book

Babeş–Bolyai Tudományegyetem Kolozsvár
Matematika és Informatika Kar
Informatika Szak

Szakdolgozat

Felhasználói értékeléseken alapuló collaborative filtering algoritmusok kiértékelése funkcionális környezetben

Témavezető:
dr. Bodó Zalán

Szerző:
Zediu
Álmos-Ágoston

2022

Babeş–Bolyai University of Cluj-Napoca
Faculty of Mathematics and Informatics
Specialization: Computer Science

Diploma Thesis

License thesis title

Advisor:
dr. Bodó Zalán

Author:
Álmos-Ágoston Zediu

2022

Universitatea Babeş–Bolyai, Cluj-Napoca
Facultatea de Matematică şi Informatică
Specializarea Informatică

Lucrare de licenta

Titlu lucrare licență

Conducător ştiinţific:
dr. Bodó Zalán

Absolvent:
Álmos-Ágoston
Zediu

2022

1.

Bevezetés

Napjainkban egyre nagyobb hangsúly kerül a különböző ajánló rendszerekre, algoritmusokra melyek felhasználási területe igencsak kiterjedt, legyen akár szó e-commerce felületek termékajánlásáról, streaming szolgáltatások ízlésmeghatározásáról, vagy pedig a közösségi média oldalak fő bevételforrásának számító személyes reklámajánlásáról.

A dolgozat fő tematikája három széles körben alkalmazott algoritmus, algoritmuscsalád ismertetése, előnyeinek és hátrányainak bemutatása, körbejárva az implementációs nehézségeket, kiértékelési metrikákat és az adott algoritmusok megfelelő környezetben való felhasználását.

A három bemutatott algoritmus a Slope One, mely egy lineáris regressziónál egyszerűbb ajánlási modell egyetlen szabad paraméterrel, a Locality Sensitive Hashing, ami a hasonló ízléssel rendelkező felhasználók értékelési vektorait egy magas ütközési rátával rendelkező hasítófüggvénnyel csoportosítja, és a Singular Value Decomposition mátrix faktorizációs módszer, ami a felhasználók és az értékelt elemek közötti legfontosabb látens faktorokat hozza napvilágra.

Egy másik bemutatott szempont az algoritmusok Clojure nyelvben való implementálása. A Clojure egy funkcionális Lispre alapuló nyelv, mely a JVM platformon fut, nagy hangsúlyt fektet az adatvezérelt programozásra, az interaktív, REPL alapú fejlesztésre és a keretrendszerek helyett az egyszerű könyvtárakra, melyeket az Unix filozófia alapján modulárisan használunk fel.

2.

TODO Az adathalmazról

Az adathalmaz a Minnesotai Egyetem MovieLens adathalmaza, mely 100 000 értékelést tartalmaz 943 felhasználótól 1682 filmről. (**?**)

3.

Clojure

A Clojure egy dinamikus funkcionális nyelv, mely ötvözi a JVM platform előnyeit a Lisp nyelvek kifejezőkészségével.

3.1 Funkcionális nyelvekről kicsit általánosan.

A funkcionális nyelvek fő alapelve az, hogy nincs mutálható memória, és ahelyett, hogy imperatív, egymáson és programállapoton alapuló utasításokkal dolgozunk, előtérbe helyezzük a kifejezéseket, és a "tiszta" mellékhatásoktól mentes függvényeket, melyeknek eredménye a bemeneti paraméterektől függ.

3.2 Funkcionális programozás Clojureben

A Clojure fő filozófiája az egyszerű adatszerkezetekkel, főleg mapekkel való modellezése a problémáknak. Ahelyett, hogy bonyolult absztrakciókat képezünk és enkapszuláljuk az adatainkat a rajtuk végzett műveletekkel (ezáltal "objektumokat", és "metódusokat képezve"), vagy pedig előbb bonyolult típusosztályokkal algebrai adatstruktúrákat képzünk, előtérbe helyezzük az egyszerű adatstruktúrákat, és az egyszerű adatstruktúrákon operáló függvényeket.

A Clojureben a függvények az elsőrendű absztrakciók, képesek vagyunk akár argumentumként is kezelni őket, stb.

```
(defn my-adder [a b] (+ a b))

(def my-five-adder (partial my-adder 5))
```

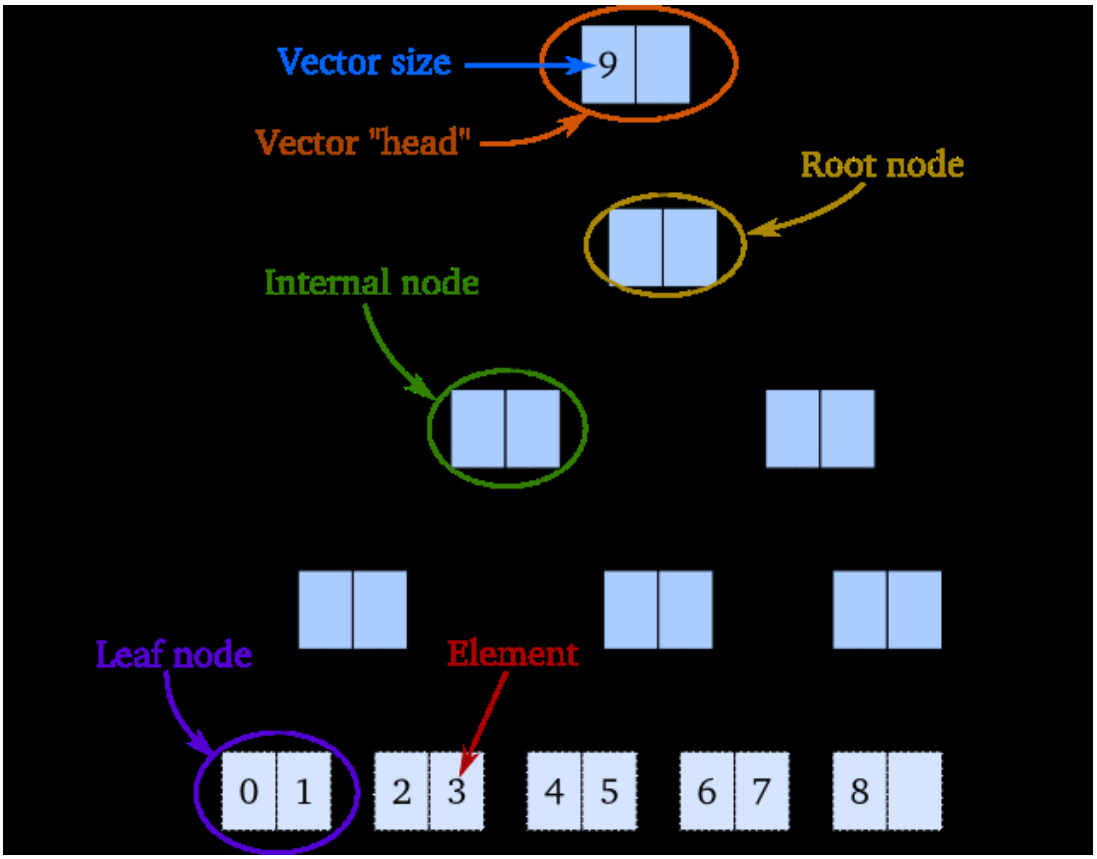
```
(map my-five-adder [1, 2, 3, 4])
```

- #'user/my-adder
- #'user/my-five-adder
- (6 7 8 9)

3.3 Perzisztens adatstruktúrák

Egy lényegi kérdés ami felmerülhet funkcionális nyelvek esetén, az a memóriahasználat, és a futási idő problémája. Mivel minden változó alapvetően konstans, ezért minden egyes olyan művelet, ami imperatív nyelvekben az eredeti adatstruktúra változtatását vette volna igénybe (új elem beszúrás, törlés, tulajdonság megváltoztatása), a funkcionális nyelvekben egy "módosított" változatát adja vissza az eredeti algoritmusnak. Egy naiv háttérbeli implementáció esetén tehát egy esetleges tömb beszúrásnál le kellene másolni az egész tömböt. Itt jön be a perziztencia, és a strukturális megosztás ötlete. Ha van olyan része a memóriának, ami változatlan marad az új adatstruktúrában is, fölösleges azon a memóriarészen levő értékeket lemásolni, és több értelme van megosztani vele.

A Clojure alap adatstruktúrái az ideális hasítófákra vannak alapozva. (**?**) (**?**). Egy konceptuális elképzelésért rátekinthetünk erre a képre:



A lényegi rész az, hogy ahhoz, hogy olyan adatstruktúrák, mint a vektorok performánsak legyenek, de perzisztensek, szükségünk van specializált bináris fák felépítésére.

3.4 Homoikonicitás

Ami talán leginkább megkülönbözteti a Lisp nyelvcsaládban levő nyelveket a többiektől, az a homoikonicitás (**?**) tulajdonság, vagyis maga a programkód formálható ugyanazzal a nyelvvel futás közben, mint amiben meg volt írva.

Hasonló viselkedést elérhetünk nem homoikonikus nyelvekben is, mint mondjuk a Java vagy a C# reflection rendszere, vagy pedig a Python dekorátor szintaxisa, viszont a Lisp nyelvek makrórendszereivel azért könnyebb valamilyen szinten dolgozni, mivel nincsenek speciálisan megkülönböztetve a programban felhasznált adatstruktúrák szintaxisai, és a programot felépítő, elágazásokat, ismétlő ciklusokat, függvénydefiníciókat jelző nyelvi struktúrák szintaxisai.

Vegyük példának okáért a következő egyszerű programot:

```
(defn add-list-numbers [number-list]
  (apply + number-list))

(add-list-numbers '(1 2 3 4 5))
```

- #'user/add-list-numbers
- 15

Látható, hogy a függvénydefiníció kerek zárójelekbe írtuk, a függvény argumentumai pedig egy vektorszerű struktúrában kaptak helyet, utána pedig maga a függvényhívás is zárójelek között volt. Érdekes módon az átadott lista szintúgy zárójelezve adódott át, viszont raktunk elé egy aposztrófot is.

Erre azért volt szükség, mivel a Lisp nyelvekben a kerek zárójel listát jelöl, és minden lista, hacsak nem jelezzük aposztrófal, függvénymeghívással jár. Annak köszönhetően viszont, hogy "listákban" programozunk, képesek vagyunk a programrészleteinket mint lista, vektor, vagy halmazelemeket módosítani átrendezni.

3.4.1 Makrók

A Lisp makrók olyan programszerkezetek, amelyek kódrészletet kapnak argumentumként, módosítják azt, és a módosított programrészlet eredményét futtatják végül le. Fontos megjegyezni, hogy a végső kód legenerálása fordítási időben történik, nem futási időben.

Egy jó példa arra, hogyan segíthet ez fejlesztésben és talán még fontosabb, adatelemzés során, az az úgynevezett "threading" makró.

```
(defn generate-masked-grouped-ratings [dataset-path]
  (-> (load-ratings dataset-path)
      (tc/dataset)
      (tc/complete :user :item)
      (tc/group-by :user {:result-type :as-seq})))
```

Szerepe tulajdonképpen abból áll, hogy az első logikai egységet ami a nyíl mellett áll, "befűzi" a következő függvényhívás első argumentumaként és azon függvényhívás eredményét pedig ugyanúgy befűzi a következő függvényhívás első argumentumaként, és így tovább.

Bár talán komplikáltnak tűnhet egy hasonló funkcionalitás implementálása, ezen makró forráskódja mindössze 10 sor, és kihasználja azt, hogy a "formok" (a Clojure kód kerek zárójelbe helyezett futtatható egysége) igazából listák, így a makró feladata egyszerűen a helyes futtatható lista megalkotása.

```
(defmacro ->
  [x & forms]
  (loop [x x, forms forms]
    (if forms
      (let [form (first forms)
            threaded (if (seq? form)
                       (with-meta `(~(first form) ~x ~@(next form)) (meta form))
                       (list form x))]
        (recur threaded (next forms)))
      x)))
```

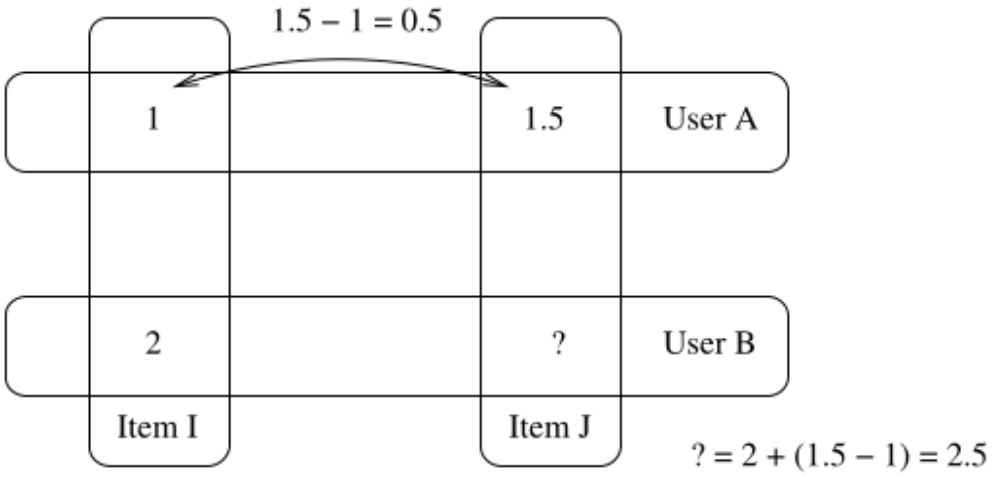
4.

Algoritmusok

4.1 Slope one

A Slope One egy egyszerűen implementálható, de ennek ellenére meglepően jó eredményekkel rendelkező algoritmuscsalád melyet Anna Maclachlan és Daniel Lemire jelentettek meg. (**?**)

Nevét onnan kapta, hogy a amíg az egyszerű lineáris regresszió esetén két paramétert becsülünk meg, itt elég csak egy paraméter, leegyszerűsítve a $f(x) = ax + b$ modellt egy $f(x) = x + b$ modellre. Abban az esetben, mikor felhasználói értékelésekről beszélünk nem egy adott termék vagy értékelendő tárgy individuális értékeléseit vizsgáljuk, hanem az egy-egy tárgy értékelései közötti átlagos különbséget.



4.1.1 Működési elv

Az algoritmus dióhéjban összesíti a tárgyak közötti szavazatkülönbségeket, utána pedig ahhoz, hogy megközelítsük egy felhasználó ismeretlen szavazatát, összeadjuk a létező szavazatait a vizsgálandó tárgy és az létező szavazatok közötti átlagos különbségekkel, és súlyozott átlagot számolunk, ahol a súly az, hogy hányan szavaztak mindkét tárgyra.

Ha a felhasználó u-ként jelöljük, a szavazatai halmazát $S(u)$-ként, akkor egy j tárgyra adott:

$$\hat{r}_{j|u} = \frac{\sum_{i \in S(u); i \neq j} (\Delta i, j + u_i) c_{j,i}}{\sum_{i \in S(u) i \neq j} c_{j,i}} \tag{4.1}$$

(**?**)

Ahol $c_{j,i} = card(S_{j,i}(R)$ vagyis a kardinalitása a j és i-re is szavazott embereknek.

4.1.2 Implementáció

Az ebben a szekcióban levő kód nagy része Henry Garner Clojureben való gépi tanulásról szóló könyvéből lett átvéve, (**?**) , és a **top-n** ajánlási mechanizmussal együtt is alig tesz ki 50 sort.

Először a listakonstruktor elvű **for** makróval tárgy párokat generálunk, majd egy üres **map** asszociatív struktúrából kiindulva leredukáljuk ezeket a párokat egy mapre, melyben minden az összes tárgyak közötti különbség el van mentve.

```
(defn conj-item-difference [dict [i j]]
  (let [difference (-  (:rating j) (:rating i))]
    (update-in dict [(:item i) (:item j)] conj difference)))

(defn collect-item-differences [dict items]
  (reduce conj-item-difference dict
          (for [i items
                j items
                :when (not= i j)]
            [i j])))

(defn item-differences [user-ratings]
  (reduce collect-item-differences {} user-ratings))
```

Ezután elmentjük a különbségek átlagát, és a közös szavazók számát.

```
(defn summarize-item-differences [related-items]
  (let [f (fn [differences]
            {:mean  (s/mean differences)
             :count (count  differences)})]
    (map-vals f related-items)))

(defn slope-one-recommender [ratings]
  (->> (item-differences ratings)
       (map-vals summarize-item-differences)))
```

A felhasználási lépésben, amikor egy adott tárgyra szeretnénk értékelést megsaccolni, hozzáadjuk a meglevő szavazatokat a különbségekhez és elvégezzük a súlyozott átlagolást.

```
(defn candidates [recommender {:keys [rating item]}]
  (->> (get recommender item)
       (map (fn [[id {:keys [mean count]}]]
              {:item id
               :rating (+ rating mean)
               :count count}))))

(defn weighted-rating [[id candidates]]
  (let [ratings-count (reduce + (map :count candidates))
        sum-rating (map #(* (:rating %) (:count %)) candidates)
        weighted-rating (/ (reduce + sum-rating) ratings-count)]
    {:item id
     :rating weighted-rating
     :count  ratings-count}))
```

A top-n ajánlás már csak annyit ad hozzá, hogy elvégzi az egész adathalmazra a megközelítéseket, kiveszi a vizsgált felhasználó már értékelt tárgyait, és csökkenő sorrendbe helyezi az értékeléseket.

```
(defn slope-one-recommend [recommender rated top-n]
  (let [already-rated  (set (map :item rated))
        already-rated? (fn [{:keys [id]}]
                         (contains? already-rated id))
        recommendations (->> (mapcat #(candidates recommender %)
                                     rated)
                             (group-by :item)
                             (map weighted-rating)
                             (remove already-rated?)
                             (sort-by :rating >))]
    (take top-n recommendations)))
```

4.2 Locality sensitive hashing

A Locality Sensitive Hashing egy olyan hasítófüggvényekre alapuló módszer, ami a legtöbb hasítófüggvény implementációval ellentétben nem minimizálja az ugyanolyan kimenetek, kulcsok számát, hanem maximalizálja, mivel a hasonló tárgyak, (esetünkben szavazatvektorok) hasonló kimenettel kell rendelkezzenek.

4.2.1 Definíció

Egy LSH séma egy olyan F hasítófüggvénycsalád, melyekre igaz, hogy a valószínűsége annak, hogy két x, y objektum függvényértéke megegyezik, megegyezik a két függvény hasonlósági távolságával valamilyen metrika szerint. (Charikar)

$$Pr_{h\in F}[h(x) = h(y)] = sim(x, y) \tag{4.2}$$

ahol $sim(x, y) \in [0, 1]$ természetesen.

Ezzel egyrészt csoportosítani tudjuk a potenciálisan hasonló ízléssel rendelkezőeket, és ugyanakkor kompaktan, kevés helyfelhasználással később is fel tudjuk használni ezen csoportokat, ami segít a futási időn is persze.

4.2.2 Véletlenszerű hiperterekre alapuló LSH

Az ötlet a következő: egy R^d-ből levő vektorcsoport esetén mintavételezzünk egy normál eloszlású $\overrightarrow{r}$ d dimenziós vektort. Ennek a vektornak a függvényében definiálhatjuk a következő $h_{\overrightarrow{(r)}}$ függvényt:

$$h_{\overrightarrow{(r)}}(\overrightarrow{u}) = \begin{cases} 1 & ha\, \overrightarrow{r} * \overrightarrow{u} \geq 0 \\ 0 & ha\, \overrightarrow{r} * \overrightarrow{u} < 0 \end{cases} \tag{4.3}$$

Ekkor $\overrightarrow{u}$ és $\overrightarrow{v}$ esetén igaz lesz, hogy:

$$Pr_{h\in F}[h(x) = h(y)] = 1 - \frac{\theta(\overrightarrow{u}, \overrightarrow{v})}{\pi} \tag{4.4}$$

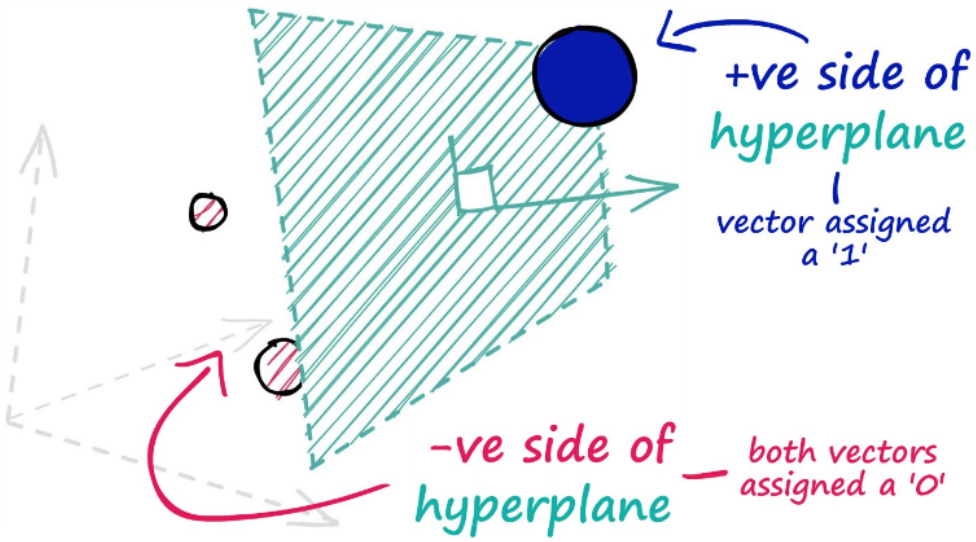
(Charikar)

Vagyis annak a valószínűsége, hogy két vektor egyenkénti skaláris szorzata a véletlenszerűvel és az erre alkalmazott előjel függvény kimenete ugyanaz legyen megegyezik a két vektor között bezárt szög koszinuszával. Ezt először Goemans és Willamson bizonyította (Goemans és Williamson, 1995) egy a MAX-CUT relaxációjával foglalkozó cikkükben.

Intuitívan arról van szó, hogy ha veszünk egy d dimenziós hiperteret, ahol d az adathalmazunkban levő tárgyak száma, akkor minden felhasználót el tudunk helyezni ebben a hipertérben, hisz a szavazataik meghatározzák a pozíciójukat, hisz d dimenziós vektorok.

Egy véletlenszerűen felvett vektor normálvektora egy a hiperteret kettéosztó hipersíknak, vagyis a vele való skaláris szorzat előjele meghatározza, hogy egy pont a hipersík melyik felén helyezkedik el.

Elég ilyen hipersíkot felvéve ki tudunk alakítani csoportokat, akik több hipersíknak is ugyanazon a felén vannak, ebből következve hasonlóak.



(Ran)

4.2.3 TODO Implementáció

Az első lépés a random normálvektorok legenerálása:

```
(defn generate-random-vectors [d nbits]
  (nrand/rand-normal! 0 1 (nnat/dge d nbits)))
```

A függvény amely kiszámítja mely "vödörbe kerül" egy felhasználó az értékelési vektora alapján

```
(def threshold
  "The sign function which translates a value into a binary 1 or 0."
  (fn [x]
    (if (> x 0.0)
      1.0
      0.0)))

(defn calculate-lsh-hash
  "Used to calculate the LSH hash of a random vector."
  [user-vector rand-normals]
  (. Integer parseInt (->> (ncore/mv rand-normals user-vector)
                           (fmap threshold)
                           (into [])
                           (map int)
                           (apply str)) 2))
```

A függvény amely elvégzi a "vödör" generálást, és feltölti őket.

```
(defn generate-lsh-buckets
  "Generating the buckets for the stuff."
  [all-items grouped-ratings rand-normals]
  (loop [current-rating (first grouped-ratings)
         remaining-ratings (rest grouped-ratings)
         current-user (get-in current-rating [:user 0])
         buckets {}]
    (let [sparse-vector (-> (get-sparse-ratings all-items current-rating)
                            (get-sparse-vector))
          lsh-hash (calculate-lsh-hash sparse-vector rand-normals)
          new-buckets (if (contains? buckets lsh-hash)
                        (update buckets lsh-hash conj current-user)
                        (conj buckets [lsh-hash [current-user]]))]
      (if (empty? remaining-ratings)
        new-buckets
        (recur (first remaining-ratings) (rest remaining-ratings) (get-in (first rema
```

4.3 Singular Value Decomposition (SVD)

4.3.1 Definíció

A Singular Value Decomposition (röviden SVD) egy mátrix faktorizációs módszer, melyben egy $m \times n$ mátrixot felbontunk $M = U\Sigma V^*$ módon, ahol U egy $m \times n$ alakú unitáris mátrix, Σ egy $m \times n$ alakú diagonális mátrix, V pedig egy $n \times n$ unitáris mátrix. Σ átlón elhelyezkedő értékei a négyzetgyökei $M * M$ sajátértékeinek, és sok SVD implementációban az értékek csökkenő sorrendben jelennek meg Σ főátlóján.

4.3.2 Felhasználás ajánlórendszerekben

Hasznossága abban rejlik, hogy ha kiválasztjuk az r legnagyobb singular value-t (vagy röviden s-értéket) akkor meg tudjuk közelíteni az eredeti M mátrixot, egy másik, $\widetilde{M}$ mátrixxal, ahol $rank(\widetilde{M}) = r$ (**?**). Ez azért tud hasznos lenni ajánlórendszerekben, mivel ezáltal le tudjuk redukálni az eltárolandó felhasználó-értékelt objektum kapcsolatok számát, és csak a redukált r rangú M, Σ, V^* mátrixokat tárolva helyet spórolunk, ugyanakkor ki tudjuk emelni az r legfontosabb "látens faktort", amik megadják az összefüggést bizonyos felhasználók és bizonyos filmekre adott értékelések között.

4.3.3 Implementáció

Az alkalmazott implementáció alapjául Badrul N. Sarwar, George Karypis etc, Minnesotai Egyetem kutatóinak cikke szolgál, (**?**) és felbontható a következő stádiumokra:

Az implementációban a normalizálás az objektumok átlagos pontszámai alapján történtek, vagyis csoportosítva lettek a szavazatok objektum szerint és onnan kivontuk az átlagszavazatot:

```
(def multiple-averages (-> grouped-ratings
                    (tc/complete :user :item)
                    (tc/group-by :item {:result-type :as-seq})))

(def averages
(map
    (fn [group]
    (let [mean (-> (tc/aggregate group #(dfn/mean (% :rating)))
```

```
                    (tc/get-entry "summary" 0))
            normalized-group (-> (tc/replace-missing group :rating :value mean)
                                 (tc/update-columns :rating [(partial map #(- % mean))]
          {:normal normalized-group
          :mean mean})) multiple-averages))
```

Az átlagos értékelések el lettek mentve, hogy az adathalmaz rekonstrukciójakor denormalizálni tudjuk a szavazatokat, megkapva az előrejelzett értékeléseket.

Az SVD kiszámítása a LAPACK lineáris algebra könyvtár Intel MKL implentációja alapján történik.

```
(def svd (-> (nnat/dge 1680 943 (:rating new-normal))
             (nlin/svd true true)))
```

Az adathalmaz rekonstrukciója során kiválasztjuk a 14 legnagyobb s-értéket, ugyanakkor az U és V^* mátrixok almátrixait, gyököt vonunk a redukált Σ mátrixokból, és újra összeszorozzuk őket, megkapva a 14-es ranggal rendelkező $\widetilde{M}$ mátrixot.

```
(def sigma (-> (:sigma svd)
                (ncore/submatrix 14 14)))
(def u (ncore/submatrix (:u svd) 1680 14))
(def vt (ncore/submatrix (:vt svd) 14 943))
(def s_root (nvmath/sqrt sigma))
(def usk (ncore/mm u s_root))
(def skV (ncore/mm s_root vt))
(def usv (ncore/mm usk skV))
```

Bibliography

Random Projection for Locality Sensitive Hashing | Pinecone. https://www.pinecone.io/learn/locality-sensitive-hashing-random-projection/.

Bagwell, P., editor. *Ideal Hash Trees*. 2001.

Brand, M. Fast online SVD revisions for lightweight recommender systems. In *Proceedings of the 2003 SIAM International Conference on Data Mining*, pages 37–46. Society for Industrial and Applied Mathematics, May 2003. ISBN 978-0-89871-545-3 978-1-61197-273-3. doi: 10.1137/1.9781611972733.4.

Charikar, M. S. Similarity Estimation Techniques from Rounding Algorithms. page 9.

Garner, H. *Clojure for Data Science: Statistics, Big Data, and Machine Learning for Clojure Programmers*. 2015. ISBN 978-1-78439-750-0.

Goemans, M. X. és Williamson, D. P. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM*, 42(6):1115–1145, Nov. 1995. ISSN 0004-5411. doi: 10.1145/227683.227684.

Jayasinghe, T., Stanek, K. Z., Kochanek, C. S., Thompson, T. A., Shappee, B. J., és Fausnaugh, M. An Extreme Amplitude, Massive Heartbeat System in the LMC Characterized Using ASAS-SN and TESS. *Monthly Notices of the Royal Astronomical Society*, 489(4):4705–4711, Nov. 2019. ISSN 0035-8711, 1365-2966. doi: 10.1093/mnras/stz2460.

Lemire, D. és Maclachlan, A. Slope One Predictors for Online Rating-Based Collaborative Filtering, Sept. 2008.

McIlroy, M. D. Macro instruction extensions of compiler languages. *Communications of the ACM*, 3(4):214–220, Apr. 1960. ISSN 0001-0782. doi: 10.1145/367177.367223.