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.

Clojure

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

2.1 Funkcionális programozás Clojureben

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

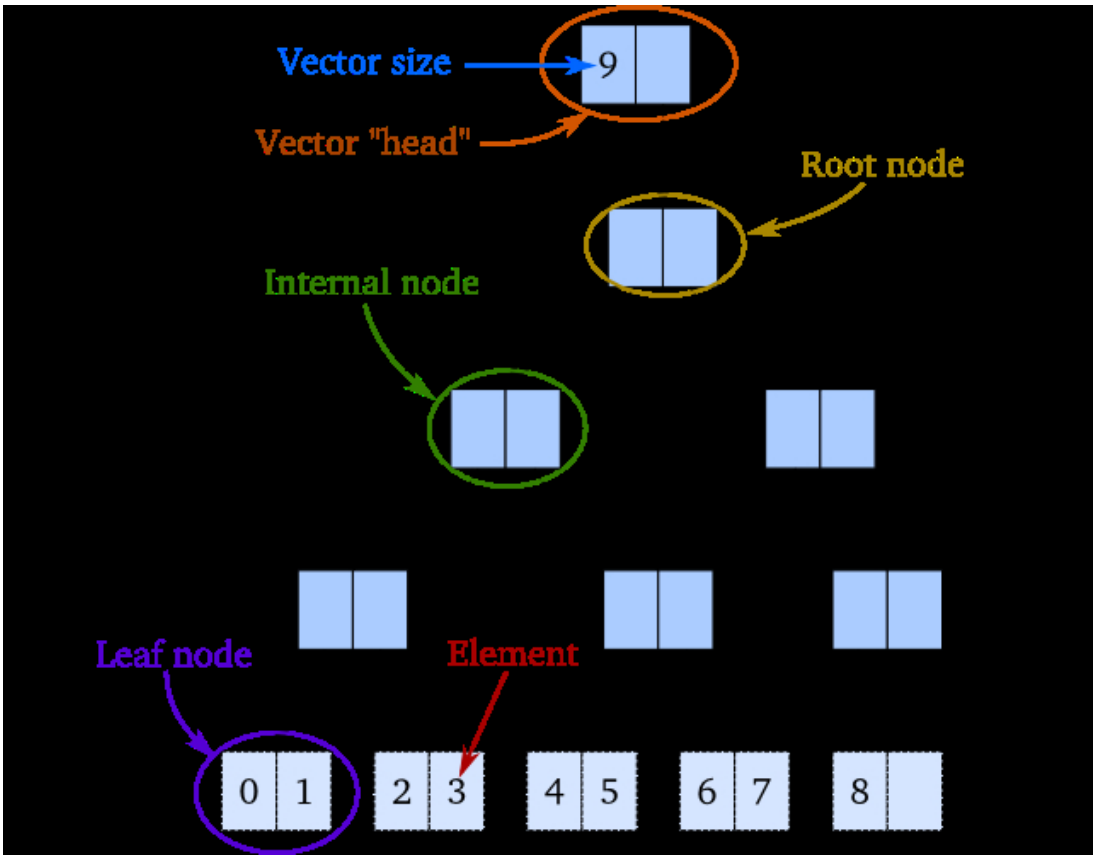
```
(defn my-adder [a b] (+ a b))

(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)

2.1.1 TODO Funkcionális nyelvekről kicsit általánosan.

2.2 Perzisztens adatstruktúrák

Rich Hickey az adatstruktúráit az ideális hasítófákra alapozta (**?**) (**?**). 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.

2.3 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.

2.3.1 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)))
```

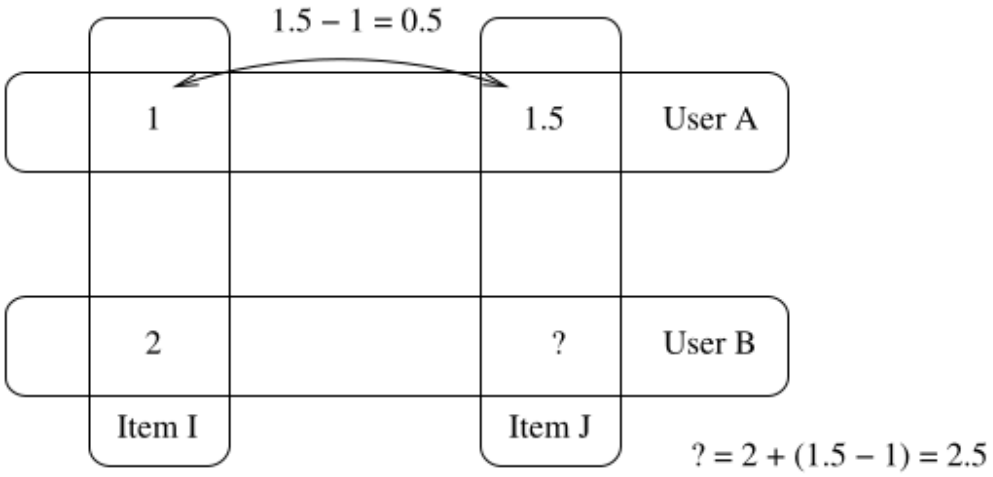
3.

Algoritmusok

3.1 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.



3.1.1 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:

$$\hat{r}_{j|u} = \frac{\sum_{i \in S(u); i \neq j} (\Delta i, j + u_i) c_{j,i}}{\sum_{i \in S(u) i \neq j} c_{j,i}} \tag{3.1}$$

(**?**)

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

3.1.2 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)))
```

3.2 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.

3.2.1 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. (**?**)

$$Pr_{h \in F}[h(x) = h(y)] = sim(x, y) \tag{3.2}$$

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.

3.2.2 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:

$$h_{\overrightarrow{(r)}}(\overrightarrow{u}) = \begin{cases} 1 & ha\, \overrightarrow{r} * \overrightarrow{u} \geq 0 \\ 0 & ha\, \overrightarrow{r} * \overrightarrow{u} < 0 \end{cases} \tag{3.3}$$

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

$$Pr_{h \in F}[h(x) = h(y)] = 1 - \frac{\theta(\overrightarrow{u}, \overrightarrow{v})}{\pi} \tag{3.4}$$

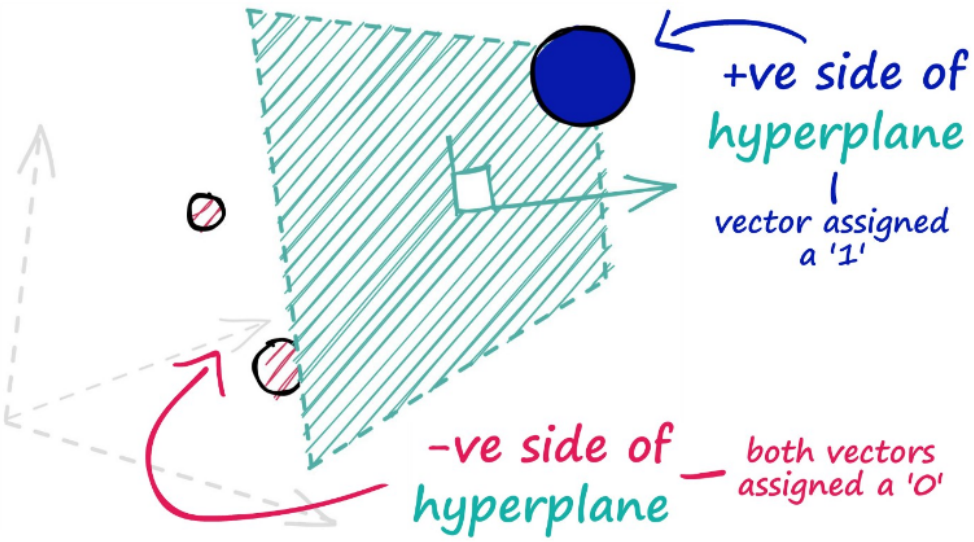
(**?**)

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 (?) 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.



(?)

3.3 SVD

(?)

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.