

Szegedi Tudományegyetem
Informatikai Tanszékcsoport

Diplomamunka

Vancsics Béla

2014

**Szegedi Tudományegyetem
Informatikai Tanszékcsoport**

Mátrix klaszterező algoritmusok összehasonlítása

Diplomamunka

Készítette:
Vancsics Béla
gazdaságinformatikus
hallgató

Témavezető:
Dr. Gergely Tamás
egyetemi adjunktus

Szeged
2014

Feladatkiírás

Egy lefedettségi mátrix sorai teszteseteket, oszlopai eljárásokat reprezentálnak, egy cella pedig a $\{0,1\}$ értékeket veheti fel aszerint, hogy az adott teszteset futása közben az adott eljárás meghívódik-e (1) vagy sem (0). A lefedettség szempontjából mind a tesztesetek, mind az eljárások lehetnek egymáshoz hasonlóak, amennyiben a tesztesetek nagyjából ugyanazokat az eljárásokat érintik, illetve az eljárásokat ugyanazok a tesztesetek futtatják. Továbbá, az egyes teszteset- illetve eljárás-klaszterek egymáshoz rendelhetőek (adott teszteset-halmaz adott eljárások tesztelésére szolgál). Ha sikerülne ilyen „klasztereket” beazonosítani, az alapjául szolgálhatna például teszt szelekciónak.

A feladat olyan algoritmusok kipróbálása, módosítása, esetleg megalkotása, amelyek lefedettségi mátrixokon képesek teszteset- és eljárás-klaszterek automatikus beazonosítására, illetve ezek összerendelésére.

Tartalmi összefoglaló

- **A téma megnevezése:**

Kód lefedettségi mátrixok klaszterezése a tesztesetek csoportosítására.

- **A megadott feladat megfogalmazása:**

Különböző klaszterező eljárások implementálása és azok függvény-teszteset mátrixokon történő futtatása. Ennek eredményeképp megkapjuk, hogy mely függvények és mely tesztesetek tartoznak - valamely közös tulajdonságuk miatt - azonos csoportba, és a klaszterezés minőségére jellemző metrikaértékeket.

- **A megoldási mód:**

A szakirodalom tanulmányozása után kiválasztottam több, különböző megközelítésen alapuló algoritmust, ill. ezek segítségével meghatároztam bizonyos eljárásokat, módszereket, valamint a klaszterezés minőségét mutató metrikát és ezeket implementáltam.

- **Alkalmazott eszközök, módszerek:**

A fejlesztés C++ nyelven történt, amihez segítségül használtam az SZTE Szoftverfejlesztési Tanszék által kifejlesztett SoDA library-t, amely a bináris mátrix beolvasásában és az eredmények vizualizációjában volt a segítségemre.

- **Elért eredmények:**

A megvalósított algoritmusok és az általam definiált metrikák segítségével különböző paraméterezések mellett megkaptam a klasztereket, valamint a klaszterekre, ill. a klaszterezett mátrixokra vonatkozó jellemzőket, információkat és meghatároztam, hogy adott feltételek mellett melyik eljárást célszerű használni.

- **Kulcsszavak:**

Klaszterezés, tesztelés, C++, Jaccard távolság, Dice távolság, Ochiai hasonlóság, klaszterezés-metrika, Hamming távolság

Tartalomjegyzék

Feladatkiírás.....	2
Tartalmi összefoglaló.....	3
Tartalomjegyzék.....	4
BEVEZETÉS	6
1. A TESZTELÉSRŐL.....	7
1.1. Különböző tesztelési módszereket.....	7
1.2. Tesztesetek szelekciója és priorizálása.....	7
1.3. Priorizálás klaszterezés segítségével.....	8
2. KLASZTEREZŐ ALGORITMUSOKRÓL.....	9
2.1. Algoritmusok csoportosítása.....	9
2.2. A bemeneti adatokról	9
3. IMPLEMENTÁLT ALGORITMUSOK	12
3.1. Hamming távolságon alapuló eljárások.....	12
3.1.1. HammingDiffClassifier.....	12
3.1.2. TreeBuild.....	14
3.2. Halmazvizsgálatokon alapuló eljárások.....	17
3.2.1. Jaccard.....	17
3.2.2. Dice.....	19
3.2.3. Ochiai.....	20
3.3. Függvények hash lenyomatain alapuló eljárások.....	20
3.3.1. Hash.....	21
3.4. A mátrixok kirajzolásáról.....	23
3.4.1. HammingDiffClassifier eredményének kirajzolása.....	24
3.4.2. A hierarchikusan klaszterező algoritmus (Tree) eredményének kirajzolása.....	24
3.4.3 A particionálón klaszterező algoritmusok eredményék kirajzolása.....	24
3.4.4. A klaszterek kiíratási sorrendjének meghatározása.....	25
3.5. Az implementációról néhány észrevétel.....	26

4. METRIKÁK BEMUTATÁSA ÉS AZ EREDMÉNYEK ÉRTÉKELÉSE	28
4.1. Metrika 1.....	28
4.2. Metrika 2.....	29
4.3. Metrika 3.....	29
4.4. Nehézségek és tapasztalatok.....	30
4.5. Eredmények és lehetőségek	31
Irodalomjegyzék.....	34
Nyilatkozat.....	35
Köszönetnyilvánítás.....	36

BEVEZETÉS

Feladatom több olyan klaszterező algoritmus felkutatása, kitalálása és implementálása volt, amelyek segítségével coverage (függvény-teszteset) mátrix alapján a tesztesetek csoportokba sorolhatóak, ezáltal később prioritizálhatóak.

A szoftverfejlesztés során komoly emberi és anyagi erőforrást emészt fel a termékek tesztelése. Fontos, hogy a módosításokkor keletkező (vagy már korábban meglevő, de eddig nem észlelt) hibákat minél hamarabb megtaláljuk és a lehető leggyorsabban kijavítsuk. Ennek egy hatékony és elterjed eszköze a tesztesetek prioritizálása.

Az én feladatom az volt, hogy különböző módszerek segítségével megadjak olyan függvénycsoport – tesztesetcsoport párokat, amelyek - valamilyen közös jellemzőjük miatt- összetartoznak. Ennek segítségével látjuk, hogy ha módosítanak a kódon valamit, akkor mely tesztek kell az elsők között lefuttatni, melyek azok, amelyekben a hiba potenciális előfordulása a legnagyobb. Emellett a klaszterekből történő teszteset-mintavétel (pl.: minden klaszterből az esetek 15% -t kiválasztjuk) esetén egyenletesebb eredményt kapunk, mintha az összes tesztből választanánk ki véletlenszerűen annyi darabot.

1. A tesztelésről

A tesztelés egy rendszer vagy program kontrollált körülmények melletti futtatása, és az eredmények kiértékelése. A tesztelés tulajdonképpen a szoftver minőségének a felmérése a hibák felfedezése által [1].

A hagyományos megközelítés szerint a tesztelés célja az, hogy a fejlesztés során létrejövő hibákat minél korábban - még az üzembe helyezés előtt- felfedezze, és ezzel csökkentse azok kijavításának költségeit [2]. A definícióból is kiderül, hogy ha egy vállalat nem fordít kellő hangsúlyt a tesztelésre vagy nem megfelelő módon végzi a folyamatot, akkor komoly, kicsi odafigyelés mellett könnyen elkerülhető költségekbe veri magát (és akkor még nem ejtettünk szót a presztízsveszteségről, valamint a vállalatról kialakult pozitív képben keletkezett károkról).

Ezért is olyan fontos a jól és hatékonyan megtervezett és kivitelezett tesztelés.

1.1. Különböző tesztelési módszereket

Alapvetően kétféle tesztelés létezik: a statikus és a dinamikus tesztelés. A statikus tesztelés bármilyen dokumentumon (vagyis nem csak a forráskódon) manuálisan vagy különböző tesztelést támogató szoftverrel elvégezhető. A dinamikus teszteléskor a program működése közben (futási időben) hajtódnak végre a tesztek. Ezeknek is két nagy típusa van: a blackbox, ill. a whitebox tesztelés. Az előbbi esetében a tesztelők mit sem tudnak a kód szerkezetéről, a funkcionalitásra helyezik a hangsúly, az utóbbi esetben a programkód struktúráját veszik górcső alá, a szerkezetre támaszkodva készítik el a tesztek és az alapján tárják fel a szoftverben levő hibákat vagy mondanak értékelést a tesztek minőségéről [3].

1.2. Tesztesetek szelekciója és prioritizálása

A szelekció lényege, hogy időt takarítsunk meg azzal, hogy a teszteléskor ne kelljen az összes tesztesetet lefuttatni, hanem annak csak egy bizonyos, számunkra - adott szempontok szerint kiválasztott- fontos részhalmazát. (Persze ettől még elkerülhetetlen, hogy később az összes tesztet le ne futtassuk, de bizonyos esetekben elég azok egy részhalmazát vizsgálnunk.)

Egy másik elterjedt eszköz a prioritizálás. Ilyenkor valamilyen szempont szerint sorrendbe rakjuk a teszteseteket és ennek a sorrendnek megfelelően hajtjuk végre a teszteket. Például a regressziós tesztek esetén előszeretettel alkalmazzák a prioritizálást mégpedig úgy, hogy azok kerülnek a lista elejére, amelyek a változtatott kódrészletet tesztelik.

1.3. Priorizálás klaszterezés segítségével

Ha adott egy függvény-teszteset mátrix - amelynek bemutatására lentebb kerül sor -, akkor különböző módszerekkel meghatározhatjuk, hogy mely tesztesetek (és mely metódusok) tartoznak egy csoportba, melyek azok, amelyek „hasonlóan viselkednek” (noha nem pont ugyanolyanok).

Ennek segítségével, ha kódban egy eljárást módosítanak, akkor azokat a teszteket kell először elvégezni, amelyek az adott metódust tesztelik, illetve azokat, amelyeket ezekhez hasonlóan viselkednek, vagyis egy klaszterbe tartoznak. Minél jobban klaszterezhető egy mátrix, vagyis az egy klaszterbe tartozó elemek minél jobban hasonlítanak egymásra (a hasonlóság fogalma később lesz konkretizálva), annál nagyobb annak az esélye, hogy a kiválasztott klaszterekben levő tesztesetek közül mindegyiket érinti a változás és emiatt mindet célszerű az elején letesztelni.

A dolgozatom további részében az általam megismert, illetve megalkotott algoritmusokat fogom részletesebben elemezni, valamint a klaszterezés minőségét jellemző metrikákat bemutatni.

2. Klaszterező algoritmusokról

Ebben a fejezetben általánosságban beszélek a klaszterezések lehetséges típusairól, valamint a klaszterezésem során használt mátrixokról, ezáltal segítve a később részletezett módszerek megértését.

2.1. Algoritmusok csoportosítása

Az algoritmusokat többféle képen is csoportosíthatjuk. Az egyik ilyen, amikor a klaszterezés „stílusa” szerint osztjuk szét az eljárásokat. Ezek szerint lehet hierarchikus, particionáló, valamint gráf alapú módszer.

Az első esetben a pontok klaszterstruktúráját klaszterek egymásba ágyazott rendszereként adjuk meg. Vagyis először minden elem külön klasztert alkot, majd amíg az általunk megadott megállási feltétel nem teljesül addig vonjuk össze a két, az algoritmus által leginkább alkalmasnak vélt klasztert.

A particionáló klaszterezés csupán a pontok diszjunkt részhalmazait adja meg. Itt előre tudjuk a klaszterek számát (míg a hierarchikusnál a megállási feltétel függvényében változhat a klaszterszám) és az elemeket egy adott célfüggvény alapján csoportosítjuk [4].

A gráf alapú klaszterezéskor a gráf csúcsainak particionálását végezzük úgy, hogy az élek minimális éleltávolítása mellett közel azonos méretű részhalmazokat eredményezzen.

A dolgozatomban az első két típusú algoritmusra hozok fel példákat.

2.2. A bemeneti adatokról

Az elemek klaszterezésekor valamilyen kiválasztott tulajdonságuk alapján soroljuk őket csoportokba. Maguk az elemek és azok tulajdonsága is sokféle lehet, különbözőképpen lehet ezeket értelmezni. Dolgozatomban az elemeket a bemeneti mátrix sorai, ill. oszlopai fogják reprezentálni, ezek fogják adni a halmazok elemeit.

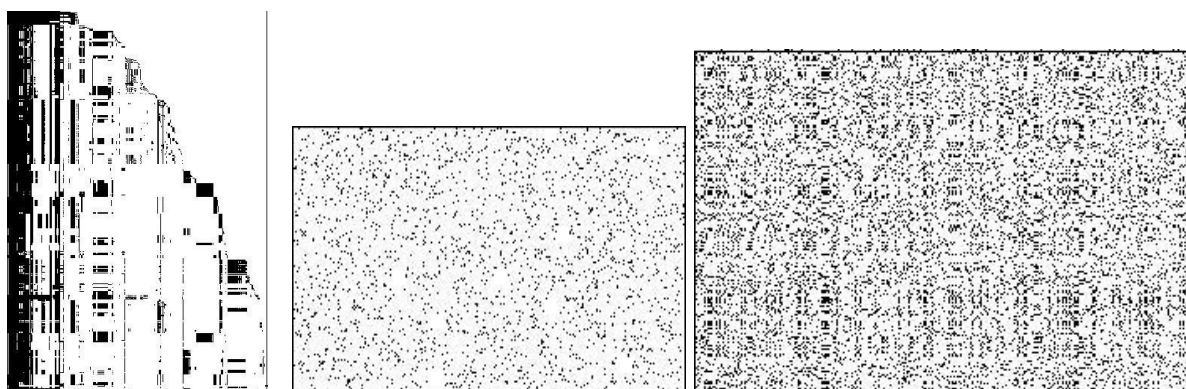
A különböző algoritmusok futtatásához különböző paraméterek megadása szükséges (ezeket a megfelelő részeknél részletesebben kifejtem) de egy valamiben mindegyik megegyezik: az összes program kér egy bináris coverage mátrixot. Ezt az SZTE

Szoftverfejlesztés Tanszék által fejlesztett SoDA library segítségével állítottam elő. Az implementáció során több, a library-ben már megvalósított metódust is felhasználtam. Így a legtöbb eszköz már rendelkezésemre állt ahhoz, hogy a kiválasztott algoritmusokat megvalósítsam.

A bemeneti fájl - a mátrix mellett - tartalmazza a függvények és a tesztesetek neveit is, ezáltal az eredmények összevethetőkké és azonosíthatókká válnak. A mátrixban csak 0-k és 1-k szerepelnek. Akkor van egy pozícióban 1, ha az adott függvény - közvetett vagy közvetlen - hívása esetében az adott teszteset megvizsgálásra kerül.

Az eredmények előállítását három típusú mátrixok végeztem el:

- valós rendszerek mért mátrixokon
- random, általam generált mátrixokon
- „speciális”-random generált mátrixokon



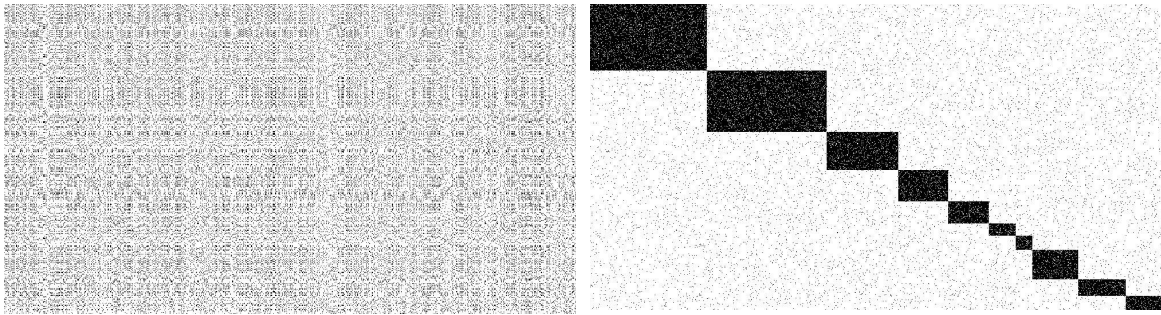
(2.1. ábra: valós, random, és „speciális”-random mátrix)

Az első, a „valós típusú” mátrixok különböző opensource projekteken mért eredményeket foglalnak magukba.

A random mátrixot egy általam implementált - szintén a SoDA-t használó - program állítja elő, aminek paraméterben meg lehet adni a méretét (hány sora, illetve hány oszlopa legyen a mátrixnak), valamint, hogy hány százalék valószínűséggel kerüljön 1 érték a mátrix adott pozíciójába. Emellett kéri, hogy adjuk meg a kimeneti mátrix nevét, egy .pgm formátumú fájlt, amibe a mátrix fekete-fehér képe kerül kirajzolásra, valamint egy számot, amely a rajzoláskor a skálázást végzi (ha túl nagy lenne a mátrix mérete, akkor arányosan csökkenti a kimeneti kép méretét).

A „speciális”-random mátrix gyakorlatilag egy klaszterezettnek generált véletlen mátrixot készít, amelynek a sorait és az oszlopait összekevertük. Itt szintén megadjuk a mátrix méretét, a skálázás mértékét, a kimeneti kép, ill. mátrix nevét. Csakhogy itt meg kell adnunk egy számot, amely a klaszterek számát adja meg. Emellett két, 0 és 100 közé eső egészet, amely megmondja, hogy hány százalék eséllyel lesz 1 érték klaszteren belüli, illetve a klaszteren kívüli pozíciókban. Ezek mellett meg kell adni még egy PGM fájlát, amibe a rendezett mátrix képe kerül (a másik képfájlban ugyanis a sorok és az oszlopok összekeverve szerepelnek).

Például a ábrán látható két mátrix megegyezik. Ekkor olyan sorrendet állítottuk fel a sorok és az oszlopok között, hogy azzal megpróbáljuk „eltüntetni” a generált klasztereket. Így első ránézésre véletlennek tűnő, ugyanakkor a belső szerkezetéből kifolyóan az elemek között - adott esetben nagy - hasonlóságokat tartalmazó mátrixot kapunk (2.2. ábra).



(2.2. ábra: a rendezetlen és a rendezett speciális random mátrix)

3. Implementált algoritmusok

Ebben a fejezetben részletesen is kifejtem az általam implementált megoldásokat, felvázolom azok lépéseit, előnyeit, hátrányait, valamint a futtatás során használt paramétereket.

A témában való elmélyüléskor próbáltam tekintettel lenni arra, hogy több megközelítésből tekintsek a problémára és más-más elvekre építő megoldásokat sikerüljön találnom. Én három főbb csoportba sorolnám az eljárásaimat, amelyeknek a szemléletmódjai - nem teljesen ugyan de- függetlenek egymástól.

3.1. Hamming távolságon alapuló eljárások

Az első ilyen csoport a Hamming távolságon alapuló klaszterező eljárások. Ilyen esetekben két azonos hosszúságú vektor azonos pozíción levő eltéréseket vizsgáljuk (3.1. ábra). Ezen eltéréseknek a száma adja meg a két vektor Hamming távolságát. A Hamming távolság gyakorlatilag az összes algoritmusban fontos szerepet játszik, de bizonyos eljárások szinte kizárólag erre a fogalomra építenek.

"**t**oned" and "**r**oses" is 3.
10**1**1101 and 100**1**001 is 2.
2**1**73896 and 2**2**33**7**96 is 3.

(3.1. példa Hamming távolságra [5])

3.1.1. HammingDiffClassifier

Az első bemutatott, particionáló algoritmus viszonylag egyszerű elven működik. Miután beolvassuk a mátrixot, megvizsgálja, hogy minden sorban, ill. oszlopban mennyi 1 érték szerepel. Ha ez a szám eléri a paraméterben megadott toleranciaszintet (amit a futtatáskor kell százalékos értékben megadni), akkor annak a sornak, illetve az oszlopnak a klaszterindexe 0 lesz. Ennek az volt a célja, hogy a sűrű, vagyis sok 1-t tartalmazó vektorok

ne rontsák el a mátrix többi részének a klaszterezését, valamint a metrikaértékeket. Ezeket az elemeket majd a képek bal, valamint a felső részére fognak kerülni.

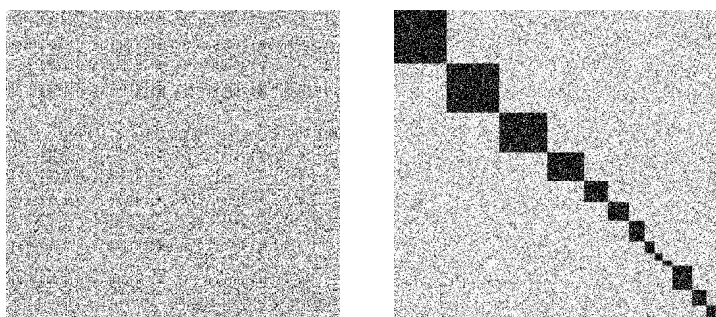
Ezután minden, eddig nem klaszterezett sorhoz megvizsgálom, hogy a többi, szintén nem klaszterezett sor mekkora (Hamming) távolságra van. Ha ez az érték kisebb, mint a paraméterben megadott küszöbérték, akkor a két klaszternek azonos lesz a klaszterindexe.

Az oszlopok klaszterezéséhez felhasználtam a fentebb vázolt eljárás során kapott klaszterindex-vektort (amely a sorok klaszterindexeit tartalmazza). Ebből határoztam meg az oszlopok indexeit. Ehhez vertikális irányban bejártam a mátrixot és összeszámoltam, hogy egy adott oszlopnál mi annak a sorklaszternek az indexe, ahol 1 szerepel a vizsgált oszlopban. Az így keletkező indexhalmazban szereplő leggyakoribb elem lett az oszlop klaszterindexe.

	2	1	2	2	3	3	2	3	2	3	3	1	1	3	3	3	2	3	3
1	0	1	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0
2	1	1	1	1	0	0	1	0	1	0	0	0	0	0	0	0	1	0	0
3	0	0	0	0	1	1	0	1	0	1	1	0	0	1	1	1	0	1	1
2	1	0	1	1	0	0	1	0	1	0	0	0	0	0	0	0	1	1	0
2	1	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0
3	0	0	0	0	1	1	0	1	0	1	1	0	0	1	1	1	0	0	1
2	1	0	1	1	0	0	1	0	0	0	0	0	0	0	0	1	1	0	0
3	0	0	0	0	1	1	0	1	1	1	1	0	0	1	1	1	0	1	1
1	1	1	1	0	0	0	0	1	0	0	0	1	1	0	0	0	0	0	0
3	0	0	0	0	1	0	0	1	0	1	1	0	0	1	1	1	0	1	1

(3.2. ábra: példa oszlopok klasztercímkezésére)

Az azonos indexek mutatják, hogy mely eljárások, valamint mely tesztesetek tartoznak egy csoportba. Az eredmény kirajzolásánál az azonos indexű elemek egymás mellé kerülnek. A képek kirajzolásának módjáról a dolgozat későbbi részében részletesebben beszámolok.



(3.3. ábra: a kiindulási és a klaszterezett mátrix)

Ennél az algoritmusnál különösen fontos a megfelelő paraméterek kiválasztása, ugyanis ezek nagyban befolyásol(hat)ják az eredményeket. Három ilyen fontos argumentum van. Az egyik, és talán legfontosabb a Hamming-tolerancia megfelelő kiválasztása (amelyet szintén százalékban kell megadni), ugyanis ez fogja megadni, hogy mekkora eltérést engedünk meg a két vizsgált elem között és így közvetetten befolyásolja a klaszterek számát is. Itt jegyezném meg, hogy az egy klaszterben levő két „szélső” - vagyis egymástól legtávolabb levő- elem távolságának a maximuma ennek a toleranciaértéknek a duplája lehet. A másik két fontos paraméterrel azt tudjuk megadni, hogy hány százalékos „telítettség” mellett (vagyis a vektorban levő elemek hány százalékának kell 1-nek lennie) tartozik a sor vagy az oszlop a nullklaszterbe.

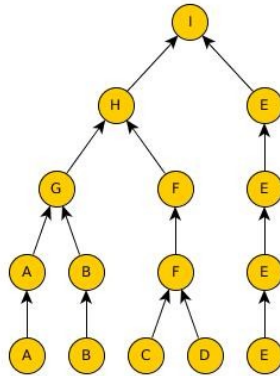
Ha a toleranciaszintet túl alacsonyan határozzuk meg akkor nagyon sok, kevés elemet tartalmazó klaszterünk lesz, ha pedig nagyra állítjuk az értéket, akkor egymástól nagy távolságra levő elemeket is egy csoportba fog sorolni. Hasonlóan ügyelnünk kell a nullklaszter küszöbértékét beállító paraméterekre is, mert ha túl sokat, vagy ha túl keveset sorol oda, akkor nem jön ki az „ötlet” gyakorlati haszna.

A programom futtatása során egyértelműen látszott, hogy csak néhány iteráció, néhány, más paraméterek melletti futtatás után sikerült közelítőleg jól beállítani ezeket az értékeket, ami a program viszonylag nagyobb rugalmasságának volt az ára.

3.1.2. TreeBuild

Ez a hierarchikus klaszterező algoritmus szintén a Hamming távolságra épít, de ez már valamivel komplexebb működésű és emiatt nagyobb erőforrás igényű is.

Ennek az alapkoncepciója az, hogy kezdetben minden elem külön klaszter és ezek közül a legmegfelelőbbnek ítélt két klasztert összeolvasztva egy új klaszter keletkezik (és a két „szülő-klasztert” nem vizsgálja a továbbiakban). Itt a program megalkotása során volt némi mozgástér, hogy mi is legyen az algoritmus megállási feltétele, valamint hogy mikor mondjuk két elemre, hogy azok az összevonásra legalkalmasabbnak ítélt komponensek. Ezen fogalmak definiálásához a legelterjedtebb és legkézenfekvőbb megközelítést használtam: a megállási feltétel egy bizonyos klaszterszám elérése, az összevonásra leginkább alkalmas elempár pedig az, amelyek között a legkisebb a Hamming távolság.



(3.4. ábra: összevonások szemléltetése)

Kezdetben létrehozom (struktúra adatszerkezetben) a kiindulási klasztereket, amik gyakorlatilag a sorok lesznek. Az azonosító a sorindex lesz, a halmazba kerülnek a klaszter elemei, amik kezdetben csak önmaguk, a két egész változóba pedig a két szülő (ami kezdetben -1 és -1).

1. struct node {
2. int azonosito;
3. set<int> klaszterelemek;
4. int apa;
5. int anya;
6. };

Ezután meghatározok két vektort (szülő- és távolságvektort), amely megadja, hogy melyik sorhoz melyik másik sor van a legközelebb és hogy mekkora ez a távolság.

Ezt követően jön az iterációs ciklus. Minden ilyen ciklusban csak egy összevonás történik, azáltal a klaszterek száma egyesével csökken (a paraméterben megadott számig).

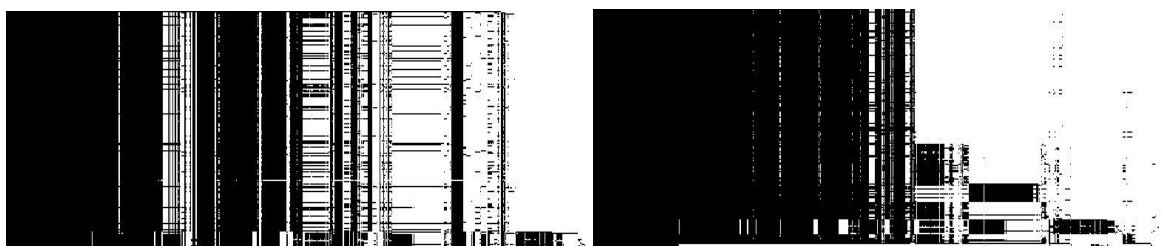
Az iteráció a következő eljárásokat foglalja magába:

- szülő klaszterek keresése
- két szülő összevonása és mátrixbővítés
- a két - fentebb említett - vektor frissítése

A távolságvektorban megkeresem a legkisebb értéket, amelynek segítségével megkapom a két szülő klasztert (a két vektor megfelelő elemeinek indexe segítségével). Ezek kerülnek összevonásra egy közös klaszterré. A gyerek klaszter azonosítója folyamatosan egyel nő, a halmaz a szülő halmazok uniójából kerül ki, a két változó pedig értelemszerűen a szülőket fogja tartalmazni. Az összevonás mellett bővíteni is kell az új klaszterrel a mátrixot, amelyet egy új sor fog reprezentálni. Az új sor adott pozíciójába a két szülő megfelelő elemeinek az átlaga kerül. Vagyis az érték továbbra is $[0,1]$ közé esik, de itt már nem csak természetes számok lehetnek a mátrixban.

Miután ezzel kész vagyunk meg kell határozni, hogy az új sorhoz melyik másik van a legközelebb és az mekkora távolságra van, valamint, hogy a szülővektorban szerepelt-e valahol az előbb összevont két struktúraindex valamelyike, vagyis ellenőriznünk és garantálnunk kell, hogy minden elemhez csak olyan elem lehessen a legközelebb, amely még nem volt szülő (ugyanis olyan elemmel nem lehetne összevonni). Ezért ha a szülő-vektorban az előbb összevont két sor indexe közül valamelyik szerepel, akkor arra a sorra újra meg kell határozni a legkisebb távolságot és a hozzá tartozó szülőpárt.

Az iteráció végén „megmaradt” struktúrák fogják alkotni a klasztereket és a halmaz elemei pedig az egy klaszterben levő sorindexeket. Ennek segítségével meghatározzuk a mátrixhoz tartozó oszlopvektort, amely a sorok klaszterindexeit tartalmazza, és amelynek segítségével - hasonlóan a HammingDiffClassifier algoritmushoz - megkapjuk az oszlopok klaszterindexét, ill. amivel ki tudjuk rajzolni a képet (de ezt egy másik program végzi, aminek a bemutatására a későbbiekben még visszatérek).



(3.5. ábra: a klaszterezés előtti és a klaszterezés utáni mátrix)

Ennél az algoritmusnál jóval kevesebb paramétert kell megadni. Elég csupán a kiindulás mátrix, a klaszterek száma, illetve egy .txt fájl, ahova a sorok klaszterindexei mentésére kerülnek (amit aztán a kirajzolásnál és a metrikaszámolásnál újból felhasználunk).

3.2. Halmazvizsgálatokon alapuló eljárások

Az ilyen típusú algoritmusoknál elsősorban nem a távolságra támaszkodunk, pontosabban nem szigorú értelemben használjuk a távolság fogalmát. Itt a vektor és a távolság helyett a halmaz, annak mérete, azok metszete, valamint uniója lesznek a kulcsfontosságú fogalmak. Egy sorhoz tartozó halmazon azokat az eljárásokat értjük, amelyek a vizsgált sorban 1-es értéket vesznek fel. Oszlopoknál azok a tesztesetek fogják alkotni a halmazt, ahol a mátrix adott oszlopában 1-es áll.

Emellett abban is eltér a Hamming távolságon alapuló eljárásoktól, hogy ezeknél az algoritmusoknál a sorok és az oszlopok klaszterezése egymástól független eljárással készül. Vagyis a sorok klaszterezésének az eredménye nem fogja közvetben befolyásolni az osztályok klasztercímkeit.

Három, egymáshoz viszonylag hasonló eljárást fogok bemutatni, amelyek lényegében „csak” a képletben fognak eltérni egymástól. Minden algoritmusnak kétféle megközelítést implementáltam. Az egyik a *k-means* [6] algoritmust használja az klaszterezéshez, amelyhez a GNU Octave [7] nevű, opensource programot hívtam segítségül. A másik megközelítés szerint a felhasználó megad paraméternek egy [0,1] közötti küszöbszámot, amely a klaszterezés elvégzéséhez szolgál alapul (lentebb részletesebben bemutatom ennek a paraméternek a szerepét és a „használatát”).

3.2.1. Jaccard

Az első ilyen típusú megoldás alapját a Jaccard hasonlóság adja, amely két halmaz egymáshoz való viszonyára ad egy 0 és 1 közötti értéket. A hasonlóság értékét a két halmaz metszetének és a két halmaz uniójának számosságának a hányadosából kapjuk.

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}.$$

(3.6. ábra: a Jaccard hasonlóság képlete [8])

Az eljárás során minden sort minden sorral, valamint minden oszlopot minden oszloppal összehasonlítottam és a Jaccard hasonlóság képlete alapján - amelyben az A és B

ezen elemeket reprezentálja - kiszámolom az adott elempárokhoz tartozó értéket. Ezáltal egy bemeneti, $n \times m$ -es mátrixból két, $n \times n$ -es, ill. $m \times m$ -es mátrix keletkezett, ahol a főátlóban mindenhol 1-k, azon kívül pedig $[0,1]$ közötti értékek szerepelnek.

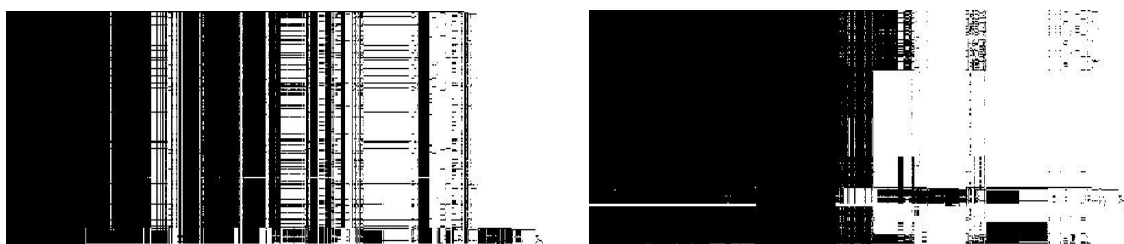
Az egyik megközelítésben ezt a két mátrixot kimentem egy-egy .txt fájlba, amit aztán a GNU Octave-ban beolvasok -a dlmread() függvény segítségével- és a programban megvalósított *k-means* algoritmus segítségével feldolgozom.

A *k-means*, vagy más néven *k*-közép klaszterezés az egyik legnépszerűbb, legelterjedtebb klaszterező algoritmus.

Az eljárást alapvetően négy lépésre lehet osztani [9]:

- kiválasztja a klaszterek számát (k).
- véletlenszerűen létrehoz k számú klasztert, és meghatározza minden klaszter közepét, vagy azonnal létrehoz k véletlenszerű klaszter középpontot.
- minden egyes pontot abba a klaszterbe sorol, amelynek középpontjához a legközelebb helyezkedik el.
- kiszámolja az új klaszter középpontokat.

Az algoritmus addig ismétli az utolsó két lépést, amíg a megállási feltétel nem teljesül. Ez a feltétel leggyakrabban az, hogy az iterációban már ne változzon egyetlen klaszterindex sem (azaz ne történjen átsorolás másik klaszterbe).



(3.7. ábra: a klaszterezés előtti és a klaszterezés utáni mátrix)

Előnye, hogy gyors és nagyméretű adatokat is hatékonyan tud kezelni, ugyanakkor a benne levő randomizálás miatt két, egymás után - azonos paraméterekkel történő - futtatás nem ugyanazt az eredményt fogja adni.

A `kmeans()` függvénynek paraméterként meg kell adni, hogy mennyi klaszterre szeretnénk bontani az inputot. Az így előállított két vektort a `dmlwrite()`-val kimentve, majd az általam implementált kirajzoló programba beolvasva megkapjuk a klaszterezett képet.

A másik megközelítés szerint paraméterben meg kell adni két, $[0,1]$ közötti számot, amely mintegy toleranciaszintként fog funkcionálni - az egyik argumentum a sorokra, a másik az oszlopokra vonatkozik. Ha két elem Jaccard hasonlósága eléri az adott szintet, akkor az a két elem egy klaszterbe tartozik. Fontos megemlíteni, hogy egy elem csak egy klaszterbe tartozhat, szóval, ha már egyszer valamikor elérte a toleranciaszintet, akkor a későbbiekben már azt az elemet nem kell vizsgálnunk. Itt szintén fontos a megfelelő toleranciaérték kiválasztása a túl nagy vagy a túl kicsi klaszterek elkerülése végett. Ahhoz, hogy ezt megtaláljuk általában - hasonlóan a HammingDiffClassifier-hez - több futtatás szükséges.

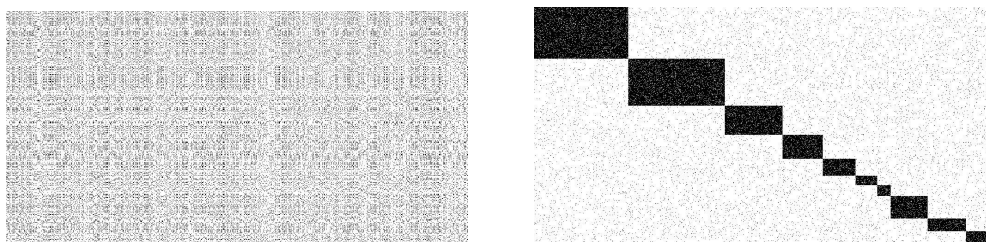
3.2.2. Dice

Ez az algoritmus nagyon hasonlít a Jaccard hasonlóság alapúra, de itt más képlettel számoljuk ki az elempárra (A-ra és B-re) vonatkozó metrikát. Ebben az esetben a képlet számlálójában a két elem metszetének a számosságának a duplája szerepel, míg a nevezőben a két elem számosságának az összege [10].

$$QS = \frac{2|A \cap B|}{|A| + |B|}$$

(3.8. ábra: a Dice hasonlóság képlete)

Ebben az esetben is implementálva lett mind a két megközelítés, amelyeknek a bemenete, a kimenete, valamint a felhasznált segédeszközei megegyeznek a Jaccard hasonlósággal klaszterező programéval ezért arra nem térnék ki ismét.



(3.9. ábra: a klaszterezés előtti és a klaszterezés utáni mátrix)

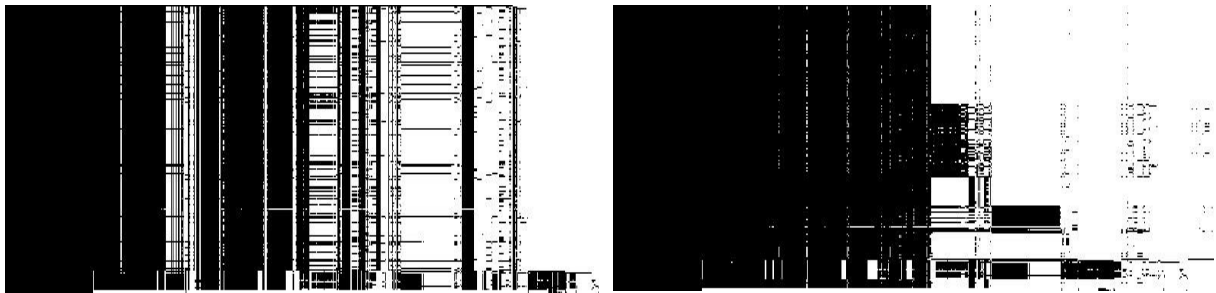
3.2.3. Ochiai

Ez a lehetséges megoldás a koszinusz hasonlóság egy speciális esetére, az Ochiai koefficiensre épít. Itt a számlálóban a vizsgált elempár (legyenek azok sorpárok vagy oszloppárok) metszetének a számossága szerepel, a nevezőben az elemek számosságának a szorzatának a gyöke.

$$K = \frac{n(A \cap B)}{\sqrt{n(A) \times n(B)}}$$

(3.10. ábra: az Ochiai együttható képlete [11])

Hasonlóan a fentebb említett két eljáráshoz, itt is megvalósítottam mind a két megközelítést, amelyek az eddigieknek megfelelő paraméterezéssel futtathatóak.



(3.11. ábra: a klaszterezés előtti és a klaszterezés utáni mátrix)

3.3. Függvények hash lenyomatain alapuló eljárások

A harmadik, a fenti kettőtől merőben eltérő hozzáállás a hash függvényekhez hasonlóan működik, de az eljárásban módosítottam egy fontos részletet. Ez egész ötlet azon alapszik, hogy több, különböző függvény esetén a közös klaszterbe tartozó elemek adott modulussal vett „kiértékeléseinek” leggyakoribb eleme ugyanaz lesz.

3.3.1. Hash

A programnak meg kell adni néhány paramétert, amely alapján elkészíti a számunkra megfelelő „minőségű” és mennyiségű hash függvényeket.

Ezek a argumentumok pedig:

- a modulus értéke
- a függvények száma
- a szorzó és a konstans maximális értéke

Ezen paraméterek megadása után a program legenerálja a kért számú és „minőségű” függvényt. Ezek $a * x + b$ alakúak lesznek, ahol az a és a b véletlen szám, de értékük nem lehet nagyobb, mint az argumentumban megadott rájuk vonatkozó maximális érték. Ezzel és a modulussal már készen is van a függvénybázisunk, amelyen elvégezhetjük a behelyettesítéseket.

Maga a kiértékelés a következőképpen zajlik: elkezdjük bejárni a mátrix sorait. Megnézzük, hogy mely pozíciókban szerepel 1-es érték. Amikor találunk egy ilyet, akkor kiszámoljuk az összes $a * x + b \bmod c$ függvény értékét úgy, hogy az x helyére az oszlopindexet helyettesítjük be. Az így keletkező eredményeknek megfelelő számlálót növeljük, majd folytatjuk a sor további vizsgálatát. Ha ismét 1-t találunk, akkor újra elvégezzük a fenti műveletet. Ha végeztünk az adott sorral, akkor megnézzük, hogy melyik számláló értéke a legmagasabb, vagyis melyik végeredmény jött ki a legtöbbször a kiértékeléskor. Ezt a $[0, \text{modulus}-1]$ érték közti egész számot fogja megkapni címkének az adott sor. Az összes sorra hasonlóan elvégezzük a bejárást.

Az algoritmus szerint azok a sorok fognak egy klaszterbe tartozni, amelyek azonos címkével rendelkeznek. A kirajzolás miatt az utolsó lépésben normalizálom az eredményt, vagyis úgy módosítom a címkézést, hogy a klaszterindexek folyamatosan, egyesével növekedjenek és ne forduljon elő olyan eset, hogy $a < b < c$ esetén a és c létező klaszterindex, b viszont nem.

Ugyanezt az eljárást elvégzem az oszlopokra is, de ott az x helyére nem az oszlop, hanem a sorindex kerül behelyettesítésre. Az így kapott két vektor pedig alkalmas arra, hogy a kirajzoló programba beolvasva előállítsuk a képeket és megkapjuk a metrikaértékeket.

Lássunk egy példát:

0	0	1	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0
1	1	1	1	1	0	0	1	0	1	0	0	0	0	0	0	0	1	0	0	0
2	0	0	0	0	1	1	0	1	0	1	1	0	0	1	1	1	0	1	1	1
3	1	0	1	1	0	0	1	0	1	0	0	0	0	0	0	0	1	1	0	0
4	1	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0
5	0	0	0	0	1	1	0	1	0	1	1	0	0	1	1	1	0	0	1	1
6	1	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0	1	1	0	0
7	0	0	0	0	1	1	0	1	1	1	1	0	0	1	1	1	0	1	1	1
8	1	1	0	0	0	0	0	1	0	0	0	1	1	0	0	0	0	0	0	0
9	0	0	0	0	1	0	0	1	0	1	1	0	0	1	1	1	0	1	1	1

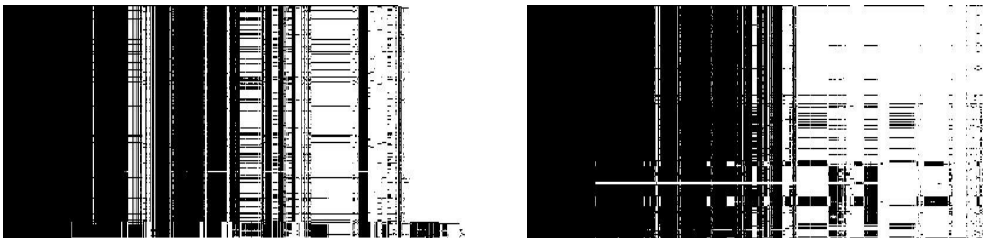
(3.12. ábra: a példa kiindulási mátrixa)

$3x + 3 \bmod 4$	$5x + 2 \bmod 4$	$3x + 7 \bmod 4$	$9x + 1 \bmod 4$	$7x + 5 \bmod 4$	$4x + 3 \bmod 4$
------------------	------------------	------------------	------------------	------------------	------------------

(3.13. ábra: a generált függvények)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
$3x + 3 \bmod 4$	3	3	2	2	2	1	2	2	0	2	2	3	3	2	2	1	2	0	2	2
$5x + 2 \bmod 4$	2	2	3	3	3	0	3	3	1	3	3	2	2	3	3	0	3	1	3	3
$3x + 7 \bmod 4$	3	3	2	2	2	1	2	2	2	2	2	3	3	2	2	1	2	0	2	2
$9x + 1 \bmod 4$	1	1	2	2	2	3	2	2	0	2	2	1	1	2	2	3	2	0	2	2
$7x + 5 \bmod 4$	1	1	0	0	0	3	0	0	2	0	0	1	1	0	0	3	0	2	0	0
$4x + 3 \bmod 4$	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
Segéd-indexek	3	3	2	2	2	3	2	2	0	2	2	3	3	2	2	3	2	0	2	2
Végső indexek	1	1	2	2	2	1	2	2	3	2	2	1	1	2	2	1	2	3	2	2

(3.14. ábra: a klaszterezés eredménye)



(3.15. ábra: a klaszterezés előtti és a klaszterezés utáni mátrix)

3.4. A mátrixok kirajzolásáról

A vizualizáció segítségével meg tudjuk jeleníteni a klaszterezés eredményét, ezzel megkönnyítve annak első körben történő – szubjektív - értékelését.

A programok alapvetően két paramétert kérnek a kirajzoláshoz: a kimeneti kép nevét, ill. egy 1 vagy annál nagyobb természetes számot, ami a skálázási érték lesz. Ez fogja majd a nagyméretű mátrixot - amely akár több tízezer sort és oszlopot is tartalmazhat - kisebb méretűre skálázni. Emellett opcionálisan megadható még két paraméter, amely szintén 1, vagy annál nagyobb egész szám kell hogy legyen. Ezek a klaszterek kirajzolásának a sorrendjét hivatottak koordinálni. Részletesebb bemutatásukra lejjebb kerül sor. Ezeken kívül még vár bizonyos paramétereket, de ez a klaszterező algoritmustól függ.

A klaszterezett kép elkészítése és kiírása háromféle képen lehetséges. Az, hogy melyiket kell használni függ attól, hogy melyik algoritmus eredményét szeretnénk megjeleníteni.

Először ezt a három „típust” vázolom fel, majd utána megemlítenék néhány, mindháromra jellemző dolgot, működési elvet.

3.4.1. HammingDiffClassifier eredményének kirajzolása

Mindközül ez a legegyszerűbb kiírás, mármint olyan szempontból, hogy a rajzolást végző folyamat a klaszterező programban van megvalósítva, nem szükséges hozzá másik program elindítása. Elegendő a klaszterező program argumentumában megadni a kép nevét (az elérési úttal), valamint a skálázási konstanst és a futás végére már el is készül a .pgm fájl. Magát, a tényleges kiírást végző függvényt nem én implementáltam, ez már rendelkezésemre állt a SoDA-ban, nekem a megfelelő sorrendben való rajzolásra kellett figyelnem.

3.4.2. A hierarchikusan klaszterező algoritmus (Tree) eredményének kirajzolása

A hierarchikus klaszterezés eredményének vizualizációjához már egy másik program elindítása szükséges. Ez a program a Tree algoritmus eredménye alapján klaszterezi az oszlopokat valamit elvégzi a kép elkészítését is.

A program hívásakor meg kell adnunk a eredeti bináris mátrixot, a Tree algoritmus eredményeként előálló oszlopvektort (ami a sorok klaszterindexeit tartalmazza), a skálázási konstanst, valamint a kimeneti fájl nevét (elérési úttal együtt). Opcionálisan megadható a fentebb említett két, a rendezést segítő paraméter is.

Első lépésként beolvassuk a mátrixot, majd a .txt fájlból a sorokra vonatkozó klaszterindexet tartalmazó oszlopvektort. Ezután a program a HammingDiffClassifier oszlopklaszterező mechanizmusa szerint megnézi, hogy mely index fordul elő leggyakrabban a vizsgált oszlopban ott, ahol a sor és oszlop találkozásban 1 érték szerepel. Ezt minden oszlopra elvégezve megkapjuk az azokhoz tartozó indexeket. Ezt követően meghatározom a kirajzolandó mind sor-, mind oszlopklaszterek sorrendjét, majd egy ciklussal beállítom az adott pozícióban a megfelelő értékeket.

3.4.3 A particionálóan klaszterező algoritmusok eredményék kirajzolása

Ennek a programnak a működése hasonlít a hierarchikus klaszterezéssel előállított eredményt megjelenítő algoritmuséra, de itt nem egy vektort kell beolvasni és egy másikat pedig a programon belül meghatározni, hanem már mind a kettő a rendelkezésünkre áll, így csak be kell olvasni őket. Ebben az esetben is a rajzolás előtt meghatározom a klaszterek kiírásának a sorrendjét.

3.4.4. A klaszterek kiíratási sorrendjének meghatározása

Erre a „trükkre”, azért volt szükség, mert ha csak például lexikografikus rendezést hajtottam volna végre akkor a képen a sötétebb klaszterblokkok a mátrixban szétszórva lettek volna. Ezzel az elgondolással viszont valamelyest sikerült a főátló menti területre irányítanom ezeket a blokkokat. Legalábbis bizonyos esetekben ...

Az első lépésként klaszterpárokat határoztam meg. Minden sorklaszterhez hozzárendeltem egy, korábban még máshoz hozzá nem rendelt oszlopklasztert. Ezen párok kialakításánál a klasztereken belüli pontok sűrűségét vettem alapul. Megnéztem, hogy az egyes sorklaszterek esetén melyik oszlopindex hányszor fordul elő azokban a pozíciókban,

ahol a mátrix értéke 1. Ezek közül a leggyakoribb, eddig más sorklaszterhez hozzá nem rendelt indexet lesz kiválasztva.

Második lépésben a sorok sorrendjének megadása végzem. Ez úgy történik, hogy megnézem az előbb meghatározott klaszterpárokban levő 1 értékek számát és a szerint csökkenő sorrendbe teszem az klaszterindexeket. Ezen kívül a korábban említett, opcionálisan megadható paraméterben kapott számot felhasználva megadhatjuk, hogy ha a második legnagyobb értéket elérő klaszterpárban levő 1 értékek száma és ennek a konstansak a szorzata nagyobb, mint a legnagyobbként kiválasztott klaszter akkor ez fog mellé - pontosabban alá - kerülni. Ez gyakorlatilag azt jelenti, hogy ha a második legtöbb 1 értéket tartalmazó klaszter adott „közelségben van” a legnagyobbhoz, akkor a képen alá fog kerülni.

A harmadik lépésben az oszlopklaszterek sorrendjének meghatározása történik. Ehhez sorba megnézem, hogy az egyes sorklaszterekhez esetén mennyi 1 érték szerepel egy-egy oszlopklaszterben. Ezek közül kiválasztom a legnagyobb, eddig még ki nem választott indexűt. Itt is, a második lépésben említetthez hasonló szerepet tölt be az opcionális konstans, vagyis egy eddig ki nem választott indexű klaszterben, ha elég sok 1 szerepel, akkor az aktuálisan kiválasztott klaszter mellett kerül majd kirajzolása.

Ha ezzel kész vagyunk, akkor a két vektor segítségével - amikben az eredményeket tároltuk - egy összetett ciklussal megoldható a megfelelő sorrendben történő megrajolás.

3.5. Az implementációról néhány észrevétel

Az algoritmusok megvalósítása közben több nehézsége is ütköztem.

Az egyik ilyen volt a műveletigény. Mivel az algoritmusok sok iterációt tartalmaznak ezért egy nagyobb mátrix esetében ez a műveletidő meredek emelkedését eredményezte. A minél rövidebb futási idő érdekében alkalmaztam néhány “trükköt”, amely nagyban segítette a felhasznált idő leredukálását.

Az egyik ilyen megoldás az volt, hogy a binárisból beolvasott *boolean* értékeket nem egyesével tároltam el, hanem a sorokat, valamint az oszlopokat adott méretű blokkokra osztottam. Ezeket az adott hosszúságú, 0-t és 1-t tartalmazó vektorokat kettes számrendszerbeli számként kezeltem, majd átalakítom decimális számrendszeri számmá. Így nem egy nagy, *boolean* tömböm lesz, hanem kettő, kisebb *integer* értékeket tartalmazó.

Ezután ha a két elem viszonyát vizsgálom (legyen szó akár Hamming távolságról vagy logikai ÉS kapcsolatáról) akkor nem az eredeti, hosszú vektort kell megnéznem, hanem ezt az új, rövidebb komponenseket. Ehhez a vizsgálathoz két függvényt használtam.

```
unsigned hamming_dist(unsigned x, unsigned y){
    unsigned dist = 0, val = x ^ y;
    while(val){
        ++dist;
        val &= val - 1;
    }
    return dist;
}

unsigned logical_AND(unsigned x, unsigned y){
    unsigned dist = 0, val = x & y;
    while(val){
        ++dist;
        val &= val - 1;
    }
    return dist;
}
```

(3.16. ábra: a két függvény a sebességnöveléshez)

Egy másik, gyakran használt eljárás a gyorsításhoz a library-ben implementált két függvényt, a *bitMatrix->getRow([index]).count()* -t és a *bitMatrix->getCol([index]).count()* -t használja. Ezek megadják, hogy a mátrix sorában, illetve oszlopában mennyi 1 érték szerepel. Ha egy elemhez keressük a legközelebbi másik elemet, és az eddig talált legkisebb távolság kisebb, mint a két elem függvényének visszaadott értékeinek a különbségének az abszolút értéke, akkor az a két elem már biztosan nem a legközelebbi.

Például: ha van 4 darab sorunk (*A*, *B*, *C*, *D*) és azt tudjuk, hogy az *A* és a *B* között az eltérés 4, valamint a *getRow(A).count()* függvény értéke 10, a *getRow(B).count()* függvényé 14, a *getRow(C).count()* -é 11, a *getRow(D).count()* -e pedig 15. Ekkor tudjuk, hogy az *A* és a *C* közötti különbség legjobb esetben 1 lehet (vagyis $|10-11|$). Emiatt ezt a két elemet össze kell hasonlítanunk. Viszont az *A* és *D* között a legjobb esetben is 5 ($|10-15|$) lesz az eltérés, ami biztosan rosszabb, mint az eddigi "lokális optimum", ezért ennek az elempárnak a részletes összehasonlításával nem kell foglalkoznunk.

Egy másik módszer a gyorsításra valamelyest ide kapcsolódik: ennek az a lényege, hogy nem vagyunk feltétlenül kíváncsiak minden elempár közötti távolságra. Amennyiben egy elemhez találunk egy olyan másikat, amely esetében a távolság 0, akkor annak az elemnek a vizsgálatát abba lehet hagynunk.

Ezen kívül elkészítettem egy programot, amely gyakorlatilag elforgatja a kiindulási mátrixot 90 fokkal így az oszlopklaszterezés szempontjából újabb lehetőségeket biztosít. Erre jó példa a HammingDiffClassifier algoritmus (részletesebben a 3.1.1 fejezetben), ugyanis ha az eredeti és az elforgatott mátrixon is lefuttatjuk az eljárást, akkor egy újfajta Hamming távolságon alapuló klaszterezést kapunk. Ebben az esetben a sorcímkéket az eredeti mátrixon végzett futtatásként kapott vektor, míg az oszlopcímkéket a transzponált mátrixon végzett futtatásként keletkező vektor adja (és nem a sorcímkék függvényében határozzuk meg, mint a 3.1.1 fejezetben). Ez módszer működik a struktúrával történő klaszterezés esetén is. Az így előállított két vektort (fájlt) a particionáló algoritmusok eredményét megjelenítendő programnak odaadva elkészíti a kimeneti képet, valamint kiszámolja a metrikaértékeket.

4. Metrikák bemutatása és az eredmények értékelése

Mivel a kép megjelenítése nem ad objektív eredményt a klaszterezés „minőségével” kapcsolatban, azaz nem lehet megbízhatóan összevetni az eredményeket ezért úgy döntöttem, hogy definiálok három metrikát, amelyek az algoritmusok jóságát hivatottak jellemezni.

A három metrika különböző szempontból, más-más megközelítést használva írja le az eljárások által létrehozott új mátrixokat, ezáltal komplexebb képet adva az algoritmusok (ilyen szempontból vizsgált) hatékonyságáról.

A definíciójuk megértéséhez két fogalom tisztázása szükséges: mikor mondjuk, hogy egy pont jól, illetve a rosszul klaszterezett.

Ha az adott pont értéke 1 és a hozzá tartozó sor- valamint oszlopcímke kitüntetett klaszterpárt alkot (lásd.: 3.4.4. fejezet) akkor azt mondjuk, hogy az adott pont jól klaszterezett. Ha viszont az oszlopcímkeje nem a sorcímkehez tartozó klaszterpár, akkor rosszul klaszterezettként tekintünk rá. Ezt a definíciót kiterjesztettem a 0 értékű pontokra is, de ezekben az esetben akkor mondjuk, hogy jól klaszterezett, ha a kitüntetett klaszterpáron kívüli a sorcímke-oszlopcímke párosítás.

Ezek tükrében az általam definiált metrikák az alábbi módon néznek ki.

4.1. Metrika 1

Az első metrika csak azokat a mezőket vizsgálja, ahol 1 érték szerepel, és ez alapján határozza meg a mutatót. Az így kapott, [-1,1] közötti érték nem az egész mátrixról, hanem csak az 1 értéket tartalmazó részéről állapítja meg, hogy mennyire jól klaszterezett.

$$M1 = \frac{\text{jól klaszterezett pontok száma} - \text{rosszul klaszterezett pontok száma}}{\text{összes pontok száma}}$$

(4.1. ábra: az első metrika képlete)

Ezzel az aránnyal lehet szemléltetni, hogy hogyan viszonyulnak a jól klaszterezhető pontok az összes ponthoz. Minél jobban közelít ez az érték az 1-hez annál jobb az algoritmus.

4.2. Metrika 2

A második metrika hasonlít az elsőhöz, de itt már nem csak az 1 értékű mezőket vizsgálom, hanem ki van terjesztve a 0 értékűekre is. Így a jól klaszterezett pontok száma két komponensből tevődik össze (lásd 4.2. ábra).

Ezáltal már az egész mátrix klaszterezettségéről kapunk egy mérőszámot, amely - az első metrikához hasonlóan- szintén $[-1,1]$ közé esik.

$$A = \text{jól klaszterezett 1 értékű pont} + \text{jól klaszterezett 0 értékű pont}$$

$$B = \text{rosszul klaszterezett 1 értékű pontok} + \text{rosszul klaszterezett 0 értékű pontok}$$

$$M2 = \frac{A - B}{\text{sorok száma} \cdot \text{oszlopok száma}}$$

(4.2. ábra: a második metrika képlete)

4.3. Metrika 3

Ez a metrika kicsit különbözik a korábbi kettőtől. Ez ugyanis nem azt vizsgálja, hogy a mátrix pontjai közül milyen arányban voltak a jól klaszterezettek, hanem azt mondja meg, hogy a jól és a rosszul klaszterezett pontok milyen arányban vannak egymáshoz képest.

$$M3 = \frac{\text{jól klaszterezett pontok száma}}{\text{rosszul klaszterezett pontok száma}}$$

(4.3. ábra: a harmadik metrika képlete)

Ennek a mérőszámnak az értéke már nem $[-1,1]$ közé esik. Itt egy 0-nál nagyobb szám lesz az eredmény. Minél nagyobb ez a szám, annál „dominánsabbak” a jól klaszterezett pontok.

Ennek a három metrikának az együttesével lehet jellemezni az algoritmusokat. Azt viszont mindenkinek magának kell eldönteni, hogy egy klaszterezésnél mire kívánja tenni a

hangsúlyt, melyik metrika a fontosabb, ugyanis ennek megfelelően más-más klaszterező eljárás lehet a jó megoldás.

Fontos még megemlíteni a metrikákkal kapcsolatban, hogy azoknál az algoritmusoknál, ahol definiáljuk és használjuk a *nullklaszter* fogalmát (vagyis azokat a sorok és oszlopok tartoznak ebbe a halmazba, amelyben sok 1 érték szerepel) lehetőségünk van dönteni arról, hogy ezek a pontok bele legyenek-e számolva a metrikaértékekbe. Ugyanis ha sok ilyen elem van, akkor azok jelentősen torzíthatják a mérési eredményeket.

4.4. Nehézségek és tapasztalatok

Az implementáció során tapasztalt, a sebességgel és a műveleti igénnyel kapcsolatos nehézségeket már említettem a 3.5. fejezetben.

Most kitérnék az eredmények értékelésével kapcsolatban felmerülő problémákra. Ennek a dolgozatnak nem célja e kérdések, problémák megoldása, de fontosnak tartom megemlíteni az ilyen irányú észrevételeimet (amelyek egy esetleges későbbi ez irányú kutatás során szintén előkerülhetnek).

A programok bizonyos információkat írnak ki a standard outputra, amelyek az algoritmusra, illetve magukra a mérési eredményekre vonatkoznak (pl.: éppen a mátrix melyik részét vizsgálja a program, klaszterek száma és azok mérete, stb.). Ezek segítségével olyan adatokhoz jutunk, amelyek segítenek eldönteni, hogy az adott paraméterezés mellett elfogadható eredményt kaptunk-e vagy újabb futtatás szükséges (más paraméterekkel).

A különböző mérések output -jait olvasva több észrevételt is tehetünk az algoritmusokkal és a mátrixokkal kapcsolatban.

Az egyik ilyen a klaszterszámra vonatkozik. Ezzel kapcsolatban kétféle típusú „módszer” van. Az egyik az, amikor közvetlenül megadjuk, hogy hány klasztert szeretnénk kapni, ezáltal megakadályozva a túl sok vagy túl kevés klaszter kialakulását. Ebbe a csoportba tartoznak a *k-means* algoritmust, valamint a struktúrát használó eljárások.

A másik csoport esetében a klaszterek száma a paraméterek függvényében alakul ki (például a HammingDiffClassifier esetében a toleranciaszinttől függ). Ezekben az esetekben a

kívánt klaszterszám eléréséhez általában több futtatás kell, ugyanis elsőre nehéz megmondani, hogy mi lesz azaz érték, ami mellett a megfelelő számú klaszter alakul ki.

Ha fontos, hogy adott számú klaszterünk legyen akkor az első típusú megoldások a célra vezetőek, ha nem ragaszkodunk a kívánt klaszterszámhoz, illetve nagyobb hangsúlyt helyezünk arra, hogy az elemek mennyire hasonlítanak egymásra akkor a másik típusú is megfelelő.

Egy másik észrevétel a halmazok méretére vonatkozik. Itt is kétféle csoportot lehet megkülönböztetni. A *k-means* algoritmust használó eljárások esetében az egyes klaszterek méretei közel azonosak, még a többi eljárás esetében néhány nagy és több kisebb méretű halmaz alakul ki. Ezek a méretbeli különbségek nagyban függenek a klaszterezés paramétereitől is.

Ezen kívül a mátrixok sűrűsége is nagyban befolyásolja a klaszterezés minőségét, ugyanis sűrű mátrix esetében nehéz úgy cserélni a sorokat, valamint az oszlopokat, hogy a kitüntetett klaszterpárokon kívül kevés 1 érték szerepeljen. Ezért az algoritmusok eredményeit közel azonos sűrűségű mátrixokon futtatva lehet csak objektíven összevetni. A vizsgálatok során ezt szem előtt tartva minden algoritmust minden mátrixon lefuttattam, ezáltal az összehasonlíthatóság szempontjából megbízható eredményeket kaptam.

4.5. Eredmények és lehetőségek

Összegezve az eddig leírtakat alapvetően háromféle bemeneti mátrixot használtam (ezekről részletesebben a 2.2 fejezetben lehet olvasni) és ezeket futtattam az általam implementált eljárásokat.

Ezek klaszterező programok két részből (két programból) állnak: az első alkalmazásban hajtódik végre a klaszterezés és ennek eredményeképpen meghatározásra kerülnek a sor-, valamint az oszlopcímkek. A második program ennek a két (vagy adott esetben csak egy) vektornak a segítségével elkészíti a PGM képet, valamint kiszámolja a metrikaértékeket. (A particionáló eljárások esetében egy köztes lépésként jelenik meg az Octave használata, aminek az eredménye később feldolgozásra kerül, valamint a HammingDiffClassifier eljárásban ez a két komponens össze lett vonva.)

Az eredményt (azaz a képet és a mérőszámokat) előállító program több egyéb információt is a rendelkezésünkre bocsát. Ilyen például az, hogy mennyi sor-, illetve oszlopklaszter van a mátrixban, ezeknek mekkora a méretük (vagyis hány elem tartozik bele) és azt, hogy hány darab 1 érték van abban a klaszterben. Ezek mind segítenek annak eldöntésében, hogy elfogadjuk-e az adott eredményt, vagy egy újabb futtatással megpróbálunk egy számunkra jobb klaszterezést elérni.

Amennyiben csak a metrikaértékeket vesszük alapul akkor a következő konzekvenciákat vonhatjuk le a mérésekből:

- Ha fontos az, hogy az általunk megadott számú klaszter alakuljon ki és azok közel azonos méretűek legyenek akkor a *k-means* algoritmust használó eljárásokat (vagy a Jaccard-ra, vagy a Dice-ra, vagy az Ochiai-ra épülőket) célszerű választani.
- Ha ragaszkodunk az általunk megadott klaszterszámhoz, de azok méretét (illetve a méretük eloszlását) kevésbé tartjuk fontosnak akkor a struktúrával történő klaszterezés jó választásnak tűnik.
- Ha rugalmasak vagyunk a klaszterszámot illetően és az egy klaszterbe tartozó elemek közötti - általunk beállítható erősségű - kapcsolatra helyezzük a hangsúlyt akkor a HammingDiffClassifier eljárást javaslom.

Az eljárások által létrehozott halmazok segítségével lehetőségünk nyílik a tesztek prioritizálására, ezek a halmazok szolgáltat(hat)ják az alapot ahhoz, hogy felállítsuk a tesztesetek futtatásának sorrendjét.

Amennyiben a forráskódban változott egy vagy több függvény akkor megnézzük, hogy a mátrixban mely teszteseteknél szerepel abban az adott pozícióban 1 érték. Ezek a lista elejére kerülnek, mivel közvetve vagy közvetlenül, de az adott függvényt tesztelik. Ez után megnézzük, hogy ezek a tesztek melyik klaszterekben vannak. A prioritási sorban a következő elemek azok lesznek, amelyek abban a klaszterben vannak, amelyekben a fent említett tesztek szerepelnek.

Egy másik felhasználása a teszteset-szelekció lehet. Ebben az esetben a fejlesztő és a tesztelő gárda a klaszterezés eredményét átbeszélve és értékelve dönthet arról, hogy mely tesztek azok, amelyeket ki lehet hagyni (mert például redundánsak, ugyanazokat a

függvényeket tesztelik), illetve melyek azok, amelyeket megfelelően átgondolva, adott koncepció mentén haladva össze lehet vonni (és a kettő vagy több teszt helyett egy újat létrehozni).

Amennyiben lehetőségem nyílik rá szeretném a későbbiekben is folytatni az ilyen irányú kutatásokat és a dolgozatomat mintegy alapként felhasználva tovább boncolgatni ezt a problémakört...

Irodalomjegyzék

- [1] : International Software Testing Qualifications Board: Certified Tester Foundation Level Syllabus, 2011
- [2] : <http://hu.wikipedia.org/wiki/Szoftvertesztel%C3%A9s>
- [3] : Ficsor Lajos, dr. Kovács László, Krizsán Zoltán, dr Kusper Gábor: Szoftvertesztelés, Kelet-Magyarországi Informatika Tananyag Tárház, 2009
- [4] : Berend Gábor: Adatbányászat c. kurzus előadásjegyzet, 2013
- [5] : http://www.princeton.edu/~achaney/tmve/wiki100k/docs/Hamming_distance.html
- [6] : <http://www.cs.uvm.edu/~xwu/kdd/Slides/Kmeans-ICDM06.pdf>
- [7] : <https://www.gnu.org/software/octave/>
- [8] : Jeff M Phillips: Jaccard Similarity and Shingling, University of Utah, 2013
- [9] : <http://hu.wikipedia.org/wiki/Klaszteranal%C3%ADzis>
- [10] : <http://sve.bmap.ucla.edu/instructions/metrics/dice/>
- [11] : http://en.wikipedia.org/wiki/Cosine_similarity

Nyilatkozat

Alulírott Vancsics Béla, gazdaságinformatikus szakos hallgató, kijelentem, hogy a dolgozatomat a Szegedi Tudományegyetem, Informatikai Tanszékcsoport Szoftverfejlesztési Tanszékén készítettem, gazdaságinformatikus diploma megszerzése érdekében.

Kijelentem, hogy a dolgozatot más szakon korábban nem védtem meg, saját munkám eredménye, és csak a hivatkozott forrásokat (szakirodalom, eszközök, stb.) használtam fel.

Tudomásul veszem, hogy szakdolgozatomat / diplomamunkámat a Szegedi Tudományegyetem Informatikai Tanszékcsoport könyvtárában, a helyben olvasható könyvek között helyezik el.

Szeged, 2014-05-15

Aláírás

Köszönetnyilvánítás

Köszönöm a dolgozat elkészítéséhez nyújtott segítséget dr. Gergely Tamásnak, aki hasznos észrevételeivel, iránymutatásával, valamint építő szándékú kritikáival nagyban hozzájárult ahhoz, hogy ez a dolgozat megszülethessen.

Emellett szeretnék köszönetet mondani az SZTE TTIK Szoftverfejlesztési Tanszékének, hogy a rendelkezésemre bocsátotta az általuk fejlesztett SoDA library-t valamint az előállított mátrixokat.

Köszönet nekik!