

Szegedi Tudományegyetem
Informatikai Tanszékcsoporthoz

**Iteratív algoritmusok folytonos változók diszkrét
kategóriába sorolásához**

TDK dolgozat

Készítette:
Bali Attila
programtervező informatikus MSc

Témavezető:
Prof. Dombi József
egyetemi docens

Szeged
2014

Tartalomjegyzék

Bevezetés.....	1
Klaszterező algoritmusok.....	2
K-közép (K-means).....	2
Fuzzy K-közép.....	3
Fuzzy maximum likelihood becslés.....	5
Klaszterezés a priori tudás nélkül.....	6
Újszerű iteratív klaszterező megoldások.....	8
Fix számú osztályokba soroló iteratív algoritmus.....	8
Előre nem meghatározott számú osztályba soroló iteratív algoritmus.....	13
Az előre nem meghatározott számú osztályba soroló iteratív algoritmus módosított változata.....	15
Tesztek.....	18
Az algoritmus futásának eredményei.....	18
Fix számú osztályokba soroló iteratív algoritmus.....	18
Előre nem meghatározott számú osztályba soroló iteratív algoritmus.....	20
Futásidő és memória tárterületre vonatkozó további tesztek.....	23
Fix számú osztályokba soroló iteratív algoritmus.....	23
Előre nem meghatározott számú osztályba soroló iteratív algoritmus.....	23
K-közép algoritmus.....	24
Fuzzy K-közép algoritmus.....	24
Fix számú osztályba soroló algoritmus futási idő és memória használat tesztjei.....	26
Fejlesztési lehetőségek.....	29
Összefoglalás.....	31
Forrás.....	32

Bevezetés

Az adatsorok jellemzéséhez elkerülhetetlen feladat az adatok csoportokba való sorolása. Például a természetes nyelvi kommunikáció során számok helyett mennyiséget jellemző szavakat használunk, amelyek számcsoportokra (intervallumokra) vonatkoznak. Nagy adattárházak információ tömörítésére is használható az ilyen csoportosítási eljárás.

Az algoritmusoknak van közgazdasági alkalmazása is. A diszkretizáló algoritmus tekinthető, mint egydimenziós klaszterezés.

A klaszterezés első megjelenése a XX. század elejére tehető. Az adatok automatikus csoportokba rendezését lehetővé tevő algoritmust viszonylag hamar felismerték. A klaszterező algoritmusok iránt megnövekedett érdeklődésért nagyrészt az internet elterjedése növelte. Az interneten az elmúlt években egyre több nagy mennyiségű információ keletkezik, sok esetben felismerhető struktúra nélkül. A felügyelet nélkül tanuló algoritmusok, így a klaszterezés legnagyobb előnye, hogy képesek feltárni az adatok belső felépítését.

A klaszterezés fő tulajdonsága, hogy az egyes adatvektorokat csoportokba (klaszterekbe) sorolja. Erre több algoritmus is ismert. A klaszterezés célja, hogy az adatokat csoportokba rendezze. Az eredmény során úgy alakulnak ki a klaszterek, hogy a hasonló elemek azonos csoportba, a különböző elemek különböző csoportba kerüljenek. Mivel a klasztereket halmazoknak is tekintjük, ezért kapcsolhatjuk a fuzzy algebrához.

A 70-es években Zadeh bemutatta a fuzzy elméletet, ami halmazelmélet egy részterülete. A motivációja bizonytalan fogalmak meghatározása volt. A fuzzy elméletet úgy írhatjuk, hogy a halmaz minden eleméhez hozzárendelünk egy 0 és 1 közötti valós számot, mely azt jelenti, hogy az adott elem „mennyire eleme” a halmaznak. A fuzzy halmaz elnevezés oka, hogy a halmazok határát nem lehet legtöbb esetben pontosan megmondani. Erre a koncepcióra teljes értékű matematikát lehet felépíteni, amelyben az alapvető műveleteknek (mint a metszet vagy az unió) is számtalan variánsa van.

Azzal, hogy egy adatvektor nem tartozik teljesen egy halmazba, lehetőségünk van arra, hogy egy elem akár több halmazhoz is tartozzon, különböző mértékben. A számítás során ezt az információt is felhasználhatjuk, amit számos alkalmazás meg is tesz, egymással átfedő klasztereket létrehozva.

A jelen dolgozatban két algoritmust implementáltunk. Mindkét algoritmus hasonló elvek alapján csoportosít. Az első algoritmus a valós számok (a, b) intervallumának szakaszokra bontását végzi, és előre megadott számú osztályba tudjuk sorolni az elemeket. A kialakult csoportosítás a teljes (a, b) intervallumot lefedik. A klaszterek között nem lehet átfedés. A másik megvalósítás amikor a bemeneti adatvektor osztályainak a számáról nincs előzetes információnk és az algoritmus feladata ezt a számot meghatározni és megadni az adatok osztályozásait. Az algoritmus határozza meg a szükséges klaszterek számát és a felbontást. Az utóbbi algoritmus egy módosított változatát is implementáltuk. Iteratív módszerrel egyesítés és szétbontás műveletét meta-algoritmus végzi, amelynek során kialakulnak a klaszterek határai. A teljes adatsor felosztása során megengedjük, hogy üres klaszterek is kialakulhassanak. A teljes (a, b) intervallumot felosztjuk és az algoritmus terminálásakor üres és nem üres osztályok csoportjait kapjuk eredményül.

Az algoritmusok implementálása során fontos kritérium volt az algoritmus sebessége. Mindkét algoritmus iteratív módszeren alapul. A gyakorlati tapasztalatok alapján, már az első iteráció után is jól közelítő eredményt kapunk.

A fent tárgyalt algoritmusok jelenleg 1 dimenziós adatsoron működnek. A felhasznált egyenletek könnyen adoptálhatók több dimenziós adatok klaszterezésére. A klaszterek és lyukak egyesítésével és szétosztásával lehetőségünk nyílik konkrét alakzatok felismerésére, ami a képfeldolgozás, alakfelismerés területén is nagyon fontos.

Klaszterező algoritmusok

Ebben a fejezetben ismertetem az ismertebb klaszterező eljárásokat és azok pszeudó MATLAB kódját, amely megtalálható Csendes Tibor könyvében.[1].

K-közép (K-means)

A felügyelet nélküli tanulás legegyszerűbb módja a K-közép (K-means) klaszterezés, amelyet MacQueen[3] használta először. Az iteratív algoritmus bemenetei az adatpontok (X) és az eredményül várt klaszterek száma (K), kimenete pedig az adatpontok hozzárendelése klaszterekhez. Az algoritmus inicializálásakor kiválasztunk K darab pontot, melyek a későbbi klaszterek középpontjai (centroid) lesznek. Az iteráció minden lépésében ezután előbb minden adatpontot hozzárendelünk a hozzá legközelebb eső centroid által reprezentált klaszterhez. Ez után a centroidokat megváltoztatjuk, hogy a hozzárendelt adatpontok átlaga legyen. Az algoritmus akkor ér véget, ha centroidok már nem változnak.

```

kmeans(X, K)
U:=0;
for k:=1 to K
  Ck := random value from X
repeat :
  for i:=1 to N
    k:=argmin(d(Ck, Xi))
  Uik:=1
  Ck := average of Xi where Uik=1
until Ck changing

```

Az algoritmus gyors, szeparálható adatokra jól működik. Hátránya, hogy egyrészt ismerni kell előre a klaszterek számát, másrészt csak kb. azonos méretű, többé-kevésbé konvex csoportokat képes felismerni. Ha a klaszterek száma túl sok, az algoritmus felosztja a nagyobb klasztereket, ha pedig túl kevés, összevon közel lévő klasztereket. Ha pedig két klaszter közül az egyik nagy, a másik kicsi, az algoritmus hajlamos rá, hogy a nagyobb klasztert felossza és egy részét a kisebb klaszterhez csatolja.

További problémát jelent, ha a klaszterek átfedőek, hiszen ekkor nem dönthető el egyértelműen, melyik elem melyik klaszterhez tartozik. Ebben az esetben van szükség a fuzzy megközelítésre, ami megengedi, hogy egy-egy elem több klaszterhez is tartozzon más-más mértékben.

Fuzzy K-közép

A K-közép algoritmus kiterjesztése fuzzy halmazokra, amelyet Bezdek[4] talált fel. Az algoritmus a következő hibafüggvény minimalizálásával éri el az eredményt:

$$J_q(U, C) = \sum_{i=1}^N \sum_{k=1}^K u_{ik}^q d^2(X_i, C_k)$$

„u” tagsági függvény kiszámítása a következő képlet alapján történik:

$$u_{ik} = \frac{\left(\frac{1}{d^2(X_j, V_k)}\right)^{\frac{1}{q-1}}}{\sum_{k=1}^K \left(\frac{1}{d^2(X_j, V_k)}\right)^{\frac{1}{q-1}}}$$

A fenti képletekben d egy belső szorzat alapú távolságmérték. A q egy 1-nél nagyobb konstans, mely az algoritmus fuzzyságát méri. Minél nagyobb q értéke, annál inkább fuzzy halmazokat kapunk eredményül. A $q = 1$ határérték pontosan a K-means algoritmust adja. Ennek oka, hogy q csökkentésével a fenti képlet szerint az u_{ik} értéke 1 lesz, ha a távolság minimális, egyébként 0. Az algoritmus MATLAB kódja:

```
function [V,U] = fuzzy_kmeans(X, K, q)
[N, M] = size(X);
eps = 1e-5;
V = rand(K, M);
A = eye(M);
for i=1:100
    D = diag(X*A*X')*ones(1,K) - (V*A*X')'
        - (X*A*V') + ones(N,1)*(diag(V*A*V')');
    U = (1./D).^(1/(q-1));
    Dnorm = 1./(U*ones(K, 1))*ones(1,K);
    U = U .* Dnorm;
    Vp = V;
    V = (U.^q)'*X ./ (((U.^q)'*ones(N,1))*ones(1,M));
    if (max(Vp - V) < eps)
        break;
    end;
end
end
```

A fenti algoritmus néhány paramétert előre rögzít az egyszerűség kedvéért: az iterációk maximális számát (100), az ε értékét (10^{-5}) és a távolság-számításhoz használt A mátrixot (egységmátrix). Emellett a kezdőpont minden esetben 0 és 1 közötti véletlen számokból álló vektorokat jelent, ami akkor használható, ha a jellemzők normalizáltak. A függvény bemenete a vizsgálandó anyag (X), a klaszterek száma (K) és q értéke. A kimenet a klaszterek középpontjai (V) és a tagsági függvény értékei (U). Az algoritmusban A egy pozitív definit mátrix, melyet a jellemzők statisztikai adatai alapján állítunk elő. Az egyszerű fuzzy K-közép algoritmus esetén A az egységmátrix, ekkor a távolság is euklideszi.

Az algoritmus jól skálázódik, nem szeparálható példák esetén jobb eredményt ad az egyszerű K-középnél. Legnagyobb hátránya, hogy csak hipergömb alakú klaszterekre működik jól, a különböző méretű és alakú klaszterek esetén még mindig a K-közép hibáit produkálja.

A fuzzy K-közép algoritmus fent említett problémájára Gath és Geva 1989-ben kidolgozott egy eljárást [2]. Ötletük az általános távolságmértéken alapul, azonban számos más tényezőt is figyelembe vettek.

Fuzzy maximum likelihood becslés

Az A mátrix szerepe akkor kerül előtérbe, ha a klaszterek eltérő a sűrűsége és számossága vagy hiperellipszoid jellege miatt az algoritmus nem működne megfelelően. Az alap algoritmus a gömbszerű klasztereket tudja legjobban felismerni, a mátrix pedig ezt a gömböt transzformálja hiperellipszoiddá. Ekkor egy maximum likelihood becslést készíthetünk a távolságmértékre, amit a posteriori valószínűségek kiszámítására tudunk felhasználni:

$$h(k|X_i) = \frac{\frac{1}{d_e^2(X_i, V_k)}}{\sum_{j=1}^K \frac{1}{d_e^2(X_i, V_j)}}$$

$$d_e^2(X_i, V_k) = \frac{\sqrt{\det(F_k)}}{P_k} e^{\frac{(X_i - V_k)^T F_k^{-1} (X_i - V_k)}{2}}$$

Ebben a képletben P_k a k . klaszter választásának *a priori* valószínűsége, F_k pedig a klaszter fuzzy kovariancia-mátrixa. Látható, hogy h megegyezik a u_{ik} -vel $q = 2$ esetén. Ez könnyen érthető, ha a tagsági függvényt a halmazhoz tartozás valószínűségként fogjuk fel. Az analógia

matematikailag nem teljesen korrekt, de jelen helyzetben segíti az intuitív megoldást. Ha tehát az algoritmusban kicseréljük u_{ik} -t $h(k|X_i)$ -re, egy új algoritmust, a fuzzy maximum likelihood becslést kapjuk. Azonban még mindig ismeretlen két tényező: P_k és F_k . Ezek kiszámítása az új centroidok kiszámítása után, a következőképpen történik:

$$P_k = \frac{1}{N} \sum_{i=1}^N h(k|X_i)$$

$$F_k = \frac{\sum_{i=1}^N h(k|X_i)(X_i - V_k)(X_i - V_k)^T}{\sum_{i=1}^N h(k|X_i)}$$

Az algoritmus nagy előnye, hogy akár nagyon eltérő jellemzővektorok és elemsűrűségek mellett is jól teljesít, de csak akkor, ha a bemenete már egy viszonylag jó közelítés. Ez a távolságfüggvény exponenciális jellegéből fakad, ami miatt az optimumot az algoritmus csak a lokális környezetben keresi. Legjobb felhasználási módja egy már korábban létrejött klaszterezés eredményének finomítása.

Klaszterezés a priori tudás nélkül

A K-közép alapú algoritmusok egyik legnagyobb problémája az inicializálás: az algoritmusok a lokális optimumhoz konvergálnak, így különböző kezdőpontok különböző eredményt hozhatnak. Számos gyakorlati alkalmazásban nincs előzetes ismeretünk a klaszterek elhelyezkedéséről. Jó példa erre a szegmentálás: tudjuk, hogy a képen hány alakzatnak kell lennie, de azt nem, hogy hol vannak. Kiindulhatunk abból, hogy ha már van egy klaszterezésünk k partícióra, akkor a $k+1$. középpontot olyan pontok közelébe logikus elhelyezni, ahol a tagsági függvények értékei alacsonyak. Innentől az algoritmus:

1. Helyezzük az első pontot az összes pont átlagának megfelelő pozícióba
2. Válasszunk ki egy új középpontot, mely minden adatponttól egyenlő messze van (nem valós pozíció)
3. Futtassuk le a fuzzy K-közép algoritmust az eddigi pontokra
4. Ha nincs még elég klaszter, vissza a 2. lépésre, egyébként kész

Nagyobb probléma, ha azt sem tudjuk, hány klaszterre van szükségünk. Tipikus példa, amikor egy hipermarket próbálja meghatározni a vásárlói szegmenseket a pénztárnál rögzített eladási lista

alapján, de lehet bármi olyan helyzet, amikor semmit sem tudunk az adatok felépítéséről. Ekkor célszerű több lehetséges K értékkel futtatni az algoritmust és kiválasztani a legjobbat. Ez azonban igényli az algoritmus eredményeinek jóság-mértékét, azaz egy adott K értékre mennyire lesznek jók a klaszterek. Feltételezhetjük, hogy a jó klaszter nem túlzottan fuzzy, így az elemek többsége esetében eldönthető, hova tartozik. Ezen felül szükséges az alábbi három feltétel:

- A klaszterek tisztán szétválaszthatóak
- Klaszterek minimális térfogata
- A pontok többsége a centroidok közelében csoportosul

Habár fuzzy környezetről van szó, az osztályozás célja jól definiált csoportok létrehozása, ami egy „crisp” halmazos felbontást eredményez. A klaszterek térfogata (inkább hipertérfogata) és átlagos sűrűsége a következő módon számítható:

$$F_{HV} = \sum_{k=1}^K \sqrt{\det(F_k)} \quad D_{PA} = \frac{1}{K} \sum_{k=1}^K \frac{S_k}{\sqrt{\det(F_k)}}$$

Ahol F_{HV} a térfogat, D_{PA} a klaszterek átlagos sűrűsége. Az utóbbiban szereplő S_k a centrális elemek, vagyis a centroidtól a standard eloszláson belül található elemek tagsági függvényének összege:

$$S_k = \sum_{i \in I} u_{ik} \quad I = \{i | (X_i - V_k) F_k^{-1} (X_i - V_k)^T < 1\}$$

Így a particióssűrűség (P_D) kiszámítása:

$$P_D = \frac{\sum_{k=1}^K S_k}{F_{HV}}$$

Ezek a változók együttesen felhasználhatóak az algoritmus jóságának mérésére. F_{HV} általában tisztán mutat egy szélsőértéket, D_{PA} pedig előnyben részesíti a sűrű klasztereket. Emiatt például sűrű és ritka klasztereket vegyesen eredményező futás jónak lesz minősítve a sűrű klaszterek miatt.

Újszerű iteratív klaszterező megoldások

A jelen dolgozatban három algoritmust implementáltunk. A három algoritmus hasonló elvek alapján csoportosít. Az első algoritmus (Fix számú osztályokba soroló iteratív algoritmus) a valós számok (a, b) intervallumának szakaszokra bontását végzi, és előre megadott számú osztályba tudjuk sorolni az elemeket. A kialakult osztályok között nem lehet átfedés. A második algoritmus (Előre nem meghatározott számú osztályokba soroló iteratív algoritmus) bemenetként megkapja az adatvektort, de a klaszterek számát nem. Az algoritmus a futása során határozza meg a szükséges klaszterek számát és a csoportok felbontást. A harmadik algoritmus, ami a második algoritmus egy módosított változata, annyiben tér el az eredeti algoritmustól, hogy itt a csoportosítás során megengedettek az üres osztályok. Az algoritmus a bemeneti adatvektort üres és nem üres osztályok sorozatára bontja fel.

Fix számú osztályokba soroló iteratív algoritmus

Az algoritmus bemenetként egy egy-dimenziós adatvektort és a leendő klaszterek számát várja. Az algoritmus a futása során a bemeneti adatsort pontosan a megadott klaszterszámú osztályra fogja felbontani. Az algoritmus minden iterációjában pontosan a megadott számú klaszterre bontja az adatvektort.

Az algoritmus a futása során 5 fázist különböztethetünk meg:

1. fázis: A bemeneti adatok iniciális klasztereinek létrehozása.

A bemeneti adatvektor elemeit a megadott klaszterszámú osztályra kell bontanunk. Az osztályozás ebben az esetben az adatsor elemszámán alapul. Első lépésnek rendeznünk kell az elemeket növekvő sorrendben. Minden klaszterben közel egyenlő számú elem tartozik. A létrejövő osztályok határait a bennük lévő első és utolsó elem határozza meg. Az első osztály

intervallumának minimumát az adatsor első eleme, a K-adik osztály intervallumának maximumát az adatsor utolsó eleme határozza meg.

... ÁBRA !!!

2.fázis: Kiszámoljuk a klaszterek súlyait.

A következő lépésben kiszámoljuk a klaszterekhez tartozó súlyokat az alábbi képlet alapján. A kiszámolt súlyok tulajdoni hányadot fejeznek ki. Egy klaszter intervallumában az adott x érték mennyire tartozik bele. K_j lesz a j-edik „szoft” klaszterbe tartozó elemek száma.

A klaszter K értékének kiszámítása:

$$K_j = \sum_0^N \delta_j(x_i)$$

ahol i : az adatsor elemének indexe
 j : az aktuális klaszter indexe
 N : az adatsor elemszáma

A k-adik klaszterre vonatkoztatva az x_i elem súlyának kiszámítása:

$$w_k(x_i) = \frac{\delta_k(x_i)}{K_k}$$

ahol N : adatsor elemeinek száma
 i, j : adatsor elemének indexe
 k : aktuális klaszter indexe

A súlyok kiszámításához használt $\delta_j(x_i)$ függvény a „felfújó” függvény. Az adott klaszter (a, b) intervallumán egy felfújást határoz meg. Az adatsor elemeinek a felfújó függvényben felvett értékei alapján végezzük a súlyok kiszámítását.

A „felfújó” függvény az alábbi képlettel számolható:

$$\delta_i^{(\lambda_1, \lambda_2)}(x) = \frac{1}{1 + \frac{\lambda_2}{\lambda_1 + \lambda_2} e^{-\lambda_1(x-a)} + \frac{\lambda_1}{\lambda_1 + \lambda_2} e^{-\lambda_2(b-x)}}$$

ahol i : aktuális klaszter indexe

x : az adatsor aktuális értéke

λ_1, λ_2 : a függvény jobb és bal oldalának meredekségét szabályzó változó

a : a klaszter intervallumának minimuma

b : a klaszter intervallumának maximuma

Az algoritmus futása során szimmetrikus λ értékeket használunk, ahol a jobb oldali és a bal oldali meredekség azonos. Tehát a fenti képlet egy egyszerűsített változatát használjuk.

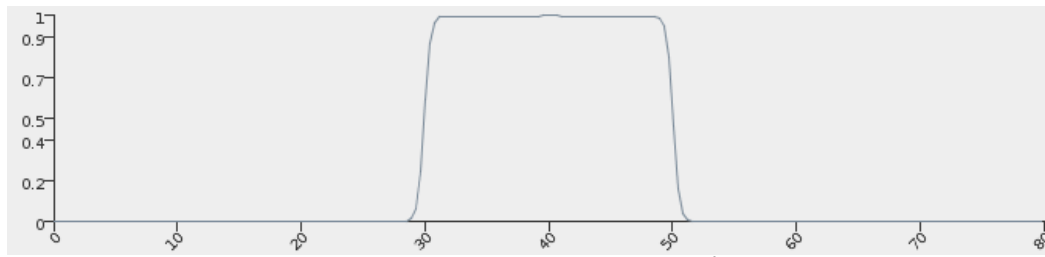
$$\delta_i^{(\lambda)}(x) = \frac{1}{1 + \frac{1}{2}(e^{-\lambda(x-a)} + e^{-\lambda(b-x)})}$$

A λ értékkel tudjuk állítani a függvény meredekségét. Kis λ értéknél laposabb függvényt kapunk, ami jobban átnyúlik a szomszéd klaszterek területeire.



$a(30, 50)$ intervallumon vett "felfújó" függvény, $\lambda = 0.5$ értékkel.

Nagy λ esetén meredek függvényt kapunk éles határokkal. A szomszédos klaszterek területére csak minimálisan nyúlik át.



a (30, 50) intervallumon vett "felfűjő" függvény, $\lambda = 4$ értékkel.

Az algoritmus iterációi során a λ értékei változnak, kezdetben kis értékkel inicializáljuk λ -t, majd fokozatosan növeljük értékét.

3. fázis: Kiszámoljuk az új centroidokat.

Az új centroidok a 2. fázisban megismert súlyok alapján számoljuk. Vesszük az összes pont súlyozott összegét az adott klaszterre vonatkozóan az alábbi képlet alapján.

$$x_j^k = \sum_{i=0}^N w_j(x_i) * x_i$$

ahol N : adatsor elemeinek száma
 i : adatsor aktuális elemének indexe
 j : aktuális klaszter index
 k : iteráció száma

Az x_j értéke meghatározza a k-adik iterációban a j-edik centroid új értékét. Az új klaszterközéppontok alapján minden klaszteren frissíteni kell az intervallum határokat. Az intervallum határok a következőképp változnak. A két szomszédos centroid által meghatározott felezőponton lesznek az új határok. Ez alól kivétel az első és az utolsó klaszter. Az első klaszter intervallumának minimuma a teljes adatsor intervallumának minimuma, az utolsó klaszter intervallumának maximuma pedig a teljes intervallum maximuma lesz.

4. fázis: Iteráció

Minden iteráció során a 2. és a 3. fázist ismételjük. Minden klaszterre kiszámoljuk az elemek súlyát a felfűjő függvény alapján, majd a kapott súlyokból kiszámoljuk az új

centroidok pozícióit. Ezt addig iteráljuk, míg a megállási feltétel nem teljesül a kialakult klaszterekre.

Egy iterációban az alábbi lépéseket kell elvégeznünk:

- beállítjuk a klaszterek határait.

Az algoritmus futása során a klaszterek középpontjait változtatjuk, ezért szükséges mindig frissítenünk a klaszterhez tartozó határokat.

- Kiszámoljuk a klaszterekhez tartozó K értékeket.

A K értékek egy iteráción belül fixek és amennyiben a határok nem változnak nem szükséges újból számolni.

- Bejárjuk a klasztereket és mindegyiknél meghatározzuk a 2. és 3. pont számításai alapján az új klaszterközéppontokat.

A fenti lépéseket addig ismételjük, amíg az algoritmus nem teljesíti a megállási feltételt.

5. fázis: Megállási feltétel.

Az algoritmus terminálási feltételeként kiszámítjuk az aktuális és az azt megelőző iteráció centroidjainak abszolútértékes különbségösszegét. Amennyiben ez kisebb, mint a teljes intervallum hosszának és a klaszterszám hányadosaként kapott érték, akkor a klaszterek középpontjai már csak kis mértékben változnak. Vagyis az algoritmus futása már nem hoz olyan változásokat, amik jelentősen befolyásolhatják a kialakult klaszterek határait.

$$\sum_0^i |x_i^k - x_i^{k+1}| < \frac{B - A}{C * K}$$

ahol i : klaszterek indexei

k : iterációk száma

A : a rendezett adatsor intervallumának minimuma

B : a rendezett adatsor intervallumának maximuma

C : egy előre meghatározott konstans skálázáshoz.

K : klaszterszám

Előre nem meghatározott számú osztályba soroló iteratív algoritmus

Az algoritmus bemenete egy egy-dimenziós adatvektor. A kapott adatvektort sok kis, egyenlő nagyságú osztályba soroljuk kezdetben, amiket a futás során majd összevonunk. Az összevonások a klaszterek relatív elemszámai alapján végezzük. Végeredményként megkapjuk a bemeneti adatvektor teljes (a, b) intervallumának felbontását. A klaszterek száma az algoritmus futása alatt változik és termináláskor alakul ki a végleges klaszterszám és az adatvektor elemeinek osztályozása.

1. fázis: A bemeneti adatok iniciális klasztereinek létrehozása.

A bemeneti adatvektort első lépésben sok kis osztályra bontjuk. Minden kezdeti klaszter azonos hosszú, ezáltal az előző algoritmussal ellentétben a klaszterek különböző számú elemeket tartalmaznak.

2. fázis: A klaszterek K értékeinek kiszámítása.

A klaszterek K értéke jellemzi a klasztert és a benne található elemek számát határozza meg. A K értéke egy iteráción belül állandó. Az alábbi képlettel tudjuk kiszámolni:

$$K_j = \sum_0^i \delta_j(x_i)$$

, ahol i : az input számsor elemeinek a száma
 j : a klaszter indexe

3. fázis: Üres klaszterek összevonása.

Az iniciális klasztereket bejárjuk és megkeressük azokat a klasztereket, amik üresek. Ezt a klaszterhez tartozó K érték jelzi számunkra. Amennyiben a K érték egy bizonyos küszöbszámnál kisebb, akkor úgy vesszük, hogy a klaszter nem tartalmaz elemet. A küszöbszám az 1 értékkel van meghatározva.

Bejárjuk a klaszterek listáját és amikor egy üresnek vélt klaszterhez érkezünk

megvizsgáljuk a klaszter jobb oldali szomszédját, ha az is üres a K értéke alapján, akkor a két klasztert összevonjuk. Az összevonás egyszerű folyamat, vesszük a két klasztert és létrehozunk egy új klasztert, ami megkapja a bal oldali klaszter intervallumának minimumát és a jobb oldali klaszter intervallumának maximumát. Ezeket az újonnan létrehozott klasztereket és a nem üres klasztereket megfelelő sorrendben egy listában tároljuk és a klaszter listájának bejárása után beállítjuk az új listát az eredeti lista helyett. A két lista közötti különbség megadja az egyesített klaszterek számát. Ezt az iterációt addig végezzük, amíg az egyesített klaszterek száma nem 0.

4. fázis: Nem üres klaszterek egyesítése.

Az iniciális klaszterek üres klasztereinek összevonása után megpróbálunk egyesíteni minden egymás melletti nemüres klasztert. A klaszterek nem üresek, ha a K értékeik egy bizonyos küszöbszámnál nagyobbak. A küszöbszám kiszámítható az egyesíteni kívánt két klaszter K értékeinek átlagával.

Az egyesítés folyamatához bejárjuk a klaszterek listáját és megvizsgáljuk a szomszédos klaszterek elemszámait. Ha a két klaszter nem üres, akkor ellenőrizzük, hogy megfelelnek-e az egyesítés kritériumának.

Amennyiben a két klaszter a K értékeik alapján nem üresek, akkor az egyesíthetőség megvizsgálásához hozzunk létre egy új klasztert, aminek a centroidja legyen a két szomszédos nemüres klaszter határa. Az új klaszter határait a két vizsgált klaszterek intervallumainak hosszából tudjuk kiszámolni. Az új klaszter határa legyen a „centroidjától” $1/8$ klaszterhossz távolságra. A két határ a centroidtól nem egyenlő távolságra lesznek, hanem a távolságuk a két vizsgált klaszter intervallumának nagyságától függ. Az egyesítést akkor végezzük el ha az újonnan létrehozott klaszter nem üres. Pontosabban, ha a klaszter K értéke egy bizonyos határérték felett van és nem tér el lényegesen az egyesítendő két klaszter K értékeinek átlagától. Ezzel azt érjük el, hogy csak akkor egyesítünk két klasztert, ha közöttük nincs lyuk egy relatív nagyságú szakaszon belül.

5. fázis: Fix számú osztályokba soroló algoritmus elvégzése.

Elvégezzük az előzőekben ismertetett klaszterező algoritmust fix klaszterszámmal. Ekkor

már a klaszterszámot fixnek vesszük. Az iteráció során a centroidok és a klaszterhatárok folyamatosan változnak. Az algoritmus terminál, ha teljesül a fix számú osztályokba soroló iteratív algoritmus kilépési feltétele. Az algoritmus befejeztével kialakulnak az új klaszterhatárok és centroidok. Az itt kialakult klasztereknél előfordulhat, hogy két nagyon közeli klasztert külön osztályba sorolja a kezdeti fix klaszterszám miatt.

6. fázis: Nemüres cellák összevonása.

Az algoritmus végső fázisában újból elvégezzük a nem üres klaszterek összevonását, de itt már más feltételekkel is dolgozunk az előző feltételek mellett. Itt az egyesítési feltételt globálisabban is vizsgálni kell a teljes (a, b) intervallum hosszának figyelembevételével. Ez azért szükséges, hogy legyen egy relatív nézőpontjuk és ne fajuljon el az egyesítés. Ugyanis előfordulhat olyan eset, hogy sok egymás melletti klaszter egyesítése után a kialakult új klaszter mérete akkorára nő, hogy a megadott $1/8$ -os határok már akkora területet ölelnek fel, amiben egészen biztosan lesz kellő számú elem.

7. fázis: Megállási feltétel.

Amennyiben két összevonás során kapott osztályok száma megegyezik az előző iteráció klaszterszámával, akkor az algoritmus terminál.

Az előre nem meghatározott számú osztályba soroló iteratív algoritmus módosított változata

A fent leírt algoritmus jól működik olyan esetekben, amikor az adatsor minden egyes elemét be kell sorolnunk egy kategóriába. De vannak olyan problémák, amikor csak az adatsor bizonyos intervallumain belül vannak csoportok, a többi esetben nem tudunk vagy nem akarunk nyilatkozni az elem kategóriájáról. Ilyenkor az adatsoron lyukakat keresünk és a lyukak nem tartalmaznak elemeket. Az osztályokba sorolást követően vizsgálnunk kell újabb elemeket és meg kell állapítanunk, hogy azok milyen osztályba tartoznak. Bizonyos esetekben szükséges, hogy legyenek olyan intervallumok, ahol nem tudunk egyértelmű döntést hozni, hogy az adott elem milyen kategóriába esik.

Az algoritmus egyes lépései megegyeznek a fentebb leírt előre nem meghatározott számú osztályba soroló iteratív algoritmus lépéseivel.

1. fázis:

Az első három fázist változatlanul végrehajtjuk. Inicializáljuk a klasztereket az adatsor egyenlő nagyságú szakaszokra való bontásával. Kiszámoljuk a klaszterek K értékeit és összevonjuk a nem üres klasztereket.

2. fázis: Nem üres klaszterek egyesítése.

Ebben a fázisban annyi módosítást kell eszközölnünk, hogy itt az eddigiekhez képest nagyobb λ értékekkel kell dolgoznunk. Az „ K ” érték alapján üresnek vélt klasztereket teljesen figyelmen kívül hagyjuk, a határait nem változtatjuk. A fázis lépései az előre nem meghatározott számú osztályba soroló iteratív algoritmus 4. fázisának lépéseivel megegyezik.

3. fázis: A lyukak között kialakult szakaszok iteratív vizsgálata

Az előző fázisban leírt lyukak közötti szakaszokat felbontjuk több kisebb klaszterre. Az itt használt klaszterszám a szakaszon belüli centroidok távolságának és a szakasz intervallum hosszának skálázott hosszából számíthatjuk ki. Amennyiben a szakaszban csak egy klaszter helyezkedik el, akkor egy a szakasz hosszának és az adatsor teljes intervallumának hányadosából származtatott számú klaszterre bontjuk az adott szakaszt.

A sok kis részre osztott szakaszra lefuttatjuk az előre nem meghatározott számú osztályba soroló iteratív algoritmust, amely során a szakaszt vagy egyben hagyja vagy több osztályra bontja szét ismét. Itt a λ értékeket megfelelő léptékben kell növelnünk a helyes eredmény érdekében.

Amennyiben a szakaszban csak egy klaszter volt, akkor a klaszter szélein üres klaszterek keletkeztek, amit a következő fázisban egyesítünk a szakaszt határoló lyukakhoz. Ha a szakaszon belül több klaszter volt, akkor klaszterek és lyukak sora keletkezik ismét. Egy előre

meghatározott számú rekurziót követően nem érdemes a szakaszok szétbontását sokszor megismételni, mert ezalatt a futásidő nagyságrendekkel növekedhet.

4. fázis: Megállási feltétel.

Amennyiben az algoritmus a szétbontás műveletét egy előre meghatározott számban rekurzívan elvégezte, akkor az algoritmus terminál. A rekurzió tipikusan 2 vagy 3 mélység, ennél több szétbontás már komoly időigénybeli többletet eredményezhet. Tesztek alapján több rekurzió elvégzése nem hoz jobb eredményt, de hatványozottan több időt vesz igénybe.

Módunkban áll nem a rekurziók számát korlátozni, hanem megadni egy küszöbértéket, ami a szétbontás műveletében határozza meg a minimális méretű klaszterek intervallumainak nagyságát. Ha a szétbontás során a küszöbértéknél kisebb nagyságú klaszterekre kéne bontanunk, akkor az adott szakaszt már nem vizsgáljuk tovább.

Tesztek

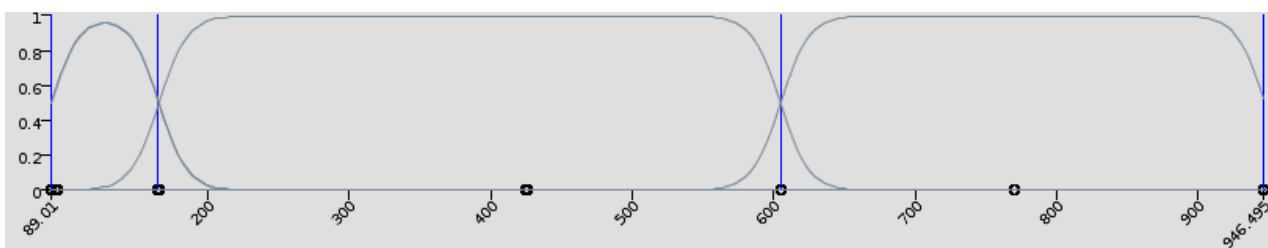
Az algoritmusok futását egy mesterséges példával végeztük el. Adott egy megadott nagyságú intervallum, amin generálunk megadott számú sűrűsödési pont pozíciókat. Ezen sűrűsödési pontok bizonyos környezetén belül elemeket generálunk. Minden sűrűsödési pont környezetében azonos számú elem helyezkedik el. A környezet nagyságát a generált pontok közötti minimális távolság $1/8$ -ad részével határozzuk meg.

Az algoritmus futásának eredményei

Fix számú osztályokba soroló iteratív algoritmus

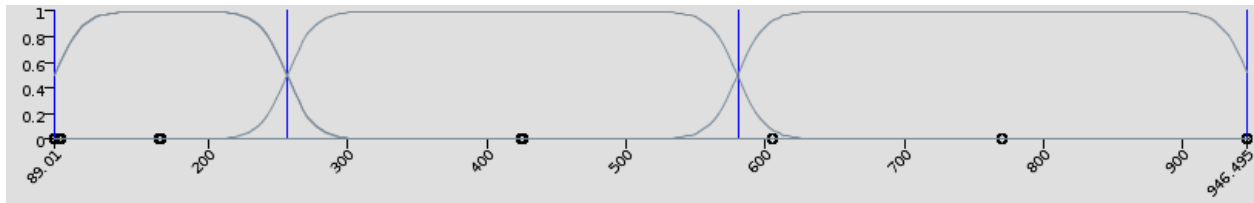
Kezdetben a bemeneti adatsort egyenlő elemszámú osztályokra bontja. A kezdeti klaszterszámot 3-ra állítottuk be.

Az alábbi példában hét sűrűsödési pont köré generálunk pontonként 1000 pontot a (0, 1000) intervallumban. Tehát, van 7000 pontunk, ami annyit tesz, hogy a sorba rendezett pontokat 2333 elemenként külön osztályba sorolja. Így kialakul a kezdeti 3 klaszter körülbelül egyenlő elemszámmal.

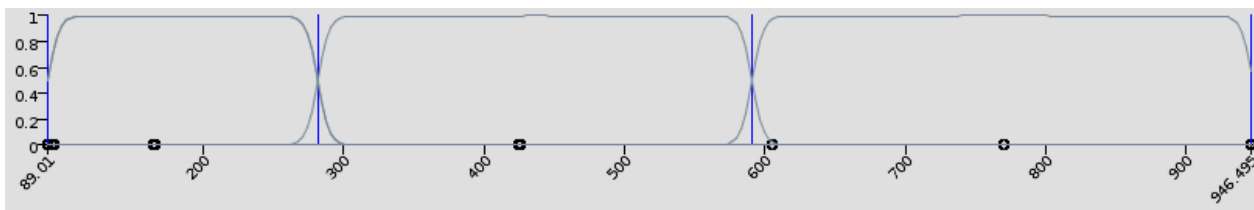


Következik az iteráció, ami során kialakulnak a végleges határok. Az első iterációban az alacsony λ érték miatt a középső klaszter felfújó függvénye átnyúlik a szomszédos klaszterekbe. A középső klaszter mindkét határa egy sűrűsödési ponton helyezkedik el, ezzel a sűrűsödési pont körüli pontokat két részre vágja. A súlyok kiszámítása és az új klaszterközéppontok kiszámításakor ezek a határok meg fognak változni. Az új centroidokhoz igazított határokat beállítjuk a klaszterekhez. A határok kiszámítása során a középső klaszter bal oldali határán levő pontok az első

klaszterhez fognak tartozni, míg a középső klaszter jobb oldali határánál elhelyezkedő pontok az utolsó klaszterhez fognak tartozni.



A következő iteráció egyben az utolsó iteráció. Az új centroidokhoz igazított határok ugyan változtak, de a klaszterközéppontok pozíciói jelentős változáson nem mentek keresztül. Így az algoritmus terminál és az alábbi három klasztert eredményezi.



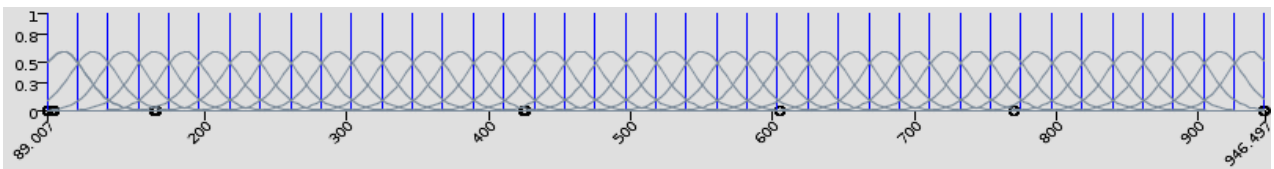
A kezdeti és a két iteráció során kialakult klaszterek felfűjt függvényeinek változása jól kivehető. Kezdetben jobban átnyúlunk a szomszéd osztályokba, majd az iterációk növekedésével egyre kisebb területeket vizsgálunk.

A futtatott példák tapasztalatai azt mutatják, hogy általában az algoritmus 3-5 iteráción belül terminál, de sok esetben elég 2 iteráció is.

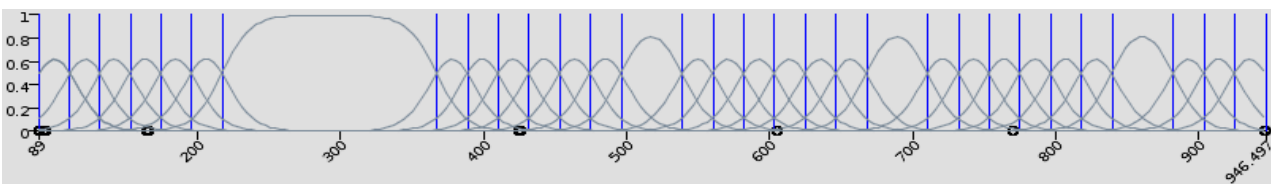
Előre nem meghatározott számú osztályba soroló iteratív algoritmus

Az algoritmus kezdetben a bemeneti adatsort egyenlő nagyságú klaszterekre bontja. A kezdeti klaszterszámnak megfelelően nagyoknak kell lennie a megfelelő eredmény érdekében. Ez lehet a bemeneti adatsor intervallumának a függvénye vagy az elemek sűrűségétől függő szám, de módunkban áll előre meghatározott számú klaszterszámmal is dolgozni.

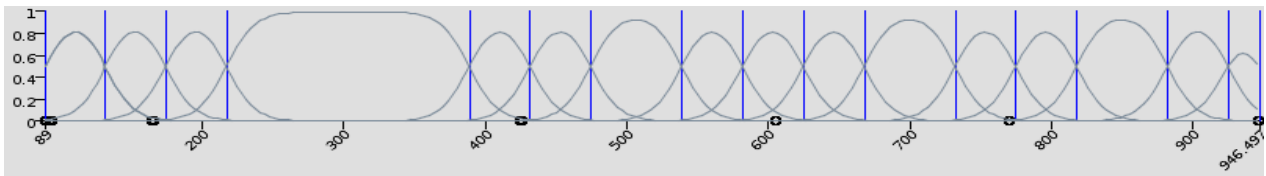
Az alábbi példában hét sűrűsödési pont köré generálunk pontonként 1000 pontot a (0, 1000) intervallumban. A kezdeti klaszterszám 40-re lett állítva. A kezdeti kis λ érték hatása látszódik az iniciális klaszterek felfújó függvényein. A felfújó függvények több klasztert is átfognak. A kezdeti lépést mutatja az alábbi ábra.



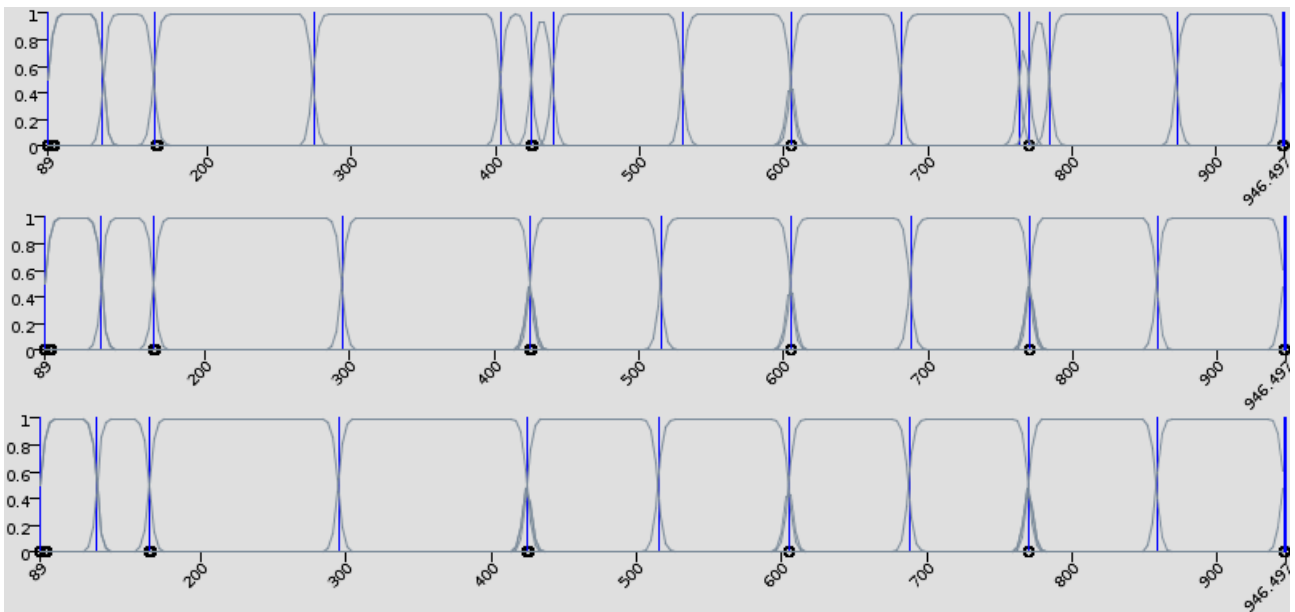
A következő lépésben az iniciális klasztereket megvizsgáljuk a K értékük szerint és az üresnek vélt klasztereket összevonjuk. Ahogy az ábrán látható a kezdeti 40 osztályból már csak 31 maradt. A sűrűsödési pontok mindegyikénél észrevehető, hogy viszonylag nagy távolságra tőlük a látszólag üres klaszterek nem lettek egyesítve. Nem vont össze minden üres osztályt. Ennek az oka a kezdetben alacsony λ értékek, amik a felfújó függvényt elnyújtják. Az üres cellák felfújó függvényei átnyúlnak a szomszéd cellába és így az osztály nem tekinthető üresnek.



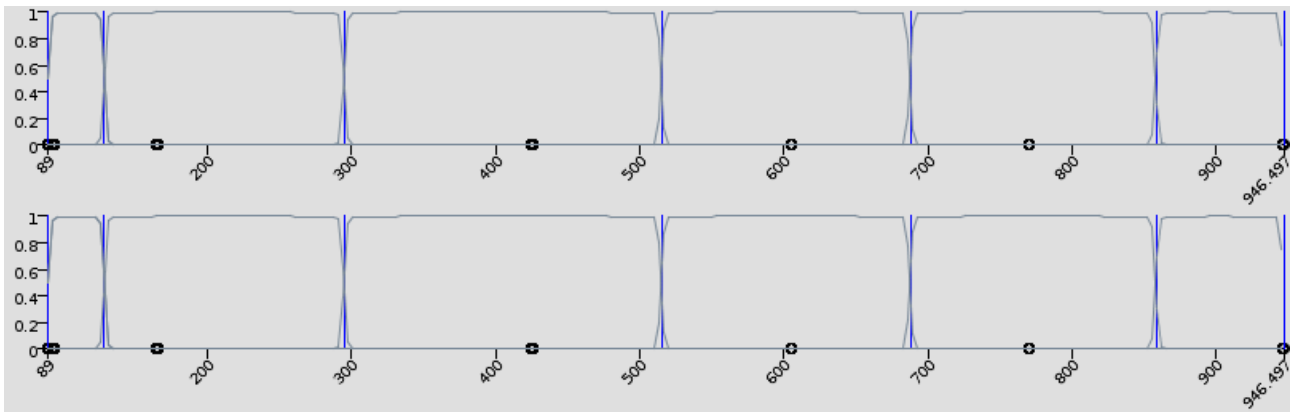
Az üres klaszterek egyesítése után a nem üres klaszterek egyesítése következik. A nem üres osztályok összevonását követően már csak 16 klaszterünk maradt.



Ebben a fázisban már nem maradt egyesítendő üres vagy nem üres klaszter. Kialakult egy fix klaszterszám, amivel végrehajtjuk a fix számú klaszterbe soroló algoritmust. Látható, hogy ebben az esetben az algoritmus másképp viselkedik, mint egyesítések nélkül futtatva, ugyanis a klaszterhatárok három iteráció alatt átmozognak a sűrűsödési pontok pozícióihoz. Az ábrán megfigyelhető, hogy az első iteráció után már csak 10 osztályunk van. Ez a klaszterhatárok feltorlódása miatt van, gyakorlatilag megmaradt a 14 klaszterünk, de egyes klaszterek határai fedik egymást.



Az utolsó fázisban újból elvégezzük a nemüres klaszterek egyesítésének műveletét. A fenti ábra utolsó iterációjánál látható, hogy a klaszterhatárok a sűrűsödési pontokon vannak. Ezért az egyesítés során a két szomszédos klaszter egyesítésre kerül, mivel a két klaszter egyesítésénél vizsgált újonnan létrejövő klaszter tartalmaz elemeket, a K értéke magas. Még két egyesítés szükséges a végfeltétel teljesüléséhez és az algoritmus terminálásához.



Az összevonás műveletének utolsó két iterációja során a klaszterközpontok és velük együtt a határok változatlanul maradtak és nem lett egyik osztály sem összevonva a szomszédjával, ezért az algoritmus elért a megállási feltételéhez és terminál.

Futásidő és memória tárterületre vonatkozó további tesztek

A teszteket a már előzetesen ismertetett véletlenszerű pontok környezetében szétszórt fix számú pont példán mértük. A teszteket változó elemszámú adatsorral futtatunk, 7 sűrűsödési ponttal a (0, 1000) intervallumon.

A továbbiakban ismertetett teszteredmények a következő konfiguráción lettek mérve:

Alaplap: Intel DQ57TM
CPU: Intel Core I5 3350P
Memória: 4 * 2GB KingMax DDR3 1333MHz
VGA: integrált Intel Graphics
HDD: 500GB
OS: Ubuntu 14.04 LTS 64bit

A teszt során a sűrűsödési pontok köré meghatározott távolságban véletlenszerű pontokat szórunk szét. A pontok véletlenszerű generálásához „seed”-et használunk, ami mindig ugyanazokat a számokat fogja generálni adott környezetben.

Fix számú osztályokba soroló iteratív algoritmus

Elemek száma	Eltelt idő (ms)	Memóriahasználat (byte)
100	00:00:00.033	2698736
1000	00:00:00.041	2708280
10000	00:00:00.099	4732416
100000	00:00:00.515	11446808
1000000	00:00:07.244	82830456
10000000	00:01:13.639	849936008

A futási időkből látható a 10-szeres elemszám hatására 10-szeres idő növekedés. Kivéve az első 3 sor adatait, ahol az eltelt idő túlzottan kevés a pontos méréshez. Sajnos a memóriahasználat számottevő, tízmillió adat számolása során 849Mb volt a memóriafoglalás.

Előre nem meghatározott számú osztályba soroló iteratív algoritmus

Elemek száma	Eltelt idő (ms)	Memóriahasználat (byte)
100	00:00:00.039	2738912
1000	00:00:00.094	2748592

10000	00:00:00.481	6051856
100000	00:00:05.623	27260088
1000000	00:00:57.321	235695384
10000000	00:10:02.737	2333242568

Az előző algoritmushoz képest itt hosszabb lefolyású az algoritmus futása, ami várható is volt, mert a fix számú osztályba soroló algoritmust használja és mellette az egyesítési algoritmust is végre kell hajtani minden iteráció során. Sajnos ez a memória használaton is érzékelhető. Az algoritmus tízmillió adaton 2.3Gb memóriát használ, ami a fix számú osztályba soroló algoritmus memóriahasználatának közel háromszorosa.

K-közép algoritmus

Elemek száma	Eltelt idő (ms)	Memóriahasználat (byte)
100	00:00:00.086	6020392
1000	00:00:00.119	11184424
10000	00:00:00.356	35637568
100000	00:00:01.881	67814256
1000000	00:00:20.593	326113768
10000000	00:04:02.140	2715921816

Fuzzy K-közép algoritmus

Elemek száma	Eltelt idő (ms)	Memóriahasználat (byte)
100	00:00:00.340	883824
1000	00:00:00.502	1811120
10000	00:00:01.969	11238584
100000	00:00:07.613	104869856
1000000	00:02:26.390	1043379512
10000000	N/A	N/A

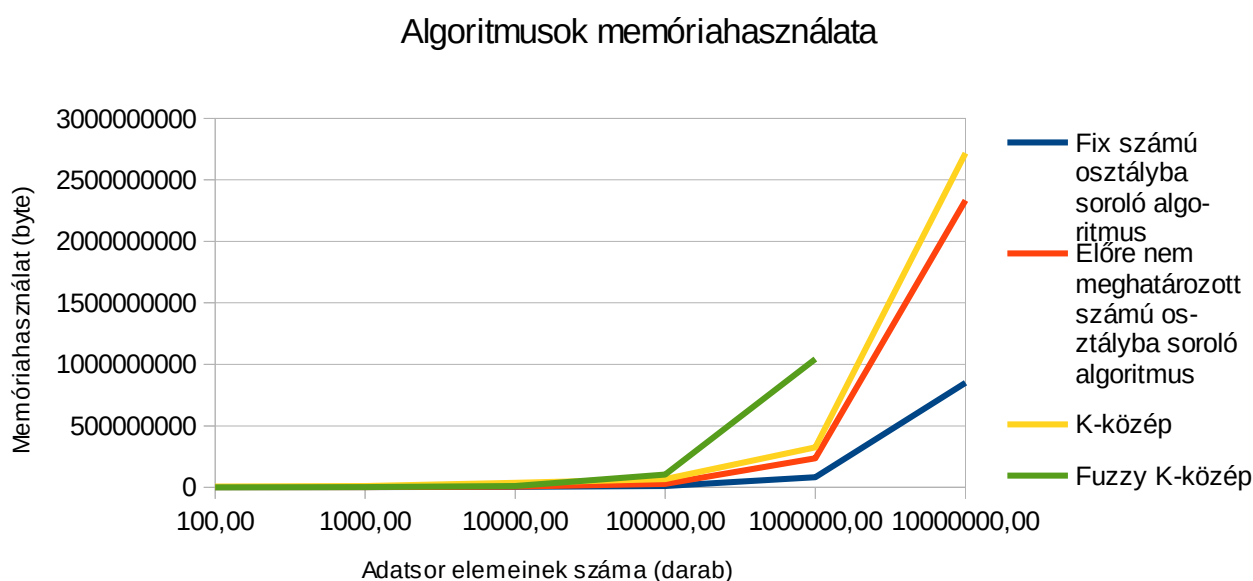
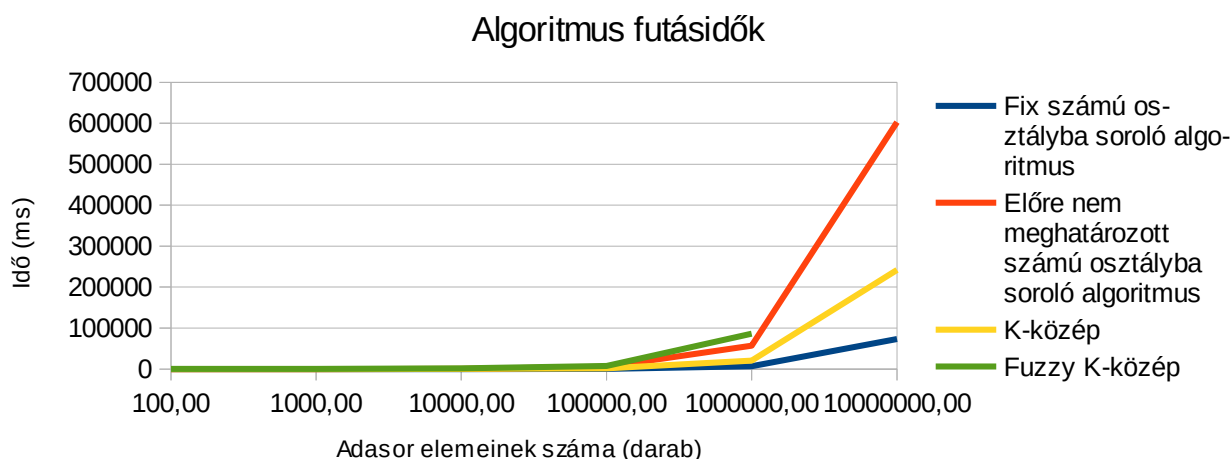
A mérések alapján a Fix számú osztályba soroló algoritmus teljesített a legjobban, mind idő, mind tárterület használatban. A legrosszabb teljesítményt a Fuzzy K-közép klaszterező hozta,

aminek az utolsó tesztjét nem tudtuk lefuttatni, mert 8Gb memóriánál többet használt volna. A teszt 20 perc után állt meg.

A K-közép algoritmus futásidőben aránylag jól teljesített, de a sok távolság számítás miatt és ezek tárolása végett a tárterület használata jelentősen nagyobb, mint a fix osztályba soroló algoritmusé, ahol nem az adatpontok távolságának mérésén alapul a klaszterezés, hanem a klaszterhatárookra ráhúzott felfújó függvények és a belőlük számított súlyok számolásán.

Az előre nem meghatározott számú osztályokba soroló algoritmus sem futásidőben, sem tárterület foglalásban nem nyújt kimagasló eredményt. Ez természetesen az algoritmus felépítéséből is adódik, hiszen a fix számú osztályba soroló algoritmust iterációinak végrehajtásán kívül még elvégzi az üres és nemüres cellák egyesítésének iteratív lépéseit.

Az alábbi ábrák szemléltetik az algoritmusok futásidő és memóriahasználat eredményeit.



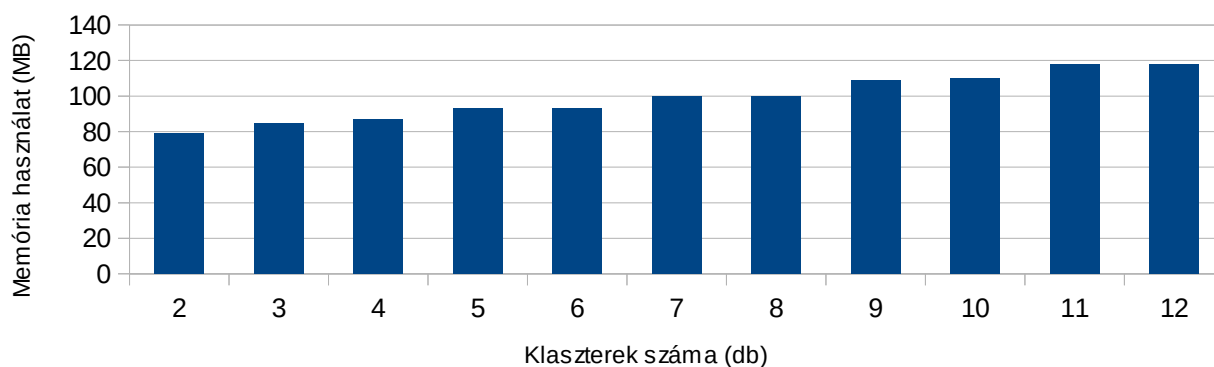
Fix számú osztályba soroló algoritmus futási idő és memória használat tesztjei

A teszteket egységesen 1 000 000 elemszámú adatsorral futtattuk, 5 sűrűsödési ponttal a (0, 1000) intervallumon. Az algoritmus klaszterszámait változtattuk minden teszt során és a mértük a futási időt, a memória használatot és az algoritmus iterációinak számát.

Klaszterek száma	Memória használat (MB)	Futási idő (s)	Iterációk száma
2	79	00:00:03.45	2
3	85	00:00:06.72	3
4	87	00:00:06.29	2
5	93	00:00:04.13	1
6	93	00:00:16.63	4
7	100	00:00:10.01	2
8	100	00:00:16.57	3
9	109	00:00:24.50	4
10	110	00:00:13.99	2
11	118	00:00:29.87	4
12	118	00:00:24.59	3

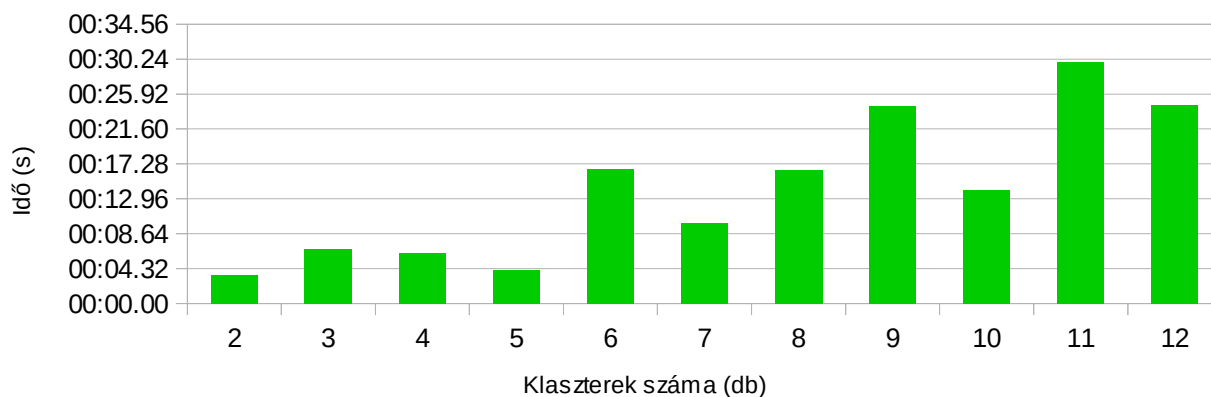
Memória használat

1 000 000 elem, 5 sűrűsödési pont, (0 1000) intervallumon



Futási idő

1 000 000 elem, 5 sűrűsödési pont, (0 1000) intervallumon



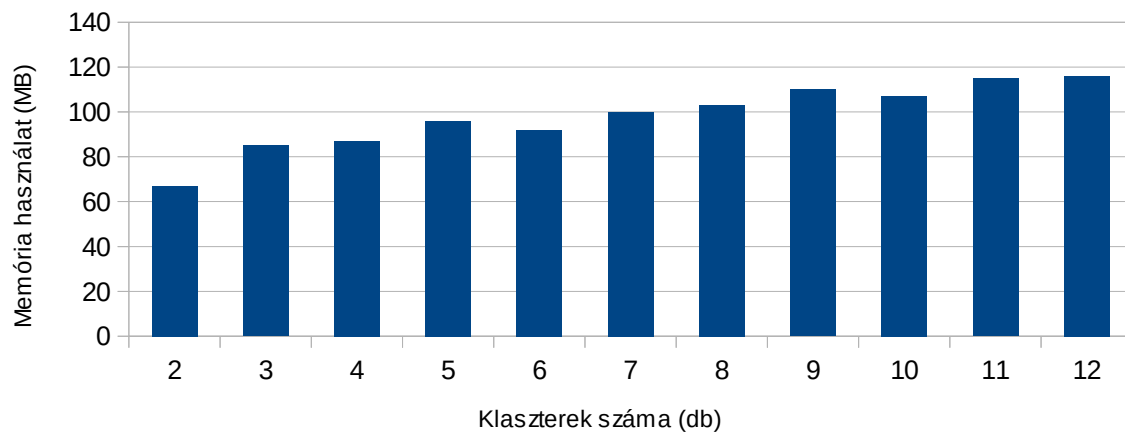
Az előző teszttel megegyező paraméterekkel futtattuk a fix számú osztályokba soroló algoritmust. Minden teszt során változó klaszterszám megadásával történt a teszt úgy, hogy a példa 12 sűrűsödési pontot generált.

Klaszterek száma	Memória használat (MB)	Futási idő (s)	Iterációk száma
2	67	00:00:02.47	1
3	85	00:00:08.26	4
4	87	00:00:08.06	3
5	96	00:00:06.79	2
6	92	00:00:15.88	4
7	100	00:00:18.35	4
8	103	00:00:15.51	3
9	110	00:00:17.39	3
10	107	00:00:19.33	3
11	115	00:00:28.01	4
12	116	00:00:08.07	1

Memória használat

1 000 000 elem, 12 sűrűsödési pont, (0, 1000) intervallumon

27



Futási idő - Fix számú osztályokba soroló algoritmus

1 000 000 elem, 12 sűrűsödési pont, (0 1000) intervallumon

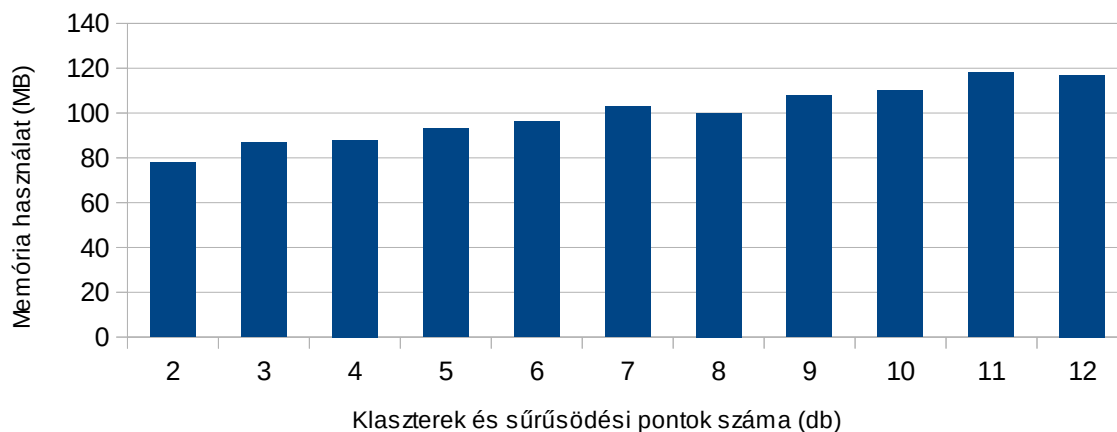


Teszteltük a fix számú osztályokba soroló algoritmust úgy, hogy a klaszterek száma megegyezik a példában generált sűrűsödési pontok számával.

Klaszterek száma	Memória használat (MB)	Futási idő (s)
2	78	00:00:02.26
3	87	00:00:02.86
4	88	00:00:03.45
5	93	00:00:04.03
6	96	00:00:04.58
7	103	00:00:05.22
8	100	00:00:05.71
9	108	00:00:06.25
10	110	00:00:06.97
11	118	00:00:07.42
12	117	00:00:08.02

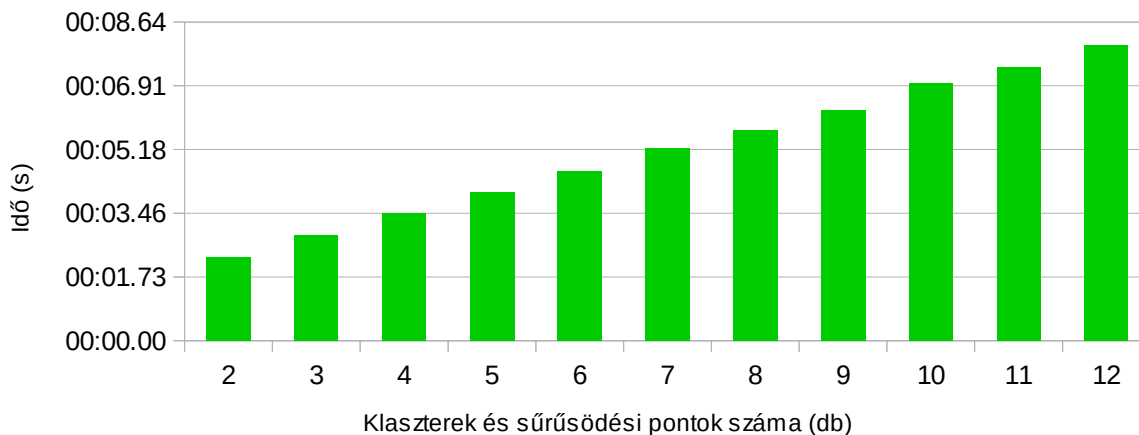
Memória használat

1 000 000 elem (0 1000) intervallumon



Futási idő

1 000 000 elem (0 1000) intervallumon

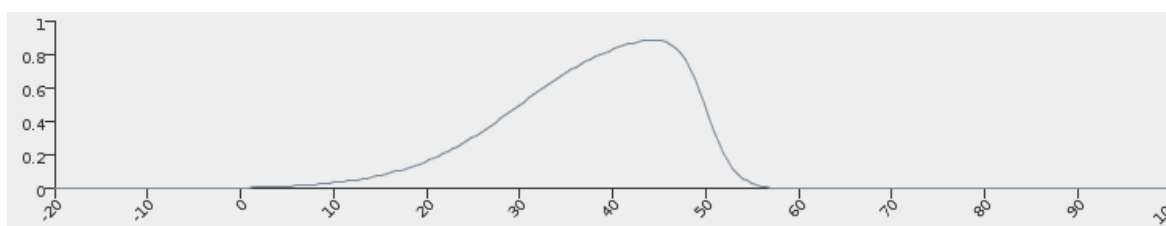


Fejlesztési lehetőségek

Az algoritmusok jelen állapotukban tartalmaznak számos olyan adatszerkezetet, amik a vizuális megjelenítésért, részeredmények számításáért, tárolásáért és kiírásáért felelősek. Ezek éles környezetben szükségtelenek és csak lassítják az algoritmusok futását.

Az adatsor elemeinek osztályozása tipikusan jól párhuzamosítható folyamat. Az adatsor iniciális klasztereinek meghatározása után a K értékek kiszámítása, az egyesítés művelete és maga az iteráció is felbontható részfeladatok sorozatára, amik alkalmassá teszik az algoritmust párhuzamos feldolgozásra. Ezzel a mostani egy processzoron történő végrehajtás futásidejét jelentősen tudnánk csökkenteni. Az maximális gyorsulás a processzormagok számától függ, nagysága elméletben az egymagos futásidő és a processzormagok számának hányadosa. Ez gyakorlatban nem szokott teljesülni, de jelentős javulást lehet elérni masszív párhuzamos feldolgozással.

A klaszterek meghatározásának alapját képező Pliant függvény jelen verzióban szimmetrikus λ értékekkel számol. Az aszimmetrikus λ értékek bevezetésével szabályozni tudjuk a függvény jobb és bal oldali meredekségét. Ehhez meg kell vizsgálni a két szomszédos klaszter nagyságát és a méretekkel arányosan beállítani a λ_1 és λ_2 értékeket. Ezzel azt érjük el, hogy nagyobb kiterjedésű szomszédos klaszter esetén kisebb meredekséget állítunk be, ami azt eredményezi, hogy nagyobb részben nyúlunk át a szomszéd területére. A kisebb méretű klaszterek esetén a meredekség nagyobb, ott csak kis részben vizsgálódunk.



A (30, 50) intervallumú klaszter felfűlt függvénye 0.16 illetve 0.65 lambda értékekkel

Ezzel a változással vélhetően az iterációk során gyorsabb konvergálást tudunk elérni.

A futásidőre és memória foglalásra vonatkozó tesztek alapján elmondható, hogy az algoritmusok még 10 millió egy dimenziós adatot képesek kezelni egy átlagos asztali számítógépen való futtatás során is. A 10 milliónál akár nagyságrendekkel nagyobb adatbázisok kezelésére is van

lehetőség. Nagy adathalmazon az algoritmus memóriakorlát miatt nem lenne képes végigfutni, de egy kisebb módosítással ez a helyzet is megoldható. Az adatbázis elemeiből valamilyen eloszlás szerint mintavételt veszünk. A mintavételezés elemszámát úgy határozzuk meg, hogy az algoritmus biztosan képes legyen a futtatására. A mintavétel elemeire elvégezzük a klaszterezést, kialakulnak a klaszterek határai. A mintavételezést és a futtatást iteratívan végrehajtjuk, addig amíg a klaszterek középpontjai számottevően változnak. Ha az algoritmus terminál, akkor nagy valószínűséggel mondhatjuk, hogy a mintavételezésekre egyenként futtatott algoritmus eredménye jól közelíti a teljes adatsor elemeivel elvégzett futás eredményét.

A jelenlegi egy dimenziós adatokon túl több dimenziós adatsorok klaszterezése lesz a következő lépcső az algoritmus fejlesztésében. A 2, 3 vagy akár több dimenzióban történő klaszterezések az algoritmus alapjául szolgáló képletek módosításával viszonylag kis fejlesztéssel adoptálhatóak. A 3 dimenzió feletti osztályozásnál már felmerülhetnek vizualizációs és tesztelési problémák, amik nehezítik az implementálás jóságának vizsgálatát.

Az 1-nél magasabb dimenzióknál az előre nem meghatározott számú osztályba soroló algoritmus módosított változata képes lesz konkrét klaszterek meghatározására a lyukak és pontokat tartalmazó klaszterek egyesítésével illetve összevonásával, ami számos területen lehet használható, mint például a képfeldolgozás és az alakfelismerés.

Összefoglalás

Jelen dolgozatban két új klaszterező eljárást dolgoztunk ki, amelyek algoritmusai hasonló elven működnek. Az egyik algoritmus a valós számok (a, b) intervallumának szakaszokra bontását végzi, és előre megadott számú osztályba tudjuk sorolni az elemeket. A kialakult csoportosítás a teljes (a, b) intervallumot lefedik és így egymás között nem lehet átfedés.

A másik megvalósítás, akkor hasznos, ha előre nem tudjuk, hogy az adatsor elemei hány kategóriába sorolhatóak. Ekkor az algoritmus határozza meg a szükséges klaszterek számát és a felbontást. Ebben az esetben az adatsoron a sűrűsödési pontok közt „lyukak” lehetnek.

Iteratív módszerrel egyesítés és szétbontás műveletét végző meta-algoritmus végzi a megfelelő csoportosítást, amelyek során kialakulnak a klaszterek határai.

Az dolgozatomban implementált algoritmusok mind futásidőben, mind tárfoglalás tekintetében jól szerepeltek. Számos javítási lehetőség ismert számunkra, amivel növelni lehet az algoritmusok hatékonyságát és felhasználhatóságát.

Az algoritmusokat JAVA nyelven implementáltuk és implementálás során fontos kritérium volt a gyors futási sebesség. A dolgozatban közölt eredmények alapján elmondható, hogy már az első iteráció után is jó közelítő eredményt kapunk.

Mivel a bemutatott algoritmusok jelenleg 1 dimenziós adatsoron működnek későbbi kutatási feladat a több dimenziós klaszterezés, amely a felfújó függvény paramétereinek módosításával viszonylag könnyen adaptálható. Az elkészül klaszterező eljárás több valós problémán is megfelelően teljesít, és felhasználási területe széleskörű.

A már említett természetes nyelvfeldolgozás problematikájában a poláris szavak megtalálásánál rendkívül jól használható. A többtényezős döntések hasznosság alapú döntéseinél jobban használhatóak a kapott diszkrét csoportok a folytonos változó értékeknél. Nagy adatbázisok, adattárházak adatbázisaiban tömörítésre alkalmazhatóak a diszkrét kategóriák. Az algoritmus által kapott eredmények tanuló algoritmusok részét képezhetik, amik a tanulás folyamatát egyszerűsítheti és gyorsíthatja.

Külön megemlítek egy konkrét felhasználási területet, az OTDK eredmények alapján az intézmények illetve a karok besorolását [5] az itt bemutatott algoritmussal lettek elvégezve.

Forrás

- [1] Csendes Tibor: Közelítő és szimbolikus számítások, Polygon, Szeged, 2007.
- [2] L. Gath, A. B. Geva: Unsupervised Optimal Fuzzy Clustering, IEEE Transactions on pattern analysis and machine intelligence, vol II., no. 7. July 1989, p. 773.
- [3] MacQueen, J. B. (1967). "Some Methods for classification and Analysis of Multivariate Observations". Proceedings of 5th Berkeley Symposium on Mathematical Statistics and Probability, University of California Press. pp. 281–297.
- [4] J. C. Bezdek, *Pattern Recognition with Fuzzy Objective Function Algorithms* (Plenum Press, New York, 1981).
- [5] Dombi József, Szendrő Péter, *Egyetemi és főiskolai karok, valamint karokra nem tagozódó felsőoktatási intézmények OTDK eredményessége (Országos Tudományos Diákköri Tanács, Budapest, 2013).*