahl der Präferenzen darstellt.

$$NDPM = \frac{2C^- + C^u}{2C^i} \tag{2.10}$$

Pro

- Gut dafür geeignet, um die Reihenfolge in der Recommendation zu beureilen. Die absoluten Werte sind nicht relevant.

Kontra

- Nicht dafür geeignet, Vorhersagen (*prediction*) zu evaluieren (nur *recommendations*).

2.2.4 Weiterführende Kriterien

Neben diesen Metriken, müssen auch andere Qualitätskriterien berücksichtigt werden.

Coverage

Die *Coverage* ist das Maß der Menge an Artikeln, über die das System eine *Prediction* oder *Recommendation* tätigen kann. Sie ist besonders wichtig für Systeme, die auf *Find All Good Items* abzielen. Die einfachste Methode, *Coverage* zu ermitteln ist, zufällige *User - Item* Paare zu generieren und festzustellen, für wie viele dieser Paare eine *Prediction* erstellt werden kann. Ähnlich wie bei *Precision* und *Recall* muss die *coverage* immer in Kombination mit *Accuracy* ermittelt werden. Ansonsten würde man der Versuchung verfallen, Pseudo *predictions* zu erstellen.

Novelty & Serendipity

Recommender Systeme mit einer sehr hohen *Accuracy* können in ihrem Nutzen im Sinne der vorgegebenen Ziele völlig ungeeignet sein. Empfiehlt ein Recommender System eines Musikhändlers beispielsweise das unter Musikliebhabern allseits bekannte „White Album“ der Beatles, dann wird der Recommender mit diesem Vorschlag mit hoher Wahrscheinlichkeit nicht falsch liegen. Es kann natürlich auch sein, dass jemand dieses Album nicht besitzt, weil er sich bewusst gegen einen Kauf entschied und nicht etwa, weil ihm das Album nicht bekannt gewesen ist. Eine Empfehlung dieses Albums wird dann mit hoher Wahrscheinlichkeit auch zu keiner Kaufentscheidung führen. Gewünscht sind demnach neuartige (*Novelty*) Empfehlungen.

Die Neuartigkeit alleine ist aber auch nicht unbedingt ausschlaggebend. Im Bereich Recommender System will man im Grunde *Serendipity* erreichen. Unter *Serendipity* versteht man das glückliche Auffinden von Produkten, die unerwarteterweise interessant sind für den Nutzer des Systems, anstatt alle Bücher eines Lieblingsautors zu listen. Anstatt Empfehlungen zu geben, auf denen der Leser ohnehin folgen würde, ist es interessanter, Bücher von Autoren zu empfehlen, die der Leser bisher noch nicht kannte. Wie man leicht sieht, folgt aus *Serendipidity* $\rightarrow$ *Novelty*. Der Umkehrschluss gilt nicht. Objektive Metriken für *Novelty* und *Serendipidity* sind schwer zu ermitteln. Am Ende hängt es von jedem einzelnen *User* ab, ob er einen Vorschlag als neuartig und nützlich betrachtet. Eine aussagekräftige Erhebung dieser Metriken ist somit nur mit einer Umfrage an die Nutzer möglich.

Confidence

Ein Recommender System befindet sich immer in der Situation, zwischen *Strength* und *Confidence* einer Empfehlung abzuwägen. Dabei versteht man unter *Strength* wie sicher das

System ist, dass dem *User* diese Empfehlung gefällt. Unter *Confidence* versteht man, wie sicher das System ist, dass eine Empfehlung passend ist. Dabei können sich *Strength* und *Confidence* widersprechen.

Hierbei handelt es sich primär um ein Verständniskonzept. Über eine Rückmeldung der *Confidence* an den *User* können Entscheidungsprozesse manipuliert werden. Eine solche Metrik lässt sich schwer in Werte fassen. Man kann es sich jedoch leicht verständlich machen. Jeder ist schon einmal in der Lage gewesen, in der ein Freund einen zu etwas überreden musste, wie beispielsweise der Fahrt auf der Achterbahn. Zunächst lehne ich die Fahrt ab. Mir gefällt der Gedanke nicht. Mein Freund weiß, dass ich zunächst ablehne. Die *Strength* seiner Empfehlung ist sehr gering. Er ist sich jedoch sicher, dass eine Fahrt auf der Achterbahn mir Spaß machen wird. Seine *Confidence* ist sehr hoch. Deswegen empfiehlt er mir mit Nachdruck, eine Fahrt zu wagen. Ich willige ein und nach der Fahrt stelle ich fest, dass die Fahrt Spaß gemacht hat. Ohne einen solchen Nachdruck wäre ich also nicht bereit gewesen, etwas Neues zu probieren.

Lern Rate

Die Qualität der *CF* basierten Recommender Systeme hängt von der Menge der Eingabedaten ab. Nimmt die Menge der Eingabedaten zu, sollte auch die Qualität der Vorhersagen steigen. Verschiedene Algorithmen weisen unterschiedliche Lernkurven auf. Für Recommender Systeme lassen sich drei Lernraten identifizieren. Da gibt es zum einen die *overall learning rate* als Qualitätsfunktion über die Gesamtzahl aller Ratings, die *per Item learning rate* als Qualitätsfunktion über die Anzahl der Ratings pro *Item* und äquivalent dazu die *per User learning rate*.

Laufzeit und Speicherverbrauch

Für die Akzeptanz von Recommender Systemen ist die Zeit bis zum Ermitteln der Vorschläge besonders kritisch. Niemand beachtet Empfehlungen, die erst Minuten später auf dem Bildschirm erscheinen. Deswegen müssen Ergebnisse in Bruchteilen einer Sekunde berechnet und auf dem Bildschirm ausgegeben werden. Für den Betreiber eines solchen Systems steht dem immer der Einsatz von Ressourcen gegenüber. Dies gilt sowohl für die Rechenkapazität als auch für den Speicherplatz. Wenn man genügend Speicherplatz besitzt, kann man im Voraus jede beliebige Empfehlung vorweg berechnen, um dann in einer Reverse Lookup Table nachzuschlagen. Die Problemstellungen können aber durchaus eine Größe erreichen, das ein kosteneffizienter Einsatz nicht möglich ist. Für alle Memory basierten Recommender Verfahren ist immer eine natürliche Grenze gesetzt. Es können nur so viele Daten verarbeitet werden, wie eine physikalische Maschine verwalten kann. Ist die Grenze erreicht, ist der Einsatz von Computer Cluster unausweichlich, ohne bei der Rechenzeit signifikante Einschnitte hinnehmen zu müssen. Bei der Implementierung der Algorithmen ist also auf eine speichersparende und parallelisierbare Umsetzung zu achten.

3 Eingangsdaten

In den folgenden Abschnitten wird kurz darauf eingegangen, wie sich die Eingangsdaten präsentieren, die als Grundlage für das Recommender System herhalten. Datenlieferant ist durchgehend das Produkt *Kunden Informations System* (*KIS*) der Firma indis Kommunikationssysteme GmbH(Mainz), einer EDV-Lösung für den Geschäftsablauf von Buchhändlern mit angeschlossenem Online-Shop.

3.1 Beschreibung der Eingabedaten

Struktur der Daten

Ein auf Benutzerwertungen basiertes Collaborative Filtering (*CF*) beruht auf den Präferenzen, die Benutzer gegenüber den Produkten äußern. Präferenzen können sich in verschiedener Weise äußern. Grundsätzlich kann man zwischen expliziten und impliziten Bewertungen unterscheiden. Bei einer expliziten Bewertung wird die jeweilige Person konkret zu ihren Vorlieben eines Produktes gefragt und muss das in der Regel anhand einer Skala bewerten.

Im Fall des Produkts *KIS* liegt eine implizite Bewertung in Form von Kaufverhalten vor. Das bedeutet, dass die Bewertung nicht als fein granulierte Bewertungsfunktion vorliegt, sondern lediglich ein monoadische Aussage getroffen werden kann. So kann ein Buch nur die Bewertung „gefällt“ annehmen oder die Wertung ist, wie in den meisten Fällen, nicht bekannt. Es kann keine Aussage über das Missfallen von Büchern getroffen werden. Ferner wird unterstellt, dass bei einem Kauf eine positive Bewertung vorliegt. Diese Annahme muss nicht immer zutreffen. Es ist allerdings davon auszugehen, dass in der Regel Bücher dann gekauft werden, wenn sie dem Käufer gefallen. Binäre Bewertungen können sich aber für die Laufzeit positiv auswirken, weil die Metriken zur Berechnung der Ähnlichkeit dadurch vereinfacht werden können. Eine explizite Bewertung von Produkten kann zwar bessere Ergebnisse liefern, weil auch negative Relationen zwischen den Produkten ausgemacht werden können, wie etwa „Harry Potter Leser mögen keine Twilight Bücher“. Das ist jedoch nicht zwingend und ist stark abhängig von der Qualität der zugrunde liegenden Datenbasis.

Bewertungsskalen bringen jedoch auch eine Reihe von Problemen mit sich, die das Ergebnis verfälschen können. So sind explizite Bewertungen oft nicht so zahlreich wie implizite. Dazu kommt, dass Bewertungen normiert werden müssen. Jemand der grundsätzlich gute Bewertungen verteilt, dessen schlechte Bewertung ist wohl signifikanter als bei jemandem, der ohnehin nur schlechte Bewertungen vergibt. Des weiteren besteht immer das Problem, dass die Personen nicht „allwissend“ sind. Angenommen ein Leser ist ein Anhänger von Fantasy Romanen. In diesem Genre hat er bevorzugte und weniger bevorzugte Autoren. Folglich wird er die Werke des einen Autors immer als schlechter bewerten. Gibt man ihm aber Groschenromane zu Lesen wird er wohl diese als schlecht bewerten. Die Fantasy Bücher des

nicht gemochten Autors steigen dadurch, relativ gesehen, im Ansehen. Diese Problematik verschärft sich bei der weit verbreiten 5 Sterne Skala. Goldberg et al.[7] stellt anhand von Jester, einem Recommender System für Witze, eine breit gefächerte Bewertungsskala vor, bei der anstelle von Sternen auf eine Bildskala geklickt werden muss. Der Nutzer kann damit oft nicht sagen, welchen Wert er wählt, sondern entscheidet mehr aus dem Bauch heraus.

In der *KIS* Datenbank ist das Kaufverhalten von Kunden mehrerer Buchhandlungen erfasst. Die Buchhandlungen können in vier verschiedene Kategorien aufgeteilt werden.

- Allgemeine Buchhandlungen (*allg*)
- Buchhandlung für medizinische Bücher (*med*)
- Buchhandlung für Steuerrecht (*steuer*)
- Interne Buchhandlung der Deutschen Bahn (*db*)

Da es sich hier jeweils um verschiedene Zielgruppen handelt und auch die Sortimente sich wenig bis gar nicht überschneiden, werden bei allen weiteren Ausführungen die Daten getrennt voneinander betrachtet. Überlegungen für einen Datensatz sind aufgrund ihrer ähnlichen Struktur aber dennoch allgemein übertragbar.

Folgende Daten stammen aus der *KIS*-Datenbank. Die ursprünglichen Quellen unterscheiden sich jedoch zum Teil deutlich. Viele Datensätze wurden aus der Warenwirtschaft importiert, andere stammen aus *KIS* direkt. Wie genau sich die Daten zusammensetzen, lässt sich im Nachhinein nicht feststellen. Datenbestände, die aus der Warenwirtschaft importiert wurden, weisen oft Unvollständigkeiten auf. Aus diesem Grund war es notwendig, die Eingabedaten nach eindeutigen Parametern zu filtern, um eine einheitliche Grundlage zu erhalten. Alle Referenznummern, die eine valide ISBN-10 oder ISBN-13 Nummer besitzen, wurden einheitlich auf ISBN-13 umgestellt und abgelegt. Die Einträge, die keine gültige ISBN tragen, werden nicht weiter berücksichtigt. In manchen Fällen ist es nicht mehr möglich, Kunden eindeutig zuzuordnen. So kann es sein, dass der gleiche Kunde unter mehreren Kennungen im System abgespeichert ist. Dies wird zu Ungenauigkeiten im Empfehlungsprozess führen. Die Zahlen in Tabelle 3.1 und die Angaben zur Verteilung beziehen sich daher ausschließlich auf gefilterte Daten. Darüber hinaus sind die Zeiträume, in denen diese Daten erfasst wurden unterschiedlich lang.

Verteilung der Daten

Für die Analyse der Daten, unter Zuhilfenahme von [6, 18, 21], wird exemplarisch der *med* Datensatz herangezogen. Untersucht werden drei Größen, die Verteilung der Bücher, der Kunden und der Äquivalenzklassen. Alle diese Größen können Einfluß nehmen auf den Ausgang eines Recommender Systems. Jeweils zwei Diagramme beschreiben die Daten. Das linke Diagramm listet auf der x-Achse den Index der betrachteten Größe, absteigend sortiert nach der Häufigkeit der auftetenden Bücher / Exemplaren. So lässt sich quantitiv ablesen,

wie viele verschiedene Bücher, Kunden und Äquivalenzklassen vorliegen und wie viele Bücher / Exemplare einem Index zugeordnet sind. Im Falle der Buchtabelle bedeutet das, dass ein Buch mit dem Index x einen Verkauf von $f(x)$ Exemplaren erzielt hat. Das rechte Diagramm zeigt die Vereilungsfunktion. Aus ihr lassen sich beliebige Quantile ablesen.

In Tabelle 3.1 sind zusammenfassend alle Datensätze aufgelistet. Sie fasst die Anzahl der Kunden und die Anzahl der Bücher pro Datensatz zusammen. Dazu wird mit $\overline{k}$ angegeben, von wie vielen Kunden ein Buch im Schnitt gekauft wurde und mit $\overline{n}$ wird erfasst, wie viele Bücher ein Kunde im Schnitt gekauft hat.

Bücher: Aus Abbildung 3.1a ist zu erkennen, dass die medizinische Buchhandlung nur etwa 40.000 Bücher in ihrem Online Sortiment gelistet hat. Das ist im Vergleich zur Gesamtanzahl aller ISBN gelisteten Bücher (ca 1,1 Millionen) sehr gering. Selbst die Gesamtgröße aller Sortimente beschränkt sich auf eine relative geringe Anzahl von 115.000 Büchern. Weiter lässt sich aus Abbildung 3.1b entnehmen, dass fast 65% der Bücher lediglich einmal verkauft wurden. Im empirischen Durchschnitt wurde jedes Buch 4,86-mal verkauft.

Kunden: Die Verteilung bei den Kunden stellt sich ähnlich dar. (Abbildung 3.2a) Die im System gelisteten Kunden beschränken sich auf etwa 15.000. Davon haben etwa 55% der Kunden nur ein einziges Buch bestellt.

Äquivalenzklassen: Ausgehend von den vorhandenen Bestellungen wurden alle Bücher in Äquivalenzklassen einsortiert. Zwei Bücher sind genau dann in einer Äquivalenzklasse, wenn sie in der transitiven Hülle der Jaccard Relation liegen. Zwei Elemente stehen in Jaccard Relation zueinander, wenn deren Jaccard Distanz in $\mathbb{Q} \setminus 0$ ist. In Abbildung 3.3 wird die Verteilung der Äquivalenzklassen aufgezeigt. Von den fast 41.000 Büchern sind etwa 34.000 Bücher in einer Äquivalenzklasse. Damit ist ein Großteil der Bücher über die verschiedenen Benutzer verknüpft. Um die 3.000 Bücher sind in ihrer eigenen Äquivalenzklasse. Diese Bücher weisen keinerlei Zusammenhang untereinander auf und es ist zu erwarten, dass diese Daten keinerlei Relevanz für den Ausgang der Empfehlung haben. Durch das Entfernen dieser Daten könnte also eine Reduktion der Problemgröße erreicht werden. Auf den Einfluss der Äquivalenzklassen auf die Speichergröße wird in Abschnitt 4.1.2 weiter eingegangen.

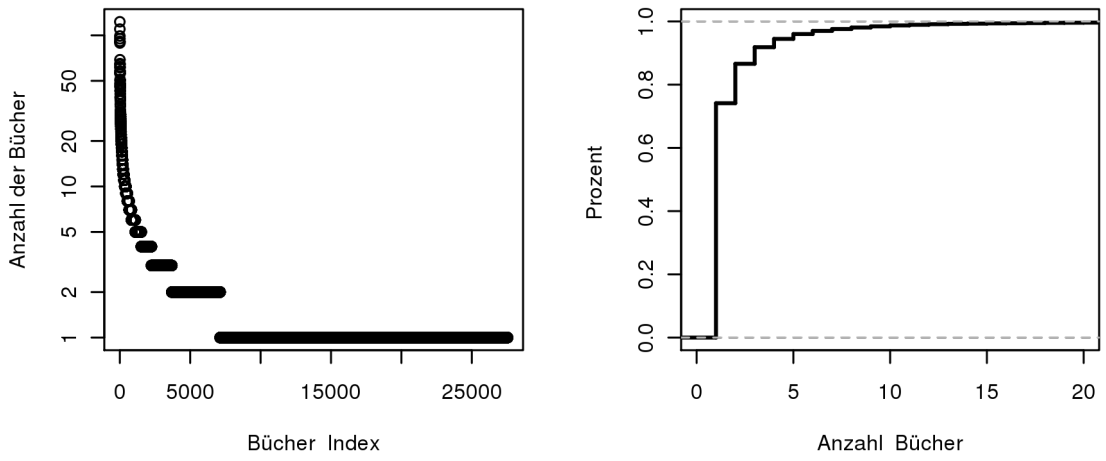
3.2 Daten Interpretation

Zusammenfassend lässt sich sagen, dass sich bei allen Datensätzen sich ein Muster beobachten lässt. Es gibt einige wenige Kunden, die sehr viele Bücher erstanden habe und eine große Masse an Kunden, die nur ein Buch gekauft haben. Bis zu einem gewissen Grad ist eine solche Verteilung zu erwarten gewesen. Sowohl der *med* als auch der *allg* Datensatz weisen extreme Werte bei einigen wenigen Kunden auf. Siehe Abbildungen 3.2a und A.2a. Im *med* Datensatz werden bis zu Größenordnung 1.000 Bücher verkauft. Dies lässt sich damit begründen, dass diese Buchhandlung an eine Universität angeschlossen ist und viele Bücher von den Fachbereichen für den Bibliotheksbestand erstanden werden. Im Fall der all-

Datensatz	Kunden	Bücher	$\overline{k}$	$\overline{n}$
steuer	1504	4324	1,69	4,86
bahn	4170	14488	1,70	5,90
med	16300	39239	2,0	4,85
allg	29626	57052	1,72	3,31

Tabelle 3.1.: Daten: Zusammenfassung

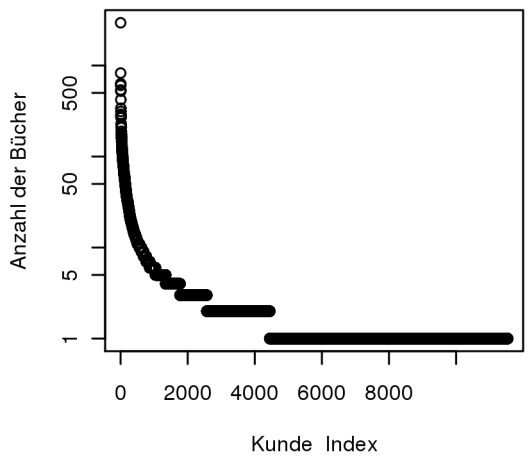
gemeinen Buchhandlungen liegt die Ursache für diese extremen Werte einiger Kunden darin begründet, dass viele Lagerbestellungen der verschiedenen Mitarbeiter der Buchhandlungen vorliegen aber auch Fehler aus Altdatenimporten sind der Grund dafür. Diese Extremwerte enthalten keine Information über individuelle Kundenvorlieben, umschreiben aber das Sortiment einer Buchhandlung und können aus diesem Grund nicht einfach herausgefiltert werden. Stattdessen muss diese Eigenschaft bei der Bewertung der Daten berücksichtigt werden.



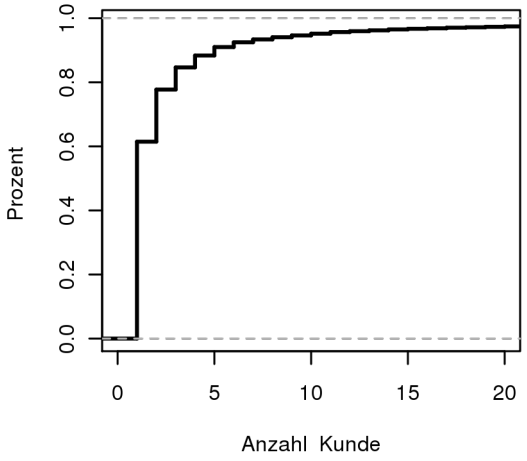
(a) Anzahl der verkauften Exemplare pro Buch

(b) Verteilungsfunktion(Bücher)

Abbildung 3.1.: *med* Daten - Bücher

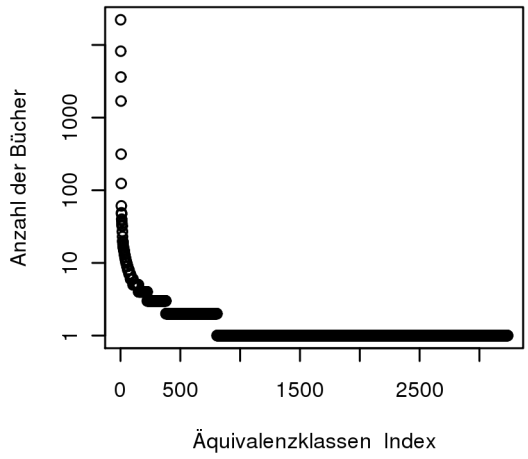


(a) Anzahl der gekauften Bücher pro Kunde

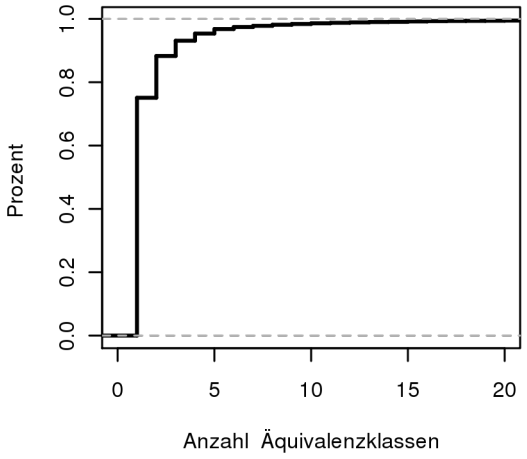


(b) Verteilungsfunktion(Kunde)

Abbildung 3.2.: *med* Daten - Kunde



(a) Größe der Äquivalenzklassen



(b) Verteilungsfunktion(Äquivalenzklassen)

Abbildung 3.3.: Äquivalenzklassen

4 Umsetzung eines *Item* basierten Recommender

Für die Umsetzung eines Recommender System besteht immer ein Tradeoff zwischen Rechenzeit, Speicherverbrauch und Qualität (Abbildung 4.1), wobei Qualität selbst ein Abwägen zwischen den Evaluationskriterien ist. Es gilt, mit den gegebenen Techniken und den vorhandenen Daten ein zufriedenstellendes Ergebnis zu erlangen und dabei diese drei Dimensionen gegeneinander abzuwägen. Die Anzahl der Kunden und Bücher bewegt sich in einem Bereich in dem Modell basierte Ansätze noch nicht nötig scheinen. Unter den Memory basierten Verfahren ist das Item basierte Verfahren vielversprechend, da der tatsächliche Rechenaufwand zur Laufzeit nur abhängig von der Anzahl der abgegebenen Bewertungen ist, wie auch von Linden et al. [14] vertreten wird.

Die Anzahl der abgegebenen Bewertungen ist im Durchschnitt bei allen Datensätzen sehr gering, womit sich dieses Verfahren gut für den Echtzeit Einsatz eignet.

Zudem kann bei dem Item basierten Verfahren Rechenleistung vorweggenommen werden. Sie wird maßgeblich durch die Anzahl der *Items* beeinflusst. Diese ist zwar durchgängig größer als die Anzahl der *User*, bewegt sich jedoch in der gleichen Größenordnung. Damit ist der Einsatz eines *Item* basierten Ansatzes gegenüber einem *User* basierten Ansatz, dessen Aufwand primär mit der Anzahl der *User* zusammenhängt, vertretbar.

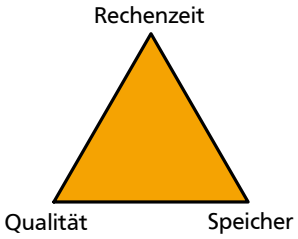


Abbildung 4.1.: Tradeoff bei der Umsetzung

Wie bereits in Abbildung 2.4 zu erkennen, ist für einen *Item* basierten Recommender das Erstellen einer Korrelationsmatrix notwendig. Für die Berechnung dieser Matrix ergeben sich zwei Probleme. Zum einen die Rechenzeit, die benötigt wird, um diese Matrix zu erstellen und zum anderen der Speicherplatz, welcher benötigt wird, um diese Matrix abzuspeichern.

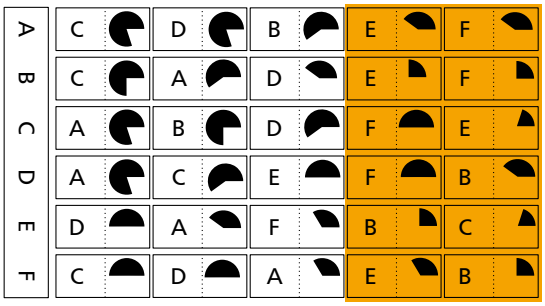


Abbildung 4.2.: Fix SmartStore

4.1 Speicher Komplexität

4.1.1 Allgemeine Betrachtung

Betrachtet man den Speicher, den man braucht, um das Beispiel aus Abbildung 2.4c abzuspeichern, wird klar, dass eine naive Implementierung n^2 Speicherplätze benötigt. Berücksichtigt man zudem, dass alle Werte an der Nebendiagonalen gespiegelt sind und die Nebendiagonale selber nicht benötigt wird, dann ist es möglich den Speicherbedarf auf $n \cdot (n-1)/2$ zu reduzieren. Problem bei dieser Form der Speicherung ist, dass es keine effiziente Möglichkeit gibt, die Präferenzen für ein *Item* aus dem Speicher zu lesen. Sind die Daten reihenweise abgelegt, und man möchte die Präferenzen für das *Item* C auslesen, ist es notwendig, Reihe C aus dem Speicher zu lesen, um die Werte für D bis F zu erhalten und danach noch die Reihen B und A, um die jeweils fehlenden Werte zu ergänzen. Um eine solche Abfrage zu beschleunigen, wäre ein Einsatz von Indizes notwendig, die aber ihrerseits Speicherplatz benötigen. Für die etwa 55.000 Einträge aus dem Allgemeinen Datensatz (Tabelle 3.1) wären allein für die Tabelle ein Speicherplatz von etwa 5,6 GiB notwendig, geht man nach der Spezifikation [9] von 4 Byte je Wert aus. Dies beinhaltet lediglich den Speicherplatzbedarf für die Werte selber. Der Speicher, der für die n *Item* Schlüssel benötigt wird und der Overhead, der in Abhängigkeit der Implementierung schwanken kann, ist hier noch nicht einmal berücksichtigt. Bei einem 32-Bit System ist hier bereits die 4 GiB Grenze überschritten.

4.1.2 Smartstore

Fix SmartStore

Für diese Implementierung wird eine alternative Herangehensweise umgesetzt. Die Speicherung der Matrix erfolgt wie in Abbildung 4.2 zu sehen ist. Ausgangpunkt ist die vollständige Abbildung aller n^2 Speicherplätze. Jede Reihe umfasst zunächst einmal theoretisch alle vorhandenen Werte, absteigend sortiert nach ihrer Relevanz. Zusätzlich wird für diesen Speicher eine fixe Grenze λ definiert, sodass nicht alle Werte, sondern nur die jeweils λ besten

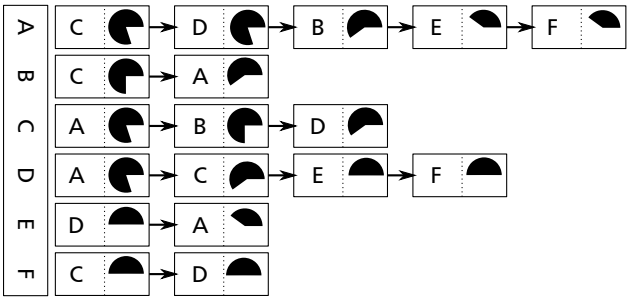


Abbildung 4.3.: Sparse SmartStore

Werte abgelegt werden. Für die oben erwähnten 55.000 Einträge zu je 4 Byte ergäbe das bei einem Schwellwert von $\lambda = 1.000$ Einträgen lediglich einen Speicherbedarf von 0,2 GiB. Der Speicherverbrauch steigt damit linear mit der Anzahl der *Items* an. Ein handelsüblicher Server mit 8 GiB Speicher könnte somit theoretisch etwa 2,2 Millionen Einträge erfassen. Dieses Vorgehen der Datenspeicherung hat zudem einen weiteren Vorteil. Die Daten können leicht in einem Cluster verteilt werden, indem man einfach die vorhandenen Daten in beliebig viele Teilmengen unterteilt. Das Weglassen der kleinen Präferenzen kann jedoch zu Fehlern in der Vorhersage führen. Wie in Abbildung 4.4 zu sehen, ist es beispielsweise möglich, dass Elemente, die eine höhere Bewertung haben als andere, trotzdem aus der Betrachtung fallen. Hier fallen die *Items* H und J aus dem Raster, obwohl die Bewertungen von O, E und X aus der 2. Liste durchgehend schlechter bewertet sind. Für ein Szenario, in dem nur sehr wenige relevante Objekte für den Benutzer interessant sind, könnte diese Fehlerquelle jedoch irrelevant sein. Es ist Teil dieser Diplomarbeit, die Auswirkung eines solchen Speichers auf den Vorhersageprozess zu untersuchen und wird in Abschnitt 5.3 behandelt.

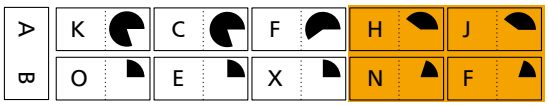


Abbildung 4.4.: Problem Speicher sparen

Sparse SmartStore

Eine alternative Möglichkeit die Daten platzsparend abzulegen, ist es, sich die Eigenschaft der dünn besetzten Daten zu Nutze zu machen. In Abbildung 4.3 wird eine Variante der platzsparenden Speicherung der Daten aufgezeigt. Gespeichert werden jeweils nur alle Werte, die nicht null ergeben. Die Werte ungleich null werden sortiert abgelegt. Das beschleunigt die Verarbeitung der Daten, wenn die Werte mehrerer Empfehlungen kombiniert werden müssen. Für die Speicherung selbst bedeutet dies nur einen geringen Mehraufwand, den man an dieser Stelle übernehmen kann.

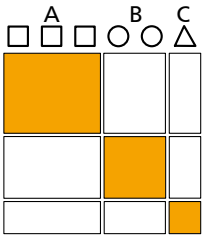


Abbildung 4.5.: Besetzung der Matrix

Seltenheit der Daten

Auch wenn eine sparse Implementierung zum Einsatz kommt, wächst der Speicherbedarf weiterhin mit $O(n^2)$, im günstigsten Fall wächst der Speicherbedarf mit $\Omega(n)$. In dem Fall ist jedoch keine Vorhersage mehr möglich, denn dann wäre jedes Element in einer eigenen Äquivalenzklasse. Deshalb ist es notwendig zu ermitteln, in welchem Maße die Matrix besetzt ist. Abbildung 4.5 zeigt, wie Wertungen von Null zustande kommen, wenn die Jaccard Metrik eingesetzt wird. Die geometrischen Formen repräsentieren die *Items*. Diese sind in ihre Äquivalenzklassen unterteilt. In diesem Fall sind das die Klassen A, B und C. Eine Wertung von Null kommt demnach immer dann zustande, wenn *Items* miteinander verglichen werden, die nicht in einer Äquivalenzklasse sind. Die farblich markierte Matrix darunter zeigt, an welchen Stellen Werte ungleich Null (farblich markiert) zu erwarten sind. Demnach können in der Matrix insgesamt $(|A|+|B|+|C|)^2$ Elemente abgelegt werden. Tatsächlich besetzt sind jedoch nur $|A|^2+|B|^2+|C|^2$ Elemente. Die Anzahl der benötigten Speicherplätze X bei m Äquivalenzklassen ermittelt sich demnach wie folgt:

$$X = \sum_{i=1}^{m} |A_i|^2 \tag{4.1}$$

Zum Abschätzen des tatsächlich belegten Speichers genügt es, lediglich die größte Äquivalenzklasse (MAX) zu betrachten, um eine gute untere und eine brauchbare obere Schranke zu ermitteln, da diese den gesamten Verbrauch dominiert. Diese Eigenschaft leitet sich aus der Verteilung der Äquivalenzklassen ab (Abbildung 3.3b) und der Eigenschaft, dass der Bedarf quadratisch wächst.

$$\begin{aligned} infX &= |MAX|^2 \\ supX &= |MAX|^2 + (n - |MAX|)^2 \end{aligned} \tag{4.2}$$

Für den *med* Datensatz errechnet sich damit eine untere Schranke von 68,8% und eine obere Schranke von 71,7%.

Kombination

Beide Strategien lassen sich auch miteinander kombinieren. Die Kombination ist nach außen hin identisch mit der Fixed SmartStore Variante, kann aber zusätzlich noch ein wenig Speicherplatz sparen, indem Nullwerte nicht abgelegt werden. Die zusätzliche Speichererersparnis ist aber bei weitem nicht mehr so groß wie es beim Sparse SmartStore der Fall ist. Die Performanz beim Füllen des Speichers ist in der einfachen Variante etwas besser, weil der gebrauchte Speicherplatz einmalig am Beginn des Prozesses angelegt werden kann. Die Daten können somit auch blockweise in Arrays abgelegt werden. Die platzsparende Variante muss den Speicher während des Berechnungsprozesses immer erweitern. Das wirkt sich je nach Implementierung auf die Rechenzeit aus. Zudem kann zusätzlicher Speicherbedarf entstehen, wenn auf verlinkte Listen als Implementierung zurückgegriffen wird.

4.2 Rechenkomplexität

Die Komplexität der Berechnungen muss in ihrer Betrachtung zweigeteilt werden. Zum einen wird die Komplexität zum Berechnen der Item-Item Matrix betrachtet. Diese Berechnung kann im Vorfeld getätigt werden. Zum anderen wird die Berechnung der Empfehlung selbst betrachtet, die während der Laufzeit in Echtzeit erfolgen muss. Zur Erinnerung sind in Tabelle 4.1 noch einmal alle relevanten Größen aufgelistet.

Größe	Bedeutung
n	Anzahl der verschiedenen *Items*
k	Anzahl der verschiedenen *User*
$\overline{k}$	Durchschnittliche Anzahl von Wertungen pro *Item*
$\overline{n}$	Durchschnittliche Anzahl von Wertungen pro *User*

Tabelle 4.1.: Komplexität relevante Größen

4.2.1 Offline

Algorithmus Jaccard Distanz

Algorithmus 4.2.1: JACCARD(`User-Item-Matrix`, $result$)

$Items \leftarrow itemVectors($`User-Item-Matrix`$)$
for each $i \in [0, \ldots, length(Items) - 1]$
do { **for each** $j \in [i+1, \ldots, length(Items)]$
do $result[i][j] \leftarrow JaccardDistance(Items[i], Items[j])$

Naive Jaccard: Für die direkte Herangehensweise des `Jaccard` Algorithmus 4.2.1 [1] vergleicht man jedes Item i mit jedem anderen Item j für $i \neq j$. `Jaccard` benötigt dafür bei n *Items* $n \cdot (n-1)/2$ Vergleiche. Ein Vergleich ist gleichbedeutend mit einem Aufruf der `JaccardDistance` Funktion B.3. Die `JaccardDistance` Funktion führt dabei $2 \cdot \overline{k}$ Operationen aus. Eine Operation entspricht dabei zwei Vergleichoperationen und zwei Speicheroperationen. Das resultiert in einer Summe von $\overline{k} \cdot n \cdot (n-1)$ Operationen und ergibt eine obere Schranke von $O(n^2 \cdot \overline{k})$.

Die Komplexität ist demnach quadratisch mit der Anzahl der enthaltenen Items. Das kann zu langen Berechnungszeiten führen. Es ist deshalb erstrebenswert, diese Zeit zu verringern, und wenn dies nicht möglich ist, eine inkrementelle Berechnung zu ermöglichen, sodass die Rechenzeit nicht am Stück vollzogen werden muss, sondern in Etappen abgearbeitet werden kann. Dies ermöglicht dann auch ein inkrementelles Update der Matrix, wenn neue Daten anfallen.

Inkrementelle Updates: Was muss beispielsweise passieren, wenn ein Bestandskunde ein neues Buch kauft. Das Recommender System hält zunächst einmal eine Interne Darstellung der *Item-Item* Matrix vor. Angenommen Kunde drei aus Beispiel Abbildung 2.4 erwirbt das Buch C neu. Damit ändern sich potentiell alle *Item-Item* Werte, die mit dem Buch C in Verbindung stehen. Abbildung 4.6 zeigt die Jaccard Matrix, wie sie nach diesem konkreten Update aussehen muß. Farblich hervorgehoben sind alle Werte, die sich gegenüber der alten Matrix (Abbildunge 2.4c) geändert haben. Für jede Bewertung, die jetzt hinzugefügt wird, ist es notwendig n Vergleiche mit jeweils $2 \cdot \overline{k}$ Operationen durchzuführen. Wollte man auf diese Weise die gesamte Matrix berechnen, also zunächst eine Matrix mit nur einer Wertung, und diese schrittweise aktualisiert, bis alle Bewertungen berücksichtigt sind, dann wären exakt $2\overline{k} \cdot \sum_{i=1}^{n-1} i = \overline{k}n^2 - \overline{k}n$ Operationen notwendig, das entspricht den $\overline{k} \cdot n \cdot (n-1)$ Operationen.

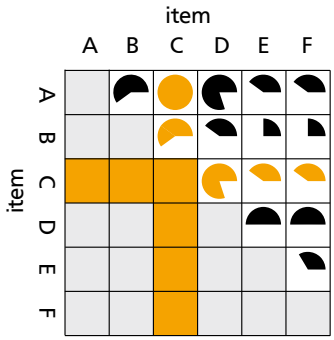


Abbildung 4.6.: Jaccard Update

Eine komplette inkrementelle Abarbeitung ist damit sehr aufwendig. In der Praxis wird es so aussehen, dass die inkrementellen Aktualisierungen blockweise durchgeführt werden. Das heißt, es werden zunächst neue Daten gesammelt ehe sie am Block abgearbeitet werden. So

[1] itemVectors: extrahiert aus der User-Item-Matrix die Spaltenvektoren

kann ein Blocks der größe b mit der Anzahl von $2\overline{k} \cdot \left[\sum_{i=1}^{n-1} i - \sum_{j=1}^{(n-1)-b} j\right] = 2\,\overline{k}\,\cdot\, b\,n + \left(-b^2\,-\,b\right)\cdot\,\overline{k}$ Operationen auskommen.

Algorithmus 4.2.2: INCREMENTALJACCARD(User-Item-Matrix, $result$)

$numerator \leftarrow 0$
$denominator \leftarrow 1$
for each $u \in userVectors$(User-Item-Matrix)
do {
  for each $p \in newItemsIndex(u)$
  do {
    for each $i \in index(itemVectors$(User-Item-Matrix))
    do {
      if incNumerator(p,i)
      $\{result[p][i][numerator] \leftarrow result[p][i]$[numerator]$+1$
      if incDenominator(p,i)
      $\{result[p][i][denominator] \leftarrow result[p][i][denominator]+1$

Incremental Jaccard: Bei einem System, welches stets aktuelle Daten fordert, ist ein klassischer Ansatz nicht praktikabel. Deshalb ist es wünschenswert, eine inkrementelle Berechnung zu verwenden, ähnlich wie es Papagelis et al. [17] für *User* basierte Recommender untersucht haben.

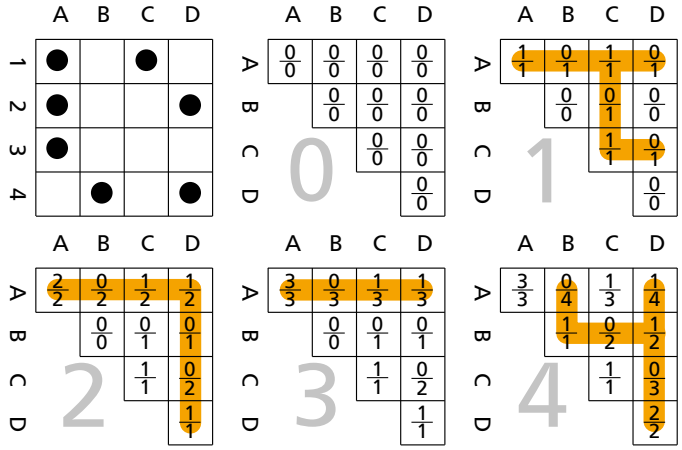


Abbildung 4.7.: Beispiel IncrementalJaccard

Der IncrementalJaccard (Algorithmus 4.2.2) geht dieses Problem an und findet eine bessere Lösung für eine rein inkrementelle Berechnung der *Item-Item* Matrix. Das Verfahren arbeitet dabei die Eingabedaten *User*-weise ab. Die Grundidee ist es, Zähler und Nenner im Quotient der Jaccard Werte getrennt voneinander zu betrachten.

$$\begin{aligned} \text{Zähler(x,y)} &= \begin{cases} +1 & \text{wenn } \texttt{is}(x) \cdot \texttt{new}(y) \vee \texttt{new}(x) \cdot \texttt{is}(y) \\ +0 & \text{sonst} \end{cases} \\ \text{Nenner(x,y)} &= \begin{cases} +1 & \text{wenn } \neg\texttt{is}(x) \cdot \texttt{new}(y) \vee \texttt{new}(x) \cdot \texttt{new}(y) \vee \texttt{new}(x) \cdot \neg\texttt{is}(y) \\ +0 & \text{sonst} \end{cases} \end{aligned} \tag{4.3}$$

An Abbildung 4.7 kann die Funktionsweise nachvollzogen werden. Gegeben ist eine *User-Item* Matrix, die zeilenweise abgearbeitet wird. In dieser sind alte Wertungen mit einem grauen Punkt markiert. Neue Wertung sind mit einem schwarzen Punkt markiert und gelöschte Wertungen sind mit einem schwarzem Kreis. Siehe auch Abbildung 4.8. In Schritt `0` wird die zu berechnende *Item-Item* Matrix mit $\left(\frac{0}{0}\right)$ initialisiert. Der besseren Übersicht wegen ist in der Zielmatrix die Diagonale noch enthalten. Diese Werte werden später nicht benötigt, da sie nach komplettem Durchlauf immer eins ergeben. Im ersten Schritt werden die beiden *Bücher* A und C, die der erste *Kunde* erworben hat, in der Zielmatrix aktualisiert. Dabei wird nach den Rechenvorschriften aus den Gleichungen 4.3[2] vorgegangen. In Rechenschritt 1 werden demnach nur die Zähler und Nenner der farblich markierten Felder, das sind diejenigen deren Koordinaten eine der neuen Bücher A oder C enthalten, entsprechend der Vorschrift um eins oder null erhöht. Diese Prozedur wird bis zum vierten und letzten Kunden wiederholt. Wie zu erwarten, ergeben alle Werte auf der Diagonalen den Wert eins. Alle anderen beziffern die jeweilige Jaccard Distanz.

Dem Algorithmus 4.2.2 können die Rechenschritte entnommen werden. Für die Berechnung müssen alle *User* durchlaufen werden. Für jeden *User* werden alle bewerteten *Items* betrachtet, welche mit allen anderen verglichen werden müssen. Das entspricht zwei Vergleichs- und zwei Schreiboperationen. Demnach sind $k \cdot n \cdot \overline{n}$ Operation für diesen Ansatz notwendig. Dieser rein inkrementelle Ansatz ist im Vergleich zu der inkrementellen Variante des vorherigen Algorithmus besser wenn Gleichung 4.4 gilt.

$$\frac{k\overline{n}}{\overline{k}n - \overline{k}} < 1 \tag{4.4}$$

`IncrementalJaccard` **Updates:** Auch bei `IncrementalJaccard` ist es möglich, nachträglich eingefügte Wertungen zu übernehmen. In Abbildung 4.8 wird zunächst die Wertung von *User* 2 zu *Item* D entnommen. Die Vorgehensweise ändert sich nicht. Die Rechenvorschrift wird durch die Gleichungen 4.5 ersetzt. Das resultiert in der *Item-Item* Matrix ohne die Wertung (2,D). Auch hier enstehen nur Änderungen für alle Kombinationen in denen D vorkommt.

In Abbildung 4.9 wird die soeben entnommene Wertung wieder hinzugefügt und nach Gleichung 4.3 berechnet. Als Ergebnis bekommt man wieder die Ausgangsmatrix. Für jedes

[2] Herleitung in Anhang B.2

neue Element fallen für ein Update $(n-1)$ Operationen an. Das entspricht einem linearen Wachstum und es ist denkbar, diese Änderungen zeitnah in das Modell einfliessen zu lassen.

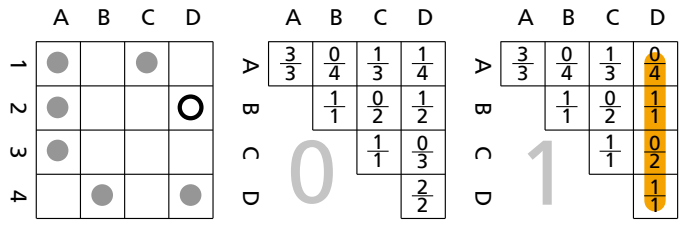


Abbildung 4.8.: Elemente entfernen

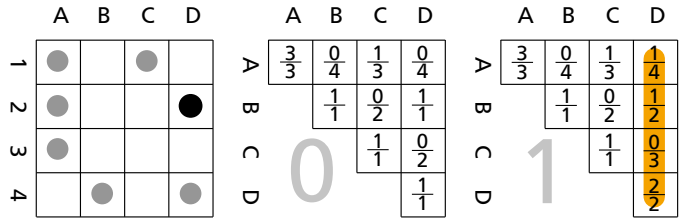


Abbildung 4.9.: Elemente hinzufügen

$$\begin{aligned}
\text{Zähler(x,y)} &= \begin{cases} -1 & \text{wenn } \texttt{is}(x)\cdot\texttt{del}(y) \vee \texttt{del}(x)\cdot\texttt{del}(y) \vee \texttt{del}(x)\cdot\texttt{is}(y) \\ +0 & \text{sonst} \end{cases} \\
\text{Nenner(x,y)} &= \begin{cases} -1 & \text{wenn } \neg\texttt{is}(x)\cdot\texttt{del}(y) \vee \texttt{del}(x)\cdot\neg\texttt{is}(y) \\ +0 & \text{sonst} \end{cases}
\end{aligned} \tag{4.5}$$

Zusammenarbeit mit Speichertechniken: Der Naive Jaccard lässt sich ohne weiteres mit den vorgestellen Speichertechniken einsetzen. Dabei sind nur die bereits erwähnten Nachteile der Speicher zu berücksichtigen. Im Fall des *Fix* SmartStore ist zusätzlich darauf zu achten, dass beim Entfernen von Wertungen Fehler entstehen können, weil schlechter bewertete *Items* unter Umständen nicht nachrücken können. Da das Entfernen von Wertung jedoch eine Ausnahme ist, wird dieser Fall an dieser Stelle nicht näher betrachtet.

Mit dem `IncrementalJaccard` verhält es sich nicht so. Aufgrund seiner getrennten Betrachtung von Zähler und Nenner können die vorgestellten Speichertechniken nicht eingesetzt werden. Zum einen ist die Nenner-Komponente immer voll besetzt. Es fallen also zwingend $n\cdot(n-1)/2$ Speicherplätze an. Damit ist ein Einsatz von *Sparse* SmartStore bereits ausgeschlossen. Zudem benötigt diese Vorgehensweise einen effizienten Zugriff auf die Datenfelder. Im optimalen Fall eine Zugriffszeit von $O(1)$. Für die Umsetzung kommt deshalb ein Treppenspeicher (Anhang B.1) in Frage.

Aufgrund des höheren Speicherbedarfs und des, in seiner Größe fixen, Treppenspeichers, wird dieser Algorithmus in den weiteren Betrachtungen nicht weiter berücksichtigt.

4.2.2 Online

Zur Laufzeit werden zu *User* $u \in \mathscr{U}$ die κ ähnlichsten *Items* ermittelt, wie es Karypis [11] vorschlägt. Während der offline Phase wurden für jedes *Item* $j \in \mathscr{I}$ die Menge $K_j = \{j_1, j_2, \cdots, j_\kappa\}$ der κ ähnlichten *Items* berechnet. Ein aktiver *User* u wird charakterisiert durch die Menge der bewerteten *Items* B. Zunächst einmal wird eine Menge an Kandidaten erstellt (C^+) und anschließend alle Elemente entfernt, die bereits von u bewertet wurden. (4.6)

$$\begin{aligned} C^+ &= \bigcup_{j \in B} K_j \\ C &= C^+ \setminus B \end{aligned} \tag{4.6}$$

Für jedes Element $c \in C$ wird im Anschluss eine Ähnlichkeit als Summe der Distanzen zwischen c und allen $j \in B$ berechnet. Die κ ersten Elemente von C, absteigend sortiert nach Ähnlichkeit, sind das Ergebnis des Problems.

Die Komplexität dieser Berechnung steigt mit $O(k|B|)$. Das Wachsen der Komplexität mit steigendem κ stellt dabei kein Problem dar. In diesem Kontext wird der Wert nicht signifikant größer werden als zehn. Problem bei der Skalierung kann aber die Größe $|B|$ werden. Je weiter das System wächst, um so mehr Daten werden pro *User* gesammelt werden. Für die vorliegenden Daten (Tabelle 3.1) sind hier jedoch keine Probleme zu erwarten.

5 Auswertung

In den folgenden Abschnitten werden die drei kritischen Größen Speicherverbrauch, Rechenzeit und Vorhersagequalität getrennt voneinander betrachtet. Dabei werden die Ergebnisse vom *Sparse* und *Fix* Speicher gegenüber gestellt. Alle Tests wurden entweder auf einem AMD 5200+ Dual core 64 Bit mit 8GB Speicher oder einem Intel(R) Pentium(R) Dual Core E2140 mit 2GB Speicher durchgeführt. Beide Spezifikationen überschreiten nicht das erforderte Limit und können daher als Referenz herangezogen werden. Bei allen Messungen ist zu beachten, dass der Einsatz eines Profilers oder die zusätzliche Ausgaben in Dateien zusätzlich Ressourcen in Anspruch nehmen können.

5.1 Speicher

So wird gemessen

In Abschnitt 4.1.2 wird der Speicherbedarf der alternativen Speichertechniken Fixed SmartStore und Sparse SmartStore betrachtet. Diese theoretische Vorgehensweise beschränkt sich jedoch auf Überlegungen, eine untere Schranke für den Speicherbedarf zu ermitteln. Auf Implementierungsdetails und Eigenschaften der Entwicklungsumgebung wird dort nicht eingegangen. Eine Abschätzung des Speicherverbrauchs gelingt also nur in einer realen Testumgebung. Der Test läuft folglich in der Umgebung, in der die Anwendung auch im Live-Betrieb arbeiten wird. Sie ist als *JavaEE* Enterprise Application geschrieben und läuft auf einem *Glassfish* Application Server (V3.1.1). Für die Auswertung des Speichers kommt der Netbeans Profiler [4] zum Einsatz. Dieser hängt sich in die aktive Glassfish Domäne ein und liest in regelmäßigen Abständen die aktuelle Heap Größe und den tatsächlichen Speicherbedarf aus. Zusätzlich zum Speicherverbrauch schreibt der Application Server im tausend Bücher Takt eine Nachricht in die Logfile, um den Fortschritt über die Zeit zu dokumentieren. Die beiden Informationsquellen zusammengenommen protokollieren den Speicherbedarf in Abhängigkeit der verarbeiteten und gespeicherten Bücher. Die Auswertungen beschränken sich dabei auf den Zeitraum in dem die *Item-Item* Matrix befüllt wird. Als Testdatensatz dient ein erweiterter *allg* Datensatz mit fast 62.000 Büchern. Damit ist gewährleistet, dass auch der größte Datensatz bearbeitet werden kann.

Die Graphen

Die beiden Abbildungen 5.1 und 5.2 veranschaulichen die Verhaltensweisen der beiden Speichervarianten *Fix Smartstore* und *Sparse Smartstore*, im Folgenden als *Fix* und *Sparse* bezeichnet. Die vom Profiler protokollierten Heapgrößen sind als graue Punktwolke über die Zeit abgetragen. Der Zusammenhang über Anzahl gespeicherter Bücher und Speicher-

verbrauch wird über eine nicht lineare parametrische Regression hergestellt. Als Regressionsfunktion wird eine Sättigungsfunktion (Gleichung 5.1) angenommen. Vertikale Linien markieren die aus dem Logfile entnommenen Fortschritte in tausender Schritten.

$$g(t) = \Theta_1 + \Theta_2 \cdot e^{-\Theta_3 \cdot t} \tag{5.1}$$

Fix SmartStore

	Wert	σ
Θ_1	505	1,008
Θ_2	-206,8	2,644
Θ_3	4,124 E-04	1,075 E-05
$\sum \sigma^2$	56035796	
Toleranz	4,279 E-08	

Tabelle 5.1.: *Fix* Parameter

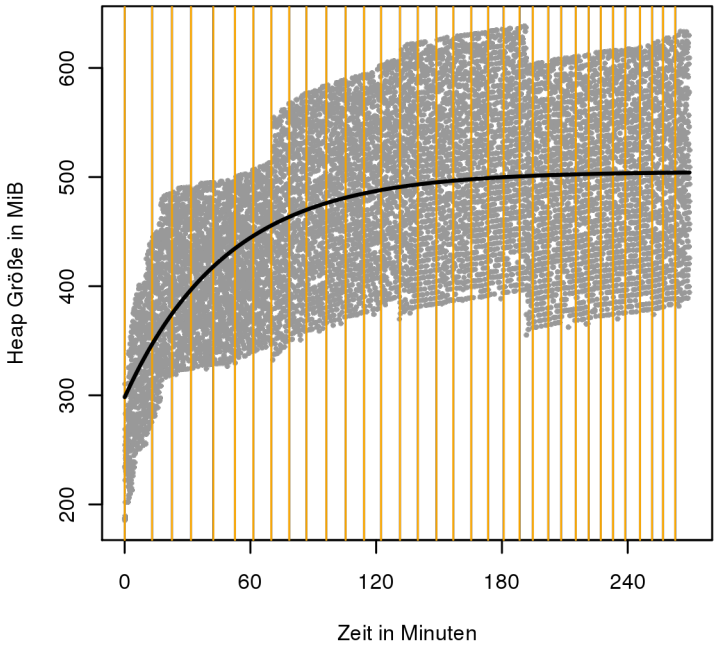


Abbildung 5.1.: Speicherverbrauch Fix $\lambda = 300$ - *allg* Daten

Der zeitliche Speicherverbrauch der *Fix* Variante (Abbildung 5.1) mit einem Limit von $\lambda = 300$ Einheiten, zeigt einen moderaten Anstieg der Daten über die Zeit. In dem beobachteten Fenster von 33.000 gespeicherten Büchern steigt der maximale Speicherverbrauch nicht über 640 `MiB`.

Für die Abschätzung des durchschnittlichen Speicherbedarfs bei gefülltem Modell lässt sich der Grenzwert der ermittelten Regressionsfunktion 5.1, definiert durch Tabelle 5.1 , errechnen. Tabelle 5.1 listet die, mit Hilfe der Methode der kleinsten Quadrate errechneten, Parameter der Regressionsfunktion auf. Explizit angegeben sind die Parameter Θ_{1-3} und deren Varianz. Zudem wird der Vollständigkeit halber die Summe der gesamten Residuen und die Toleranz des konvergierenden Verfahrens angegeben. So ergibt sich eine Abschätzung von:

$$\lim_{t\to\infty} g_{fix}(t) = \lim_{t\to\infty}(-206{,}8\ \texttt{MiB} \cdot e^{-4{,}124E-04t} + 505\ \texttt{MiB}) = 505\ \texttt{MiB} \tag{5.2}$$

Diese deckt sich mit den tatsächlich gemessenen 509 `MiB`.

Der Verlauf als Sättigungskurve lässt sich durch zwei Eigenschaften erklären. Zum einen wird für die Berechnung der Item-Item Matrix jedes Item mit jedem anderen Item verglichen. Das führt dazu, dass mit zunehmendem Verlauf immer weniger Vergleiche für ein Item durchgeführt werden. Das erzeugt weniger Hits und damit weniger Daten, die im Speicher abgelegt werden. Auch die limitierte Größe des Speichers führt dazu, dass der Gesamtverbrauch immer langsamer ansteigt, da mit Füllen des Speichers mit der Zeit immer seltener Einträge getroffen werden, die mit weniger als $\lambda = 300$ Daten gefüllt sind.

Bei genauer Betrachtung der tausender Marken fällt auf, dass die Abstände im Verlauf immer geringer werden. Dies macht sich in der Rechenzeit bemerkbar. Mehr dazu in Abschnitt 5.2

Weiter fällt auf, dass die Varianz des Speicherverbrauchs zunehmend ansteigt. Diese Eigenschaft lässt sich durch den vorliegenden Algorithmus nicht erklären und ist wahrscheinlich auf das Speichermanagement der eingesetzten `JVM` zurückzuführen und soll hier nicht weiter untersucht werden.

Sparse SmartStore

	Wert	σ
Θ_1	3.897	25,61
Θ_2	-3.816	23,62
Θ_3	-2,620E-05	2,816E-07
$\sum\sigma^2$	1,347 E+09	
Toleranz	1,727 E-06	

Tabelle 5.2.: Sparse SmartStore Parameter

Im Fall der *Sparse* Variante (beschrieben durch Abbildung 5.2) wurde der Speicherverbrauch für die ersten 15.000 Bücher protokolliert und musste abgebrochen werden, weil das Speicherlimit des Testsystems erreicht wurde. Der Anstieg ist zunächst hoch und flacht nur langsam ab. Daraus resultiert ein hoher Speicherbedarf. Bei diesem Test wurden zeitweise bis zu 2.960 `MiB` reserviert.

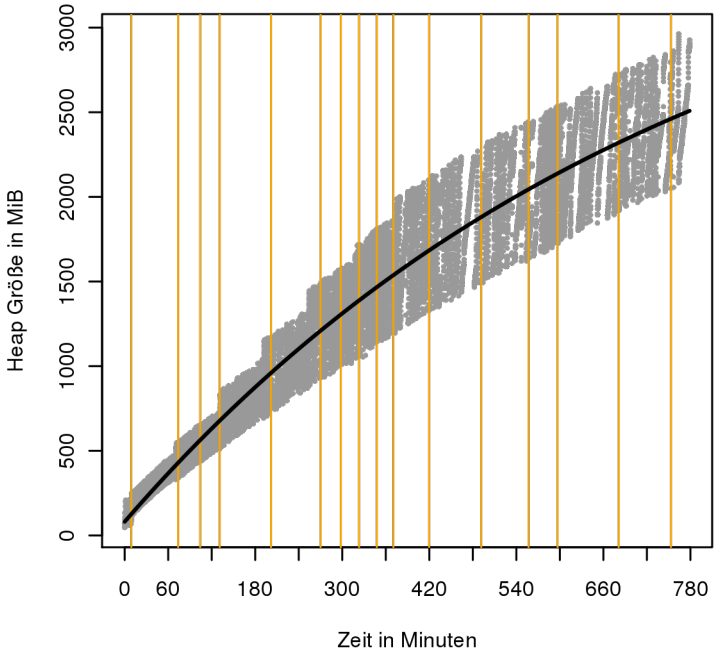


Abbildung 5.2.: Speicherverbrauch Sparse SmartStore - *allg* Daten

Eine vollständige Analyse des Speicherverbrauchs war in diesem Fall nicht möglich, da das vorliegende Testsystem über nicht mehr als 3GiB Speicher für die `JVM` zur Verfügung stellen kann. Da der reservierte Speicher der `JVM` am Stück reserviert werden muss, hängt die maximale Speichergröße nicht nur von dem verfügbaren Speicher ab, sondern auch von dem Betriebsystem. Oft kann nur ein Bruchteil des physikalischen Speichers der `JVM` zur Verfügung gestellt werden. Ein umfassende Analyse, wie viel Speicher letztendlich reserviert werden kann, würde an dieser Stelle zu weit führen, als Daumenregel kann man jedoch die Hälfte des Verfügbaren Speichers annehmen. Mit Hilfe der Regressionsfunktion 5.1, beschrieben durch die Parameter in Tabelle 5.2, lässt sich der gesamte Bedarf abschätzen durch:

$$\lim_{t\to\infty} g_{sparse}(t) = \lim_{t\to\infty}(-3.816\ \texttt{MiB}\cdot e^{2.620E-05t} + 3.897\ \texttt{MiB}) = 3.897\ \texttt{MiB} \tag{5.3}$$

Die Abstände der tausender Marken scheinen hier keinem Muster zu folgen, insbesondere werden die Abstände mit der Zeit nicht geringer.

Auch hier steigt die Varianz in den Daten mit zunehmender Zeit und ist auch auf die `JVM` zurückzuführen.

Fazit Speicher

Der Speicherbedarf von *Sparse* überschreitet mit 3.897 `MiB` den von *Fix* und ist damit fast acht mal größer. Kritisch ist diese Zahl vor allem für 32 Bit Systeme, die am Stück nur 4GiB Blöcke adressieren können. Damit ist der Wert nahe an einem Limit, welches die Machbarkeit gefährdet, da zusätzliche Items schnell zum Überschreiten des physikalisch adressierbaren Adressraums führen können. Auf Bewertungsbasis des Speicherverbrauchs ist *Fix* SmartStore dem Sparse SmartStore also überlegen.

5.2 Zeit

Für den Betrieb eines Recommenders ist vorwiegend die Zeit relevant, die für einen einzelnen Empfehlungsvorgang benötigt wird. Nur so ist es möglich, eine große Anzahl von Anfragen in kurzer Zeit zu bewältigen. Um eine Echtzeit-Empfehlung zu garantieren, müssen Teile der Berechnungen ausgelagert und im Vorfeld also offline berechnet werden. Diese offline Berechnungen müssen mit zusätzlichem Speicherbedarf und offline Rechenzeit bezahlt werden. Ein Verfahren wie dieses kann also Rechenzeit auslagern. Das kann jedoch nicht in beliebiger Größenordnung geschehen. Ein Algorithmus, der in $O(1)$ beliebig große Probleme lösen kann, dafür aber unendlich lange Vorberechnungen durchführen muss, ist in der Praxis nicht anwendbar.

Offline

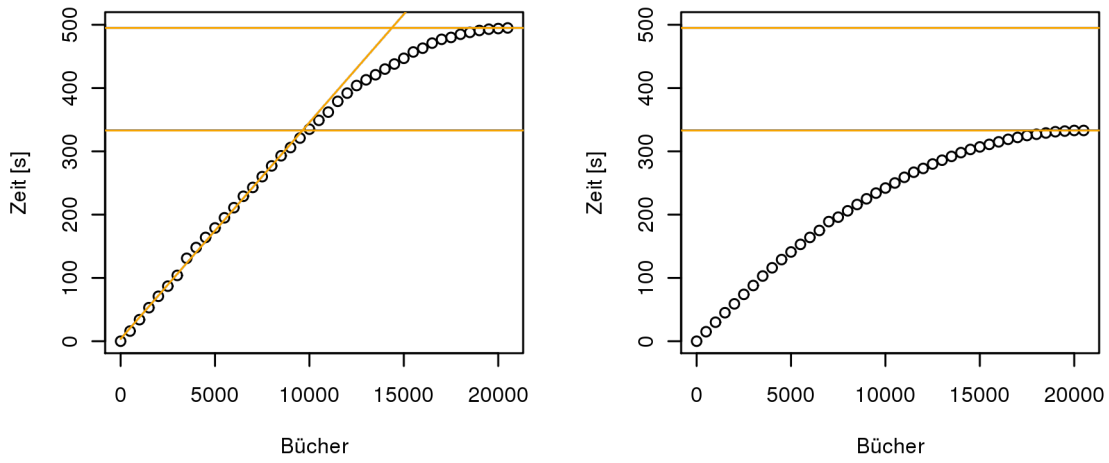
So wird gemessen: Die Messung des Rechenzeitbedarfs erfolgt über die bereits erwähnten Serverlogs. Um das Laufzeitverhalten abschätzen zu können, wird betrachtet, wie sich die Rechenzeit für fixe Intervalle von Büchern verhält. Hinweise auf das Verhalten sind bereits in den Abbildungen 5.2 und 5.1 erkennbar. Die Abstände der vertikalen Markierungen lassen darauf schließen, dass im Fall des Sparse Smartstore die Rechenintervalle konstant bleiben, wohingegen im Fall von Fix Smartstore die Intervalle immer geringer werden. Dies resultiert in einer insgesamt geringeren Rechenzeit für Fix gegenüber von Smart bei gleichen Startintervallen.

Da der *Sparse* Test über die Gesamtdaten nicht bis zum Ende laufen konnte, weil der Speicher auf dem Testsystem nicht ausreichte, wurde das Experiment mit dem *db* Datensatz wiederholt. Die neuen Messwerte sind in Abbildung 5.3 abgetragen. Dieser ist kleiner und kann daher in beiden Fällen bis zum Ende durchlaufen. Das Messinterval wurde von 1000 auf 500 Bücher verkürzt, um mehr Messdaten zu erhalten.

Ergebnis: Im Fall *Fix* Smartstore (Abbildung 5.3b) benötigt das Programm 333 Sekunden, um die komplette Item-Item Matrix zu berechnen. Für die ersten 500 Bücher werden 15 Sekunden benötigt, die Rechenzeit für alle weiteren Intervalle nimmt kontinuierlich ab. Dieses Verhalten war zu erwarten, schließlich nehmen die Vergleiche, die für jedes weitere Intervall benötigt werden, kontinuierlich ab.

Im Fall *Sparse* (Abbildung 5.3a) benötigt das Programm 495 Sekunden, um die komplette Item-Item Matrix zu berechnen. Wie zu erwarten, nimmt irgendwann die Zeit, die für ein fixes Intervall von 500 Büchern benötigt wird, ab. Dieser Effekt stellt sich jedoch erst spät ein. Es scheint so, als würde für die erste Hälfte der Rechnungen die Zeit linear ansteigen. Dies ist angedeutet durch die Ausgleichsgerade, die aus den ersten 21 von 42 Datenpunkten abgeleitet ist. Dieses Verhalten war auch für den *allg* Datensatz zu beobachten, der nicht bis zum Ende laufen konnte. Das Laufzeitverhalten der beiden Datensätze ist demnach ähnlich und lässt sich Datensatz übergreifend vergleichen. Erst in der zweiten Hälfte der Eingabedaten macht sich ein Abflachen der Kurve bemerkbar. Im Vergleich zu der Fix Smartstore Variante äußert sich das in einer höheren Gesamtlaufzeit, da die Anfangssteigung in beiden Fällen identisch ist.

Die Ursache für ein solches Verhalten kann darin begründet sein, dass die Elemente sortiert im Speicher abgelegt werden. Zu Beginn werden viele neue Elemente im Speicher abgelegt. Die zusätzliche Zeit, die benötigt wird, Elemente in eine sortierte Liste einzufügen, wirkt dem Effekt der weniger werdenden Vergleiche entgegen. Erst gegen Ende, wenn kaum noch neue Einträge eingefügt werden, wird die Rechenzeit merklich geringer. Dies bedeutet, dass die Rechenzeit für die Fix Smartstore Variante zwingend besser ist, als für die Sparse Smartstore Variante. In diesem Fall ergibt sich eine Ersparnis von fast 33 Prozent.



(a) Sparse SmartStore **(b)** Fix SmartStore

Abbildung 5.3.: Rechenzeit über Bücher - *db* Datensatz

Online

So wird gemessen: Das Testen der online Rechenzeit geschieht über einen Stresstest auf den ausführenden Webservice. Für die Umsetzung des Tests wird auf die Software JMeter™ [2] zurückgegriffen.

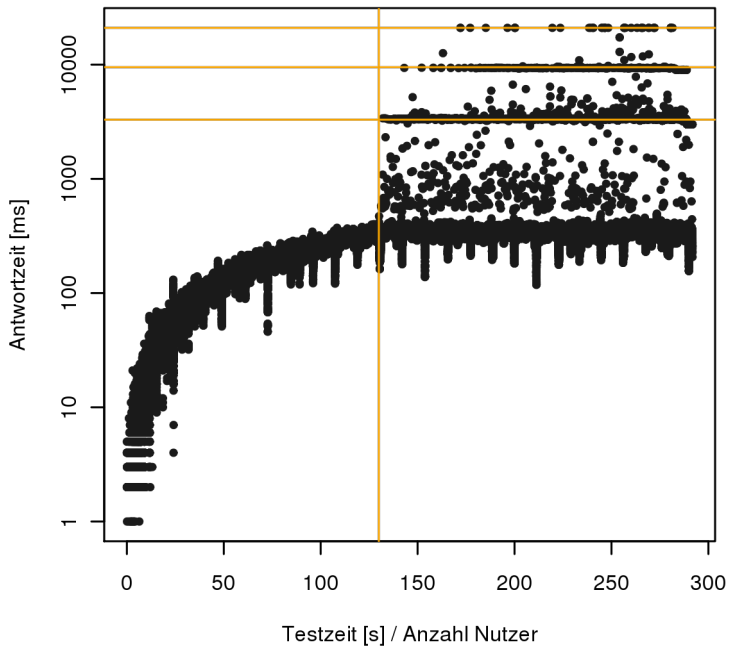


Abbildung 5.4.: Stresstest Fix $\lambda = 300$ - *db* Daten

Getestet wird ein hohes Anfrageaufkommen an den Webservice. Dabei schicken bis zu 300 simulierte Nutzer gleichzeitig Anfragen an den Webservice ab. Der Test ist dabei so aufgebaut, dass zunächst nur ein Nutzer Anfragen stellt. Nach jeder verstrichenen Sekunde kommt ein weiterer Nutzer hinzu, bis nach fünf Minuten eine Gesamtzahl von 300 simulierten Nutzern erreicht wird. Jeder aktive Nutzer stellt alle 6 ms eine zufällige Anfrage an das System. Dabei sind die 6 ms nicht fest, sondern normalverteilt mit einer Standardabweichung von 2. Damit wird verhindert, dass Muster in den Abfragen entstehen. Am Ende des Tests werden folglich bis zu 49.800 Anfragen in der Sekunde an das System geschickt. Gemessen wird die Antwortzeit die für jede Anfrage gebraucht wird. Ein Anfrage ist eine Empfehlungsanfrage für ein einzelnes Buch.

Ergebnis: Der Ausgang des Tests ist in Abbildung 5.4 dargestellt. Auf der x-Achse abgetragen ist die verstrichene Testzeit in Sekunden. Das ist äquivalent zu der Anzahl simulierter Nutzer. Auf der logarithmisch skalierten y-Achse sind die benötigten Antwortzeiten in Millisekunden abgetragen. Zu Beginn des Test liegen die Antwortzeiten noch bei einer bis fünf Millisekunden. Das bedeutet, dass eine einzige Empfehlung, inklusive Anfrage und Antwort über *SOAP* in einem einstelligen Millisekunden Bereich abgearbeitet werden kann. Die Antwortzeiten wachsen mit zunehmenden Anfragen zunächst linear an. Ab etwa 130 parallelen Nutzern, das entspricht 21.800 Anfragen in der Sekunde, treten zunehmend Messwerte auf, die sich durch signifikant längere Laufzeiten auszeichnen. Auffällig sind die Häufungen bei 3.300, 9.500 und 21.000 ms. Dieses Verhalten ist durch die Load Balancing Strategie des

Quantil	5%	10%	15%	20%	25%
Antwort[ms]	39	83	125	163	201
Quantil	30%	35%	40%	45%	50%
Antwort[ms]	233	258	288	310	321
Quantil	55%	60%	65%	70%	75%
Antwort[ms]	328	335	343	349	356
Quantil	80%	85%	90%	95%	100%
Antwort[ms]	363	371	380	397	21005

Tabelle 5.3.: Quantile Stresstest Fix $\lambda = 300$ - *db* Daten

Application Servers zu erklären. Offensichtlich werden die Anfragen nach Prioritätswarteschlangen abgearbeitet.

Der Graph alleine erweckt den Eindruck, als wären die Antwortzeiten ab 130 Nutzern für einen Realbetrieb nicht mehr brauchbar. In Tabelle 5.3, die alle Quantile in 5% Intervallen auflistet über die gesamte Messreihe, sieht man jedoch, dass diese Abweichler in der Minderheit sind. 95% aller Anfragen wurden in weniger als 397 ms abgearbeitet und das bei bis zu 49.800 Anfragen in der Sekunde. Teilt man die Messdaten in mit und ohne Load Balancing auf, dann können ohne Load Balancing 95% der Anfragen unter 337 ms abgearbeitet werden und 90% der Anfragen in 394 ms oder weniger, im Load Balancing Bereich. Die volle Auflistung findet sich in Anhang A.2.

Quantil	5%	10%	15%	20%	25%
Abweichung	87,1%	50,6%	32%	20,3%	10,4%
Quantil	30%	35%	40%	45%	50%
Abweichung	7,7%	8,5%	6,6%	3,9%	3,4%
Quantil	55%	60%	65%	70%	75%
Abweichung	3,3%	3%	2%	2%	2%
Quantil	80%	85%	90%	95%	100%
Abweichung	1,7%	1,6%	1,8%	5,0%	0,2%

Tabelle 5.4.: Abweichung Quantile Sparse gegenüber Fix $\lambda = 300$

Die vorliegenden Daten zum Stresstest sind alle mit *Fix(300)* auf dem *db* Datensatz getestet worden. Der gleiche Test mit *Sparse* ergibt sehr ähnliche Ergebnisse. Er weist die gleichen charakteristischen Merkmale auf. Auch hier setzt ab 130 parallelen Nutzern das Load Balancing ein und die charakteristischen Marken bei 3.300, 9.500 und 21.000 zeigen sich ebenfalls. Tabelle 5.4 zeigt, um wie viel die *Sparse* Quantile gegenüber den Ergebnissen aus Tabelle 5.3 abweichen. Insgesamt benötigt die *Sparse* Variante im Schnitt 12,7% länger, als die *Fix* Variante. Der Unterschied ist sehr gering und beide Varianten kommen zur gleichen Zeit in den kritischen Bereich in dem der Application Server sein Load Balancing aktivieren muss. Aus diesem Grund lassen sich beide Varianten als gleichwertig betrachten.

Fazit Zeit

Ausgehend von der offline Rechenzeit, von etwa 5 1/2 Stunden für die *Fix* Variante, kann der Aufwand als machbar bezeichnet werden. Dieser Zeitrahmen ist durchaus dafür geeignet, um in zeitgesteuerten Prozessen über Nacht die benötigten Vorberechnungen zu erledigen. Zudem ist es auch möglich, das Vorverarbeiten der Daten inkrementell und intervallweise zu gestalten. So ist es beispielsweise denkbar, jede Nacht eine vorher festgelegte Anzahl von neuen Daten dem System zuzufügen. Das System wäre dann zwar zeitweise nicht auf dem aktuellsten Stand bis alle Daten verarbeitet wurden, wäre dafür verfügbar.

Die Ergebnisse aus dem Stresstest ergeben eindeutig, dass diese Implementierung geeignet ist auch eine große Anzahl von gleichzeitigen Anfragen zu bewältigen. Der Test ist absichtlich so gestaltet, dass eine sehr hohe Anfragefrequenz von bis zu fast 50.000 Hz geprüft wird. Selbst bei höchster Last schafft es das System noch, die Anfragen in weniger als einer halben Sekunde zu verarbeiten. Nur einige wenige Ausreißer erreichen die Timeout Grenze des Application Servers. In diesem Szenario ist allerdings eine wesentlich geringere Abfragefrequenz zu erwarten, sodass der kritische Bereich wohl im Normalbetrieb nicht erreicht wird.

5.3 Recall & Precision

Die Evaluation der Qualität wird auf die Betrachtung von *Recall* und *Precision* reduziert, da es primär darum geht, zu untersuchen, ob ein besserer Speicherverbrauch zulasten der Qualität geht. Eine umfassende Analyse, wie in Abschnitt 2.2.1 erörtert, ist dafür nicht notwendig.

So wird getestet

Das Testen von Recall und Precision findet aus Laufzeit Gründen nicht auf dem *alg* Datensatz, sondern auf dem *med* Datensatz statt. Für diesen Zweck werden die Eingangsdaten zufällig aufgeteilt in Trainings- und Testdaten im Verhältnis 9:1. Für jeden *User* im Testdatensatz wird eine Anfrage an den Recommender gestellt, der die besten 30 Empfehlungen zurückliefert. Die Beschränkung auf nur 30 Elemente hat den Grund, dass bei einer Empfehlung die weniger relevanten *Items* nicht betrachtet werden. In der Einsatzumgebung werden nur die ersten fünf bis sieben besten Treffer angezeigt und nach Miller[15] ist nicht davon auszugehen, dass wesentlich mehr wahrgenommen werden. Für jede Anfrage wird danach Recall und Precision für die Ränge eins bis dreißig ermittelt und anschließend über Macro Averaging gemittelt.

Insbesondere interessiert hier, wie sich der Fix SmartStore mit unterschiedlichen Limits $\{1, 5, 20, 50\}$ auf Precision und Recall im Vergleich zu Sparse SmartStore verhält, da dieses verlustbehaftet ist.

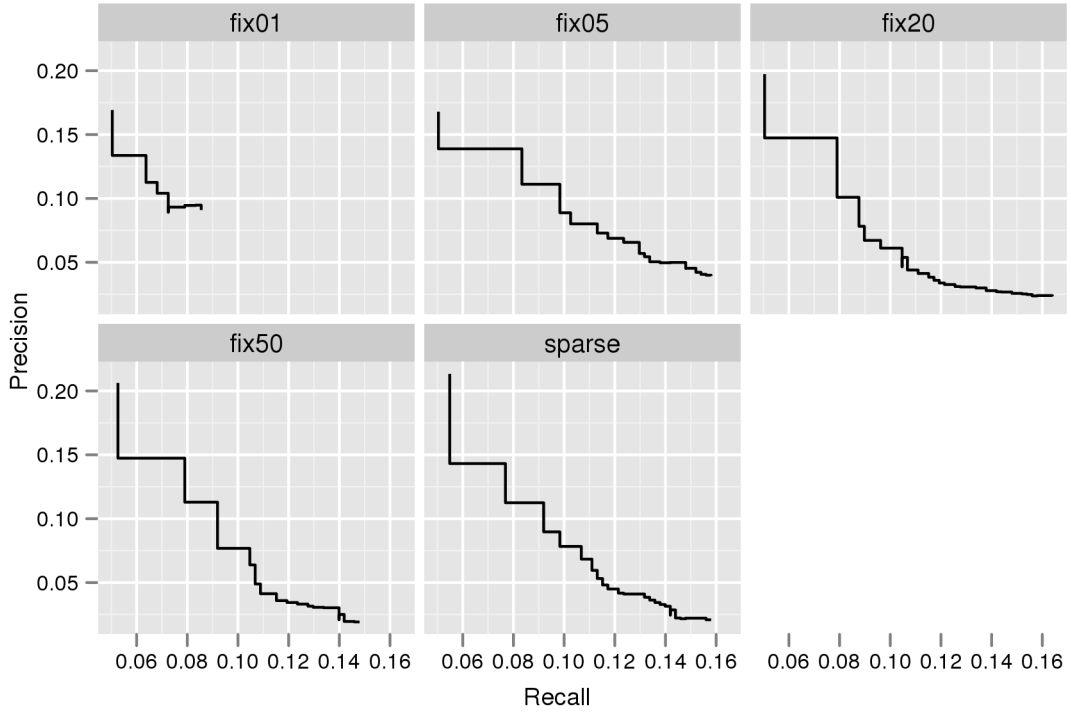


Abbildung 5.5.: Recall Precision SmartStore-

Ergebnis

In Abbildung 5.5 sind alle Ergebnisse abgetragen. Die erreichte Precision bewegt sich dabei zwischen null und 22 Prozent. Der Recall erreicht Werte bis zu 16 Prozent. Beide Wertebereiche sind durchaus aussagekräftig, zumal es sich um Durchschnittswerte handelt. Dass der Recall nicht signifikant höher wird, ist auch der Tatsache geschuldet, dass im Test nur die ersten dreißig Elemente betrachtet werden. Für alle Testeinträge, die mehr als 30 Bücher gelistet haben, können keine hundert Prozent Recall erreicht werden und drücken somit den Durchschnitt nach unten.

Vergleicht man die verschiedenen Testreihen miteinander, lässt sich zunächst einmal feststellen, dass der beste Precision Wert von knapp 22% wie zu erwarten von der *Sparse* Variante erreicht wurde. Der Unterschied zu den größeren Fix Varianten ist allerdings nur marginal. Der beste Recall Wert wurde von der Fix 20 Variante erreicht. Die einzige Variante, die sich im Recall abhebt, ist die Fix 01 Variante, die nicht über einen Recall von 8,5% hinauskommt. Dies lässt darauf schließen, dass Bücher, wenn sie in einer Listung auftauchen, auch oft ähnlich gut bewertet werden und damit weit vorne in der Auflistung auftauchen.

Vergleicht man die Sparse Variante mit der Fix 50 Variante, stellt man erst bei genauem Hinsehen Unterschiede in den Kurven fest. Der Verlust in der Qualität der Empfehlung ist demnach nur gering. In einigen Fällen wirkt sich das Einschränken der Speichergröße sogar positiv auf die Vorhersage aus. So sind die Precision Werte der Fix 05 Variante im obe-

ren Recall Bereich besser, teilweise sogar doppelt so hoch, wie bei der verlustfreien Sparse Variante.

Fazit Recall & Precision

Recall und Precision Werte von 16% bzw. 20% hören sich zunächst nicht nach viel an. Bedenkt man jedoch die minimalen Informationen, die als Basis für die Empfehlung dienen und zieht man das Rauschen hinzu, mit dem man in allen Realdaten rechnen muss, ist das Ergebnis sehr zufriedenstellend. Hinzu kommt, dass wie bereits in Abschnitt 2.2.1 erwähnt, Precision und Recall keinesfalls als einziges Gütekriterium für einen Recommender herangezogen werden darf. Im Gegenteil würde eine Optimierung auf Precision dem eigentlichen Ziel, dem Benutzer neuartige Empfehlungen zu geben, sogar entgegenwirken.

Der Verlust, den man durch das verlustbehaftete Fix SmartStore Speicherverfahren hinnehmen muss, ist nahezu vernachlässigbar. Im Einzelfall ergeben sich sogar Verbesserungen. Die beiden Speichervarianten sind in der Frage von Precision und Recall demnach bei geeignetem Limit als gleichwertig zu betrachten.

6 Fazit

Das Ziel, einen Recommender in Echtzeit mit geringen Hardwareanforderungen zu betreiben wurde erreicht. Als ein Hauptproblem hat sich nicht, wie vorher zu erwarten, im Bereich der schnellen Berechnung einer Empfehlung ergeben, sondern in der Problematik des Speicherverbrauchs. Zwar sind Rechner heute mit 8 GiB und mehr ausgestattet, diese versorgen jedoch eine Vielzahl von virtuellen Systemen, die für sich genommen meist weniger Speicher zur Verfügung haben. Aus diesem Grund beschäftigt sich ein großer Teil der Arbeit damit, die Anforderungen an den Speicher so gering wie möglich zu halten. In diesem Rahmen wurde das Konzept des verlustbehafteten Speichers vorgestellt, der auf dem Grundgedanken beruht, dass nicht alle, sondern nur die besten Gegenstände als Grundlage für eine Empfehlung herangezogen werden. Auswertungen haben ergeben, dass der Verlust, der zwangsläufig mit solchen Verfahren verbunden ist, gering genug ist für ein Produktivsystem wie z.B. im Bucheinzelhandel.

Der Ansatz des `IncrementalJaccard` wurde nicht weiter verfolgt, da sein Hauptnachteil der hohe Speicherverbrauch ist, wobei es nicht ausgeschlossen ist, dass diese Eigenschaft durch Ausnutzen von Gesetzmäßigkeit gesenkt werden kann. Er ist dennoch ein legitimer Ansatz, wo es darum geht, die Datenbasis möglichst aktuell zu halten und die Updateintervalle zu verkürzen.

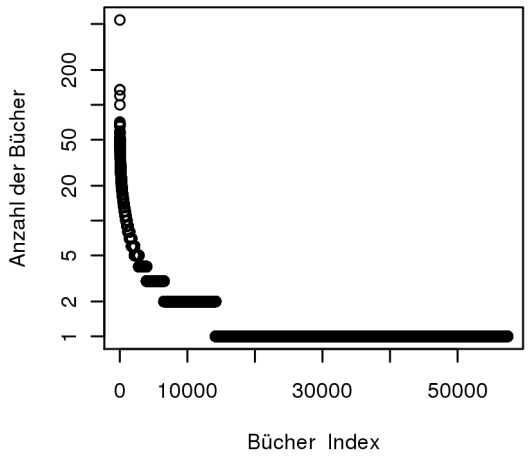
Bei den Anforderungen an die Vorberechnungen geht das System nahe an das Limit heran. Es ist zu erwarten, dass solche rechenintensiven Prozesse außerhalb der üblichen Nutzungs- und Wartungszeiten geschehen müssen. Ein Zeitfenster von etwa fünf Stunden ist dabei nicht leicht in einen Zeitplan zu fassen. Die Möglichkeit eines blockweisen Updates wurde hier vorgestellt mit dem Nachteil, dass die Datenbasis nicht immer aktuell ist. Problematisch wird dieser Ansatz nur dann, wenn in einem Intervall der Datensammlung mehr Daten anfallen, als in dem Intervall der Datenverarbeitung abgearbeitet werden können.

Die Problemdimension zu verringern, wurde nicht weiter untersucht. Es könnte also durchaus von Interesse sein, ob und wie sich die hier theoretisch vorgestellten Ansätze mit der gewählten Vorgehensweise kombinieren lassen und welche Auswirkungen das nach sich zieht.

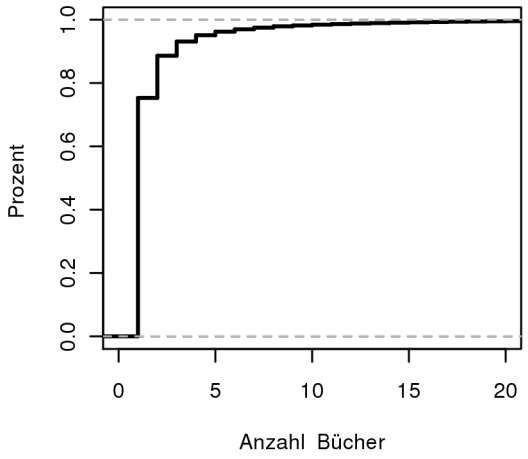
Die stellenweise erwähnten Möglichkeiten, das Problem zu parallelisieren, um es auf Rechenclustern laufen zu lassen, sind auch Punkte, die einer tiefergehenden Betrachtung wert sind. Es ist zu erwarten, dass die Zukunft der Server im Cloud Computing liegt. Daher ist davon auszugehen, dass der Einsatz von massiv paralleler Verarbeitung in absehbarer Zukunft kostengünstig zu haben ist.

A Statistiken

A.1 Eingabedaten

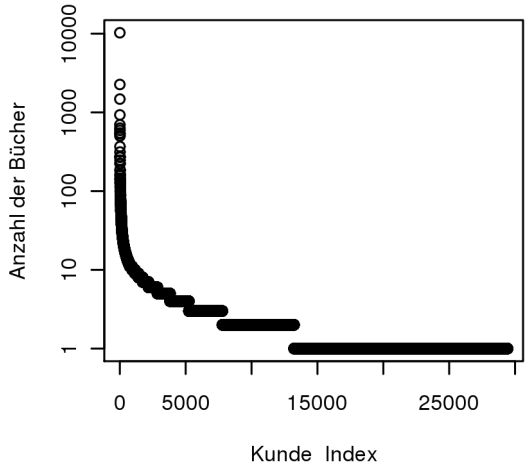


(a) Anzahl verkaufter Exemplare pro Buch

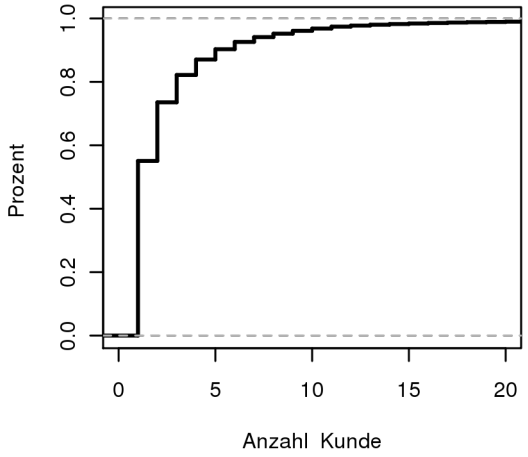


(b) Verteilungsfunktion der verkauften Bücher

Abbildung A.1.: *allg* Daten - Bücher

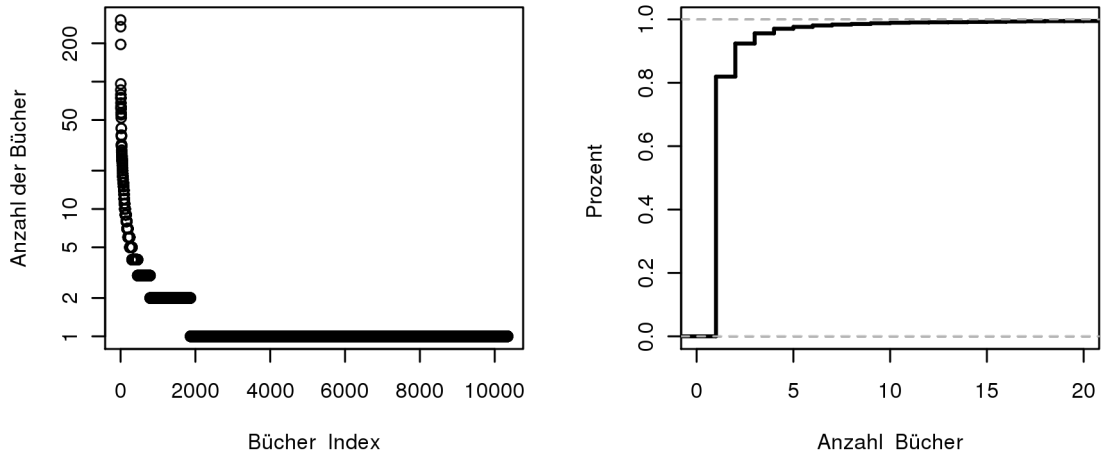


(a) Anzahl der vom Kunden gekauften Bücher



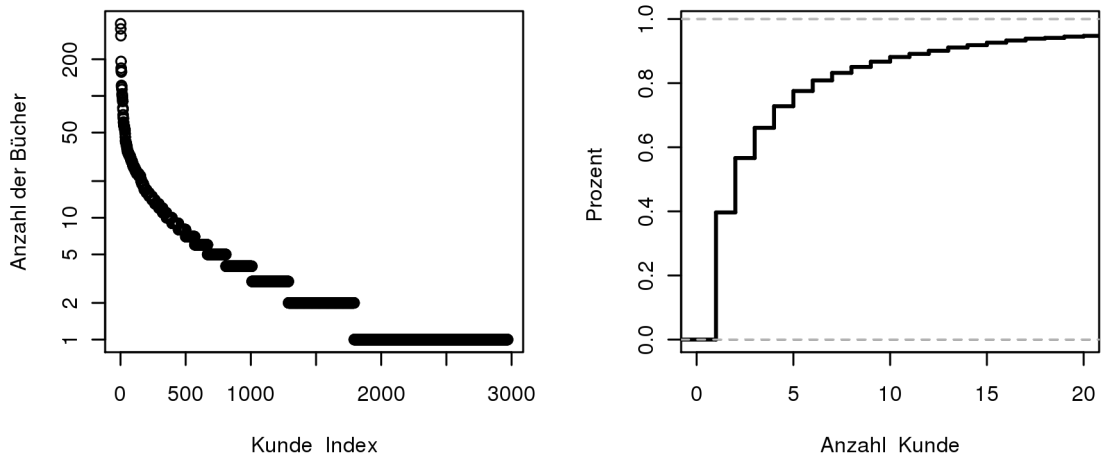
(b) Verteilungsfunktion von Büchern

Abbildung A.2.: *allg* Daten - Kunde



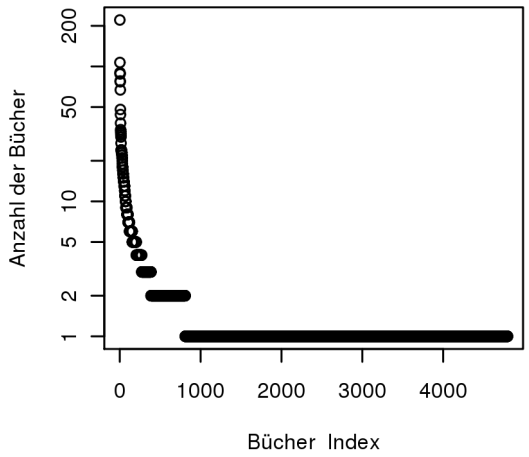
(a) Anzahl verkaufter Exemplare pro Buch **(b)** Verteilungsfunktion der verkauften Bücher

Abbildung A.3.: *DB* Daten - Bücher

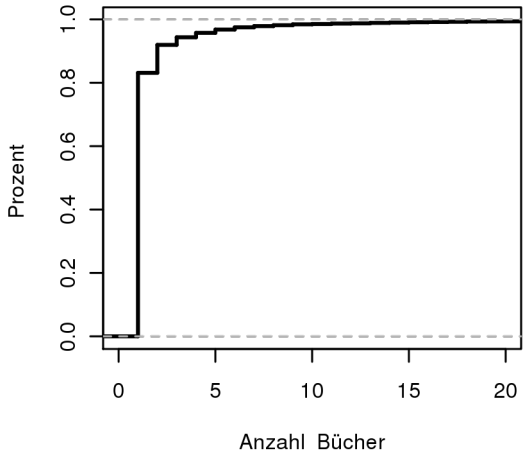


(a) Anzahl der vom Kunden gekauften Bücher **(b)** Verteilungsfunktion von Büchern

Abbildung A.4.: *DB* Daten - Kunde

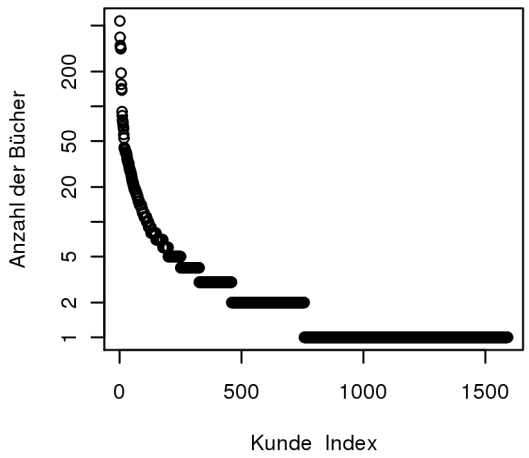


(a) Anzahl verkaufter Exemplare pro Buch

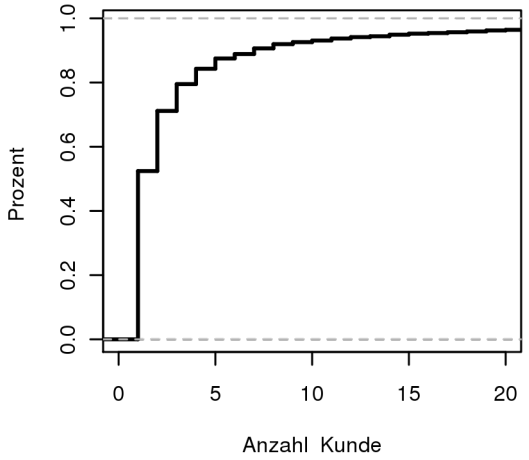


(b) Verteilungsfunktion der verkauften Bücher

Abbildung A.5.: *steuer* Daten - Bücher



(a) Anzahl der vom Kunden gekauften Bücher



(b) Verteilungsfunktion von Büchern

Abbildung A.6.: *steuer* Daten - Kunde

A.2 Stresstest

Die Tabellen A.1 und A.2 listen die Quantile für die Stresstestdaten auf. Und zwar für jeweils die Daten im Bereich mit und ohne Load Balancing, also die Daten vor und nach der "130 Sekunden Marke“. Alle Tests wurden mit Fix $\lambda = 300$ auf dem *db* Datensatz durchgeführt.

Quantil	5%	10%	15%	20%	25%
Antwort[ms]	9	33	52	73	89
Quantil	30%	35%	40%	45%	50%
Antwort[ms]	109	126	145	160	176
Quantil	55%	60%	65%	70%	75%
Antwort[ms]	194	211	228	243	257
Quantil	80%	85%	90%	95%	100%
Antwort[ms]	274	296	318	337	401

Tabelle A.1.: Stresstest - Ohne Loadbalancing

Quantil	5%	10%	15%	20%	25%
Antwort[ms]	269	305	316	322	337
Quantil	30%	35%	40%	45%	50%
Antwort[ms]	341	337	341	346	350
Quantil	55%	60%	65%	70%	75%
Antwort[ms]	354	358	363	367	372
Quantil	80%	85%	90%	95%	100%
Antwort[ms]	377	384	394	3300	21005

Tabelle A.2.: Stresstest - Mit Loadbalancing

B Definitionen und Herleitungen

B.1 Treppenspeicher

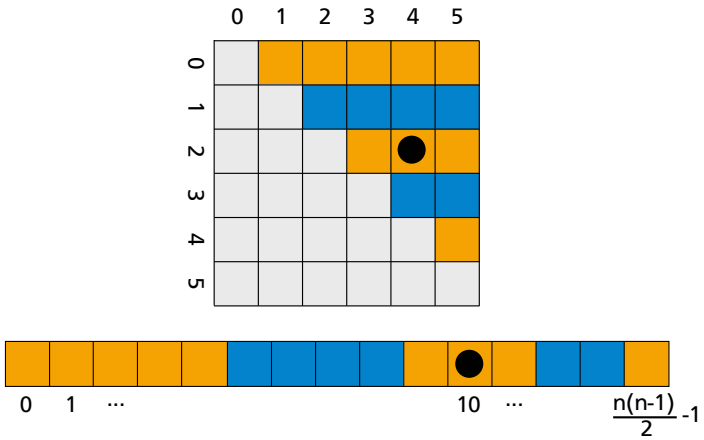


Abbildung B.1.: Treppenspeicher

Der Treppenspeicher ist eine platzsparende und zugriffeffiziente Möglichkeit, eine zweidimensionale Speicherstruktur der Dimension $n \times n$ in einer eindimensionalen Struktur zu implementieren. Der Vorteil einer solchen Implementierung ist die sehr effiziente Möglichkeit der Speicherreservierung. Diese kann im Vorfeld geschehen und ermöglicht einen zusammenhängenden Speicherblock, der sehr effizient verwaltet werden kann.

Gebraucht werden von allen Koordinaten (x, y) nur diejenigen, bei denen $x < y$ gilt, da das Array spiegelsymmetrisch ist und die Werte auf der Diagonalen nicht relevant sind. Für die Umsetzung wird das zweidimensionale Array, wie in Abbildung B.1 zu sehen, zeilenweise in eindimensionale Arrays zerlegt und nach Größe geordnet aneinandergehängt. Das resultierende Array belegt danach nur $n(n-1)/2$ Speicherplätze. Die Umrechnung der zweidimensionalen Array Koordinaten erfolgt nach Gleichung B.1:

$$index(n, x, y) = y - \frac{x^2}{2} + \left(n - \frac{3}{2}\right) \cdot x - 1 \tag{B.1}$$

So errechnet sich beispielsweise aus der zweidimensionalen Koordinate $(2, 4)$ der eindimensionale Index 10, der die Position dieses Wertes markiert. Damit ist ein effizienter Zugriff auf alle Positionen im Array gegeben. Ändert sich die Ausgangsdimension, muss das eindimensionale Array neu erstellt werden, weil sich damit alle Indizes verschieben.

B.2 Herleitung - `IncrementalJaccard`

Die Aktualisierungsfunktionen des `IncrementalJaccard` ergeben sich aus den Tabellen B.1 und B.2. Sie berechnen sich aus dem Istwert und der Angabe, ob ein Wert aktuell neu hinzugekommen (new) bzw. entfernt (del) wurde. Mit • markierte Werte sind solche, die nicht definiert sind und demnach beliebige Wahrheitswerte annehmen können. Manche Werte sind deshalb nicht definiert, weil Abhängigkeiten bestehen. Aus new(x) folgt zwingend is(x). Wertekombination, die den Gleichungen B.2 widersprechen, sind folglich nicht möglich.

$$\begin{aligned} \texttt{new}(x) \rightarrow \texttt{is}(x) \quad &\Rightarrow \quad \neg\texttt{is}(x) \rightarrow \neg\texttt{new}(x) \\ \texttt{del}(x) \rightarrow \neg\texttt{is}(x) \quad &\Rightarrow \quad \texttt{is}(x) \rightarrow \neg\texttt{del}(x) \end{aligned} \tag{B.2}$$

is		new			
x	y	x	y	Zähler	Nenner
0	0	0	0	0	0
0	0	0	1	•	•
0	0	1	0	•	•
0	0	1	1	•	•
0	1	0	0	0	0
0	1	0	1	0	1
0	1	1	0	•	•
0	1	1	1	•	•
1	0	0	0	0	0
1	0	0	1	•	•
1	0	1	0	0	0
1	0	1	1	•	•
1	1	0	0	0	0
1	1	0	1	1	0
1	1	1	0	1	0
1	1	1	1	1	1

Tabelle B.1.: Elemente hinzufügen - Zähler & Nenner

is		del			
x	y	x	y	Zähler	Nenner
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	0	1
0	0	1	1	1	1
0	1	0	0	0	0
0	1	0	1	•	•
0	1	1	0	1	0
0	1	1	1	•	•
1	0	0	0	0	0
1	0	0	1	1	0
1	0	1	0	•	•
1	0	1	1	•	•
1	1	0	0	0	0
1	1	0	1	•	•
1	1	1	0	•	•
1	1	1	1	•	•

Tabelle B.2.: Elemente entfernen - Zähler & Nenner

B.3 JaccardDistance

Der Algorithmus `JaccardDistance` B.3.1 errechnet aus zwei Vektoren die Jaccard Distanz. Dabei beträgt die Anzahl der Iterationen $|ItemsA| + |ItemsB|$. Folgende Funktionsaufrufe werden in diesem Algorithmus benötigt.

getSortedIndex(List) Liefert eine sortierte Liste von Indizes, bei dem der Wert der Eingangsliste ungleich null ist.

hasMoreItems(List) Liefert wahr zurück, solange die Liste nicht leer ist.

first(List) Liefert den ersten Wert der Liste zurück. Ist die Liste leer wird ∞ ausgegeben.

pop(List) Löscht das erste Element in der Liste und liefert wahr zurück genau dann, wenn die Liste zum Zeitpunkt des Aufrufs nicht leer ist.

Algorithmus B.3.1: JACCARDDISTANCE($ItemsA, ItemsB$)

$u \leftarrow getSortedIndex(ItemsA)$
$v \leftarrow getSortedIndex(ItemsB)$
$newU, newV \leftarrow true$
$numerator, denominator \leftarrow 0$
while $hasMoreItems(u)$ **or** $hasMoreItems(v)$
do
- **if** $first(u) == first(v)$
  - **then** $numerator \leftarrow numerator + 1$
  - **else**
    - **if** $newU$
      - **then**
        - $denominator \leftarrow denominator + 1$
        - $newU \leftarrow false$
    - **if** $newV$
      - **then**
        - $denominator \leftarrow denominator + 1$
        - $newV \leftarrow false$
- **if** $first(u) < first(v)$
  - **then** $newU \leftarrow pop(u)$
  - **else** $newV \leftarrow pop(v)$

return $(numerator/denominator)$

Literaturverzeichnis

[1] Dblens collaborative filtering research engine, . http://dblens.sourceforge.net/.

[2] Jmeter, . http://jmeter.apache.org/.

[3] Apache mahout, . http://mahout.apache.org/.

[4] Netbeans profiler, . http://profiler.netbeans.org/.

[5] Daniel Billsus and Michael J. Pazzani. Learning collaborative information filters. In *ICML '98: Proceedings of the Fifteenth International Conference on Machine Learning*, pages 46–54, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc. ISBN 1-55860-556-8.

[6] Ludwig Fahrmeir, Rita Künstler, Iris Pigeot, and Gerhard Tutz. *Statistik. Der Weg zur Datenanalyse*. Springer, Heidelberg, 1999.

[7] Ken Goldberg, T. Roeder, D. Gupta, and C. Perkins. Eigentaste: A constant time collaborative filtering algorithm. Technical Report UCB/ERL M00/41, EECS Department, University of California, Berkeley, 2000.

[8] SongJie Gong. A collaborative filtering recommendation algorithm based on user clustering and item clustering. *Journal Of Software Vol.5 No.7*, pages 745–752, July 2010.

[9] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Prentice Hall, 2005.

[10] J.L. Herlocker, J.A. Konstan, L.G. Terveen, and J.T. Riedl. Evaluating collaborative filtering recommender systems. *ACM Transactions on Information Systems*, pages 5–53, 2004.

[11] George Karypis. *Evaluation of Item-Based Top-N Recommendation Algorithms*. ACM, New York, NY, USA, 2001. ISBN 1-58113-436-3.

[12] Daniel Lamire. Cofi: A java-based collaborative filtering library. http://www.nongnu.org/cofi/.

[13] Daniel Lemire, Harold Boley, Sean McGrath, and Marcel Ball. Collaborative filtering and inference rules for context-aware learning object recommendation. *International Journal of Interactive Technology and Smart Education*, 2(3), August 2005.

[14] Greg Linden, Brent Smith, and Jeremy York. Amazon.com recommendations: Item-to-item collaborative filtering. *IEEE Internet Computing*, 7:76–80, 2003.

[15] G. A. Miller. The magical number 7, plus or minus 2: Some limits on our capacity for processing information. *Psychological Review*, 63(2):81–97, 1956.

[16] Sean Owen, Robin Anil, Ted Dunning, and Ellen Friedman. *Mahout in Action*. Manning Publications Co., 2011. ISBN 978-1-935182-68-9.

[17] Manos Papagelis, Ioannis Rousidis, Dimitris Plexousakis, and Elias Theoharopoulos. Incremental collaborative filtering for highly-scalable recommendation algorithms. In Mohand-Said Hacid, Neil V. Murray, Zbigniew W. Ras, and Shusaku Tsumoto, editors, *ISMIS*, Lecture Notes in Computer Science, pages 553–561. Springer, 2005. ISBN 3-540-25878-7.

[18] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2008. URL `http://www.R-project.org`. ISBN 3-900051-07-0.

[19] Anand Rajaraman and Jeffrey D Ullman. Mining of massive datasets. *Lecture Notes for Stanford CS345A Web Mining*, 30(3), 2011. URL `http://infolab.stanford.edu/~ullman/mmds.html`. ISBN 978-1-107-01535-7.

[20] Badrul Sarwar, George Karypis, Joseph Konstan, and John Reidl. Item-based collaborative filtering recommendation algorithms. In *Proceedings of the 10th international conference on World Wide Web*, WWW '01, pages 285–295, New York, NY, USA, 2001. ACM. ISBN 1-58113-348-0.

[21] Günther Sawitzki. *Computational statistics*. Chapman & Hall/CRC Press, Boca Raton, Fla. [u.a.], 2009. ISBN 978-1-4200-8678-2.

[22] Xiaoyuan Su and Taghi M. Khoshgoftaar. A survey of collaborative filtering techniques. *Advances in Artificial Intelligence.*, 2009:4:2–4:2, January 2009.

[23] Ian H. Witten, Eibe Frank, and Mark A. Hall. *Data Mining - Practical Machine Learning Tools and Techniques*. Morgan Kaufman Publisher Inc., January 2011. ISBN 978-0-12-374856-0.

[24] Zhou Yunhong, Wilkinson Dennis, Schreiber Robert, and Pan Rong. Large-scale parallel collaborative filtering for the netflix prize. In *AAIM '08: Proceedings of the 4th international conference on Algorithmic Aspects in Information and Management*, pages 337–348, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-68865-5.