

INSTITUT FÜR  
INFORMATIK  
Computer Vision, Computer  
Graphics and Pattern Recognition

Universitätsstr. 1  
D-40225 Düsseldorf



# Collaborative Filtering Kollaboratives Filtern

Ole Kiefer

Bachelorarbeit

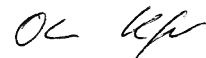
Beginn der Arbeit:	22. Januar 2016
Abgabe der Arbeit:	10. März 2016
Gutachter:	Prof. Dr. Stefan Harmeling Prof. Dr. Stefan Conrad



## Erklärung

Hiermit versichere ich, dass ich diese Bachelorarbeit selbstständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Düsseldorf, den 10. März 2016



---

Ole Kiefer



## Zusammenfassung

Beim kollaborativen Filtern versucht man die meist wenigen Informationen einer dünn besetzten Matrix zu verwenden, um diese sinnvoll aufzufüllen. In der Anwendung wird dieses Verfahren beispielsweise benutzt, um Vorschläge für einen Nutzer zu kreieren oder Vorlieben eines Nutzers heraus zu finden. Dazu werden Distanzen zwischen den Zeilen oder Spalten der Matrix berechnet, um ähnliche Einträge zu finden und neue Informationen zu gewinnen.

In dieser Arbeit stelle ich einige gängige Verfahren des kollaborativen Filterns vor, wie den Pearson Correlation Coefficient, Adjusted Cosine Similarity oder den Slope One Algorithmus. Mit dem Floyd-Warshall-Algorithmus wird eine Idee aus der Graphen-Analyse mit diesem Thema in Verbindung gebracht. Zusätzlich habe ich mir Gedanken gemacht, wie ich die simple Metrik des euklidischen Abstandes verbessern kann, um bessere Ergebnisse mit diesem Algorithmus zu erzielen.

Der Vergleichstest arbeitet auf der frei verfügbaren MovieLens-Datenbank 100k, in der 943 User 1682 Filme bewertet haben. Die Datenbank wurde aufgeteilt in eine Trainings- und eine Testmenge. Die Algorithmen berechnen mögliche Filmbewertungen und werden mit der Testmenge verglichen. Der durchschnittliche Fehler dient als Vergleichsgröße wie gut der Algorithmus funktioniert. Zusätzlich vergleiche ich die Verfahren auf einer großen Matrix mit niedrigem Rang, in der viele Zeilen eine Linearkombination einer kleinen Menge linear unabhängiger User sind.

Im Kapitel 4 wird erst die Parameterwahl einiger Algorithmen getestet. Die produzierten Fehler werden miteinander verglichen und die Ergebnisse interpretiert, zum Beispiel in welchem Szenario sich welcher Algorithmus eignet. Überraschenderweise kann die Methode mit dem euklidischen Abstand mit Strafe ebenfalls sehr gute Ergebnisse erzielen und schlägt die meisten Algorithmen. Das Hybrid Verfahren, dass Euklid mit Slope One kombiniert, verbessert noch einmal das Ergebnis und erzielt in diesem Szenario das beste Ergebnis.

Im Anhang ist mein Quellcode zur Auswertung der Datenbank, mittels der untersuchten Algorithmen, in Python zu finden. In der Recommender Klasse sind alle Algorithmen implementiert. Die Testklasse dient als eine Art Toolbox um den Fehler zur Testmenge zu berechnen.



# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>3</b>
<b>2</b>	<b>Algorithmen zum kollaborativen Filtern</b>	<b>4</b>
2.1	User-basierte Algorithmen . . . . .	5
2.1.1	Euklidische Distanz . . . . .	5
2.1.2	Pearson Correlation Coefficient (PCC) . . . . .	6
2.1.3	Floyd-Warshall (FW) . . . . .	7
2.2	Item-basierte Algorithmen . . . . .	8
2.2.1	Adjusted Cosine Similarity (ACS) . . . . .	8
2.2.2	Slope One . . . . .	8
2.3	Hybrid . . . . .	9
<b>3</b>	<b>Vorhersage zu der Filmdatenbank MovieLens</b>	<b>11</b>
3.1	Filmdatenbank MovieLens . . . . .	11
3.2	Aufteilung der Datenmenge in Test- und Trainingsdaten . . . . .	12
3.3	Matrix mit niedrigem Rang . . . . .	12
3.4	Berechnung des Fehlers . . . . .	13
<b>4</b>	<b>Vergleich verschiedener Algorithmen</b>	<b>14</b>
4.1	Wahl der Parameter . . . . .	14
4.1.1	Vergleich der drei Euklid Varianten . . . . .	14
4.1.2	Wahl von $k$ bei den kNN im Pearson Algorithmus . . . . .	15
4.1.3	Wahl von $k$ bei den kNN im Floyd Warshall Algorithmus . . . . .	16
4.1.4	Wahl von $a$ beim Hybrid Algorithmus . . . . .	17
4.2	Auswertung der Fehlerberechnung . . . . .	17
<b>5</b>	<b>Schlussfolgerung</b>	<b>21</b>
<b>A</b>	<b>Anhang: Quellcode</b>	<b>23</b>
A.1	Übersicht aller Funktionen . . . . .	23
A.2	Quellcode . . . . .	27

<b>Literatur</b>	<b>51</b>
<b>Abbildungsverzeichnis</b>	<b>52</b>
<b>Tabellenverzeichnis</b>	<b>52</b>



# 1 Einleitung

Jeder Homepagebetreiber hat das Ziel den Benutzer möglichst lange auf der eigenen Homepage zu halten, um möglichst viele Klicks und Werbeeinspielungen zu generieren. Ein Weg dies zu erreichen: personalisierte Webseiten, Werbung und Kaufvorschläge. Eine Online-Zeitung schlägt deshalb beim Lesen eines Artikels weitere Artikel vor, die einen interessieren könnten. Marketing-Firmen analysieren den Benutzer, um mit gezielter Werbung mehr Umsatz zu erzielen. Ein 20-jähriger Mann wird eher Autowerbung interessant finden als Make-Up-Werbung. Online-Shops geben Kaufvorschläge zu Artikeln, die man gerade ansieht oder in den Warenkorb gelegt hat, um weitere Käufe anzuregen. Online-Video- oder Musik-Plattformen schlagen ähnliche Filme oder Musikstücke vor, um den Nutzer zu halten.

Die Daten, die nötig sind, um den User zu analysieren und zu bewerten, liefert dieser meistens selbst. Tracker verfolgen jeden Klick, jede Mausbewegung wird mit Timern analysiert, Warenkörbe werden ausgewertet, Cookies speichern Daten für die nächste Sitzung. Daraus lassen sich viele Informationen ableiten, man kann abschätzen, wen man zu Besuch hat und womit man Aufmerksamkeit und letztendlich Klicks erreichen kann. In vielen Bereichen liefern die Benutzer ganz bewusst Informationen. Wenn sie etwas bewerten, ihr Profil ausfüllen oder auf den „like“-Button drücken. Hierbei treten jedoch zwei Probleme auf. Sehr wenige Nutzer bewerten regelmäßig und extreme Bewertungen (sehr schlecht oder sehr gut) kommen häufiger vor, als die differenzierten Zwischenstufen.

Um verschiedene Musikstücke genauer bewerten zu können, setzt der Musik-Streaming-Dienst PANDORA<sup>1</sup> deshalb auf ein Expertensystem. Im Music Genome Project<sup>2</sup> werden 450 charakteristische Merkmale in Liedern von trainierten Musikern bewertet und analysiert. Studierte Musiker geben hier qualitativ hochwertige Bewertungen ab, um die Musikstücke korrekt zu kategorisieren.

Beim kollaborativen Filtern nutzt man diese Informationen über die User und Items, um Interessen zu vergleichen, Ähnlichkeiten zu bewerten und Vorschläge zu erstellen.

In dieser Arbeit stelle ich sechs Algorithmen vor und teste diese am MovieLens Datensatz 100k (s.Unterabschnitt 3.1 Filmdatenbank Movielens), um sie miteinander vergleichen zu können.

---

<sup>1</sup>[www.pandora.com](http://www.pandora.com)

<sup>2</sup><http://www.pandora.com/about/mgp>

## 2 Algorithmen zum kollaborativen Filtern

Beim kollaborativen Filtern ist die grundsätzliche Frage: Wie ähnlich sind sich zwei Objekte? Mathematisch ausgedrückt fragt man nach der Distanz zweier Objekte. Ein Ansatz ist nach der Distanz zweier User zu fragen, der andere die Ähnlichkeit zweier Items zu untersuchen. Um diese Distanz zu berechnen, gibt es verschiedene Methoden.

Betrachtet man die Bewertungen als Matrix  $R$  mit den Usern als Zeilen und den Items als Spalten, so wird die Ähnlichkeit zweier User  $u$  und  $v$  durch die Distanz ihrer Zeilenvektoren bewertet. Die Distanz zweier Items  $i$  und  $j$  ist der Abstand der Spaltenvektoren.  $r_{u,i}$  ist der  $i$ -te Eintrag der  $u$ -ten Zeile von  $R$  und damit die Bewertung des Users  $u$  zum Film  $i$ .  $\bar{r}_u^{user}$  ist seine durchschnittliche Bewertung.  $\hat{r}_{u,i}$  ist die in den Bereich  $[-1, 1] \subset \mathbb{R}$  normierte Bewertung.  $R_u^{user}$  bezeichne, die Menge der Filme, die von  $u$  bewertet wurden,  $R_i^{item}$  die Menge der User, die Film  $i$  bewertet haben.  $\bar{r}_i^{item}$  ist die durchschnittliche Bewertung des Items  $i$  aller User.

$$\begin{aligned}
 R_u^{user} &:= \{i | r_{u,i} > 0\} & R_i^{item} &:= \{u | r_{u,i} > 0\} \\
 \bar{r}_u^{user} &:= \frac{\sum_{i \in R_u^{user}} r_{u,i}}{|R_u^{user}|} & \bar{r}_i^{item} &:= \frac{\sum_{u \in R_i^{item}} r_{u,i}}{|R_i^{item}|} \\
 \min_R &= \min_{(u,i) \in R} r_{u,i} & \max_R &= \max_{(u,i) \in R} r_{u,i} \\
 \hat{r}_{u,i} &:= \frac{2(r_{u,i} - \min_R) - (\max_R - \min_R)}{(\max_R - \min_R)}
 \end{aligned} \tag{1}$$

Im speziellen Fall der MovieLens-Daten ist  $\min_R = 1$  und  $\max_R = 5$ , die minimale bzw. maximale Bewertung eines Filmes.

Ein einfacher Weg einen neuen Vorschlag zu erzeugen, ist den ähnlichsten Nutzer zu User  $u$  zu finden und ein Item vorzuschlagen, dass  $u$  noch nicht bewertet hat. Um ein wenig mehr Informationen zu nutzen und unabhängiger von persönlichen Vorlieben zu werden, benutzt man nicht nur einen, sondern  $k$  ähnliche User. Die Menge der  $k$  nächsten Nachbarn  $kNN_d(u)$ , sei gefüllt mit  $k$  Usern mit dem geringsten Abstand, berechnet durch  $d$ , zu User  $u$ . Da der Test in Abschnitt 3 (Vorhersage zu der Filmdatenbank MovieLens) das Rating für bestimmte Filme benötigt, werden in diesem Test nur Nachbarn in Betracht gezogen, die den Film  $i$  auch bewertet haben. Damit wird sicher gestellt, dass die ausgewählten Nachbarn den Film bewertet haben.

Um ein Rating für einen Film zu erzeugen, kann man nun den Mittelwert der

Bewertungen der  $k$  nächsten Nachbarn bilden für diesen Film.

$$r_d(u, i) := \frac{\sum_{v \in \text{kNN}_d(u)} r_{v,i}}{|\text{kNN}_d(u)|} \quad (2)$$

Diese Formel kann man als Grundformel für User-basierte Verfahren ansehen.

In den nächsten Abschnitten werden die verschiedenen Algorithmen beschrieben. Die euklidische Distanz dient als Einstieg in die Abstandsberechnung. Die Verfahren Pearson Correlation Coefficient [Zar15, Kap. 2, S. 23], Adjusted Cosine Similarity [Zar15, Kap. 3, S. 16] und Slope One [Zar15, Kap. 3, S. 28] sind beschrieben in dem Buch „A Programmer’s Guide to Data Mining: The Ancient Art of the Numerati“ [Zar15] von Ron Zacharski. Es folgt eine Idee aus der Graphentheorie. Mittels Floyd-Warshall werden kürzeste Wege zwischen Usern gesucht, um weitere Ähnlichkeiten zu errechnen. Den Abschluss bildet ein Hybrid-Verfahren, eine Kombination aus Euklid und Slope One.

Für alle Formeln und Algorithmen gilt die Einschränkung für  $R_u^{user} \cap R_v^{user} \neq \emptyset$  bzw.  $R_i^{item} \cap R_j^{item} \neq \emptyset$ . Ist die Schnittmenge zwischen zwei Usern oder Items leer, wird keine Distanz zwischen diesem Paar definiert.

## 2.1 User-basierte Algorithmen

### 2.1.1 Euklidische Distanz

Eine simple Metrik ist die euklidische Distanz:

$$d_{\text{euclid}}(u, v) := \sqrt{\sum_{i \in R_u^{user} \cap R_v^{user}} (r_{u,i} - r_{v,i})^2} \quad , \text{ für } R_u^{user} \cap R_v^{user} \neq \emptyset \quad (3)$$

Ist die Schnittmenge zwischen User  $u$  und User  $v$  leer, so wird keine Distanz zwischen ihnen definiert. Die euklidische Distanz zwischen zwei Benutzern  $u$  und  $v$  wird berechnet durch alle Items, die sowohl von User  $u$  als auch von User  $v$  bewertet wurden. Eine geringe Distanz suggeriert eine hohe Ähnlichkeit. Dies führt jedoch zu dem Problem, dass zwei Personen über die sehr wenig gemeinsame Informationen verfügen, ähnlicher bewertet werden können, als zwei Personen über die man sehr viele gemeinsame Informationen hat. Jede Information die man nutzt, vergrößert im Allgemeinen den Abstand zwischen zwei Objekten.

Eine Idee, dies zu verbessern, wäre eine Strafe einzubauen für Informationen, die man über den User  $u$  kennt, aber über User  $v$  nicht. Ich habe mich dazu entschieden die Distanz zur mittleren Bewertung  $(\frac{1}{2}(\max_R - \min_R) + \min_R)$  als Strafe

einzubauen. Als Beispiel: User  $u$  hat Film  $i$  mit 5 bewertet, User  $v$  hat keine Bewertung zu diesem Film abgegeben, so wird der Wert 2 als Fehler addiert.

$$d_{\text{euclidpenalized}}(u, v) := \sqrt{\sum_{i \in R_u^{\text{user}} \cap R_v^{\text{user}}} (r_{u,i} - r_{v,i})^2 + \sum_{i \in R_u^{\text{user}} \setminus R_v^{\text{user}}} (r_{u,i} - (\frac{1}{2}(\max_R - \min_R) + \min_R))^2} \quad (4)$$

Eine weitere Möglichkeit Nachbarn zu suchen, die viele Informationen teilen, ist durch die Anzahl der Überschneidungen zu dividieren, um eine mittlere Distanz zwischen zwei Items zu ermitteln. Damit wird die euklidische Distanz zum mittleren, quadratischen Fehler.

$$d_{\text{euclidnormalized}}(u, v) := \frac{\sqrt{\sum_{i \in R_u^{\text{user}} \cap R_v^{\text{user}}} (r_{u,i} - r_{v,i})^2}}{|R_u^{\text{user}} \cap R_v^{\text{user}}|} \quad (5)$$

Eine Bewertung wird mittels folgender Formel berechnet.

$$r_{d_{\text{euclid}}}(u, i) := \frac{\sum_{v \in \text{kNN}_{d_{\text{euclid}}}(u)} r_{v,i}}{|\text{kNN}_{d_{\text{euclid}}}(u)|} \quad (6)$$

### 2.1.2 Pearson Correlation Coefficient (PCC)

Der Pearson Algorithmus errechnet eine Ähnlichkeit zwischen allen User mit Hilfe des Pearson Korrelations Koeffizienten. Der Wert zwischen  $[-1, 1] \subset \mathbb{R}$  wird auch als Pearson Score bezeichnet. Wenn sich zwei User in ihren Bewertungen übereinstimmen, haben sie eine Pearson Score von +1. Sind beide User komplett verschieden, bekommen sie eine Score von -1.

$$d_{\text{pearson}}(u, v) := \frac{\sum_{i \in R_u^{\text{user}} \cap R_v^{\text{user}}} (r_{u,i} - \bar{r}_u^{\text{user}})(r_{v,i} - \bar{r}_v^{\text{user}})}{\sqrt{\sum_{i \in R_u^{\text{user}} \cap R_v^{\text{user}}} (r_{u,i} - \bar{r}_u^{\text{user}})^2} \sqrt{\sum_{i \in R_u^{\text{user}} \cap R_v^{\text{user}}} (r_{v,i} - \bar{r}_v^{\text{user}})^2}} \quad (7)$$

Von jeder Bewertung  $r_{u,i}$  des Users  $u$  für Item  $i$ , wird der Durchschnitt aller Bewertungen des Users  $\bar{r}_u^{\text{user}}$  abgezogen. Dadurch können sich zwei User ähnlich sein, unabhängig davon ob, der eine am oberen Ende der Skala und der andere am unteren Ende der Skala bewertet. Diese zwei User würden einen großen euklidischen Abstand haben und damit als sehr verschieden gelten. Hat man die Distanzen zwischen dem User  $u$  und allen anderen Usern errechnet, findet man den ähnlichsten User  $v$ , indem man nach dem Pearson Score sortiert. Dementsprechend beinhaltet die Menge der  $k$  nächsten Nachbarn die User, mit dem

höchsten Score. Eine weitere Optimierung im PCC ist, dass die Bewertungen der Nachbarn mit dem Pearson-Score gewichtet gemittelt werden. Dies erzeugt eine Liste an Vorschlägen mit Items die  $u$  gefallen könnten. Gleichung 2 wird mit der Gewichtung angepasst:

$$r_{d_{\text{pearson}}}(u, i) := \frac{\sum_{v \in \text{kNN}_d(u)} r_{v,i} \cdot d_{\text{pearson}}(u, v)}{\sum_{v \in \text{kNN}_d(u)} d_{\text{pearson}}(u, v)} \quad (8)$$

### 2.1.3 Floyd-Warshall (FW)

Ermittelt man die Distanz zwischen allen Usern, so erhält man eine User-User-Matrix. Betrachtet man diese Matrix als Graphen kann man mit Graphen-Algorithmen Beziehungen zwischen Usern finden.

Die Gewichte des Graphen sind die Distanzen zwischen den einzelnen Userpaaren. Zur Anwendung kommt hier der Euklid-Algorithmus mit Strafe aus Gleichung 4. Dadurch entsteht zwischen einigen Usern ein gewichteter Pfad. User die keine Übereinstimmungen haben, bekommen die Distanz 1000.

$$d_{\text{flowar}}(u, v) := \begin{cases} d_{\text{euclidpenalized}}(u, v) & \text{if } R_u^{\text{user}} \cap R_v^{\text{user}} \neq \emptyset \\ 1000 & \text{else} \end{cases} \quad (9)$$

Floyd-Warshall berechnet nun zwischen allen Knoten den kürzesten Pfad. Der Pfad mit der kürzesten Gesamtlänge zwischen Knoten  $u$  und  $v$  wird als Distanz zwischen diesen Beiden aktualisiert.

Floyd-Warshall-Algorithmus

```

foreach  $w$  in  $User$  do
  foreach  $u$  in  $User$  do
    foreach  $v$  in  $User$  do
       $d[u][v] = \min ( d[u][v], d[u][w] + d[w][v] )$ 
    end
  end
end

```

Wie beim Euklid-Algorithmus werden die  $k$  ähnlichsten User, mit der niedrigsten Distanz, betrachtet, um ein gemittelttes Rating zu erzeugen.

$$r_{d_{\text{flowar}}}(u, i) := \frac{\sum_{v \in \text{kNN}_d(u)} r_{v,i}}{|\text{kNN}_d(u)|} \quad (10)$$

Die Idee Algorithmen aus der Graphen-Theorie für das kollaborative Filtern zu nutzen wird in dem Paper „Studying Recommendation Algorithms by Graph Analysis“[MKR] von Mirza, Keller und Ramakrishnan näher erläutert.

## 2.2 Item-basierte Algorithmen

### 2.2.1 Adjusted Cosine Similarity (ACS)

Adjusted Cosine Similarity ist ein Item-basierter Algorithmus. Statt die Ähnlichkeit zweier User zu berechnen, sucht ACS nach der Ähnlichkeit zweier Items.

$$d_{\text{acs}}(i, j) := \frac{\sum_{u \in R_i^{\text{item}} \cap R_j^{\text{item}}} (r_{u,i} - \bar{r}_u^{\text{user}})(r_{u,j} - \bar{r}_u^{\text{user}})}{\sqrt{\sum_{u \in R_i^{\text{item}} \cap R_j^{\text{item}}} (r_{u,i} - \bar{r}_u^{\text{user}})^2} \sqrt{\sum_{u \in R_i^{\text{item}} \cap R_j^{\text{item}}} (r_{u,j} - \bar{r}_u^{\text{user}})^2}} \quad (11)$$

Die Formel berechnet die Ähnlichkeit zwischen Item  $i$  und  $j$ . Im Gegensatz zu PCC summiert ACS nicht über alle Items, die zwei User verbinden, sondern über alle User, die zwei Items verbinden.

Um mit diesen Ähnlichkeiten jetzt ein Rating zu erzeugen wird die folgende Funktion benötigt.

$$\hat{r}_{\text{acs}}(u, i) := \frac{\sum_{j \in R_u^{\text{user}}} (d_{\text{acs}}(i, j) \cdot \hat{r}_{u,j})}{\sum_{j \in R_u^{\text{user}}} (|d_{\text{acs}}(i, j)|)} \quad (12)$$

$j$  sind alle Items die bisher von  $u$  bewertet wurden.  $\hat{r}_{u,j}$  ist das normalisierte Rating im Wertebereich  $[-1, 1] \subset \mathbb{R}$ . Das heißt, man betrachtet alle Items die bisher vom User  $u$  bewertet wurden und multipliziert die Ähnlichkeit zu Item  $i$ . Dividiert durch die Summe aller Ähnlichkeiten.

Danach wird das normalisierte Rating wieder in den ursprünglichen, nichtnormalisierten Ratingbereich transformiert.

$$r_{\text{acs}}(u, i) := \frac{1}{2}(\hat{r}_{u,i} + 1)(\max_R - \min_R) + \min_R \quad (13)$$

### 2.2.2 Slope One

Slope One ist ebenfalls ein Item-basierter Algorithmus. Hier wird die Distanz zwischen zwei Items als durchschnittliche Abweichung aller Bewertungen definiert. Aus diesen erzeugt man dann ein Rating für das neue Item.

$$\text{freq}(i, j) := \text{card}(R_i^{\text{item}} \cap R_j^{\text{item}}) = |R_i^{\text{item}} \cap R_j^{\text{item}}| \quad (14)$$

$$d_{\text{slope1}}(i, j) := \sum_{u \in R_i^{\text{item}} \cap R_j^{\text{item}}} \frac{r_{u,i} - r_{u,j}}{\text{freq}(i, j)} \quad (15)$$

$\text{freq}(i, j)$  berechnet die Anzahl der User, die sowohl  $i$  als auch  $j$  in ihren Bewertungen haben.  $d_{\text{slope1}}$  berechnet die Item-Item Matrix mit den Abweichungen zwischen allen Items. Um jetzt eine Vorhersage für den User  $u$  für das bisher nicht von ihm bewertete Item  $i$  machen zu können, kann man die vorher berechneten Abweichungen nutzen.

$$r_{\text{slope1}}(u, i) := \frac{\sum_{j \in R_u^{\text{user}}} (d_{\text{slope1}}(i, j) + r_{u,j}) \text{freq}(i, j)}{\sum_{j \in R_u^{\text{user}}} \text{freq}(i, j)} \quad (16)$$

Der Zähler bedeutet: Für jedes von User  $u$  bewertete Item  $j$  addieren wir zu  $r_{u,j}$  die Abweichung  $d_{\text{slope1}}(i, j)$ . Dies wird mit der Anzahl der User multipliziert, die beide Items  $i$  und  $j$  bewertet haben. Danach wird durch die Anzahl aller User geteilt, die sowohl Item  $i$  als auch Items des Users  $u$  mit einer Bewertung versehen haben. Die Abweichungen als auch die Anzahl der Bewertungen, die in die mittlere Abweichung einbezogen wurden, kann man vorweg berechnen und in Item-Item-Matrizen speichern. Eine neue Bewertung  $r_{u,i}$  kann, durch die Frequenz-Matrix, sehr schnell in die Abweichungen hinzu gerechnet werden, ohne dass die komplette Matrix neu berechnet werden muss.

$$\forall j \in R_u^{\text{user}} : \quad d_{\text{slope1}}(i, j) := \frac{(d_{\text{slope1}}(i, j) \text{freq}(i, j) + (r_{u,i} - r_{u,j}))}{\text{freq}(i, j) + 1} \quad (17)$$

## 2.3 Hybrid

Hybrid ist ein Mix aus dem Verfahren des Euklidischen Abstandes mit Strafe und SlopeOne. Es wird also einmal eine Bewertung über die User-Ähnlichkeit und einmal über die Item-Ähnlichkeit berechnet und diese beiden Bewertungen werden gemittelt. Ein Parameter  $a$  entscheidet, wie stark die Ratings der beiden Algorithmen ins Gewicht fallen.

$$r_{\text{hybrid}}(u, i) := (1 - a)r_{\text{euclidpenalized}}(u, i) + ar_{\text{slope1}}(u, i) \quad (18)$$

Dieser Algorithmus ist ein Versuch, wie stark eine Kombination aus zwei Methoden sein kann.

Tabelle 1: Liste der Algorithmen

	Euklidische Distanz
User-basiert	Pearson Correlation Coefficient (PCC)
	Floyd-Warshall (FW)
	Hybrid
Item-basiert	Adjusted Cosine Similarity (ACS)
	Slope One



## 3 Vorhersage zu der Filmdatenbank MovieLens

Um die sechs Algorithmen miteinander vergleichen zu können, treten sie in verschiedenen Szenarien (s. Unterabschnitt 3.2 Aufteilung der Datenmenge in Test- und Trainingsdaten) gegeneinander an. Als Datenbank dient das MovieLens Dataset 100k. Als zweiten Test habe ich eine niedrig dimensionierte, randomisierte Matrix erstellt. Die Frage ist, ob man parallelen zwischen den beiden Datenmengen erkennt und dieses Wissen nutzen kann.

### 3.1 Filmdatenbank Movielens

Der verwendete Datensatz, auf dem die Algorithmen laufen, ist der MovieLens Datensatz 100k<sup>3</sup>. Dieser ist frei verfügbar und ist seit 1998 unverändert online, um erzielte Ergebnisse miteinander vergleichen zu können. Die Datensätze sind durch Userbewertungen, die auf Movielens<sup>4</sup> abgegeben wurden, entstanden. Im 100k Datensatz sind folgende Daten enthalten:

- 943 User
- 1682 Filme
- 100.000 Ratings
- Ratings von 1-5
- Jeder User hat mindestens 20 Filme bewertet
- Demographische Informationen wie Geschlecht, Alter, Wohnort

Im Laufe der Jahre sind weitere, größere Datensätze entstanden. Ebenfalls verfügbar sind die Datensätze 1M mit einer Million Bewertungen von 6000 User zu 4000 Filmen. Im Jahr 2009 ist der 10M Datensatz zur Verfügung gestellt worden. Zusätzlich zu den 72.000 User und 10.000 Filmen wurden noch 100.000 Tags der Datenbank hinzugefügt. Diese Informationen helfen die Vorschläge stetig zu verbessern und zu optimieren. Im Jahr 2015 hat sich die Datenbank noch einmal verdoppelt auf 20M (27k Filme / 138k User / 100k Tags). Weiterhin gibt es zwei Datensätze die sich regelmäßig verändern und somit ständig neue Voraussetzungen bieten.

---

<sup>3</sup><http://grouplens.org/datasets/movielens/>

<sup>4</sup><https://movielens.org/>

### 3.2 Aufteilung der Datenmenge in Test- und Trainingsdaten

Die Datenmenge wurde aufgeteilt in Test- und Trainingsdaten. In den Trainingsdaten sind die Informationen enthalten, die der Algorithmus nutzen kann um Vorhersagen zu generieren. Die Testdaten werden genutzt um die Vorhersagen zu überprüfen mittels der echten Bewertungen der User. Jeder User hat mindestens 20 Filme bewertet. Um verschiedene Szenarien testen zu können werden  $l$  Bewertungen pro User aus der Datenmenge in die Testmenge geschoben, mit  $l \in \{1, 5, 10, 19\}$ . Alle anderen Bewertungen befinden sich in den Trainingsdaten. Für das Test-Szenario  $l = 1$  haben alle User noch mindestens 19 Bewertungen in den Trainingsdaten. Das heißt, dass sehr viele Informationen zur Verfügung stehen. Dies beeinflusst die Wahl ähnlicher User und natürlich das Wissen über einen bestimmten User. Wie wertet er im Mittel? Welche Genres findet er gut oder schlecht? Im Fall  $l = 19$  haben einige Nutzer nur noch einen Film in der Trainingsmenge. In diesem Szenario wird es sehr viel schwieriger eine passende Vorhersage zu ermitteln.

### 3.3 Matrix mit niedrigem Rang

Als zweiter Testdatensatz dient eine zufällig erstellte Matrix mit niedrigem Rang. Die Frage, die sich hier stellt ist, kann man die Datenmenge auf eine Basis von typischen User reduzieren. Wenn man solche User finden kann, kann man mit weiteren Verfahren eine geeignete Linearkombination für jeden User finden, um approximativ die fehlenden Einträge der Matrix zu berechnen. In dieser Arbeit teste ich die oben genannten Algorithmen an einer solchen Menge, um eventuell Parallelen zwischen den beiden Datensätzen zu erkennen. Die Testmenge besteht aus 30 typischen, linear unabhängigen Usern die alle 1682 Filme zwischen 1 und 5 bewertet haben. Diese Matrix wird per Linearkombination der 30 Basis-User auf 943 User erweitert, mit einem Rang 30.

$$o = 30$$

$$m = 943$$

$$n = 1682$$

$$A1 = \text{random.uniform}(1,5,(m,o))$$

$$A2 = \text{random.uniform}(1,5,(o,n))$$

$$B = A1^T A2$$

Matrix  $B$  hat jetzt eine Größe von  $943 \times 1682$  und Rang 30. Um die Matrix an die MovieLens-Daten anzupassen, habe ich die Einträge in den Bereich 1 bis 5 transformiert und zufällig 93,7% der Bewertungen entfernt, um die gleiche Dichte wie in den MovieLens-Daten (6,3%) zu erreichen. Mit diesem Datensatz wird nur das Szenario  $l = 1$  untersucht. Da der ganze Vorgang zufällig ausgewählt wird, ist in diesem Szenario nicht garantiert, dass über jeden User noch genug Informationen

durch Bewertungen in der Trainingsmenge vorliegen.

### 3.4 Berechnung des Fehlers

Die Güte der Algorithmen wird über den durchschnittlichen absoluten Fehler (Mean Absolute Error) ermittelt. Jede User-Item-Bewertung aus der Testmenge  $R^{test}$  wird der Vorhersage des Algorithmus gegenüber gestellt. Der absolute Fehler wird aufsummiert und durch die Anzahl der Datensätze in der Testmenge dividiert.

$$\text{MAE} := \frac{\sum_{(u,i) \in R^{test}} |r(u,i) - r_{u,i}^{test}|}{\text{card}(R^{test})} \quad (19)$$

Ein MAE von 1 bedeutet demnach, dass der Algorithmus mit seinem Vorschlag um durchschnittlich einen Stern daneben liegt. Bei einem MAE von 0 ist die Testmenge perfekt getroffen. Die Bewertungen in der Datenmenge liegen im Ganzzahlbereich 1, 2, 3, 4, 5. Die Algorithmen erstellen Ratings im Bereich  $[1, 5] \subset \mathbb{R}$ . Dies führt zu einem Wertebereich des MAE von  $[0, 4] \subset \mathbb{R}$ .

## 4 Vergleich verschiedener Algorithmen

### 4.1 Wahl der Parameter

#### 4.1.1 Vergleich der drei Euklid Varianten

Euclid (standard) $k$ = Anzahl Nachbarn, $l$ = Anzahl unbekannter Filme pro User									
k	1	5	10	15	20	25	50	100	200
$l=1$	0,93513	0,86575	0,83903	0,82847	0,82402	0,8271	0,82197	0,82663	0,83957
$l=5$	0,93291	0,85851	0,82809	0,81509	0,80803	0,80551	0,81162	0,81936	0,83285
$l=10$	0,93814	0,87524	0,8482	0,83736	0,8308	0,82894	0,82729	0,83385	0,84658
$l=19$	0,9353	0,89642	0,86939	0,85977	0,85862	0,85634	0,85114	0,85345	0,85783
mean	0,93537	0,87398	0,8461775	0,8351725	0,8303675	0,8294725	0,828005	0,83332	0,8442075
Euclid (normalized)									
k	1	5	10	15	20	25	50	100	200
$l=1$	0,83438	0,79505	0,78685	0,78748	0,78434	0,78884	0,80209	0,8213	0,84519
$l=5$	0,83768	0,80484	0,79295	0,78873	0,78536	0,78595	0,79431	0,81294	0,83385
$l=10$	0,85808	0,81426	0,80163	0,79769	0,801	0,802	0,81183	0,82723	0,84781
$l=19$	0,88172	0,85219	0,83798	0,83473	0,83428	0,83401	0,84036	0,84861	0,8586
mean	0,852965	0,816585	0,8048525	0,8021575	0,801245	0,8027	0,8121475	0,82752	0,8463625
Euclid (penalized)									
k	1	5	10	15	20	25	50	100	200
$l=1$	0,91092	0,79328	0,78224	0,77823	0,78341	0,78364	0,79445	0,80971	0,82294
$l=5$	0,90081	0,79033	0,7749	0,76986	0,77163	0,77062	0,78346	0,79762	0,81348
$l=10$	0,89898	0,80311	0,79214	0,78652	0,78673	0,78898	0,8003	0,81281	0,82751
$l=19$	0,94419	0,8215	0,80532	0,80168	0,80176	0,80154	0,8095	0,82136	0,83293
mean	0,913725	0,802055	0,78865	0,7840725	0,7858825	0,786195	0,7969275	0,81038	0,824215

Abbildung 1: MAE im Euklid Algorithmus in Abhängigkeit der Parameter

Die Werte sind mit einer Farbskala von Rot über Gelb nach Grün je Zeile formatiert, um den maximalen Fehler (rot) und den minimalen Fehler (grün) in einem Szenario besser erkennen zu können.

Wie man an den Tabellen sehen kann, verbessert die Normalisierung den Standard-Euklid-Algorithmus aus Gleichung 3. Eine stärkere Verbesserung wird jedoch durch die eingebaute Strafe erzielt. Mit den Parametern  $k = 15$  und der Strafe kann der Fehler minimiert werden. In den folgenden Vergleichen werden immer diese Parameter benutzt.

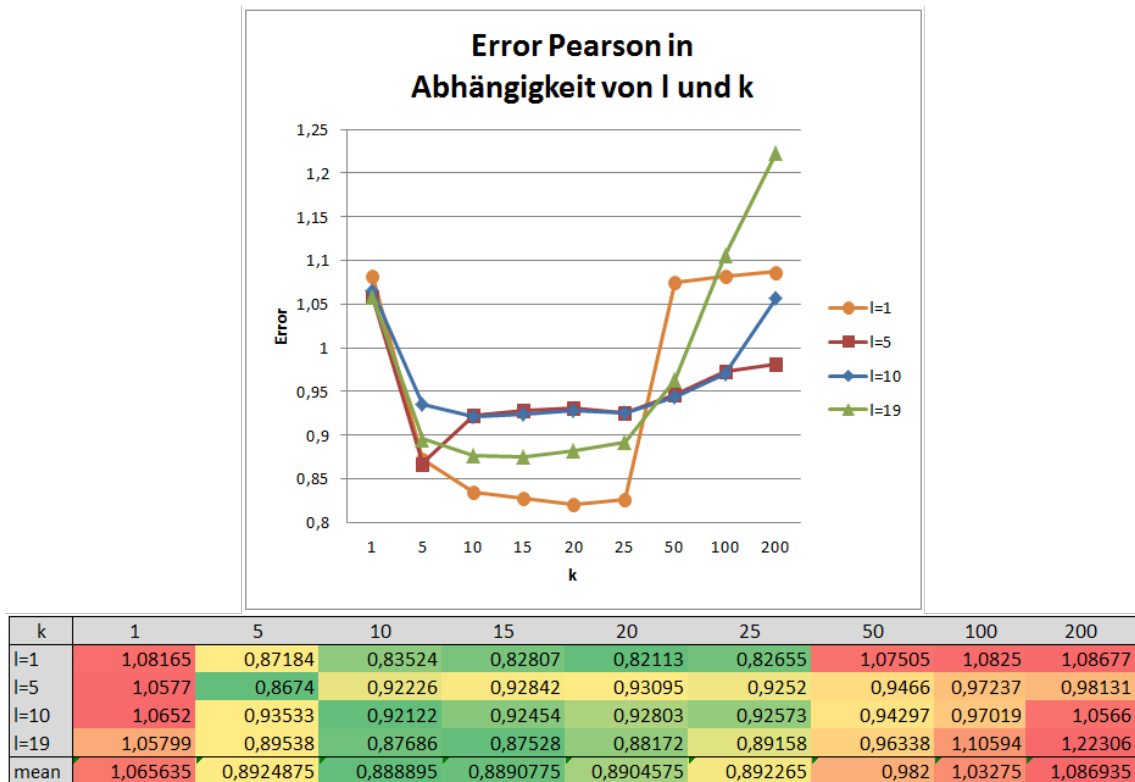
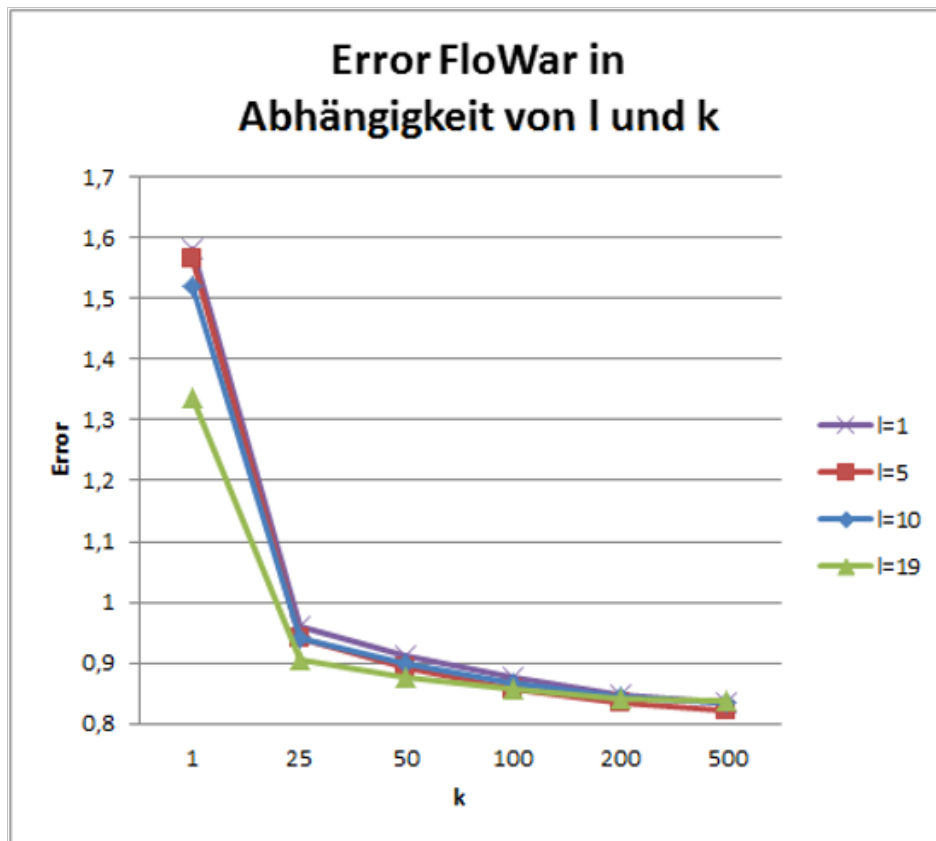
4.1.2 Wahl von  $k$  bei den kNN im Pearson Algorithmus

Abbildung 2: MAE im Pearson Algorithmus in Abhängigkeit der Parameter

Eine Wahl von  $k = 10$  nächsten Nachbarn optimiert den Pearson Algorithmus. Im Szenario  $l = 1$  zeigt der Algorithmus seine Stärken. Wenn viele Informationen vorhanden sind, können mit dem Pearson Algorithmus gute Bewertungen vorhergesagt werden. Wenn man die mittleren Szenarien  $l = 5$  und  $l = 10$  mit  $l = 19$  vergleicht, fällt auf, dass das letzte Szenario am besten abschneidet. Da hier 19 Fehler berechnet und gemittelt werden. Im Fall  $l = 5$  werden nur fünf Fälle verglichen, dadurch entsteht im Mittel eine größere Abweichung zwischen der Vorhersage und der tatsächlich abgegeben Bewertung.

4.1.3 Wahl von  $k$  bei den kNN im Floyd Warshall Algorithmus

k	1	25	50	100	200	500
l=1	1,58218	0,96124	0,91318	0,87614	0,84624	0,83418
l=5	1,56422	0,94203	0,89428	0,85883	0,83381	0,82344
l=10	1,51881	0,94071	0,89796	0,8662	0,84417	0,8355
l=19	1,33583	0,904	0,87606	0,85691	0,84257	0,83788
mean	1,50026	0,936995	0,89537	0,86452	0,8416975	0,83275

Abbildung 3: MAE im Floyd Warshall Algorithmus in Abhängigkeit der Parameter

Man sieht in Abbildung 3, dass Floyd-Warshall bei großem  $k$  ( $=500$ ) am besten funktioniert. Es wird das Minimum aus den  $k$  nächsten Nachbarn und den  $n$  nächsten Nachbarn, die den aktuell zu bewertenden Film überhaupt bewertet haben, genommen, um das zu erwartete Rating zu ermitteln. Dies tendiert gegen den Durchschnittswert aller User, die die fehlenden Items bewertet haben. Somit ist der FW-Algorithmus fast gleichzusetzen mit dem Itemmean.

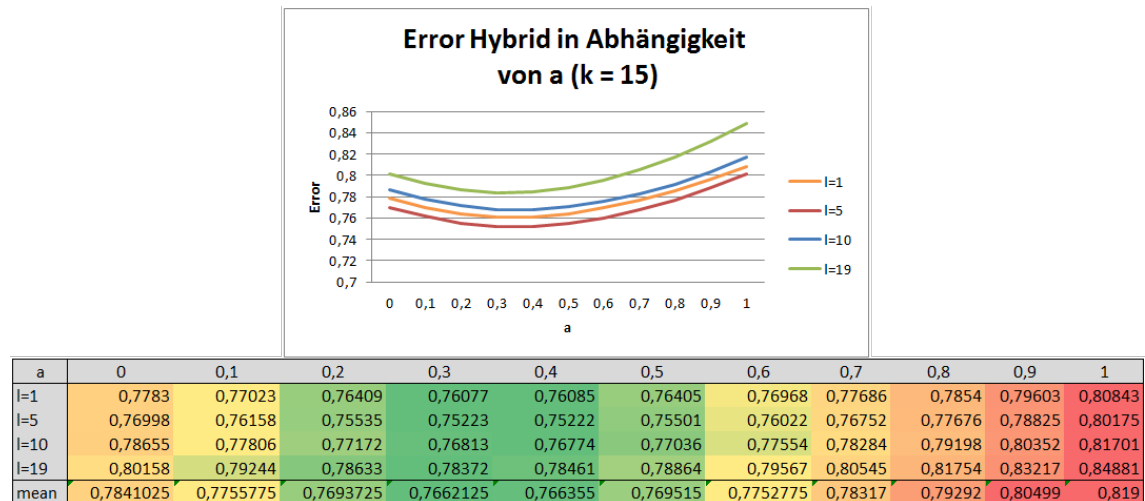
4.1.4 Wahl von  $a$  beim Hybrid Algorithmus

Abbildung 4: MAE im Hybrid Algorithmus in Abhängigkeit der Parameter

Ein Verhältnis aus 70% Euklid und 30% Slope One minimiert den Fehler. Während der Euklid Algorithmus nur ähnliche Nachbarn untersucht, hilft das Wissen aus dem Item-basierten Algorithmus Slope One den MAE weiter zu minimieren.

## 4.2 Auswertung der Fehlerberechnung

Neben den sechs genannten Algorithmen (s. Liste der Algorithmen) habe ich noch drei weitere Algorithmen für den Vergleichstest implementiert. Der einfachste, Random, erzeugt einen randomisierten Float zwischen  $[1, 5] \subset \mathbb{R}$ . Usermean  $\bar{R}_u^{user}$  ermittelt den Durchschnitt des Users, der gerade betrachtet wird. Itemmean  $\bar{R}_i^{item}$  berechnet die durchschnittliche Bewertung aller User des Items, das gerade im Fokus ist (s. Gleichung 1). Gegen diese drei simplen Algorithmen werden die MAE der sechs oben beschriebenen Funktionen gemessen. Wobei wieder alle vier Szenarien bezüglich des Parameters  $l$  betrachtet werden.

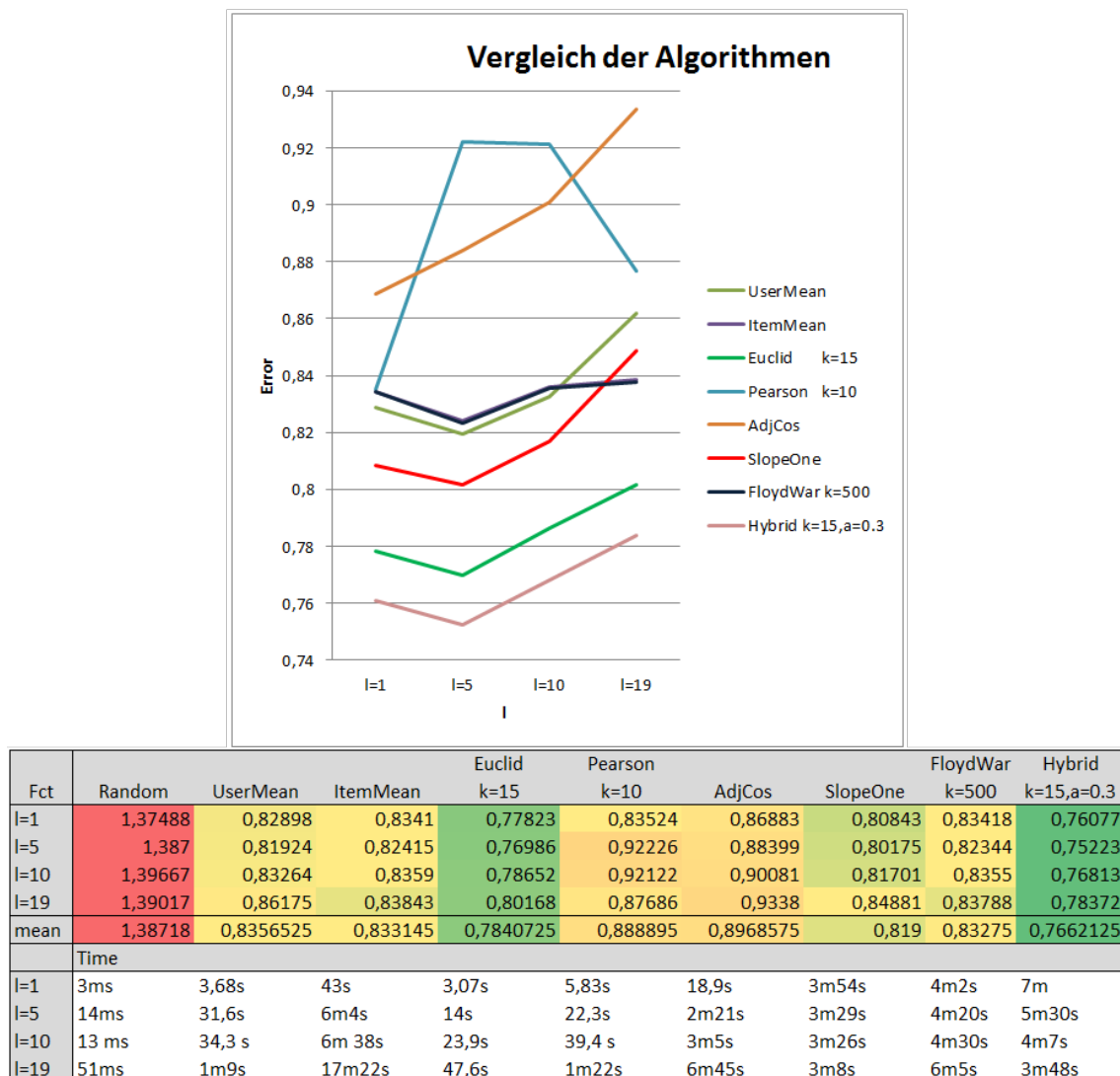


Abbildung 5: MAE Vergleich aller Algorithmen

Die Laufzeiten dienen als zusätzlicher Vergleich zwischen den Algorithmen. Die Daten sind in einer Hashmap gespeichert und es wird nur ein Kern der CPU ausgelastet. Durch andere Implementierungen und andere PC-Setups können die Zeiten variieren.

Usermean und Itemmean erzielen im Vergleich kein schlechtes Ergebnis, zählen sogar laut MAE mit zu den besten Algorithmen. Usermean ist für Vorhersagen nicht brauchbar, da jedes unbekannte Item das gleiche Rating bekommt. Einen Einsatz findet dieser Algorithmus nur, wenn kein anderer User diesen Film bisher bewertet hat. Bei Itemmean können zwar allgemein beliebte Filme vorgeschlagen werden, die Vorlieben des Users werden aber komplett ignoriert. Pearson erkennt sogar Ähnlichkeiten in relativen Abweichungen und kommt



somit mit einem geringem  $k$  im  $k$ -Nearest-Neighbors aus. Das genaue Gegenteil ist bei Floyd-Warshall der Fall. Der sehr groß eingestellte Parameter  $k$  führt dazu, dass FW ähnlich wie Itemmean die bekannten Informationen über den User vernachlässigt. Da mit dem Itemmean eine bessere durchschnittliche Abweichung erzielt wird, als über die entdeckten Pfade zu anderen User. Adjusted Cosine Similarity ist in diesem Test nicht sehr erfolgreich. Obwohl es mit dem gleichen Score arbeitet wie der PCC und dort alle bekannten Informationen über die zwei Items verwendet, können keine guten Vorhersagen erzielt werden. Bessere Vorhersagen erzielt man mit dem Item-basierten Algorithmus Slope One. Die absolute durchschnittliche Differenz zwischen den Items verwendet alle verfügbaren Informationen aus dem Datensatz. Neben den Differenzen zwischen allen Items muss auch noch eine Frequenzmatrix gespeichert werden. Neue Informationen können dadurch ohne großen Rechenaufwand in die Abstandsmatrix eingearbeitet werden. Die Vorberechnungen im Slope One machen allerdings nur ca. 30 Sekunden der gesamten Zeit aus. Damit braucht der Algorithmus auch recht lange, um Vorhersagen für viele Nutzer gleichzeitig zu berechnen. Der Algorithmus mit dem euklidischen Abstand und Strafe schneidet sehr gut ab und hat dazu sehr kurze Berechnungszeiten. Ein noch besseres Ergebnis liefert das Hybrid Verfahren. Das gemittelte Rating zu 70% aus dem besten User-basierten (Euklid mit Strafe) und zu 30% aus dem besten Item-basierten Algorithmus (Slope One) minimiert in diesem Test den MAE. Jedoch verlängert Slope One die Laufzeit immens.

Low-rank-matrix		6,3% matrix entries							
Fct	Random	UserMean	ItemMean	Euclid (k=15)	Pearson(k=10)	AdjCos	SlopeOne	ydWar(k=5)	Hybrid
l=1	1,10586	0,39276	0,5966	0,33236	0,38796	0,37396	0,38757	0,5342	0,32259

Low-rank-matrix		30% matrix entries							
Fct	Random	UserMean	ItemMean	Euclid (k=15)	Pearson(k=10)	AdjCos	SlopeOne	ydWar(k=5)	Hybrid
l=1	1,08811	0,35599	1,11413	0,22248	0,28867	0,44678	0,35599	0,65072	0,25614

Abbildung 6: MAE Vergleich mit einer Matrix mit niedrigem Rang

Bei der Konstruktion mit niedrigem Rang erkennt man, dass die User-basierten Algorithmen richtig gut werden. Das ist aber auch zu erwarten, denn die User ähneln sich sehr stark, da sie alle aus einer Kombination aus dreißig Basis-Usern entstanden sind. Die Bewertungen wurden zufällig erstellt, dadurch können auch keine Ähnlichkeiten zwischen den Filmen entstehen. Dies wertet die Item-basierten Algorithmen zusätzlich ab. Sind viele Informationen bekannt, ändert sich nicht viel an der Reihenfolge der Algorithmen.

Einen Vergleich zu der realen Datenbank von MovieLens kann man nur mit Vorsicht tätigen. Wenn man die Histogramme der beiden Matrizen betrachtet, wird klar, dass die Konstruktion der Matrix mit niedrigem Rang nicht sehr nah an die Verteilung der MovieLens-Daten heran kommt.

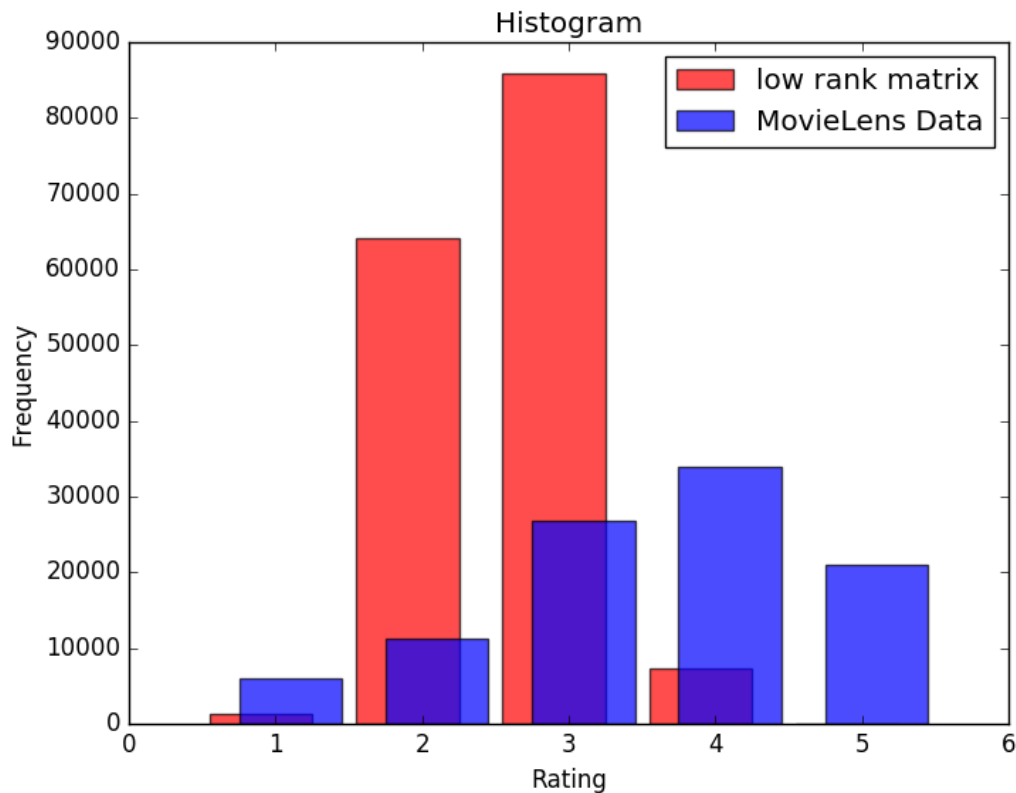


Abbildung 7: Histogramm Vergleich

Beim Versuch die Bewertungen gleichmäßiger zu verteilen, zerstört man die lineare Abhängigkeit der Zeilen. Doch genau diese Eigenschaft der Bewertungen wollte ich mit diesem Test überprüfen. Überraschend ist die Verteilung der MovieLens-Daten. Ich hätte erwartet, dass die 1 häufiger von enttäuschten Usern als negative Bewertung benutzt wird.

Fct	Low-rank-matrix		10% matrix entries		without normalization				
	Random	UserMean	ItemMean	Euclid (k=15)	Pearson(k=10)	AdjCos	SlopeOne	ydWar(k=5)	Hybrid
l=1	259,59216	17,72952	31,4272	15,65126	15,73972	250,71992	17,05056	16,2612	9,65739

Abbildung 8: MAE Vergleich ohne Normalisierung

In den Daten in Abbildung 8 wurde bewusst vermieden, die Bewertungen wieder in den Wertebereich 1, 2, 3, 4, 5 zu normieren. Dadurch wird die lineare Abhängigkeit der Zeilen nicht zerstört. Diese erzielten Fehler können deshalb nicht mit den anderen Szenarien verglichen werden. Was man aber erkennen kann, dass die User-basierten Verfahren sehr viel besser abschneiden, als der Item-basierte Algorithmus Adjusted Cosine Similarity. Slope One kann auch hier mit den vollen Informationen der Matrix ein sehr gutes Ergebnis erzielen.

## 5 Schlussfolgerung

Die Entscheidung, welchen Algorithmus man verwenden sollte, liegt an vielen Faktoren. Hat man sehr viele User auf wenige Items, trumpfen Item-basierte Algorithmen auf. Die Item-Item-Matrix der Distanzen zwischen zwei Items hat in diesem Fall kleinere Dimensionen. User-basierte Verfahren müssen erst die Abstände zu allen anderen Nutzern berechnen, um die ähnlichsten Nutzer zu finden. Ist das Verhältnis User zu Items anders herum, können User-basierte Algorithmen schneller sein. Alle User zu betrachten ist unter diesen Umständen eventuell schneller oder nicht so speicherintensiv wie eine sehr große Item-Item-Matrix. Generell wird eine Kombination aus mehreren Verfahren in der Praxis die besten Ergebnisse erzielen. Der Hybrid Algorithmus erzielt in diesem Test durchweg die besten Ergebnisse. Weitere Kombinationen sind durchaus vorstellbar, wie eine Linearkombination aus mehreren Verfahren, oder Abfragen ob viel oder wenig über den User bekannt ist und dementsprechend die Stärken der Algorithmen gezielt auszunutzen.

Zudem haben alle Algorithmen Schwierigkeiten damit, einem neuem User Vorschläge zu bereiten, da sein persönlicher Geschmack noch nicht bekannt ist. Hier kann der Itemmean seinen Nutzen zeigen, dieser kann allgemein beliebte Items vorschlagen. Eine erfolgreiche Methode, um neue Items zu integrieren, könnte sein über Tags, die z.B. Vertreter des gleichen Genres verlinken, die Items zu Klassifizieren. Vor allem professionell erstellte Tags haben enormes Potenzial, ähnliche Filme auf verschiedene Eigenschaften zu kategorisieren. Das erfordert eine Art Expertensystem, wie Pandora es nutzt.

Demographische Informationen aus dem Userprofil bieten weitere Informationen um Ähnlichkeiten zwischen Usern zu erkennen.

Ein überraschendes Ergebnis, in diesem Test, konnte der Algorithmus mit dem euklidischem Abstand erzielen, ein simples Verfahren, dass leicht zu verstehen ist. Mit der Strafe als Anpassung kann der Fehler sehr gut minimiert werden. In Verbindung mit der kurzen Laufzeit hat diese Methode eine sehr gute Gesamt-Performance. Die aufwändigeren Verfahren brachten keine Verbesserung in der Fehlerminimierung. Das hybride Modell minimiert weiter den Fehler.

Die Idee aus der Graphentheorie, kürzeste Wege über mehrere ähnliche Nachbarn zu gehen, um weiter entfernte Nachbarn näher zu bringen, hat in diesem Szenario nicht funktioniert. Die Fehlerminimierung ist nicht sehr gut und der Parameter  $k$  ist, im besten Fall, viel zu groß. Damit sind die meisten, erzeugten Ratings gleich zu setzen mit dem Itemmean.

Abschließend lässt sich keine große Aussage aus dem zweiten Test mit der zufällig erstellten Matrix erzielen. Die User-basierten Algorithmen sind per Konstruktion sehr gut. Die Item-basierten Verfahren können durch die zufälligen Bewertungen keine Ähnlichkeiten zwischen den Items erkennen. Zusätzlich ist die Verteilung in der Konstruktion weit entfernt von der realen Verteilung der

MovieLens-Daten.

## A Anhang: Quellcode

Da ich vor Beginn dieser Arbeit noch nicht mit Python gearbeitet habe und mit der Syntax noch nicht vertraut war, habe ich mich dazu entschlossen als Start meines Quellcodes, die Recommender Klasse von Ron Zacharski zu nutzen. In seinem Buch „A Programmer’s Guide to Data Mining: The Ancient Art of the Numerati“ [Zar15] stellt er neben den Erklärungen auch Python Quellcode<sup>5</sup> zur Verfügung. In Chapter 3 befindet sich die Python Datei "recommender3.py". Darin enthalten sind der Import der Movie-Lens-Daten, der Pearson und der Slope One Algorithmus. Des weiteren befindet sich die Funktion `computeSimilarity(band1, band2, userRatings)` in der Datei "cosineSimilarity.py". Diese Funktion war meine Ausgangslage, um das Verfahren Adjusted Cosine Similarity zu implementieren. Den Code habe ich an meine Bedürfnisse angepasst, damit ich den automatisierten Test mit den Testdaten ausführen konnte. Für diese Zwecke habe ich die Testklasse erstellt. Mit der Funktion `testit(self, fct, k = 10, a = 0.5)` kann man den Testlauf für eine der Verfahren starten. Der Parameter  $k$  entscheidet die Anzahl der Nachbarn,  $a$  wird als Parameter für den Hybrid Modus benötigt.

$$fct \in \{\text{'random'}, \text{'itemmean'}, \text{'usermean'}, \text{'euclid'}, \text{'pearson'}, \text{'slopeone'}, \text{'floydwarshall'}, \text{'lusgus'}, \text{'adjcos'}\} \quad (20)$$

Der Quellcode und der MovieLens-Datensatz ist online in meinem Github Repository<sup>6</sup> zu finden.

### A.1 Übersicht aller Funktionen

```
""" Packages in use """
import codecs
import timeit
import random
from math import sqrt
from math import fabs
import numpy as np
import matplotlib.pyplot as plt
import pandas

class recommender: #(page 27)
    """this class is based on the recommender class of
    Guide2DataMining"""
```

<sup>5</sup><https://github.com/zacharski/pg2dm-python>

<sup>6</sup><https://github.com/kralle71/Collaborative-Filtering>

```

def __init__(self, data, k=1, n=5, metric='pearson'):
    """initialize recommender"""

def convertProductID2name(self, id):
    """Given product id number return product name"""

def userRatings(self, id, n):
    """Return n top ratings for user with id"""

def showUserTopItems(self, user, n):
    """ show top n items for user"""

def loadMovieLens(self, path=''):
    """import MovieLens Dataset"""

def computeDeviations(self):
    """Precalculations for SlopeOne"""

def slopeOneRecommendations(self, user, item='0'):
    """SlopeOne Recommendations"""

def sloOne(self, user, item):
    """sloOne is used to speed up the test"""

def computeUserAverages(self):
    """Computes user averages"""

def computeItemAverages(self):
    """Computes item averages"""

def computeSimilarity(self, item1, item2):
    """Computes similarity for adjcos"""

def normalizeRating(self, rating, minRating=1, maxRating=5):
    """normalizes to range [-1,1]"""

def denormalizeRating(self, rating, minRating=1, maxRating=5):
    """transforms range [-1,1] to range [1,5]"""

def adjcos(self, user, diffItem):
    """Adjusted Cosine Similarity Recommendations"""

```

```

def pearson(self, rating1, rating2):
    """computes pearson score between rating1 and
    rating2"""

def computeNearestNeighbor(self, username, rateItem = '
0'):
    """creates a sorted list of users based on their
    distance to username"""

def computeAdjacentMatrix(self):
    """Precalculations for FW"""
    """computes the weights (euclid distance) between
    edges of the graph"""

def floydwarshall(self):
    """computes shortest paths"""
    """https://jlmedina123.wordpress.com/2014/05/17/
    floyd-warshall-algorithm-in-python/"""

def flowar(self, user, k, item):
    """FloydWarshall Recommendations"""

def usermean(self, user):
    """Usermean Recommendations"""

def itemmean(self, item):
    """Itemmean Recommendations"""

def euclDist(self, rating1, rating2):
    """computes euclid distance between rating1 and
    rating2"""

def recommend(self, metric, user, k, rateItem = '0', n
=30):
    """Pearson or Euclid Recommendations (based on
    parameter metric)"""

class tester: #(page 41)

    def __init__(self, r, data):
        """initialize tester"""

```

```

def loadMovieLens(self , path=''):

def testinput(self , fct , k , a):
    """input test for parameters given"""

def testit(self , fct , k=10 , a=0.5):
    """toolbox for the test of different algorithms"""

"""Load MovieLens-Dataset"""
l = 1
path = 'PathtoData\Data'

r = recommender(0)
r.loadMovieLens(path+'\ml-L='+str(l)+'/')
r.computeUserAverages()

t = tester(r,0)
t.loadMovieLens(path+'\ml-L='+str(l)+'/')

```



## A.2 Quellcode

```
# coding: utf-8

# In[ ]:

import codecs
import timeit
import random
from math import sqrt
from math import fabs
import numpy as np
import matplotlib.pyplot as plt
import pandas

# # recommender class

# In[ ]:

class recommender:
    """this class is based on the recommender class of
    Guide2DataMining"""

    def __init__(self, data, k=1, n=5, metric='pearson'):
        """initialize recommender currently, if data is
        dictionary the recommender is initialized to it.
        For all other data types of data, no initialization
        occurs k is the k value for k nearest neighbor
        metric is which distance formula to use n is the
        maximum number of recommendations to make"""

        self.k = k
        self.n = n
        self.username2id = {}
        self.userid2name = {}
        self.productid2name = {}
```

```

# The following variables are used for the
  different algorithms
self.frequencies = {}
self.deviations = {}
self.adjMatrix = {}
self.floydwarDist = {}
self.userAvg = {}
self.soRec={}
self.euclMatrix = {}
self.penalty=False
self.normalized=False

# for some reason I want to save the name of the
  metric
self.metric = metric
if self.metric == 'pearson':
    self.fn = self.pearson

# if data is dictionary set recommender data to it
if type(data).__name__ == 'dict':
    self.data = data

def convertProductID2name(self, id):
    """Given product id number return product name"""
    if id in self.productid2name:
        return self.productid2name[id]
    else:
        return id

def userRatings(self, id, n):
    """Return n top ratings for user with id"""
    print ("Ratings_for_" + self.userid2name[id])
    ratings = self.data[id]
    print(len(ratings))
    ratings = list(ratings.items())[:n]
    ratings = [(self.convertProductID2name(k), v) for (
        k, v) in ratings]
    # finally sort and return
    ratings.sort(key=lambda artistTuple: artistTuple
        [1], reverse = True)
    for rating in ratings:

```

```

        print("%s\t%i" % (rating[0], rating[1]))

def showUserTopItems(self, user, n):
    """ show top n items for user """
    items = list(self.data[user].items())
    items.sort(key=lambda itemTuple: itemTuple[1],
               reverse=True)
    for i in range(n):
        print("%s\t%i" % (self.convertProductID2name(
            items[i][0]), items[i][1]))

def loadMovieLens(self, path=''):
    """import MovieLens Dataset"""
    self.data = {}

    # first load movie ratings

    i = 0

    # First load book ratings into self.data

    #f = codecs.open(path + "u.data", 'r', 'utf8')
    #f = codecs.open(path + "u.data", 'r', 'ascii')
    f = codecs.open(path + "ua.base", 'r', 'ascii')
    # f = open(path + "u.data")
    for line in f:
        i += 1
        #separate line into fields
        fields = line.split('\t')
        user = fields[0]
        movie = fields[1]
        rating = int(fields[2].strip().strip(''))
        if user in self.data:
            currentRatings = self.data[user]
        else:
            currentRatings = {}
        currentRatings[movie] = rating
        self.data[user] = currentRatings
    f.close()

    # Now load movie into self.productid2name
    # the file u.item contains movie id, title, release

```

```

        date among
# other fields

#f = codecs.open(path + "u.item", 'r', 'utf8')
f = codecs.open(path + "u.item", 'r', 'iso8859-1',
               'ignore')
#f = open(path + "u.item")
for line in f:
    i += 1
    #separate line into fields
    fields = line.split('|')
    mid = fields[0].strip()
    title = fields[1].strip()
    self.productid2name[mid] = title
f.close()

# Now load user info into both self.userid2name
# and self.username2id

#f = codecs.open(path + "u.user", 'r', 'utf8')
f = open(path + "u.user")
for line in f:
    i += 1
    fields = line.split('|')
    userid = fields[0].strip(' "')
    self.userid2name[userid] = line
    self.username2id[line] = userid
f.close()
print(i)

def computeDeviations(self):
    """Precalculations for SlopeOne"""
    # for each person in the data:
    # get their ratings
    for ratings in self.data.values():
        # for each item & rating in that set of ratings
        :
        for (item, rating) in ratings.items():
            self.frequencies.setdefault(item, {})
            self.deviations.setdefault(item, {})
            # for each item2 & rating2 in that set of

```

```

        ratings:
    for (item2, rating2) in ratings.items():
        if item != item2:
            # add the difference between the
            # ratings to our
            # computation
            self.frequencies[item].setdefault(
                item2, 0)
            self.deviations[item].setdefault(
                item2, 0.0)
            self.frequencies[item][item2] += 1
            self.deviations[item][item2] +=
                rating - rating2

    for (item, ratings) in self.deviations.items():
        for item2 in ratings:
            ratings[item2] /= self.frequencies[item][
                item2]

def slopeOneRecommendations(self, user, item='0'):
    """SlopeOne Recommendations"""
    recommendations = {}
    frequencies = {}
    # for every item and rating in the user's
    # recommendations
    for (userItem, userRating) in self.data[user].items
        ():
        # for every item in our dataset that the user
        # didn't rate
        for (diffItem, diffRatings) in self.deviations.
            items():
            if diffItem not in self.data[user] and
                userItem in self.
                deviations[diffItem]:
                freq = self.frequencies[diffItem][
                    userItem]
                recommendations.setdefault(diffItem,
                    0.0)
                frequencies.setdefault(diffItem, 0)
                # add to the running sum representing
                # the numerator
                # of the formula
                recommendations[diffItem] += (

```

```

        diffRatings[userItem] +
                                userRating) *
                                freq
        # keep a running sum of the frequency
        # of diffitem
        frequencies[diffItem] += freq

recommendations = [(k, v / frequencies[k])
                    for (k, v) in recommendations.
                        items()]
        # finally sort and return
recommendations.sort(key=lambda artistTuple:
                    artistTuple[1],
                        reverse = True)
# Return some recommendations for the user, or a
# particular rating of an item for the user
if item == '0':
    return recommendations[:50]
else:
    return recommendations[item]

def sloOne(self, user, item):
    """sloOne is used to speed up the test"""
    if user not in self.soRec.keys():
        # save the recommendations for any user in
        # soRec
        self.soRec[user] = self.slopeOneRecommendations
            (user)
    for i in range(len(self.soRec[user])):
        # look up the specific item and return the
        # value
        if self.soRec[user][i][0]==item:
            result = self.soRec[user][i][1]
            return result
    return self.userAvg[user]

def computeUserAverages(self):
    """Computes user averages"""
    results = {}
    for (key, ratings) in self.data.items():
        results[key] = float(sum(ratings.values())) /
            len(ratings.values())
    self.userAvg = results

```

```

def computeItemAverages(self):
    """Computes item averages"""
    sumRating = {}
    freqRating = {}
    results = {}
    for (key, ratings) in self.data.items():
        for (k) in ratings:
            if k not in sumRating:
                sumRating[k] = ratings[k]
                freqRating[k] = 1
            else:
                sumRating[k] += ratings[k]
                freqRating[k] += 1
    for key in sumRating:
        results[key] = float(sumRating[key])/freqRating[key]
    return results

def computeSimilarity(self, item1, item2):
    """Computes similarity for adjcos"""
    num = 0
    dem1 = 0
    dem2 = 0
    #for every user check if they rated both items and
    #count them for the similarity
    for (user, ratings) in self.data.items():
        if item1 in ratings and item2 in ratings:
            avg = self.userAvg[user]
            num += (ratings[item1]-avg) * (ratings[item2]-avg)
            dem1 += (ratings[item1]-avg)**2
            dem2 += (ratings[item2]-avg)**2
    if dem1*dem2!=0:
        return num / (sqrt(dem1) * sqrt(dem2))
    return -3

def normalizeRating(self, rating, minRating=1, maxRating=5):
    """normalizes to range [-1,1]"""
    return float((2*(rating-minRating)-(maxRating-minRating)))/(maxRating-minRating)

```

```

def denormalizeRating(self, rating, minRating=1, maxRating=5):
    """transforms range [-1,1] to range [1,5]"""
    return (float((rating+1)*(maxRating-minRating))/2)
        +minRating

```

```

def adjcos(self, user, diffItem):
    """Adjusted Cosine Similarity Recommendations"""
    # calculates the rating of diffItem for User given
    num = 0
    dem = 0
    if diffItem not in self.data[user].keys():
        for (uItem, uRating) in self.data[user].items():
            :
            # for all items of the user calculate the
            # similarity to the diffItem
            similarity = self.computeSimilarity(
                diffItem, uItem)
            if similarity != -3: # avoid dividing with
                0, if noone rated both Items
                nRating = self.normalizeRating(uRating)
                #calculate the rating
                num += similarity*nRating
                dem += fabs(similarity)
        if dem == 0:
            return self.userAvg[user]
        else:
            return float(self.denormalizeRating(float(
                num/dem)))
    else:
        print("already rated with %10.3f" % (self.data
            [user][diffItem]))

```

```

def pearson(self, rating1, rating2):
    """computes pearson score between rating1 and
        rating2"""
    sum_xy = 0
    sum_x = 0

```



```

sum_y = 0
sum_x2 = 0
sum_y2 = 0
n = 0
for key in rating1:
    if key in rating2:
        n += 1
        x = rating1[key]
        y = rating2[key]
        sum_xy += x * y
        sum_x += x
        sum_y += y
        sum_x2 += pow(x, 2)
        sum_y2 += pow(y, 2)
if n == 0:
    return 0
# now compute denominator
denominator = sqrt(sum_x2 - pow(sum_x, 2) / n) *
               sqrt(sum_y2 - pow(sum_y, 2)
               / n)
if denominator == 0:
    return 0
else:
    return (sum_xy - (sum_x * sum_y) / n) /
           denominator

def computeNearestNeighbor(self, username, rateItem = '
0'):
    """creates a sorted list of users based on their
    distance to username"""
    distances = []
    for instance in self.data:
        if instance != username:
            if rateItem == '0':
                #compute nearest neighbors
                distance = self.fn(self.data[username],
                                   self.data[instance])
                distances.append((instance, distance))
            else:
                #compute nearest neighbors, who rated
                the given item
                if rateItem in self.data[instance]:

```

```

        distance = self.fn(self.data[
            username],
            self.data[instance])
        distances.append((instance,
            distance))
# sort based on distance — closest first
if self.fn == self.pearson:
    distances.sort(key=lambda artistTuple:
        artistTuple[1],
        reverse=True)
elif self.fn == self.euclDist:
    distances.sort(key=lambda artistTuple:
        artistTuple[1])
return distances

def computeAdjacentMatrix(self):
    """Precalculations for FW"""
    """computes the weights (euclid distance) between
        edges of the graph"""
    distlist = []
    (r.penalty, r.normalized)=(True, False)
    self.fn = self.euclDist
    for (user) in self.data:
        self.adjMatrix[user]={}
        distlist = self.computeNearestNeighbor(user)
        for (u,i) in distlist:
            self.adjMatrix[user][u]=i

def floydwarshall(self):
    """https://jlmedina123.wordpress.com/2014/05/17/
        floyd-warshall-algorithm-in-python/"""
    # Initialize dist and pred:
    # copy graph into dist, but add infinite where
        there is
    # no edge, and 0 in the diagonal
    self.computeAdjacentMatrix()
    graph = self.adjMatrix
    dist = {}
    pred = {}
    for u in graph:
        dist[u] = {}
        pred[u] = {}
        for v in graph:

```

```

        dist[u][v] = 1000
        pred[u][v] = -1
    dist[u][u] = 0
    for neighbor in graph[u]:
        dist[u][neighbor] = graph[u][neighbor]
        pred[u][neighbor] = u

    for t in graph:
        # given dist u to v, check if path u - t - v is
        # shorter
        for u in graph:
            for v in graph:
                newdist = dist[u][t] + dist[t][v]
                if newdist < dist[u][v]:
                    dist[u][v] = newdist
                    pred[u][v] = pred[t][v] # route new
                    # path through t

    self.floydwarDist = dist

def flowar(self, user, k, item):
    """FloydWarshall Recommendations"""
    self.k = k
    # find nearest neighbors
    dist = list(r.floydwarDist[user].items())
    dist.sort(key=lambda ks: ks[1], reverse = True)
    n = 0
    nearest=[]
    for (distItem) in dist:
        if item in r.data[distItem[0]]:
            nearest.append((distItem))
            n += 1
        if n == k:
            break
    self.k = min(len(nearest), k)
    if self.k ==0:
        return self.userAvg[user]
    recommendation = 0
    totalDistance = 0.0
    for i in range(self.k):
        if item in self.data[nearest[i][0]]:
            totalDistance += 1
    # now iterate through the k nearest neighbors

```

```

# accumulating their ratings
if totalDistance == 0:
    return self.userAvg[user]
for i in range(self.k):
    if item in self.data[nearest[i][0]]:
        # compute slice of pie
        weight = 1 / totalDistance
        # get the name of the person
        name = nearest[i][0]
        # get the ratings for this person
        neighborRatings = self.data[name]
        # get the name of the person
        # now find bands neighbor rated that user didn't
        recommendation += neighborRatings[item] *
            weight
return recommendation

def usermean(self, user):
    """Usermean Recommendations"""
    self.computeUserAverages()
    return self.userAvg[user]

def itemmean(self, item):
    """Itemmean Recommendations"""
    avg = self.computeItemAverages()
    if item not in avg:
        return 0
    else:
        return avg[item]

def euclDist(self, rating1, rating2):
    """computes euclid distance between rating1 and
    rating2"""
    sum = 0
    freq = 0
    nooverlap = True
    for key in rating1:
        if key in rating2:
            sum += (rating1[key]-rating2[key])**2
            freq +=1
            nooverlap = False
        elif self.penalty:

```

```

        #add penalty for items in rating1 but not
        #in rating2
        sum += ((rating1[key]-3))**2
        freq +=1
    if freq == 0 or nooverlap:
        return 1000
    else:
        result = sqrt(sum)
        if self.normalized:
            result /= freq
        return result

def recommend(self, metric, user, k, rateItem = '0', n
=30):
    """Pearson or Euclid Recommendations (based on
    parameter metric)"""
    """Give list of recommendations. rateItem.default
    is set to '0', which means we will get the n
    best
    recommendations. If the parameter is given, it will
    rate just this item."""

    self.k = k
    self.n = n
    recommendations = {}
    recommendation = 0
    if metric == 'pearson':
        self.fn = self.pearson
    elif metric == 'euclid':
        self.fn = self.euclDist

    # first get list of users ordered by nearness
    if rateItem == '0':
        nearest = self.computeNearestNeighbor(user)
    else:
        nearest = self.computeNearestNeighbor(user,
        rateItem)
    # if fewer neighbors than k rated the item
    self.k = min(len(nearest), k)

    #
    # now get the ratings for the user

```

```

#
userRatings = self.data[user]
#
# determine the total distance
totalDistance = 0.0
for i in range(self.k):
    if self.fn == self.pearson:
        totalDistance += nearest[i][1] #weighted
        mean
    elif self.fn == self.euclDist:
        totalDistance += 1 #classic mean
# now iterate through the k nearest neighbors
# accumulating their ratings
if totalDistance == 0:
    return self.userAvg[user];
for i in range(self.k):
    # compute slice of pie
    if self.fn == self.pearson:
        weight = nearest[i][1] / totalDistance
    elif self.fn == self.euclDist:
        weight = 1 / totalDistance
    # get the name of the person
    name = nearest[i][0]
    # get the ratings for this person
    neighborRatings = self.data[name]
    # get the name of the person
    # now find bands neighbor rated that user didn't
    if rateItem == '0':
        for artist in neighborRatings:
            if not artist in userRatings:
                if artist not in recommendations:
                    recommendations[artist] =
                        neighborRatings[artist] *
                        weight
                else:
                    recommendations[artist] =
                        recommendations[artist] +
                        neighborRatings[artist] *
                        weight
    else:
        recommendation += neighborRatings[rateItem]

```

```

        * weight

    if rateItem == '0':
        # now make list from dictionary and only get the
        # first n items
        recommendations = list(recommendations.items())
                           [:self.n]
        # recommendations = [(self.convertProductID2name(k),
        #                      v)
        #                    for (k, v) in recommendations]
        # finally sort and return
        recommendations.sort(key=lambda artistTuple:
                              artistTuple[1],
                              reverse = True)
        return recommendations
    else:
        return recommendation

# # test class

# In[ ]:

class tester:

    def __init__(self, r, data):
        """ initialize tester """

        self.username2id = {}
        self.userid2name = {}
        self.productid2name = {}

        if type(data).__name__ == 'dict':
            self.data = data

    def loadMovieLens(self, path=''):
        self.data = {}

        # first load movie ratings

```

```

i = 0

# First load book ratings into self.data

#f = codecs.open(path + "u.data", 'r', 'utf8')
#f = codecs.open(path + "u.data", 'r', 'ascii')
f = codecs.open(path + "ua.test", 'r', 'ascii')
# f = open(path + "u.data")
for line in f:
    i += 1
    #separate line into fields
    fields = line.split('\t')
    user = fields[0]
    movie = fields[1]
    rating = int(fields[2].strip().strip('\"'))
    if user in self.data:
        currentRatings = self.data[user]
    else:
        currentRatings = {}
        currentRatings[movie] = rating
    self.data[user] = currentRatings
f.close()

# Now load movie into self.productid2name
# the file u.item contains movie id, title, release
#   date among
# other fields

#f = codecs.open(path + "u.item", 'r', 'utf8')
f = codecs.open(path + "u.item", 'r', 'iso8859-1',
                'ignore')
#f = open(path + "u.item")
for line in f:
    i += 1
    #separate line into fields
    fields = line.split('|')
    mid = fields[0].strip()
    title = fields[1].strip()
    self.productid2name[mid] = title
f.close()

# Now load user info into both self.userid2name
# and self.username2id

```



```

#f = codecs.open(path + "u.user", 'r', 'utf8')
f = open(path + "u.user")
for line in f:
    i += 1
    fields = line.split('|')
    userid = fields[0].strip(' ')
    self.userid2name[userid] = line
    self.username2id[line] = userid
f.close()
print(i)

def testinput(self, fct, k, a):
    """input test for parameters given"""

    fctlist = ['adjcos', 'hybrid', 'floydwarshall', 'pearson', 'slopeone', 'euclid', 'usermean', 'itemmean', 'random']

    try: #k is integer and positiv
        k = int(k)
        if k < 0:
            print("\nPlease insert a positive k amount of neighbors")
            return False
    except ValueError:
        print("\nPlease only use integers")
        return False

    try: #a is float and in range [0,1]
        a = float(a)
        if a < 0 or a > 1:
            print("\nPlease use the paramater a in range [0,1]")
            return False
    except ValueError:
        print("\nPlease only use integers")
        return False

    if fct not in fctlist: #fct is in fctlist
        print("\nPlease use a function like ['%s']" % "','".join(map(str, fctlist)))
        return False

```

```

return True

def testit(self, fct, k=10, a=0.5):
    """toolbox for the test of different algorithms"""
    if not self.testinput(fct,k,a):
        return

    sumall = 0
    lenall = 0
    avgall = 0

    #precalculations
    if fct == 'slopeone' or fct == 'hybrid':
        r.computeDeviations()
    if fct == 'floydwarshall':
        r.floydwarshall()

    for user in self.data.keys():
        sum = 0
        len = 0
        avg = 0
        for (item) in self.data[user]:
            #calculate the error between predicted rating
            and rating from testdata
            if fct == 'adjcos':
                sum += fabs(float(self.data[user][item]) - r.adjcos(user, item))
            elif fct == 'hybrid':
                sum += fabs(float(self.data[user][item]) - ((1 - a) * r.recommend('euclid', user, k, item) + a * r.sloOne(user, item)))
            elif fct == 'floydwarshall':
                sum += fabs(float(self.data[user][item]) - r.flowar(user, k, item))
            elif fct == 'slopeone':
                sum += fabs(float(self.data[user][item]) - r.sloOne(user, item))
            elif fct == 'pearson':
                sum += fabs(float(self.data[user][item])

```

```

        ])-r.recommend('pearson', user, k,
            item))
    elif fct == 'euclid':
        sum += fabs(float(self.data[user][item]
            ])-r.recommend('euclid', user, k,
            item))
    elif fct == 'usermean':
        sum += fabs(float(self.data[user][item]
            ])-r.usermean(user))
    elif fct == 'itemmean':
        sum += fabs(float(self.data[user][item]
            ])-r.itemmean(item))
    elif fct == 'random':
        sum += fabs(float(self.data[user][item]
            ])-random.uniform(1,5))
    len += 1
    avg = float(sum)/len #avg error for user
    sumall += avg
    lenall += 1
    avgall = float(sumall)/lenall #avg error for all
    user
    return (avgall)

```

```

# # initialising the classes and import data

```

```

# In[ ]:

```

```

l = 1
path = 'PathtoData\Data'

r = recommender(0)
r.loadMovieLens(path+'\\ml-L='+str(l)+'/')
r.computeUserAverages()

t = tester(r,0)
t.loadMovieLens(path+'\\ml-L='+str(l)+'/')

```

```

# possible folders are ml-L=1 / 5, 10, 19. .
# determines how many ratings per user will be in testdata

```

```

# # ToyData low rank matrix
#

# In[ ]:

"""creating low rank matrix"""
#setting parameters
l = 1
o = 30
m = 943
n = 1682

matrixdensity = 0.063
#matrixdensity = 0.30
normalize=True

#create random low rank matrix with rank o
A1 = np.random.uniform(1,5,(m,o))
A2 = np.random.uniform(1,5,(o,n))

B = np.dot(A1,A2)
#np.linalg.matrix_rank(B)

if normalize:
    # normalize to range [1,5]
    Dmax = np.max(B)
    Dmin = np.min(B)
    for (i,j), value in np.ndenumerate(B):
        B[i,j] = round(r.denormalizeRating(r.
            normalizeRating(B[i,j],Dmin,Dmax),0.51,5.49),0)

# killing matrixdensity of the entries
C = np.random.rand(m,n)

for (i,j), value in np.ndenumerate(C):
    if C[i][j]>matrixdensity:
        C[i,j]=0
    else: C[i,j]=1

D = B*C

```

```

np.linalg.matrix_rank(D)

#convert to dictionaries Test and Train(l items of each user)
traindata = {}
testdata = {}
testcount={}

for (i,j), value in np.ndenumerate(D):
    istr = str(i+1)
    traindata.setdefault(istr, {})
    testdata.setdefault(istr, {})
    testcount.setdefault(istr, 0)

r = recommender(0)
for (i,j), value in np.ndenumerate(D):
    istr = str(i+1)
    jstr = str(j+1)
    if D[i,j]>0:
        if testcount[istr]<1:
            #testdata[istr][jstr] = round(r.
                denormalizeRating(r.normalizeRating(D[i,j],
                    Dmin,Dmax),0.51,5.49),0)
            testdata[istr][jstr] = D[i,j]
            testcount[istr]+=1
        else:
            traindata[istr][jstr]=D[i,j]

r = recommender(traindata)
r.computeUserAverages()
t = tester(r, testdata)

# ### Create histograms to compare the randomized low rank
matrix to the MovieLens data

# In[ ]:

#convert the MovieLens-Dataset to a dataframe to create the
histogram
df = pandas.DataFrame.from_dict(r.data, orient='index').
    fillna(0)

```

```

histML, binsML = np.histogram(df, bins=(1,2,3,4,5,6))
histLR, binsLR = np.histogram(D, bins=(1,2,3,4,5,6))
#plot both histograms together
centerML = (binsML[:-1] + binsML[1:]) / 2
centerLR = (binsLR[:-1] + binsLR[1:]) / 2
plt.bar(centerLR-0.1, histLR, color='r', align='center',
        width=0.7, alpha=0.7, label='low_rank_matrix')
plt.bar(centerML+0.1, histML, color='b', align='center',
        width=0.7, alpha=0.7, label='MovieLens_Data')
fig1 = plt.figure(1)
rect = fig1.patch
rect.set_facecolor('white')
plt.title('Histogram')
plt.xlabel('Rating')
plt.ylabel('Frequency')
plt.legend()
plt.show()

# ## precalculations for floydwarshall for test purposes

# In[ ]:

get_ipython().run_cell_magic(u'time', u'', u'r.
    floydwarshall()\nam=_r.adjMatrix\nfw=_r.floydwarDist')

# In[ ]:

r.adjMatrix = am
r.floydwarDist = fw

# # Parameter test for Euclid, Pearson, FloydWarshall and
    Hybrid

# In[ ]:

get_ipython().run_cell_magic(u'time', u'', u'(r.penalty, r.
    normalized)=(True, False)\nk=15\nalist
    =[0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1]\nfor_a_in_
    alist:\n    print_("%3.1f:_%10.5f"%_(_a,t.testit('\
    hybrid\' ,k,a)))')

```

```
# In[ ]:
```

```
get_ipython().run_cell_magic(u'time', u'', u'klist=_
[1,25,50,100,200,500]\nfor_k_in_klist:\nprint_("%i:_
%10.5f"%_t.testit(\ 'floydwarshall\' ,k)))')
```

```
# In[ ]:
```

```
get_ipython().run_cell_magic(u'time', u'', u'klist=_
[1,5,10,15,20,25,50,100,200]\nfor_k_in_klist:\nprint_
_("%i:_%10.5f"%_t.testit(\ 'pearson\' ,k)))')
```

```
# In[ ]:
```

```
get_ipython().run_cell_magic(u'time', u'', u'klist=_
[1,5,10,15,20,25,50,100,200]\n#(r.penalty ,r.normalized)
=(False ,False)\n(r.penalty ,r.normalized)=(True ,False)\n
#(r.penalty ,r.normalized)=(False ,True)\nfor_k_in_klist:\n
nprint_("%i:_%10.5f"%_t.testit(\ 'euclid\' ,k)))'
)
```

```
# # Calculation of the error for a given algorithm
```

```
# In[ ]:
```

```
get_ipython().run_cell_magic(u'time', u'', u'print_"Hybrid
"\nprint_("%10.5f"%_t.testit(\ 'hybrid\' ,15,0.3)')
```

```
# In[ ]:
```

```
get_ipython().run_cell_magic(u'time', u'', u'print_"
FloydWarshall"\nprint_("%10.5f"%_t.testit(\ '
floydwarshall\' ,10)')
```

```
# In[ ]:
```

```
get_ipython().run_cell_magic(u'time', u'', u'print "\nRandom\nnprint "%10.5f"%_t.testit(\'random\')')
```



## Literatur

- [MKR] MIRZA ; KELLER ; RAMAKRISHNAN: *Studying Recommendation Algorithms by Graph Analysis*. <http://people.cs.vt.edu/~ramakris/papers/receval.pdf>
- [Zar15] ZARCHARSKI, Ron: *A Programmer's Guide to Data Mining: The Ancient Art of the Numerati*. <http://guidetodatamining.com/>. Version: 2015

## Abbildungsverzeichnis

1	MAE im Euklid Algorithmus in Abhängigkeit der Parameter . . . .	14
2	MAE im Pearson Algorithmus in Abhängigkeit der Parameter . . .	15
3	MAE im Floyd Warshall Algorithmus in Abhängigkeit der Parameter	16
4	MAE im Hybrid Algorithmus in Abhängigkeit der Parameter . . .	17
5	MAE Vergleich aller Algorithmen . . . . .	18
6	MAE Vergleich mit einer Matrix mit niedrigem Rang . . . . .	19
7	Histogramm Vergleich . . . . .	20
8	MAE Vergleich ohne Normalisierung . . . . .	20

## Tabellenverzeichnis

1	Liste der Algorithmen . . . . .	10
---	---------------------------------	----