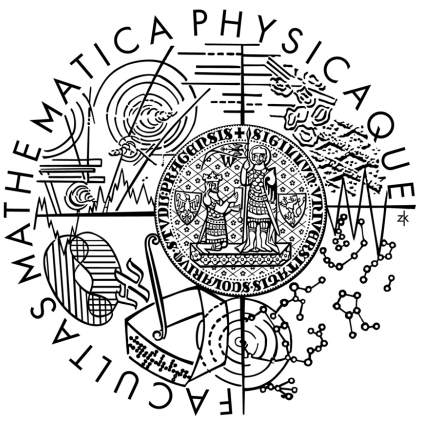
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Dominik Franěk

Klastrovací algoritmy

Katedra teoretické informatiky a matematické logiky

Vedoucí práce: Doc. RNDr. Iveta Mrázová, CSc.
Studijní program: Obecná informatika

2008

Prohlašuji, že jsem svou bakalářskou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Praze dne Jméno Příjmení

Obsah

Název práce: Klastrovací algoritmy
Autor: Dominik Franěk
Katedra (ústav): Katedra teoretické informatiky a matematické logiky
Vedoucí bakalářské práce: Doc. RNDr. Iveta Mrázová, CSc.
e-mail vedoucího: Iveta.Mrazova@mff.cuni.cz

Abstrakt: Tato práce je přehledem některých z nejpoužívanějších algoritmů pro shlukovou analýzu a některých kritérií pro validaci výsledků clusterování. Obsahuje popis a informace o algoritmech, dále pak jejich implementaci (v příloze) a nakonec výsledky testování na různých datech.

Klíčová slova: Shluková analýza, Klastrovací algoritmy, Validační kritéria

Title: Clustering algorithms
Author: Dominik Franěk
Department: Department of Theoretical Computer Science and Mathematical Logic
Supervisor: Doc. RNDr. Iveta Mrázová, CSc.
Supervisor's e-mail address: Iveta.Mrazova@mff.cuni.cz

Abstract: This work is an overview of some of the most frequently used algorithms for cluster analysis and some criteria for validation of the analysis results. It includes description and informations about the algorithms, its implementation (in the attachment) and, finally, results of tests on various data.

Keywords: Cluster analysis, Clustering algorithms, Validation criteria

Kapitola 1

Úvod

Techniky clusterování (klastrování) patří do oblasti data-miningu. Možnosti získávání dat zdaleka předběhly možnosti jejich zpracování a získání užitečného výstupu. Z různých průzkumů, pozorování, testů atd. získáváme obrovské objemy dat (například řádově GB/s z urychlovačů) a je nutné je zpracovat. Používají se různé přístupy, například zřejmě nejběžnější statistika.
Techniky clusterování se snaží data rozdělit do disjunktních skupin podle jejich vlastností, podobná data do stejné skupiny, naopak data v různých skupinách by se měla co nejvíce lišit. Tyto výsledky jsou obvykle předzpracováním dat, dalším krokem může být například zjištění, čemu vlastně tyto skupiny odpovídají, využití ke klasifikaci nebo jiným postupům.
Hlavní výhodou clusterování je, že vlastnosti, podle kterých dokáže data rozdělit, často nejsou vůbec zřejmé, jedná se o kombinace vlastností, které již nelze jednoduše rozlišit.

Příklady aplikací:
Marketing: Rozlišování skupin zákazníků podle známých údajů pro lepší zaměření nabídek.
Pojištění: Hledání vztahů (kombinace vlastností : riziko)
Využití půdy - vyhledávání vhodných lokalit ke stavbě ve městě, hledání minerálních ložisek, ...
Věda - analýza výsledků pokusů

Cílem práce bude vypracování přehledu clusterovacích algoritmů především s učením bez učitele a některých validačních kritérií. Dále implementace vlastního programu, který bude některé z těchto algoritmů a kritérií provádět a vhodným způsobem zobrazovat výsledky, který bude možno dobře rozšiřovat a použít k dalšímu výzkumu v této oblasti.

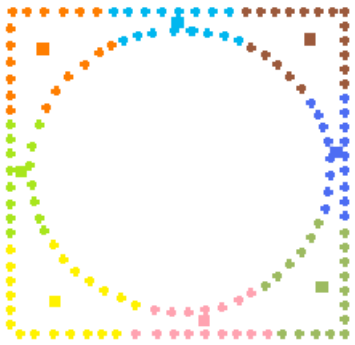
Kapitola 2

Klastrovací techniky

2.1 Obecně

Vstupní data

Základním vstupem je nějaká množina objektů s porovnatelnými vlastnostmi. V praxi to může být téměř cokoliv, požadujeme pouze aby je bylo možno popsat nějakou sadou atributů, které je vystihnou a bude je možné u každých dvou porovnat. Prakticky to znamená, že objekt je reprezentován reálným vektorem $(x_1, \ldots, x_n)$, kde každá složka x_i odpovídá jedné vlastnosti. Takto reprezentované objekty si lze představit jako body v n-rozměrném prostoru.
Předpokládáme, že všechny vlastnosti jsou známy, opačným případem se zabývají zvláštní modifikace clusterovacích algoritmů. Tento předpoklad není vůbec samozřejmostí, v praxi bývá obvyklejší, že některé informace chybí.
Dále předpokládáme, že pokud se dva body v nějaké složce liší jen málo, je odpovídající vlastnost u nich podobná. Například pokud se výška dvou osob liší o pár milimetrů, můžeme říct, že jsou podobně vysoké. Příkladem opaku je obrázek 2.1. Puntíky jsou nějaké vstupní body, zřejmě tvořící čtverec a kruh a do těchto dvou množin by bylo přirozené je zařadit. Pokud bychom ale tyto body rozdělili do clusterů podle nejběžnějším přístupem, tedy množin do s co největší podobností danou vzdáleností od jejich středu, získali bychom něco zdaleka jiného.



Obrázek 2.1: Výsledek clusterování ve dvou dimenzích - barvy odpovídají clusterům, čtverce jsou jejich středy.

Dohoda

Protože všechny objekty, se kterými zde budu pracovat (nejen vstupy, ale clusterové prototypy, neurony, ...) budou reprezentovány reálnými vektory, v konkrétní aplikaci stejné délky, budu psát některé operace jednodušeji. Součin vektorů x_1 a x_2 bude automaticky součinem skalárním, pokud nebude řečeno jinak a bude psán jako $x_1 * x_2$ a rovná se tedy $\sum_1^n x_{1,i} * x_{2,i}$. Mocnina vektoru bude $x^k = \sum_{i=1}^{n} x_i^k$.

Shluky

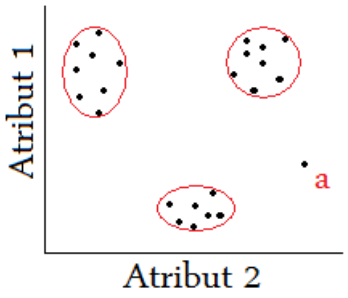
Pokud je malý rozdíl v atributu dvou vstupních bodů, jsou si v této vlastnosti podobné. Samozřejmě čím větší je počet atributů, ve kterých se podobají, tím jsou si objekty podobné celkově. Tuto podobnost vystihuje vzdálenost mezi body v prostoru. Pro naše účely budeme používat euklidovskou metriku. Použitelných metrik je více a dávají různé definice clusterů a různé výsledky ([10]), tyto ale práce neobsahuje. Tedy vzdálenost d dvou vstupních bodů x_1 a x_2 je definována jako $d(x_1, x_2) = ||x_1 - x_2|| = \sqrt{x_1 * x_2}$.

Víme-li tedy, že podobné objekty se budou v prostoru nacházet blízko sebe, víme již co hledat. Podobné objekty se budou sdružovat poblíž sebe a tvořit

v prostoru shluky. Takovýto shluk dat se vyznačuje tím, že vzdálenosti mezi body uvnitř něj jsou malé, zatímco vzdálenost k dalším bodům (shlukům) v prostoru bude v poměru k nim velká.

Chceme-li tedy data rozdělit do nějakých skupin podobných vlastností, je potřeba tyto shluky najít a umět o vstupech říci, zda do daného shluku patří, či nikoliv. Takto označené shluky se nazývají clustery (či česky klastry) a jejich hledáním se zabývá právě clusterovaní. Nejpoužívanější algoritmy pro tento úkol jsou k-means a fuzzy k-means.

Kromě toho lze v prostoru nalézt různé izolované body, neboli šum. Mohou to být odchylky, chyby měření, nevýznamné menšinové údaje, nebo nějaké "průměrné", inertní pozadí. Většinou jsou tato data nezajímavá a moc nám neřeknou. Pro některé algoritmy mohou znamenat značný problém, takže jejich rozpoznání je také důležitým úkolem.

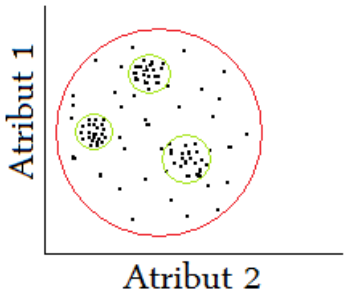


Obrázek 2.2: Ukázka dvoudimenzionálního prostoru se třemi shluky a izolovaným bodem (a)

Tím však možnosti ještě vyčerpány nejsou. Dále nás mohou zajímat vztahy mezi shluky. Zatímco to, že dva vstupy mají podobné hodnoty vlastností můžeme poznat z jejich popisu přímo, zjistit vztah mezi shluky bodů už tak jednoduché není, protože takové shluky budou sice v prostoru blízko sebe, ale vlastnosti bodů v nich se již mohou podstatně lišit (proto jsou v různých shlucích). Tyto údaje jsou přitom velmi důležité, protože nám dávají mnoho informací, které bysme skutečně z dat rádi zjistili - například jaké kombinace vlastností člověka častěji vedou k nehodám. Tímto problémem se zabývá algoritmus Kohonenových map.

Jednoznačnost

Bohužel neexistuje jednotný návod na správné clusterování. Existují různé definice, například v závislosti na použité metrice. Další nejednoznačnost plyne z toho, že data mohou obsahovat různé vnitřní struktury v závislosti na měřítku, ve kterém je sledujeme. Například obrázku 2.3 je vidět jeden velký shluk obsahující tři menší. Jaký přístup a parametry použít závisí na konkrétním případu. Mnoho snah se dnes upírá k automatickému určení těchto parametrů, přesto je obvykle nutné, aby je zvolil uživatel. Na pomoc jsou však k dispozici různé pomocné algoritmy a validační kritéria.



Obrázek 2.3: Zde je zřejmě rozdělení do clusterů nejednoznačné

Hierarchie

Pokud budeme pokračovat s příkladem obrázku 2.1, začne být zřejmé, že pokud shluky z různých úrovní rozlišení dáme přes sebe, vznikne nám jakási hierarchie - větší shluky obsahují menší atd. až k jednotlivým bodům. Toto odpovídá i hierarchii ve skutečnosti. Pokud menší shluky z obrázku budou například třešně, broskve a švestky, větší shluk může reprezentovat druhy a odrůdy peckového ovoce.
Tyto struktury hledají hierarchické algoritmy.

Clusterování

Hlavním rysem clusterování je rozdělení dat na disjunktní množiny - clustery. Snahou pak je, aby tyto množiny odpovídaly shlukům v datech a vzdálenosti mezi body uvnitř byly výrazně menší, než mezi body z různých

clusterů. Tyto clustery mohou být navíc reprezentovány nějakým prototypem, nazývaným střed nebo centroid. Například u algoritmu k-means mu odpovídá těžiště clusteru, u k-medoids (algoritmus podobný k-means) je to jeden z bodů clusteru.

2.2 Algoritmy

2.2.1 K-means

Obecně

Tento algoritmus je velice jednoduchý a populární, především díky své rychlosti a tím pádem možnosti zkoušet mnoho variant. Ve své základní verzi přiřazuje vstupní body k jednotlivým středům, což jsou zvláštní body $c_1, \ldots, c_k$ přidané do vstupního prostoru. Tyto středy jsou prototypy clusterů a jejich počet odpovídá počtu shluků, jaký v datech očekáváme, tedy k. Vstupy jsou přiřazovány vždy k nejbližšímu středu (používá se klasická euklidovská metrika), v důsledku je tedy prostor rozdělen na oblasti náležící jednotlivým středům. V iteracích algoritmu se středy přesouvají do těžiště "svých" clusterů a tím pádem mají tendenci putovat do oblastí s co nejvíce vstupními body - do středů shluků, což je právě požadovaná vlastnost.

Průběh a výsledek algoritmu značně závisí na počáteční konfiguraci - na počtu a poloze středů. Především je předem nutné určit jaký počet shluků očekáváme a podle toho na začátku do prostoru umístit náležitý počet středů. Druhou částí je jejich počáteční rozložení, které průběh algoritmu také velmi ovlivňuje. Zvláště ve složitějších datech algoritmus pravděpodobně skončí pouze v lokálním minimu, takže pro získání dobrého výsledku je nutné jej spustit vícekrát (i stovky opakování) s různou inicializací.

Kromě této základní verze existuje řada modifikací:
Fuzzy c-means - každý vstup náleží do všech středů, do každého různou měrou podle toho, jak dobře jej střed vystihuje. Viz. sekce 2.2.2.
Exact k-means - v klasickém k-means se v každém kroku přesouvají středy do těžiště svých dat, čímž se opět změní náležitost dat k nim. Zde se nejdříve před každým přesunem zkontroluje, zda se tím skutečně sníží celková chyba ($\equiv$ energie).

Cíl

Cílem algoritmu je dosáhnout stavu s co nejnižší "energií", tj. s co nejmenším součtem vzdáleností dat k jejich středům. Energie (nebo též celková chyba) je definována následujícím vzorcem (viz. také index 2.3.1):

$$E_{km} = \sum_{i=1}^{k} \sum_{x \in v_i} ||c_i - x||^2 \tag{2.1}$$

kde v_i jsou clustery, c_i jim náležící středy a x vstupy.

Je zřejmé, že nejnižší energie odpovídá stavu, kde středy leží v těžištích nejvýznamnějších shluků dat.
Pokud v takovém stavu přiřadíme každý vstup nejbližšímu středu, získáme středy jako reprezentanty shluků a dat v nich. Navíc ale středy nasbírají i šum v prostoru mezi shluky, který algoritmus nedokáže rozlišit, což je jedna z jeho slabin.
Výsledek závisí na inicializaci, pouze při správné inicializaci bude dosaženo globálního minima E_{km}. Při náhodné inicializaci je nejčastěji dosahováno pouze minima lokálního.

Inicializace

Inicializace má dvě části. Určení počtu středů a jejich počáteční rozmístění. Oba kroky mají značný vliv na průběh a výsledek algoritmu.

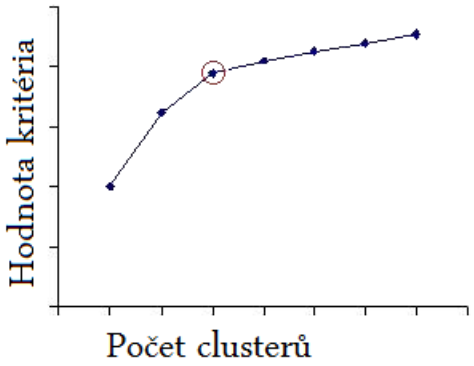
Pro správný počet může být víc možností, viz. sekce o jednoznačnosti (2.1) - můžeme chtít hledat v různém rozlišení.

Máme-li toto rozlišení už zvolené, stále je potřeba vědět, kolik shluků chceme najít, což obvykle předem nevíme.
Je-li hledán větší, než skutečný počet, některé shluky nebudou vůbec nalezeny (≡ explicitně označeny), pouze se, stejně jako šum, stanou součástí jednoho nebo víc clusterů, do jejichž oblasti spadnou.
Při zadání většího počtu budou, v případě ideálního počátečního rozložení, nalezeny všechny skutečné shluky, ostatní středy zůstanou ve volném prostoru a bude možné je identifikovat podle příliš malého počtu jim náležících

vstupů. V horším případě vlezou do již obsazeného clusteru a tento si rozdělí s druhým středem.

Volba počtu středů k:
Předpokládejme nyní, že pokud již zvolíme k, druhá část inicializace proběhne správně.
Zde (i v druhé části) se uplatní rychlost algoritmu. Algoritmus spouštíme opakovaně s různým k a výsledek vždy vyhodnotíme hodnotícími kritérii, podle kterých vybereme te nejlepší počet. Použitelných kritérií je mnoho, lze použít přímo energetickou funkci (2.3.1), nebo pokročilejší kritéria, jako Dunnovo kritérium (2.3.2) nebo SD index (2.3.3). Pro konkrétní případy jsou lepší různá kritéria, je potřeba jich vyzkoušet více.

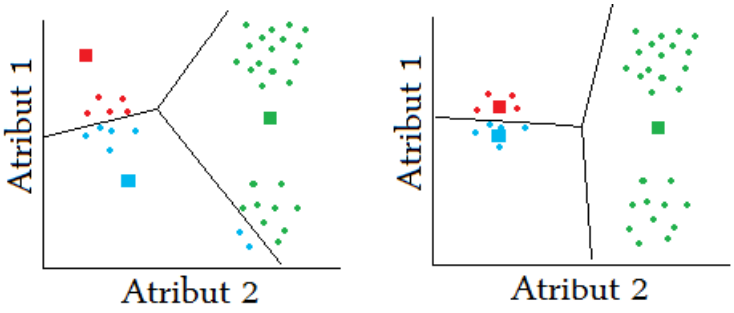
Vezměme nejjednodušší příklad a to energetickou funkci. Pokud budeme zvyšovat počet středů, bude se energie snižovat (ve chvíli, kdy by byl každý vstup obsazen jedním středem by byla nulová). Největšího zlepšení se bude dosahovat dokud není dosaženo skutečného počtu shluků, přidání dalších středů už energii tolik nezlepší.
Toto chování je obecně patrné jako tzv. elbow criterion (kritérium ramene) podle tvaru funkce popisující vztah počtu středů a hodnoty kritéria. Cílem je najít nejvýraznější zlom kriteriální funkce, ve kterém pravděpodobně bude ležet optimální počet středů. Tento postup platí obecně, ne jen pro k-means.



Obrázek 2.4: Závislost zadaného počtu clusterů a obecné hodnotící funkce

Volbu počátečního rozložení je možné provádět náhodně, což se v praxi dělá, nebo je možné nejdříve prostor nějak prohledat a podle výsledku středy

rozmístit (například pomocí kd-stromu).
Špatné rozložení je takové, kde si od počátku nějaké dva středy nárokují jeden shluk do něhož už zpočátku oba vstoupí, tento si rozdělí a zůstanou v něm. Při správně zadaném počtu středů navíc zůstane nějaký shluk neobsazený (nebo spíš bude označen jako jeden společně s jiným). Popsat správné rozložení je obtížné, ale je asi vidět, že při náhodné inicializaci není moc pravděpodobné.



Obrázek 2.5: Stav po inicializaci a po skončení algoritmu při špatné inicializaci.

Vhodná data

Jelikož z výstupu nelze rozlišit, zda vstupní bod náležící danému středu je skutečně součástí clusteru, nebo je pouze šumem spadající do jeho oblasti, v zašuměných datech algoritmus ztrácí na vypovídací schopnosti. V datech se šumem nám tedy dá informaci o poloze shluků, v čistých datech se můžeme spolehnout i na příslušnost dat do nich. Ideálními daty jsou, jako ostatně vždy, dobře oddělené kulové shluky. Algoritmus lze ale díky jeho jednoduchosti uplatnit pro různé účely i v dost typově rozdílných datech.

Průběh

Inicializace:
1) Volba počtu středů (k)
2) Přiřazení počáteční polohy vektorům $c_1, \ldots, c_k$ mezi vstupními daty. Toto může být provedeno náhodně, nebo s využitím předchozího prohledání prostoru například kd-stromem.

Průběh:
Prováděj
1) Každý vstup x_j přiřaď k nejbližšímu středu c_i ($\equiv$ do clusteru v_i, jehož je c_i středem).
2) Přesuň každý střed c_i do těžiště jeho vstupů. Tedy $c_i := \frac{1}{n} \sum_{\{x \in v_i\}} x$. dokud není dosaženo stabilního stavu.

Konvergence

V nejjednodušší variantě se nemusí algoritmus zastavit.
Když se v kroku algoritmu přesune vstupní bod z jednoho shluku do jiného, změní se i těžiště těchto shluků a tím pádem se mohou změnit i náležitosti dalších vstupů. V některých případech může nastat situace, kdy nějaké body mezi shluky oscilují a není možné dosáhnout stabilního stavu.
Aby byl algoritmus konečný, používá se v praxi nějaká pomocná podmínka pro ukončení, jako malé nebo žádné zlepšení nějaké ohodnocovací funkce. Další možností je použít nějakou variantu k-means, například exaktní k-means.

Složitost

Nechť n je počet vstupních bodů, k zadaný počet středů, d dimenze, i počet iterací.
V jednom kroku algoritmu se pro každý z n vstupů hledá nejbližší mezi k středy, což je $n * k$. Dále se každý z k středů přesouvá do těžiště svých dat. Těchto je sice pouze část z celku (součet přes všechny středy je teprve n), ale jelikož $k \ll n$, asymptoticky je to n, takže složitost tohoto kroku je opět $k*n$. Toto se provádí v i iteracích, takže celková složitost je $i*2*k*n = O(i*k*n)$. Dimenze se obvykle v teorii nezahrnuje, v praxi je ale významná a výpočty se vždy provádí v každé dimenzi. Takže reálná složitost je cca $i*2*k*n*d$. Zdaleka nejvýznamnější složkou je počet vstupů n, vůči kterému je složitost lineární, proto je algoritmus tak rychlý.

Exaktní K-means

Tato varianta mírně mění jednu část základního algoritmu a to kritérium pro přesun bodu do jiného clusteru.
V původním k-means se bod přesouval za podmínky, že vzdálenost k jinému clusterovému středu byla menší, než k jeho vlastnímu. Tento přesun následně změní i polohy těchto středů - původní se posune dál od tohoto vstupu a nový k němu, čímž se vzdálí či přiblíží k jiným bodům. Tato změna obvykle hodnotu chybové funkce ($\sum_{i=1}^{k}\sum_{x\in c_i}||c_i - x||^2$) zlepší, ale nemusí to tak nutně být a ve složitých datech také často není. Proto se zde zavádí nové kritérium, které říká, že k tomuto přesunu dojde pouze za předpokladu, že se tím sníží hodnota energetické funkce.

Výhodou tohoto přístupu je, že algoritmus vždy konverguje v konečném počtu kroků, na rozdíl od originálního (v práci [5] nazývaného naivní) k-means.
Nevýhodou je zřejmé zvýšení složitosti. Touto změnou chybové funkce jsou zasaženy pouze dva zmiňované clustery, což ale asymptoticky stále dává zvýšení složitosti o krát n.

2.2.2 Fuzzy K-means

Základ

Tento algoritmus je nejčastěji používanou modifikací k-means. Je o něco pomalejší, než základní verze, ale robustnější, dokáže si poradit se zašuměnými daty.
Od základní verzi se liší tím, že vstupní bod nenáleží plně k jednomu středu, ale ke všem, pouze s různou mírou náležení. Ta odpovídá tomu, jak dobře je vstup tím kterým středem vystižen. Algoritmus probíhá podobně jako u k-means, navíc ale získáváme informace o tom, zda a jak dobře vstupní bod patří do nějakého clusteru. Bod blízko nějakého středu, neboli ve středu nalezeného shluku, bude mít k tomuto středu velkou náležitost, oproti tomu vstup, který je od všech shluků a tím pádem i středů daleko (šum), bude ke všem středům náležet málo a je rozlišitelný.

Cíl

Cílem je zde opět minimalizace energetické funkce. Podobá se energetické funkci k-means, je ale navíc rozšířena o hodnoty u_{ij}^m, kde u_{ij} $\forall i, j$ je míra náležení vstupu x_i ke středu c_i vypočtená algoritmem a m je parametr algoritmu, $1 < m < \infty$. Již zde je tedy vidět zvýšení složitosti - do vzorce vstupuje $k * n$ nových prvků u, k počet středů, n počet vstupů.

Energie se vypočte následovně:

$$E_{fkm} = \sum_{i=1}^{k}\sum_{j=1}^{n} u_{ij}^m ||c_i - x_j||^2 \tag{2.2}$$

c_i středy, x_j vstupy, u_{ij} míra náležení x_i k c_i, m parametr > 1.

Stejně jako u k-means je pro dosažení globálního minima potřeba správná inicializace.

Inicializace

Inicializace je stejná jako u k-means a platí pro ni stejná pravidla.

Průběh

Zde se nepřesouvají vstupy mezi clustery, místo toho se přepočítává jejich náležení do všech clusterů. V druhém kroku se opět středy přesouvají do polohy s nejnižší energií vůči energetické funkci při daných mírách náležení. Jelikož se náležitost dat do clusterů nemění diskrétně, ale spojitě, není počet kroků teoreticky omezen. K ukončení nestačí požadavek dosažení stabilního stavu, kde se již náležitost nemění, je třeba určit toleranci, která bude uživateli stačit.

Průběh:
1) Inicializuj středy $c_1, \ldots, c_k$ jako u k-means.

Prováděj:
2) Vypočti míru náležení všech bodů ke všem středům. u_{ij} je míra náležení i-tého středu (c_i) k j-tému vstupu (x_j), $m > 1$ je parametrem algoritmu, k počet středů, n počet vstupů.

$$u_{ij} = \frac{1}{\sum_{l=1}^{k} \left(\frac{\|x_j - v_i\|^2}{\|x_j - v_l\|^2} \right)^{1/m-1}} \quad 1 \leq i \leq k,\ 1 \leq j \leq n \tag{2.3}$$

3) Přesuň středy do polohy s nejnižší energií.

$$c_i = \frac{\sum_{j=1}^{n} u_{ij}^m \ x_j}{\sum_{j=1}^{n} u_{ij}^m} \quad 1 \leq i \leq k \tag{2.4}$$

dokud není dosaženo koncové podmínky. Tou může být maximální počet kroků, daný rozdíl objektivní funkce mezi dvěma kroky (> 0) nebo dosažení hodnoty nějakého indexu.

Složitost

Nechť n je počet vstupních bodů, k zadaný počet středů, d dimenze, i počet iterací.
V prvním kroku algoritmu se pro každý z n vstupů přepočítává náležitost ke všem k středům. Při výpočtu náležení vstupu ke středu se prochází všechny středy, takže výpočet všech náležení obnáší $n * k^2$. Druhá část je přesun každého z k středů do nové pozice. Tato se vypočítává ze všech dat a jejich náležitosti k aktuálnímu středu, takže složitost tohoto kroku je $k * n$. Toto se provádí v i iteracích, takže celková složitost je $O(i * k^2 * n)$. Po započtení dimenze je složitost $i * k^2 * n * d$. Změny oproti základní verzi jsou především k^2 oproti k a větší počet iterací. Algoritmus je však stále velice rychlý.

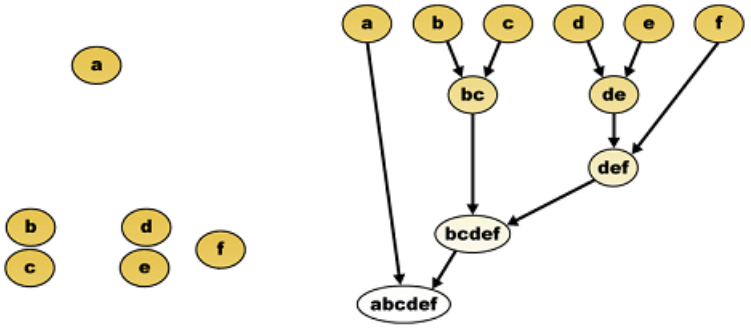
2.2.3 Hierarchické clusterování

Obecně

Hierarchický přístup lze do algoritmů zavést, pokud předem neznáme nejvhodnější rozlišení, nebo nás zajímají výsledky na různých úrovních. Zde nahlédnu metody pracující přímo hierarchickým způsobem.
Tyto metody fungují na principu spojování či naopak rozkládání clusterů na různých úrovních. Dělí se na metody aglomerativní (propojovací) a dělící.

Aglomerativní algoritmus začíná s jednotlivými vstupními body a ty postupně spojuje do větších a větších clsuterů až na úroveň, kdy propojí všechny. Dělící funguje obdobně, avšak v opačném směru. Dále se algoritmy liší podle přístupu, kterým se vybírají skupiny ke spojení, nebo pravidel pro rozdělení skupiny na menší, podle směru.

Práci algoritmu lze velmi dobře zachytit pomocí stromu (obr. 2.6), kde kořen obsahuje všechna data a v listech jsou jednotlivé body. Tím spíše je vidět, že jsou dělící a spojovací metody duální, pouze prochází stromem z opačných stran. To je však pravdou pouze teoreticky, v praxi jsou značné rozdíly. Jedním z důvodů je, že aglomerativní metody pracují s lokální informací a počáteční rozhodnutí mohou být chybná, zatímco dělící algoritmy pracují na začátku s plnou informací a tudíž se mohou kvalifikovaněji rozhodovat.



Obrázek 2.6: Vlevo jsou data a vpravo odpovídající strom. (Wikipedia)

Pokud u výsledného stromu vybereme úroveň, dostaneme clusterování daného rozlišení. Algoritmus tedy můžeme nechat běžet do zadané hloubky, ukončit, když vzdálenost mezi clustery odpovídá předem zadanému požadavku, nebo jej nechat proběhnout celý a mít k dispozici všechny výsledky.

Aglomerativní clusterování

Při tomto přístupu jde o propojování clusterů od triviálních až k celé vstupní množině.
V každém kroku jsou nalezeny dva nejbližší clustery a tyto jsou sloučeny do nového. Toto slučování probíhá až do sloučení všech, nebo do zadaného rozlišení, tedy počtu kroků nebo nalezených clusterů, což je téměř to samé.

Obvyklý přístup ke zjišťování vzdálenosti je sestavení matice vzdáleností, neboli matice $a_{ij} = d(c_i, c_j)$ (c_k jsou clustery). Při každém sloučení se sloučí náležité řádky a sloupce a matice se postupně zmenšuje. Zřejmě matice má velikost n^2, což nám dává nemalý základ pro složitost, ale z podstaty problému (hledání dvojic nejbližších prvků) to lépe nepůjde.

Otázkou, která algoritmy rozděluje, je podle čeho počítat vzdálenost mezi shluky.
Nechť $c_i = (x_k)_{k \in I}, c_j = (x_k)_{k \in J}$ jsou spojované clustery a $c_{ij} = (x_k)_{k \in I \cup J}$ cluster vzniklý jejich spojením, c_l jiný cluster, ke kterému chceme přepočítat vzdálenost.
Zde jsou některé možné přístupy.

i) vzdálenost nejbližších bodů
ii) vzdálenost nejvzdálenějších bodů
iii) průměrná vzdálenost bodů mezi clustery ($d(c_i, c_j) = \frac{1}{|c_i| * |cj|} \sum_{x \in c_i} \sum_{y \in c_j} ||x - y||$)
iv) celkový rozptyl uvnitř nového clusteru
v) nárůst chyby nového clusteru oproti původním

V prvních dvou případech je přepočítávání nových vzdáleností triviální, poslední tři přístupy jsou účinnější, ale také náročnější.
V případě i) je $d(c_l, c_{ij}) = min(d(c_l, c_i), d(c_l, c_j))$, případ ii) obdobně. V obou případech je tedy výpočet jedné vzdálenosti konstantní, celkem v jednom kroku přepočítání vzdálenosti ke všem ostatním, tedy řádově n. Vypočítat vzdálenost mezi clustery v bodě iii) má složitost $|c_l| * |c_{ij}|$, tento výpočet je třeba provést ke všem clusterům.
Vypočítat varianci clusteru je lineární úkol k jeho velikosti, je ale potřeba ji vypočítat ve všech dvojicích c_{ij}, c_l $\forall l$, což dává složitost n^2 v každém kroku v případech iv) a v).

Algoritmus lze ukončit, je-li to žádoucí, po daném počtu kroků, tedy při žádaném počtu vzniklých clusterů (ten se rovná počtu všech bodů - počet kroků). Druhá možnost je zadat prahovou vzdálenost mezi clustery, ve které by již spojování, podle předem známých skutečností, nevedlo k žádoucímu výsledku.

Dělící clusterování

Jedna z výhod této metody vzniká, když nechceme nechat výpočet proběhnout až do konce, protože pak pracujeme s menším počtem clusterů, než u aglomerativní varianty, ikdyž bude počet kroků stejný. Je to tím, že zde začínáme jedním clusterem a počet po krocích stoupá, zatímco u aglomarativního začínáme na maximálním počtu, který pak klesá.
Druhou výhodou, jak již bylo řečeno, je globální přehled o datech při rozhodování, což tuto variantu činí robustnější. Je to však vyváženo větší náročností při využívání těchto údajů.
Nelze dát jednoduchý návod, jakým způsobem clustery dělit, je třeba zapojit nějaký další algoritmus, který toto rozdělení vyhledává. Možnou variantou je použití jiného rychlého clusterovacího algoritmu, který najde nejvhodnější rozdělení dané podoblasti (děleného clusteru) na dva (nebo více) nové.
Díky své rychlosti je vhodnou volbou k-means. Složitost k-means v jednom kroku bude $O(n * i * k)$ (i je počet iterací). To při řádově n dělení v hlavním algoritmu dává $O(n^2 * i)$, kde i je velmi malé, k předpokládáme jako malou konstantu, nejspíše 2.

Složitost

Složitost této třídy algoritmů obecné začíná na n^2, výjimkou mohou být dělící algoritmy, které při rozhodování nevyužívají celou dostupnou informaci. Tato složitost je pro běžný clusterovací problém vysoká, ale pokud požadujeme znalost hierarchického rozložení clusterů, je tento přístup vhodný.

2.2.4 Kohonenovy mapy

Obecně

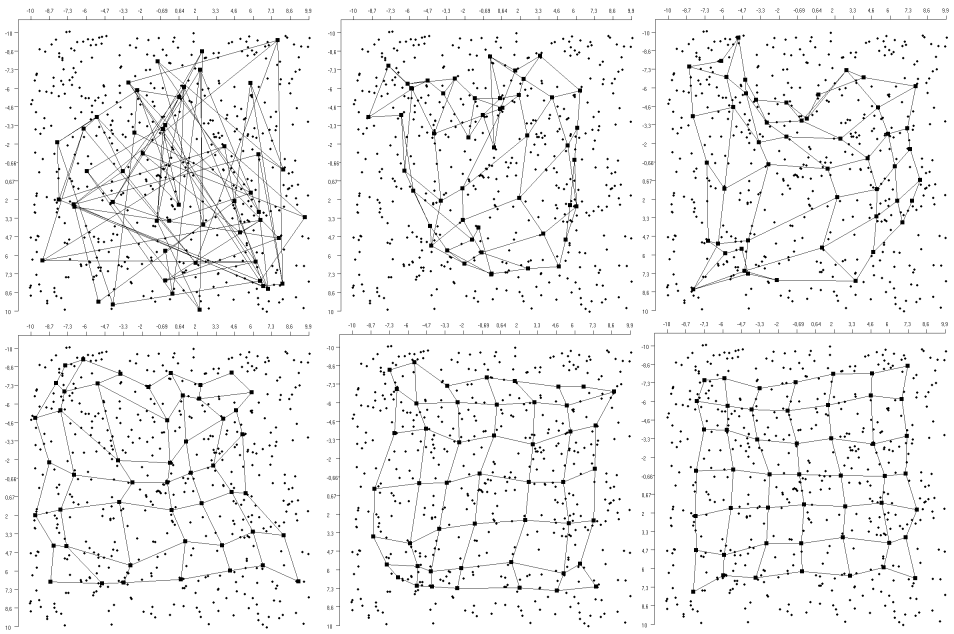
Tento algoritmus spadá už spíše do oblasti neuronových sítí. Využívá se k nízkodimenzionálnímu zobrazení dat s vysokou dimenzí a k hledání složitějších vztahů v datech. Další obvyklý název je samoorganizující se mapa (SOM).
Zatímco základem předchozích byly v podstatě nezávislé středy, které se organizovaly podle všech dat, centra zde, nebo zde spíše již neurony, jsou propojeny sítí vazeb, díky které se navzájem ovlivňují a učí se v každém kroku podle jednoho vstupu. Vazby jsou dány již při inicializaci a pevně určují topologii sítě. Neurony se snaží co nejlépe rozmístit mezi data, ale přitom zachovávají původně zadané rozložení.

Stručně k fungování: v každé iteraci se síti předloží jeden vstup, určí se jemu nejbližší neuron sítě, označovaný jako vítěz a tento se v prostoru přiblíží ke vstupu, takže neuron pak tento vstup lépe vystihuje. Kromě vítěze se takto adaptují i neurony v jeho blízkém okolí, ovšem okolí je zde dáno hloubkou v sítí, nikoliv vzdáleností v prostoru. Toto se provádí po předem daný počet kroků, přičemž na začátku je adaptace výrazná a postupně se její míra snižuje až k nule - síť postupně "chladne".
Tento algoritmus není ideální na hledání shluků, jelikož vazby mezi neurony jsou v tomto ohledu jakýmsi omezením.
Výsledkem je přibližné určení shluků dat, ale co je důležitější - nalezení souvislostí mezi shluky a různými oblastmi v datech, díky tomu, že blízké shluky budou nyní pravděpodobně nalezeny neurony, které jsou si blízké v síti.
Další aplikací Kohonenových sítí je nízkodimenzionální reprezentace složitých dat. Zatímco data mohou mít mnoho (často stovky) dimenzí, síť se používá 1-3 rozměrná, dvourozměrná nejčastěji. Poté, co se síť roztáhne mezi data a její neurony se ztotožní s oblastmi dat, ve kterých se nachází, můžeme tato data zobrazit ve dvou, resp. třech dimenzích a nejvýraznější rysy zůstanou v síti zachyceny.
Samozřejmě možností, jak taková data proložit jednoduchou sítí, je velice mnoho, takže je vhodné použít algoritmus víckrát s různou inicializací, parametry a pořadím předkládání vstupů, čímž se získají další výsledky.

V této části se budu zabývat základní verzí s pevnou, na začátku danou topologií a učením bez učitele, zdrojem je kniha autora této třídy algoritmů profesora T. Kohonena ([11]).
Další varianty zahrnují učení s učitelem, hierarchický přístup, dynamickou topologii nebo síť rostoucí z pár neuronů na začátku.

Cíl

U tohoto algoritmu není specifikována jedna exaktní chybová (energetická) funkce, jelikož algoritmus má mnoho variant už jen v závislosti na parametrech a síti.
Celkově vzato se jedná opět o minimalizaci součtu vzdáleností mezi vstupy a neurony (aby byly vstupy neurony co nejlépe vystiženy), navíc zde ale vystupují vazby uvnitř sítě, které závisí na volbě topologie a funkce přenosu adaptace neuronů po síti.



Obrázek 2.7: Ukázka práce Kohonenovy mapy 7x7 v iteraci 0, 85, 140, 250, 550, 2000. Vstupy jsou 1000 bodů s uniformní distribucí.

Inicializace, topologie, parametry

Parametry

Parametry základní verze Kohonenových map jsou:
- Velikost a topologie sítě : topologie se u 2d sítě standardně využívá čtvercová, trojúhelníková a hexagonální.
- Počet kroků : síť by měla všechny vstupy projít nejméně jednou, lépe vícekrát. V praxi bývají iterací řádově tisíce až desítky tisíc.
- Parametr učení a funkce jeho postupného utlumování v závislosti na počtu kroků : Počáteční hodnota by měla být z intervalu (0,1], funkce snižování v závislosti na počtu iteraci může být víceméně libovolná.
- Funkce přenosu adaptace z vítěze na jeho sousedy v závislosti na síťové vzdálenosti od něj. Použitá funkce může být opět různá.

Průběh

Inicializace:
1) Vytvoření sítě podle zvolených parametrů
2) Umístění jejích neuronů do prostoru, obvykle náhodně mezi vstupy

Průběh:
Prováděj 1) a 2):
1) Vezmi vstupní bod x (měly by přicházet náhodně) a najdi k němu nejbližší neuron n^*
2) Aktualizuj tento vítězný neuron podle vzorce 2.5 a podobně aktualizuj i jeho sousedy v síti až do zadané maximální hloubky.

Dokud není dosaženo daného počtu kroků.

Vzorec pro aktualizaci neuronu n:

$$n = n + \alpha * \theta(n, n^*) * (x_k - n) \tag{2.5}$$

x_k - vstup, podle kterého aktualizujeme
n^* - vítězný neuron
α - s počtem kroků klesající parametr učení
$\theta(n, n^*)$ - funkce, která určuje, o kolik se bude aktualizovat neuron n v závislosti na jeho vzdálenosti (v síti) od vítěze n^*
Vítězný neuron, společně s jeho blízkými sousedy, se tedy ke vstupu přiblíží.

Složitost

Složitost je zde O(nk), kde n je počet předložených vstupů. Nemusí to být to samé, jako celkový počet vstupů, obvykle se vstupy prochází opakovaně. V každém kroku se hledá nejbližší neuron, tedy *k. Konstanta je zde vyšší, obsahuje počet neuronů, které jsou v jednom kroku aktualizovány - např. při čtvercové 2D topologii a hloubce 1 jich je 5.

2.2.5 Rostoucí Kohonenovy mapy

Obecně

Obvyklejší název je rostoucí samoorganizující se mapy (GSOM - growing self-organizing map). Princip algoritmu a struktura, se kterou zde pracu-

jeme, je stejná, jako u klasických Kohonenových map. Tato varianta je však pružnější, protože tvar mapy se přizpůsobuje konkrétním podmínkám. V klasickém algoritmu pracujeme s pevnou mapou zadanou před jeho začátkem, v této variantě algoritmus začíná jen s nějakou základní buňkou (čtverec, šestihran... podle topologie), která se podle potřeby rozrůstá, dokud nejsou data dostatečně pokryta.
Výhodou tohoto přístupu je, že se síť lépe přizpůsobí konkrétním datům a lépe vystihne jejich tvar. Nevýhodou je, že nemáme nad algoritmem takovou kontrolu a výstup nemusí být podle našich představ. Například pro zobrazení výsledků je vhodnější pevná topologie, kde pak máme pokrytou celou dvourozměrnou plochu sítě a můžeme síť dobře vykreslit.
Nepodařilo se mi nalézt dostatečnou dokumentaci algoritmu, takže zde předložím a popíši vlastní doplnění některých jeho částí. Vycházel jsem přitom z práce [8].

Inicializace

Při inicializaci je vytvořena pouze základní buňka sítě. Algoritmus by mohl začít i s jedním bodem (alespoň některé varianty), ale jistě můžeme předpokládat, že se tato buňka využije.
Dále podle uživatelem zadaného parametru rychlosti růstu SF a dimenze D určíme růstovou hranici GT (growth threshold) podle vzorce

$$GT = -D^* ln(SF) \tag{2.6}$$

Podle mého názoru není tento vzorec ideální, protože nebere v potaz samotná data a hodnota GT je přitom v algoritmu porovnávána s hodnotou, jež závisí na velikosti dat a zvoleném měřítku. V mé variantě je tedy navíc tato hodnota vynásobena dalším číslem odhadnutým (velmi jednoduše a určitě to jde lépe) z velikosti vstupních dat (ve smyslu vzdálenosti od jednoho konce k druhému).

Průběh

Algoritmus se obvykle popisuje jako dvoufázový - první je růstová fáze, druhá je vyhlazovací. Síť se rozrůstá pouze v první fázi, kdy se však též přizpůsobuje jako v SOM, druhá fáze je několik přidaných zakončovacích iterací, ve kterých se síť již jen aktualizuje. Můj algoritmus pracuje jen s růstovou fází, druhá fáze není nutnou podmínkou k fungování, je spíše volitelným rozšířením.

Varianty se liší samozřejmě topologií. Nejčastější je trojúhelníková, o něco náročnější, ale zřejmě s lepšími výsledky je varianta šestiúhelníková. Třetí možností je obdélníková topologie.
U rostoucích map je dalším rozdílem technika růstu sítě. Ve chvíli, kdy je rozhodnuto, že se má neuron rozrůst, mu můžeme buď přidat jednoho souseda a toho napojit do sítě, nebo jich přidat najednou více. Druhý přístup je takovým kompromisem mezi přizpůsobivostí sítě a kontrolou jejího tvaru - růstová pravidla mohou být volena tak, aby síť stále udržovala žádoucí tvar a vlastnosti.

Krok a délka algoritmu nemusí být v GSOM tak přesně stanoven jako v SOM.
Kroky se dělí na dva druhy. Větší iterace jsou iterace růstové - jeden růst sítě odpovídá jedné iteraci. Jejich počet může být pevně zvolen, nebo můžou další kroky přibývat, dokud síť roste dostatečnou rychlostí.
Druhý typ kroku je obsažen v jedné růstové iterací a podobá se (svojí délkou) kroku SOM.
Popíši algoritmus tak, jak byl popsán v [8] a k němu svoje úpravy či doplnění.

Předpokládejme, že je již síť inicializována. Dále máme vypočtenu hodnotu růstového prahu GT a parametr učení učení α, který v každém kroku klesá. Oproti SOM obsahuje každý neuron navíc další hodnotu a tou je jeho kumulativní chyba E, na počátku nulová.

Růstová fáze:
1) Zvolíme vstupní bod x a najdeme k němu vítězný neuron n^*, tzn. neuron, který je v euklidovské metrice nejblíže.

2) Aktualizujeme tento neuron a jeho blízké okolí stejně, jako v SOM. Tedy neuron se aktualizuje podle $n = n + \alpha * \theta(n, m) * (x_k - n)$.
V [8] funkce síťové vzdálenosti od vítěze θ chybí, nevidím k tomu však důvod.

3) Ke kumulativní chybě E^* vítězného neuronu n^* přičti vzdálenost n^* od x. Tedy $E^* := E^* + ||n^* - x||$.

4) Pokud chyba některého neuronu n_i překročila růstový práh ($E_i > GT$), jsou dvě možnosti.
Je-li neuron n_i vnitřní (má-li všechny sousední pozice obsazené), propaguj

chybu k sousedům.
Je-li neuron n_i okrajový, proveď růst. To znamená přidání jednoho nebo více neuronů do sousedství n_i, dále vynulování E_i (není v [8]) a nastavení parametru učení α na počáteční hodnotu. 5) Prováděj 1)-4), dokud síť roste dostatečnou rychlostí.
Jinou podmínkou může být zadaná maximální velikost sítě.
V mém přístupu je dán pevný maximální počet kroků, přičemž počet provedených kroků je vynulován při každém růstu (a tím pádem i α, která na něm závisí). Pokud se síť nerozroste dříve, než je dosažen maximální počet kroků, algoritmus končí. Jelikož byla tím pádem celá tato iterace využita pouze na přizpůsobování sítě klasickým způsobem aniž by rostla, považuji další (vyhlazovací) fázi za zbytečnou.

Vyhlazovací fáze:
Přizpůsobuj síť jako v klasickém SOM (nyní již bez růstu) po zadaný počet kroků.

Síť se tedy rozrůstá především do směrů, ve kterých se nachází nejvíce vstupních bodů. Pokud je nějaký výraznější shluk již sítí překryt, pak chyba, kterou v síti generuje do vnitřních, nerostoucích vrcholů, se propaguje do stran a růst po obvodu umožňuje zahuštění sítě v místě tohoto shluku. Tedy síť se chová podle očekávání.

Složitost

Složitost závisí především na počtu růstů a každá růstová iterace se dá srovnat s jedním průběhem klasického SOM, avšak s menší sítí a menším počtem kroků. Složitost by tedy mohla být přibližně $O(i*n*k)$, kde i je počet iterací (růstů), n počet vstupů předložených v jedné iteraci a k velikost sítě.

2.2.6 LVQ

Obecně

Tento algoritmus je jedinou zde zmíněnou ukázkou algoritmu učení s učitelem. Je to teda algoritmus určený k rozdělení vstupů do tříd, který se učí podle vzorů s již známou klasifikací. Předpokladem je, že víme, do kolika tříd budeme chtít data klasifikovat. Druhým je, že máme nějakou dostatečně velkou sadu dat, u kterých již známe třídy, do kterých patří a podle kterých můžeme

tuto "síť" (žádná síť to není) naučit.
Popsán zde bude LVQ 1. LVQ 2 v učící fázi aktualizuje dva neurony najednou, LVQ 3 zavádí další omezení pro aktualizaci neuronů při učení. Informace jsou čerpány z [9].

Síť

Síť sestává z k neuronů, které mají dánu pozici v prostoru a každý je reprezentantem jedné z m tříd. $m <= k$, často se k reprezentaci jedné třídy používá neuronů několik.

inicializace

Nejdříve je potřeba znát počet tříd m, které chceme rozlišovat. Ten je obvykle znám z podstaty dat, nebo se může zjistit předzpracováním nějakým clusterovacím algoritmem.
Dále zvolíme počet neuronů k, kterými budeme vstupy rozlišovat (zřejmě je $k >= m$) a každý neuronu pevně přiřadíme k jedné z tříd. Pro vhodný počet neexistuje žádná formule, obvykle se iterativně upravuje podle spokojenosti s výsledkem učení.
Tyto neurony jsou pak rozmístěny do oblasti vstupních dat. Protože zatím není známo, ve které oblasti se budou nacházet vstupy patřící do třídy reprezentované tím kterým neuronem (zjistit toto je úkolem algoritmu), je obvyklé rozmístit neurony náhodně.
Posledně zvolíme parametr učení α, $0 < \alpha < 1$, a funkci jeho snižování v závislosti na počtu iterací t. α může být konstantní, pan Kohonen v [9] doporučuje monotónní snižování. Dalším rozšířením je přiřazení parametru učení každému neuronu zvlášť, přičemž pak samozřejmě klesá rychleji.

Učení

Učení probíhá kompetičním způsobem. Máme připravenu učící množinu - tedy množinu vstupů, u kterých je známa příslušnost do rozlišovaných tříd. Tyto vstupy jsou postupně síti předkládány, jednou, je-li jich dostatek, jinak vícekrát.
V iteraci je síti předložen vzor $x(t)$ a nalezen vítězný neuron n^*, to jest ten, který je k $x(t)$ ze všech nejblíže. Tento $n^*(t)$ je adaptován podle vzorce

$n^*(t+1) := n^*(t) + \alpha(t)(x - n^*(t))$, je-li $x(t)$ ze třídy reprezentované $n^*(t)$

$n^*(t+1) := n^*(t) - \alpha(t)(x - n^*(t))$, je-li $x(t)$ z jiné třídy,

ostatní neurony se nemění.

Je vidět, že se tímto způsobem neurony postupně dostanou dovnitř shluků dat z jejich třídy. Algoritmus se vlastně podobá k-means, ale má mnohem jednodušší práci, protože má při adaptaci přesné informace o tom, kam neurony patří a kam ne. Ovšem za předpokladu, že učící množina dostatečně dobře reprezentuje reálná data.

Klasifikace

Klasifikace nějakého prvku x je pak jeho předložení síti, tzn. nalezení euklidovsky nejbližšího (vítězného) neuronu n^*, přičemž o x je tím rozhodnuto, že patří do třídy reprezentované n^*.

Složitost

V učící fázi je procházeno n vzorů (opakování počítáno vícekrát) a každý je porovnáván s k neurony pro určení vzdálenosti, což nám dává složitost $O(nk)$
Složitost klasifikace je stejná, pouze $O(k)$ pro jeden klasifikovaný vzor. Šla by zlepšit na logaritmickou využitím nějaké prostorové vyhledávací struktury, např. quad-stromem.

2.3 Indexy

Do třídy indexů, nebo též validačních kritérií, spadají různé hodnotící funkce přiřazující výsledku (nebo stavu) algoritmu nějakou reálnou hodnotu a jsou snahou o co nejjednodušší postihnutí jeho kvality. Nejjednodušší kritéria pracují pouze se vzdálenostmi bodů a středů, složitější berou v potaz i další vlastnosti, jako poměry hustot dat v prostoru a v nalezených clusterech. Indexy se dají použít k ohodnocení výsledku a tím pádem k výběru nejvhodnějšího agoritmu a parametrů, nebo k ohodnocení stavu algoritmu v jeho průběhu a určení, zda se již má skončit.

2.3.1 Energetická funkce

Nejedná se o validační kritérium jako takové, ale jako funkci ohodnocující clusterování jsem ji zařadil sem. Jsou zde dvě verze energetické funkce - pro diskrétní a fuzzy clusterování. K minimalizaci funkce vede, při pevném počtu středů, jejich umístění do těžišť shluků dat. S přidáváním středů se hodnota indexu výrazně snižuje, dokud je počet středů $\leq$ počtu shluků. Pokud jsou všechny shluky již pokryty, s dalšími středy bude hodnota funkce dále klesat, ale již znatelně méně.

Základní nergetická funkce:

$$E_{km} = \sum_{i=1}^{k} \sum_{x \in c_i} ||c_i - x||^2 \tag{2.7}$$

Energetická funkce pro fuzzy verzi:

$$E_{fkm} = \sum_{i=1}^{k} \sum_{j=1}^{n} u_{ij}^{m} ||c_i - x_j||^2 \tag{2.8}$$

c_i jsou středy, x_j vstupy a m parametr > 1

2.3.2 Dunnovo kritérium

Velmi jednoduchá, ve většině případů nepoužitelná technika. Je však základem geometrického přístupu k hodnocení clusterování. Vzdálenosti všech bodů a středů zde nehrají příliš roli, rozhodující je rozložení shluků samotných. Celý index je pouze poměrem vzdálenosti mezi dvěma nejbližšími clustery (vzdálenosti dvou nejbližších bodů) a průměru největšího clusteru (vzdálenost dvou nejvzdálenějších bodů). Na velikosti dat nebo počtu clusterů tedy nezáleží, ze všech nakonec roli hrají pouze nejvýše tři clustery. Algoritmus může fungovat, pokud v datech není šum, protože pokud by do nějakého clusteru patřil jediný vzdálený bod, jeho průměr by se počítal podle něho a vzdálenost k sousednímu clusteru také. Nejlepšího výsledku (největší hodnoty indexu, zde se maximalizuje) dosáhnou malé (kulové) dobře oddělené a pravidelně rozptýlené clustery.

Maximalizuje se funkce:

$$\frac{min(d(c_i, c_j))}{max(diam(c_k))} \tag{2.9}$$

kde c_n jsou clustery, $d(c_i, c_j)$ je nejmenší vzdálenost bodů ve shlucích c_i, c_j a $diam(c_k)$ je průměr shluku = největší vzdálenost dvou bodů v něm.

2.3.3 SD index

Tento index je již složitější a využívá rovnou několik pohledů na geometrii dat a výsledku clusterování.
Hodnota indexu se skládá ze dvou částí:

$$SD = \alpha * scat + dist \tag{2.10}$$

kde hodnota $\alpha > 0$ určuje poměr významnosti těchto dvou složek a index je tedy minimalizován, jsou minimalizovány obě složky.

První část - *scat* - udává poměr variance dat vůči varianci uvnitř shluků. Variance všech dat σ se vypočte jako vektor $\sigma = (\sigma^p)_{p=1}^d$ o složkách

$$\sigma^p = \frac{1}{n}\sum_{i=1}^{n}(x_k^p - \bar{x^p})^2 \quad \forall p = 1, \ldots, d \tag{2.11}$$

$\bar{x}$ je těžiště všech dat, x_k vstupy, n je počet vstupů, d dimenze.

Variance shluku se vypočítává stejně, jen v rámci jednoho shluku, tedy $\sigma_i = (\sigma_i^p)_{p=1}^d$ a složky jsou

$$\sigma_i^p = \frac{1}{n}\sum_{x \in c_i}(x^p - \bar{c_i^p})^2 \quad \forall p = 1, \ldots, d \tag{2.12}$$

kde c_i tentokrát znamená celý shluk a $\bar{c_i}$ jeho těžiště.

Celé *scat* se vypočte jako

$$scat = \frac{\frac{1}{k}\sum_{i=1}^{k}||\sigma_i||}{||\sigma||} \tag{2.13}$$

k je počet shluků.

Výpočet se podobá základní energetické funkci, navíc ale zohledňuje varianci celých dat je tedy vhodnější i pro srovnání výsledků na různých datech.

Druhá část, *dist*, neboli rozdělení shluků, určuje, jak dobře jsou shluky v prostoru rozmístěny. Nejlépe to řekne vzorec:

$$\frac{max(||c_i - c_j||)}{min(||c_i - c_j||)} \left(\sum_{j=1,i=1}^{k} ||c_i - c_j|| \right)^{-1} \tag{2.14}$$

kde všechna c_i jsou středy clusterů. Pro optimální hodnotu tedy vzorec vyžaduje co nejpravidelnější rozložení clusterů.

Tento index vypadá složitě, ale náročnost je dosti nízká. Složitost první části je $O(n)$, druhé $O(k^2)$.

2.3.4 Entropy index

Tento index je rozšířením energetické funkce fuzzy k-means o funkci entropie, která je od E_fkm odečtena, přičemž se vyvíjí obdobně (klesá s rostoucím počtem clusterů) a staví tím minimum indexu do nějakého neokrajového bodu na oboru počtu clusterů. Při vhodné volbě váhového parametru α umožňuje minimalizace tohoto indexu nalezení vhodného optima pro počet clusterů k a používá se pro plně autonomní clusterování.
Informace k tomuto indexu jsou čerpány z práce [13].

O mírách náležení vstupů ke clusterům $u_{i,j},\ i \in 1,\ \ldots,\ k,\ j \in 1,\ \ldots,\ n,$ platí, že $u_{ij} \in [0, 1]$ a $\sum_{i=0}^{k} u_{ij} = 1$. $u_{i,j}$ lze tedy interpretovat jako předpokládanou pravděpodobnost, že vstup x_j patří do clusteru v_i. Tím pádem pravděpodobnost clusteru v_i (p. že do v_i náhodně vybraný vstup patří) se rovná:

$$p_i = \frac{1}{n} \sum_{j=0}^{n} u_{ij}, \tag{2.15}$$

Z teorie pravděpodobnosti pak máme celkovou entropii dánu jako $\sum_{i=0}^{k} p_i log(p_i)$.
Celý index se pak rovná

$$I_{entr} = \sum_{i=1}^{k} \sum_{j=1}^{n} u_{ij}^m ||c_i - x_j||^2 - \alpha \sum_{i=0}^{k} p_i log(p_i). \tag{2.16}$$

Kapitola 3

Implementace klastrovacích algoritmů

3.1 Analýza

Pro clusterování je v dnešní době k dispozici nespočet nástrojů. První skupinou jsou různé aplikace vyvinuté pro studijní účely, obvykle open-source. Těchto existuje velice mnoho, univerzity a studenti je vyvíjejí pro vlastní testování. Kvalita a použitelnost se velmi různí. Velice povedeným příkladem je programová sestava Weka z univerzity Waikato na Novém Zélandu. Jako druhou skupinu bych pojal placené programy, které jsou však stále ještě využívány k vývoji a vědeckým účelům, například MatLab. Třetí skupinou jsou programy čistě komerční, určené přímo pro koncové uživatele.
Pro svou práci potřebuji program, který bych nemusel platit a mohl libovolně upravovat a rozšiřovat. Komerční programy odpadají z obou hledisek, jsou velmi drahé a nepočítají s modifikací. Druhá skupina je jaksi mezi. Programy jsou stále placené, ale je do nich lépe vidět. Jelikož mi nyní nejde o řešení reálých problémů, není rychlost rozhodující, takže nejvhodnější jsou nekomerční open-source programy. Dalším hlediskem, které je neméně důležité, je vlastní studium a nejlépe se člověk naučí to, co sám dělá. Takže konečnou volbou je program vlastní, při jehož psaní nejlépe proniknu do problematiky programování a nároků těchto algoritmů. Bezplatnost tento splňuje automaticky. Jaké budou další cíle?

Možnost úprav - rozhodující je mít možnost zasahovat do kterékoli části programu, algoritmy vylepšovat a přizpůsobovat aktuální potřebě.

Rozšiřitelnost - možnost stále přidávat další algoritmy pro jejich testování a studium.

Rychlost - rychlost ve skutečnosti není rozhodující a nepředpokládám, že bych mohl překonat komerční programy, ale je přesto důležitým cílem, protože je potřeba vyvíjet algoritmy co nejlépe.

Přehlednost kódu - programovat se dá přehledně i nepřehledně. Přehlednost dává omezení efektivitě, zde je však důležitější.

Nyní ještě nahlédnu, jak si v těchto ohledech stojí dva zmíněné programy - Weka a Matlab. Rychlost zatím vynechám, bude součástí testování později.

Program	Zdarma	Úpravy	Rozšíření	Přehlednost
Matlab	ne	ne	ano	ne
Weka	ano	ano	ano	ano

Matlab neumožňuje přepisování vlastního kódu, je pouze možné vytvořit si pomocí jeho nástrojů vlastní funkce.
Nevýhodou Weky je velmi malý počet algoritmů pro clusterování, většina funkčnosti je zaměřena na statistické algoritmy.

3.2 Nástroje

Jazyk

Zvolený jazyk je C++. S ohledem k objektovému přístupu (přehlednosti) by bylo možno použít i Javu, C++ je však vhodnější vzhledem k větší efektivitě a v neposlední řadě protože jej umím. Na seznamy je použit vector ze stdlib.

Prostředí

Program byl nejdříve vyvíjen jako konzolová aplikace ve Visual Studiu .NET. Následně ale přišly značné nároky na interface kvůli nimž jsem přešel do Borland C++ Buildera 5.0.
Programová část je od rozhraní dobře oddělena, takže přenést program v případě potřeby jinam je určitě možné.

VS: Plus je značná efektivita, podpora, kvalitní prostředí. Interface lze vytvořit ve standardní knihovně MFC, je však velice složitá a bez značných znalostí špatně použitelná. Existuje však řada dalších knihoven.

Builder: Výhodou zde byla možnost vytvořit v editoru velmi rychle libovolný interface, společně s mou předchozí dlouhou praxí ze střední školy. Borlandu je vytýkána značná neefektivita compileru, žádný rozdíl jsem ale zatím nepozoroval.

Použité knihovny

V progamu je použitá jediná cizí komponenta a tou je malá knihovna generující náhodná čísla s uniformním rozdělením. Viz. sekce 3.4.7.

3.3 Architektura

3.3.1 Možnosti

Algoritmy umělé inteligence lze obecně implementovat dvěma způsoby:
- entity algoritmu (jednotlivá data, neurony, ...) považovat za samostatné prvky a pracovat s nimi objektově
- entity uložit jako řádky/sloupce matic a pracovat maticově.

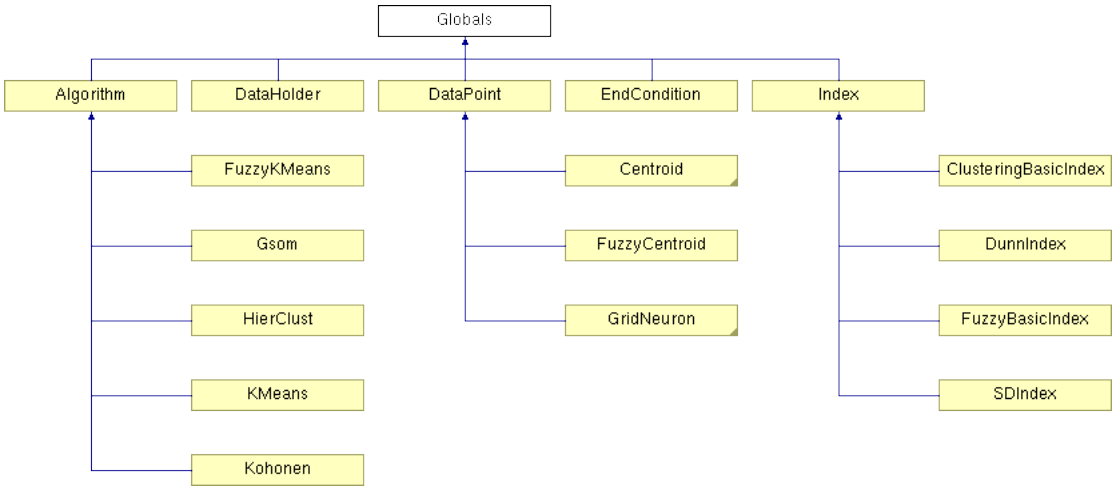
Efektivnější je samozřejmě druhý přístup. Je maximálně kompaktní a lze uplatnit celou řadu optimalizací vyvinutých pro maticovou aritmetiku.
Tento přístup však není tak názorný a intuitivní, jako první. Protože se jedná o studijní, nikoliv komerční projekt, je toto kritérium důležitější a objektový přístup je použit v mém programu.

3.3.2 Základní princip fungování

Program udržuje v paměti jeden seznam vstupních dat. Do něj lze body přidávat, načíst je ze souboru, nebo smazat.
Dále má přístup k seznamu clusterů a neuronů SOM pro vykreslování, nějaký obsah však bude mít pouze právě využívaná struktura. Dále je k dispozici vždy jeden zvolený algoritmus a jeho výsledky, dokud není zvolen jiný, či smazána data. Algoritmus se nejdříve inicializuje se zadanými parametry,

poté se buď mohou provádět jednotlivé kroky algoritmu, nebo se může spustit celý algoritmus, proběhne pak až do splnění zadané koncové podmínky.
Každý algoritmus obsahuje ukazatel na vector datových položek, které potřebuje. Jsou to obvykle vstupy a nějaká vlastní struktura. Například K-means obsahuje vector<DataPoint $*>$ $*$ pro vstupy a vector<Centroid $*>$ $*$ pro clustery.
Vstupní data, inicializované části algoritmu a výsledky lze průběžně graficky zobrazovat příslušnými příkazy.
Program též obsahuje možnost debugovacích výstupů, jsou-li povoleny. Tato možnost však není oficiální a aktivuje se přímo ve zdrojovém kódu.

3.3.3 Objektový model



Obrázek 3.1: Objektový model

Základní třídou je třída globals, která obsahuje globální defince, které využívají třídy ostatní. Dále se třídy rozvětvují na datové struktury, algoritmy a indexy, samostatně pak stojí hlavní třída DataHolder, starající se o celý program a třída EndCondition, která má na starosti ukončování algoritmů.

3.4 Části programu

Proměnné budou popsány pouze stručně, detaily je možné najít ve zdrojovém kódu.
(například místo vector<DataPoint $*>$ $*$ data napíši data - vector DataPointů)

3.4.1 Globals

Společným předkem všech tříd je třída Globals obsahující globální definice a deklarace.
Jsou zde deklarovány knihovny využívané všemi (většinou) tříd v programu. Jsou to: vector, iostream, fstream, math.h, float.h.

Dále se zde nastavuje přesměrování výstupu.
Možnosti jsou:
output pro standardní výstup
error pro chybová hlášení
debug pro debugovací výpisy.
Protože je ale aplikace okýnková, nikoliv konzolová, jsou užitečné pouze pro ladění, při normální práci nejsou využívány.

3.4.2 Data pro algoritmy

DataPoint

Nejjednodušší data programu. Reprezentuje jeden bod v n-rozměrném prostoru (n je libovolné, v rozumných mezích). Obsahuje vector<double> jako své souřadnice a metody pro různé vektorové operace. Nejsou implementovány všechny možné a v budoucnu může být nutné je dopsat.
Je předkem všech dalších typů - stará se o správu jejich souřadnic.

Centroid

Centroid reprezentuje jeden cluster pro práci diskrétních clusterovacích algoritmů. Obsahuje souřadnice prototypu (předek DataPoint) a vector<DataPoint $*>$, což je seznam vstupních bodů, které do tohoto clusteru patří.

Speciální metody:
clearOwnDataPoints - smaže seznam vlastních vstupů
AddOwnDataPoint - přiřadí jeden vstup do seznamu vlastních vstupů
MoveToDataCenter - přesune centroid do těžiště vlastních dat.

FuzzyCentroid

Tato struktura je určena algoritmu Fuzzy K-means. V tomto algoritmu se počítá s náležením všech dat všem centrům s ohodnocenou mírou náležitosti. Tedy vztah všechna data : všechna centra. Otázkou tedy je, zda mají data znát centra, nebo centra znát data. V algoritmu se opakují dva kroky. Přepočítání náležení dat k centrům a přesun center do nových pozic. První krok by šel v obou směrech stejně, druhý však vyžaduje reprezentaci centra znají data a byl tedy zvolena. Tato reprezentace je problematická při zpětném převodu na diskrétní reprezentaci (kde každý vstup je pouze u jednoho centra), ta však není součástí základního algoritmu. Jeden FuzzyCentroid je tedy clusterovým prototypem pro fuzzy clusterování, obsahuje seznam všech dat a jejich náležitosti.

Data:
Souřadnice v předkovi DataPoint a seznam ukazatelů na všechna data. Ty jsou zabaleny do vlastní třídy FuzzyData, která obsahuje ukazatel na vstup a ohodnocení jeho náležitosti k tomuto centru.

Zvláštní metody:
moveToOwnDataCenter - přesune centroid do polohy s nejnižší energií vzhledem k fuzzy energetické funkci a vstupům s ohodnoceným náležením.

GridNeuron

Základní jednotka Kohonenových map. Jeden neuron obsahuje DataPoint pro své souřadnice, dále vector ukazatelů (vector<GridNeuron $*$ >) na některé další neurony, čímž neurony tvoří síť. Vazba mezi dvojicí je vždy obousměrná.

Zvláštní metody:
clearNeighbours - smaže seznam sousedů
addNeighbour - přidá souseda do seznamu
update - hlavní metoda. Provádí aktualizaci neuronu podle zadaných para-

metrů a volá tuto funkci rekurzivně na sousedy, čímž vzniká procházení sítě do hloubky, oříznuté parametrem maxDepth. Tato aktualizace je bod 2) v popisu algoritmu.

GrowingGN

Název je zkratkou Growing Grid Neuron. Jedná se tedy o neuron používaný pro rostoucí kohonenovy mapy, neboli GSOM. Aktuálně je naprogramován pouze pro dvourozměrnou síť se čtvercovou topologí. Je rozšířením GridNeuronu o síťové souřadnice, které umožňují hledání pozic sousedů, kteří se mají k novému neuronu připojit a o nastřádanou chybu, která se používá pro určení, zda se má síť rozrůstat.

Nové proměnné:
cx, cy - síťové souřadnice neuronu.
error - kumulativní chyba tohoto neuronu.

Nové metody:
- getFreeCoords - vrátí síťové souřadnice náhodné z volných sousedských pozic. Využívá se pro hledání pozice pro růst.
- propagateError - tato metoda dostane jako parametr chybu, která má být na neuron přidána a ta se přičte k jeho error. Pokud error přesáhne growthThreshold, jsou dvě možnosti. Buď neuron nemá obsazeny všechny sousední pozice, pak může růst a nechá se tak jak je, růst provede algoritmus později. V opačném případě propaguje polovinu své chyby pomocí stejné metody mezi své sousedy.

3.4.3 Algoritmy

Algorithm

Předek určující rozhraní algoritmů. Předepisuje pouze metody run() a step() pro spuštění a krokování algoritmu.

KMeans

Tato třída prování algoritmus K-means. Algoritmus ke své práci využívá vektor instancí třídy centroid, tedy seznam clusterů s jejich středy. Každý

centroid kromě středu obsahuje seznam svých dat a všechny centroidy dohromady mají všechna data. Také provádějí velkou část práce, třída KMeans jejich práci organizuje.

Vlastní proměnné:
stepCount - počet zatím provedených kroků
maxStepCount - maximální počet kroků
evalFunctionLimit - limitní hodnota indexu pro skončení algoritmu

Datové struktury:
data - vector vstupních bodů (DataPoint)
centers - vector centroidů (Centroid)

Důležité metody:
KMeans(konstruktor) - pouze konstruktor se všemi důležitými parametry provede zároveň inicializaci. U ostatních je nutné ji provést zvlášť.

init - provádí inicializaci algoritmu. Kromě nastavení proměnných vytvoří centra a rozmístí mezi data, náhodně do hyperkvádrové oblasti dané nejvzdálenějšími vstupními body v jednotlivých dimenzích. Poté provede přiřazení vstupních bodů nejbližším centroidům, takže lze již po inicializaci data zobrazovat nebo počítat indexy.

run - spustí algoritmus, který proběhne až do splnění některé koncové podmínky. V cyklu ověřuje ukončení a volá metodu step().
Koncové podmínky:
- dosažení maximálního počtu kroků
- index dosáhne zadané limitní hodnoty (je-li tato -1, nebude vůbec použita). Toto se provádí skrze třídu EndCondition, která dostane jako parametr typ indexu, limitní hodnotu a ukazatel na požadovanou datovou strukturu - zde centers. V cyklu se pak volá metoda check() z EndCondition.
- pokud je rozdíl použitého indexu mezi dvěma kroky menší, než daná hodnota Stability difference. Jde též skrze EndCondition.
step - provede jeden krok algoritmu. Obsahuje tyto podkroky:
- moveCentersToCenters - u každého centroidu zavolá metodu moveToDataCenter(), tím se centroid přesune do těžiště dat, která má přiřazena.
- clearCentersData - u každého z center zavolá clearOwnDataPoints(), čímž smaže seznamy jejich dat, ta budou opět přidána při dalším přepočítání.

- assignDataToCenters - pro každý vstupní bod zavolá vlastní metodu KMeans::findClosestCenter(),
která k danému vstupu najde nejbližší centroid a přiřadí vstup k němu.

FKmeans

Tato třída provádí algoritmus Fuzzy K-means a je rozšířením K-Means.

Vlastní proměnné:
algExp - parametr algoritmu (m ve vzorci)
stepCount - počet zatím provedených kroků
maxStepCount - maximální počet kroků
evalFunctionLimit - limitní hodnota indexu pro skončení algoritmu

Datové struktury:
data - vector vstupních bodů (DataPoint)
centers - vector centroidů (FuzzyCentroid).

Důležité metody:
Pozn.: Konstruktor, init, run fungují stejně, jako u K-means.

FKMeans(konstruktor) - pouze konstruktor se všemi důležitými parametry provede zároveň inicializaci. U ostatních je nutné ji provést zvlášť.

init - provádí inicializaci algoritmu. Kromě nastavení proměnných vytvoří centra a rozmístí mezi data, náhodně do hyperkvádrové oblasti dané nejvzdálenějšími vstupními body v jednotlivých dimenzích. Poté provede přiřazení vstupních bodů nejbližším centroidům, takže lze již po inicializaci zobrazovat nebo počítat indexy.

run - spustí algoritmus, který proběhne až do splnění některé koncové podmínky. V cyklu ověřuje ukončení a volá metodu step().
Koncové podmínky:
- dosažení maximálního počtu kroků
- index dosáhne zadané limitní hodnoty (je-li tato -1, nebude vůbec použita). Toto se provádí skrze třídu EndCondition, která dostane jako parametr typ indexu, limitní hodnotu a ukazatel na požadovanou datovou strukturu - zde centers. V cyklu se pak volá metoda check() z EndCondition.
- pokud je rozdíl použitého indexu mezi dvěma kroky menší, než daná hod-

nota Stability difference. Jde též skrze EndCondition.
step - provede jeden krok algoritmu. Obsahuje tyto podkroky:
- moveCenters - u každého centra zavolá metodu moveToOwnDataCenter(), která jej přesune do pozice s minimální energií.
- recalculateMemberships - projde všechna centra a u každého seznam jeho dat (což jsou všechny vstupy) a přepočítá k nim míru náležení.

getCenters - metoda pro převod center z fuzzy do diskrétní reprezentace. Je velmi složitá, což je způsobeno reprezentací dat ve FuzzyCentroid, která pro tento převod není vhodná.

Kohonen

Tato třída provádí základní algoritmus Kohonenových map (2.2.4).

Vlastní proměnné:
stepCount - počet zatím provedených kroků
maxStepCount - maximální počet kroků
diameter - maximální vzdálenost v síti od vítězného neuronu, do které se ještě propaguje aktualizace podle vstupu
vigilance - učící konstanta. Není to tak úplně konstanta, s počtem kroků klesá k 0.
interactionFunction - řetězec určující interakční funkci. Podle této funkce s aktualizují neurony v závislosti na vzdálenosti od vítěze.

Datové struktury:
data - vector vstupních bodů (DataPoint)
neurons - vector všech neuronů.

Důležité metody:
Kohonen(konstruktor) - pouze konstruktor se všemi důležitými parametry provede zároveň inicializaci. U ostatních je nutné ji provést zvlášť.

init - inicializuje proměnné a volá initNeurons.

initNeurons - vytváří neuronovou síť. Program zatím umí síť v 1-2 dimenzích se čtvercovou topologií. Další možnosti jde samozřejmě přidat.

run - spustí algoritmus, který běží dokud není dosaženo maximálního počtu kroků.

step - provede jeden krok algoritmu. Obsahuje tyto podkroky:
vypočítá učící konstantu v závislosti na počáteční konstantě a počtu kroků/celkový počet. Vybere náhodný vstup
findClostestNeuron - najde neuron nejbližší vybranému vstupu, na tomto neuronu je zavolána metoda update (jeho aktualizace), která se už dál šíří sítí samostatně.

Gsom

Tato třída je (značnou) modifikací třídy Kohonen a provádí algoritmus rostoucí Kohonenovy mapy(GSOM - 2.2.5). Prvek adaptace sítě je obdobný, incializace je zde velice jednoduchá a průběh mnohem složitější. Třída se liší natolik, že je samostatným potomkem Algorithm, tedy nedědí od Kohonen. V této verzi funguje pouze pro čtvercovou topologii ve dvou dimenzích.

Vlastní proměnné:
stepCount - počet zatím provedených kroků
maxStepCount - maximální počet kroků
growthThreshold - Hranice kumulativní chyby neuronu, za kterou následuje růst do jeho okolí. diameter - maximální vzdálenost v síti od vítězného neuronu, do které se ještě propaguje aktualizace podle vstupu. Na rozdíl od třídy Kohonen tato není parametrem, upravuje se průběžně podle velikosti sítě.
vigilance - učící konstanta. S počtem kroků klesá k 0.
interactionFunction - řetězec určující interakční funkci. Podle této funkce se aktualizují neurony v závislosti na vzdálenosti od vítěze.

Datové struktury:
data - vector vstupních bodů (DataPoint)
neurons - vector všech neuronů (GrowingGn).

Důležité metody:
Gsom(konstruktor) - pouze konstruktor se všemi důležitými parametry provede zároveň inicializaci.

init - inicializuje síť do počátečního stavu, tedy vytvoří základní buňku sítě. V tomto případě - čtvercová 2d topologie - je to jeden čtverec ze 4 neuronů.

run - zde algoritmus nemá předepsanou maximální délku. Zastaví se ve chvíli, kdy až do zadaného počtu kroků v jedné iteraci nedojde k růstu. Růst počet kroků iterace vynuluje.

step - provede jeden krok algoritmu. Postupně provádí tyto operace:
Zkontroluje, zda nemá algoritmus skončit - skončí, pokud počet kroků dosáhl daného maxima. Jinak zvolí náhodný vstup a najde vítězný neuron. Tento aktualizuje jako v Kohonen a zvýší jeho chybu. To provede tak, že na něj zavolá rekurzivní funkci propagateError, které předá přírustek chyby. Pokud neuron nemůže růst, pošle chybu dál do sítě.
Nyní algoritmus projde všechny neurony a hledá ty, u kterých chyba přerostla GrowthThreshold. Tyto již růst mohou a tento růst je proveden. Vždy je vytvořen nový neuron, pro něj nalezeny vhodné volné souřadnice a je zapojen do sítě. Po všech růstech tento krok končí.

HierClust

Tato třída je navržena pro provádění různých variant hierarchického clusterování (2.2.3), v této implementaci pouze pro aglomerativní algoritmus single linkage, neboli spojování podle minimální vzdálenosti.

Vlastní proměnné:
stepCount - počet zatím provedených kroků
maxStepCount - maximální počet kroků

Datové struktury:
data - vector vstupních bodů (DataPoint)
centers - vektor shluků, se kterými algoritmus pracuje. Jsou typu Centroid, ale středy zde nejsou využity. distances - vzdálenostní matice. Je typu vector<vector<double> >, tedy dvojrozměrné pole. Je navržena úsporně, obsahuje pouze pravou horní polovinu bez diagonály (protože matice má na diagonále nuly a je podle ní symetrická). Detaily implementace jsou popsány ve zdrojovém kódu.

Důležité metody:

HierClust(konstruktor) - inicializuje vše potřebné. Zde je vyžadováno dodání vektoru dat i prázdného vektoru pro centra.
init - inicializuje a vypočítává vzdálenostní matici.

run - spustí algoritmus, který běží dokud není dosaženo maximálního počtu kroků. Tento je menší ze zadaného parametru a počtu vstupních bodů -1, protože dále algoritmus nemá smysl.

step - provede jeden krok algoritmu. Obsahuje tyto podkroky:
- najde nejmenší hodnotu vzdálenostní matice, tzn. řádek i a sloupec j.
- do sloupce i zapíše nové vzdálenosti shluku i,j k ostatním
- to samé pro řádek i
- vymaže sloupec a řádek j
- sloučí skutečné shluky, tedy seznamy vlastních bodů náležitých shluků, do shluku i
- smaže shluk j

3.4.4 Indexy

Index

Rozhraní indexovacích tříd.
Proměnné:
IndexValue - zde se uchovává poslední vypočtená hodnota indexu
Metody:
setter, getter pro IndexValue
calculateIndex - vypočítá hodnotu indexu
reachedValue - vrací true nebo false podle toho, zda již index překročil danou hodnotu (protože některé indexy maximalizují, jiné minimalizují)

ClusteringBasicIndex

Základní energetická funkce (2.3.1) K-means (2.2.1).
Proměnné:
pointer na vector center (centra obsahují ve svých seznamech i všechny vstupy)
Metody:
Pouze podle rozhraní - calculateIndex a reachedValue.

FuzzyBasicIndex

Základní energetická funkce (2.3.1) Fuzzy K-means (2.2.2).
Proměnné:
pointer na vector fuzzy center (centra obsahují i všechny vstupy s ohodnoceními)
algExp - parametr Fuzzy K-means algoritmu.
Metody:
Pouze podle rozhraní - calculateIndex a reachedValue.

DunnIndex

Další index k diskrétní reprezentaci clusterování (vector<Centroid $*$ >). Viz. 2.3.2
Rozhraní je shodné s ClusteringBasicIndex.
Proměnné:
pointer na vector center
Metody:
calculateIndex a reachedValue.

SDIndex

Tento index je opět pro standardní interpretaci clusterů (vector<Centroid $*$ >). Viz. 2.3.3
Rozhraní je shodné s ClusteringBasicIndex.
Proměnné:
pointer na vector center
Metody:
calculateIndex a reachedValue.

3.4.5 Další třídy

EndCondition

Tato třída se stará o kontrolu koncových podmínek algoritmů - dosažení hodnoty indexu a dosažení (přibližně) stabilního stavu - rozdíl hodnoty indexu mezi dvěma kroky je menší, než daná prahová hodnota.

Proměnné:
index - ukazatel na používaný index. Používá se jedna instance po celý průběh algoritmu a v ní je uložena hodnota indexu.
limitIndexValue - prahová hodnota indexu
stableDifference - prahová hodnota rozdílu hodnoty indexu mezi dvěma po sobě následujícími kroky.
Metody:
EndCondition (konstruktory) - Pro každou rozdílnou sadu parametrů má zvláštní konstruktor. Rozdílem je například jiná datová struktura používaná pro výpočet indexu.
check - zkontroluje, zda bylo dosaženo některé koncové podmínky. Je-li hodnota prahové proměnné -1, tato koncová podmínka se nepoužívá.

3.4.6 Programové rozhraní

DataHolder

Tato třída se stará o celý běh programu a tvoří programové rozhraní, s nímž komunikuje rozhraní uživatelské.
Při běhu programu obsahuje instanci nejvýše jednoho algoritmu. Dále obsahuje vstupní data a jejich generování, načítání, ukládání, ta zapůjčuje algoritmům, které též spravuje, a centroidy (základní verzi), ve kterých se uchovává výsledek proběhlého algoritmu.

Proměnné:
data - ukazatel na vector dat. Vector se inicializuje v konstruktoru.
centers - ukazatel na vector centroidů. Inicializují se předtím, než jsou potřeba - před inicializací algoritmu.
min, max - vector<double> - souřadnice protilehlých rohů minimální kvádru, který obsahuje všechna vstupní data a má hrany rovnoběžné s osami. Tato lze vypočítat z dat, ale používají se často (nejvíce při krokování algoritmu a vykreslování), vyplatí se je tedy uschovat a počítat jen při změně dat.
kMeans, fKMeans, kohonen, hierClust - pointery na algoritmy. Tyto se inicializují až na příkaz uživatele v náležitých metodách initXXX

Seznamy fuzzy centroidů (F. K-means) nebo nebo neuronů (Kohonen) nejsou standardním výstupem a jsou uloženy pouze ve svých algoritmech, odkud je možno je získat.

Zvláštní metody:
settery: data - vector vstupů, centra - vector centroidů
gettery: data, centra,
neuronGrid - vector neuronů,
stepCount - počet provedených kroků algoritmu,
indexValue - hodnota zvoleného indexu u aktuálního algoritmu,
Min, Max - protilehlé rohy hyperkvádru obalujícího data

loadDataFromFile - načte vstupy ze souboru. Formát: jeden vstup na řádek, na řádku souřadnice oddělené mezerami.
saveDataToFile - uloží vstupy do souboru.
print vytiskne data na daný výstup

addUniformData - do vectoru dat přidá zadaný počet vstupních bodů uniformě náhodně rozložených v zadané oblasti. Počet souřadnic hranic udává dimenzi, ta musí souhlasit s dimenzí již vytvořených dat.
addNormalData - do vectoru dat přidá data s normálním rozdělením. Počet souřadnic střední hodnoty dává dimenzi.

step - provede jeden krok zadaného algoritmu.

initKMeans, initFKMeans, initKohonen - provede inicializaci algoritmu. Je-li již nainicializován jiný, smaže se. Dokud není zavolána další inicializace, pracuje se pořád se stejným algoritmem. runKMeans, runFKMeans, runKohonen - spustí běh zvoleného algoritmu, ten proběhne do splnění některé ze zadaných koncových podmínek.

3.4.7 Knihovny

Operations

Obsahuje různé užitečné funkce.

pointDistance - vrací Euklidovskou vzdálenost dvou vector<double>
getMin - vstupem je vector vstupů (DataPoint), vrátí vector<double>, který v každé složce obsahuje nejmenší hodnotu této složky mezi všemi vstupy.
getMax - jako getMin, ale největší.
random - vrací uniformní náhodné číslo ze zadaného intervalu. Double a int

verze.
standardNormal - vrací číslo ze standardního normálního rozdělení N(0,1).
normal - vrací číslo z normálního rozdělení N(mean, variability).
createUniformRandomVector - vytvoří vector<double> s hodnotami ze zadaného uniformního rozdělení.
createNormalRandomVector - vytvoří vector<double> s hodnotami ze zadaného normálního rozdělení.

mtRand

Cizí knihovna generující uniformě rozdělená data různých typů.
Od Takuji Nishimura a Makoto Matsumoto, Keido University.

Uživatelské rozhraní

Uživatelské rozhraní je sestaveno prostředky VCL knihoven, které jsou nadstavbou WinApi. Využívá se ve dvou případech - pro ovládání programu a pro grafický výstup. Rozhraní komunikuje výhradně s programovým rozhraním DataHolder.

Main

Main je metodou Form1 - hlavního okna programu. Tento soubor obsahuje mnoho metod pro ovládání různých prvků okna, blíže rozeberu jen ty důležité. Dělají tři různé věci: mění prvky okna (při změně vybraného algoritmu se nabízí jiné parametry), komunikují s programovým rozhraním (DataHolder) a volají vykreslování.

Proměnné:
data - pointer na instanci DataHolder
mtDRand - globální proměnná - instance generátoru náhodných double čísel z intervalu [0,1)
mtIRand - globální proměnná - generátor náhodných 32-bit integerů

Zvláštní metody:
Konstruktor (jmenuje se složitě, nadpis MAIN) - v konstruktoru se inicializuje náhodný generátor, který je sdílený pro celý program a DataHolder - vlastní program. Je-li v Globals povolen log výpis, inicializuje se výstupní

soubor.
readParamterList - pomocná funkce, ze stringu vytvoří vector<double>
translate - převádí názvy algoritmů a indexů z interfacových názvů do programových.
draw - vykreslí data podle požadavků (spouští Plot, druhou část uživatelského rozhraní). Otevře vykreslovací okno.

Další funkce jsou vázány na akce prvků interface:
Změna interface:
newDataDistComboBoxChange - změní parametry v Generate data podle zvolené distribuce
algChoiceComboBoxChange - změní parametry (a pár dalších věcí) podle zvoleného algoritmu.

Volání DataHolder:
generateDataButtonClick - přidá vstupní data do vectoru data v DataHolder
plotData(Clusters,Grid)ButtonClick - Zavolá draw s parametrem podle typu vykreslení
eraseDataButtonClick - zavolá eraseData DataHolderu, která smaže data a centra.
algInitButtonClick - zavolá inicializaci náležitého algoritmu v DataHolder.
algRunButtonClick - zavolá spuštění náležitého algoritmu v DataHolder (je-li algoritmus inicializován).
algStepButtonClick - provede krok algoritmu v DataHolder (je-li algoritmus inicializován).
indexCalculateButtonClick - zavolá getIndexValue v DataHolder, získá hodnotu zvoleného indexu u aktuálního algoritmu.

Plot

Vykresluje grafický výstup. Vykreslení se provádí na TBitmap, která se vykreslí na Form (okno). Před začátkem vykreslování je nutné spustit init, ve které se třídě vykreslovacího formu předá ukazatel na DataHolder, aby měla přístup k datům.
Parametry vykreslení jsou rozměry, které se posílají z uživatelského rozhraní, když se volá draw, která otevře vykreslovací okno a konstanta udávající typ vykreslení. Konstanty jsou 0 - pouze vstupy, 1 - vstupy a centra s barevným

rozdělením, 2 - vstupy a Kohonenova síť.
Další parametry jsou absolutní a poměrné velikosti různých prvků, které jsou definovány na začátku Plot.cpp.

Vykreslování se provádí ve dvou částech. Jednou je vykreslení souřadnicových os v drawCoordinate, druhou je vykreslení dat v plot.

3.5 Rozšiřitelnost

Uvedu zde dvě možnosti rozšíření programu a to přidání dalšího algoritmu a indexu. Jiné možnosti ponechám fantasii.

3.5.1 Nový algoritmus

1) Implementace s rozhraním Algorithm (Algorithm.h)
Požadován je konstruktor, init pro inicializaci všeho potřebného pro průběh, step pro jeden krok, run pro celý průběh. Pro dobré používání by metoda step měla být bezparametrická. Další metody jsou vhodné pro spolupráci s DataHolder, nejsou však povinné. Například getStepCount, pokud algoritmus počítá kroky nebo getIndexValue pro zjištění hodnoty nějakého indexu. V run lze použít třídu EndCondition pro kontrolování koncových podmínek dosažení hodnoty indexu a stability, nejlépe podle vzoru některého hotového algoritmu.

Vlastní datové struktury, jsou-li třeba, mohou být implementovány libovolně. Nejspíše bude praktické dědit od DataPoint. Měla by mít metodu print pro textový výpis (např. pro uložení do souboru).
Pro výstup by struktura buď měla mít převod na nějakou známou, která už má implementováno například vykreslování, nebo se musí zpřístupnit pro DataHolder a vytvořit pro ni vlastní výstupní metody.

2) Začlenění do DataHolder.cpp
Header:
- include algoritmu
- include datové struktury, pokud je nutné, aby byla dosažitelná z venku, například pro vykreslování
- přidání pointeru na algoritmus do proměnných.

- přidání metody initAlg a runAlg, která zavolá náležitou funkci uvnitř algoritmu.
Body:
- inicializace nových proměnných v konstruktorech DataHolderu
- implementace init a run.
- jelikož je povolena instance jen jednoho algoritmu, je třeba přidat delete tohoto nového do initu ostatních.
- přidat jako variantu do metod step, getIndexValue, getStepCount
- je-li to nutné, přidat další metodu pro přístup k algoritmu
- přidat delete algoritmu do destruktoru.

3) Začlenění do user interfacu (Main.cpp)
- přidat potřebné komponenty a varianty ve výběrech na formu
- Přidat změny interfacu při výběru algoritmu – algChoiceComboBoxChange
- přidat algoritmus do volání stisknutí tlačítek init, run, step

3.5.2 Nový index

1) Implementace s rozhraním Index (Index.h)
Kromě konstruktoru a destruktoru musí obsahovat proměnnou s poslední vypočtenou hodnotou indexu, pro ni getter a setter (setIndexValue, getIndexValue). Dále funkci calculateIndex, která spočte hodnotu indexu a metodu reachedValue, která vrátí, zda index překročil danou hodnotu (např. zda je index nižší než daná hodnota, při minimalizaci).

2) Začlenění do DataHolder.cpp
DataHolder o indexech nic neví, pouze hodnoty předává, algoritmus se volí skrz parametr string indexType.

3) Začlenění do dalších tříd
Pokud se má index využívat u algoritmu, je třeba ho přidat do getIndexType, kde se již typ rozlišuje a do run, kde se s ním instanciuje EndCondition. Dále pak do náležitých konstruktorů EndCondition (tyto se liší podle parametrů, které předává volající algoritmus)

3) Začlenění do user interfacu (Main.cpp)
- přidat do výběrů u koncové podmínky algoritmů, kde má být

- přidat nový název do translate (pokud se název používaný na formu liší od indexType v programu
- Přidat jako další item v ComboBoxech interfacu při výběru algoritmu – indexComboBoxChange

3.6 Testování

Nakonec bude provedeno testování výsledného programu proti srovnávaným konkurenčním programům - MatLabu a Weka. Bude provedeno na jedné sadě reálných dat pro porovnání úspěšnosti, poté na dostatečně velkých generovaných datech pro porovnání rychlosti.

Použitý systém:
CPU: Intel Core2 Duo 2.33GHz, FSB 1333MHz, 4MB L2 Cache
RAM: 2x1GB dual channel, 800MHz
Systém: Windows Vista Bussines SP1

Srovnávaný software:
Matlab 7.1 R14
Weka 3.4

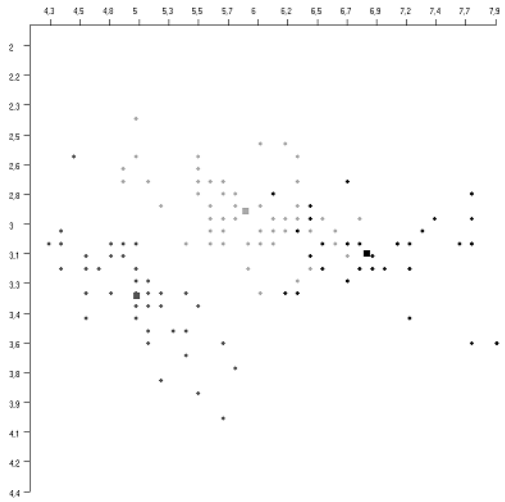
Test na reálných datech

K testování použiji známou sadu dat Iris. Obsahuje údaje o třech poddruzích kosatce, od každého padesát kusů. Jsou popsány čtyřmi hodnotami - délka a šířka okvětních lístků a těch zelených lístků pod nimi.

Pokusy jsou samozřejmě prováděný se třemi clustery.

K-means

Můj program
Z 50 pokusů bylo dosaženo nejlepšího (ne nutně globálního) minima (hodnota en. funkce 78,9) ve 43 případech. Úspěšnost klasifikace byla v tomto minimu 133/150, tedy 88,6%.
Weka



Obrázek 3.2: První dvě dimenze množiny Iris po K-means clusterování

Bylo dosaženo shodného výsledku.

MatLab
Úspěšnost MatLabu byla 134/150 - 89,3%.
K-means patří v MatLabu do Statistics Toolboxu.

Fuzzy k-means

Parametry: Exponen 1,1 (blíže k 1 byl se výsledek zlepšuje), ukončení při rozdílu en. funkce 0,001.

Můj program
V 9/50 pokusech bylo dosaženo jiného, než nejlepšího minima. Úspěšnost v nejlepším minimu (hodnota en. funkce po diskretizaci 78,95) je 134/150, tedy 89,3%.

Weka
Weka v tuto chvíli neobsahuje fuzzy k-means.

MatLab
Výsledek je přesně shodný.

Test rychlosti

Pro testování bylo použito 5000 bodů o 10-ti dimenzích, náhodně generovaných s uniformním rozdělením na intervalu $[-10, 10]^{10}$. Měření času bylo prováděno pouze pomocí stopek, takže chyba je v řádu jedné sekundy.

Parametry:
K-means: 50 clusterů.
Fuzzy k-means: 50 clusterů, exponent 1,5, ukončení při rozdílu energetické funkce 0,01.
Hierarchické clusterování (single linkage): Proběhne všech 5000 kroků.
Kohonenovy mapy: čtvercová síť 10x10, 5000 učících iterací (při náhodném výběru tedy nebudou pravděpodobně použita všechna data). Další parametry se různí podle přístupu jednotlivých programů.

Algoritmus	Můj program	Weka	MatLab
K-means	16s	18s	21s
Fuzzy k-means	187s	N/A	6s
Single linkage	5m	N/A	15s
Kohonenovy mapy	7s	N/A	1h

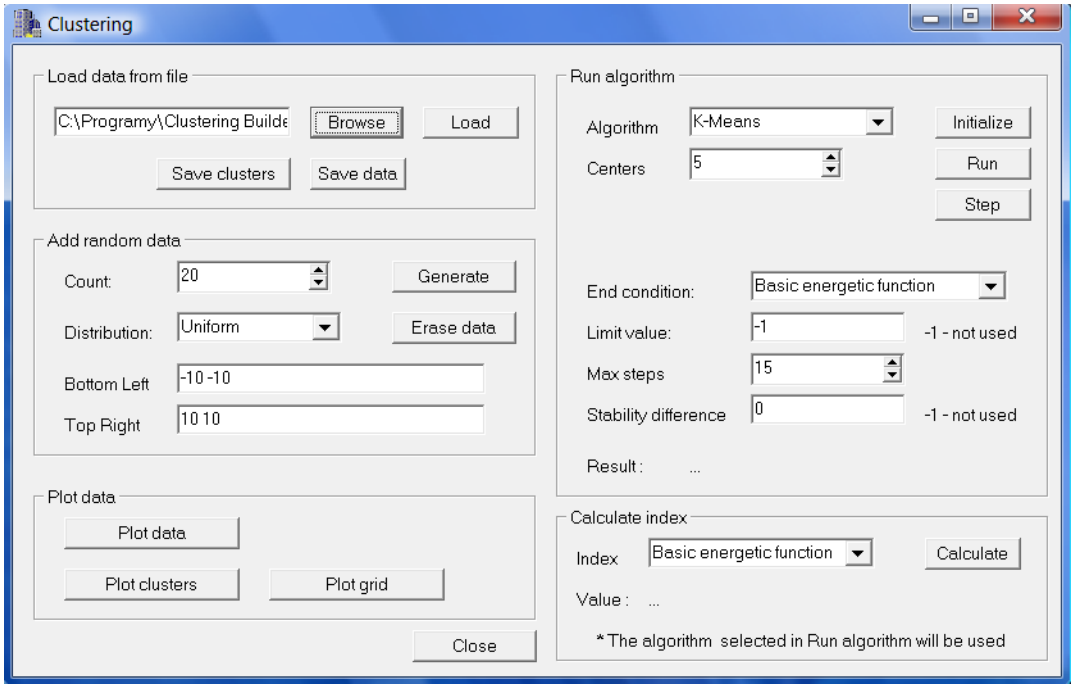
Zhodnocení

Ve většině případů jsou rychlosti programů značně rozdílné. MatLab jistě využívá nejnovější optimalizace algoritmů, moje algoritmy jsou napsány pouze podle definic s vylepšeními, která mě napadla. Někdy testy dopadly lépe pro mě, důvodem bude nejspíše zapojení mnoha komponent matlabu do výpočtu, který lze provést přímočaře. Je zřejmé, že by bylo dobré některé algoritmy vylepšit, aby rychlostí dosahovaly alespoň k MatLabu.

Kapitola 4

Uživatelská příručka

Program je ovládán z jednoho okna (na obrázku), dále budou popsány jeho části a funkce.



Obrázek 4.1: Interface programu

Okno pro ovládání obsahuje tyto části:

4.1 Load data from file

Možnosti práce s datovými soubory. Umožňují nahrát a uložit vstupní data z .dat souborů.
Formát: Jeden řádek = jeden vstup. Na řádku double čísla oddělená mezerami jsou souřadnice v jednotlivých dimenzích.

Další možností je uložení klasifikvaných dat, tedy export vektoru centroidů do souboru. Program umožňuje dvě reprezentace, bez změny parametru v kódu je přístupná pouze první. Tuto změnu je možno provést v metodě náležící tlačítku "Save clusters" v Main.cpp.
Parametr 0 (defaultní) : Nejdříve vypíše seznam clusterů s jejich čísly (číslo: souřadnice), následují všechny vstupy na jednotlivých řádcích, poslední souřadnicí je číslo clusteru.
Parametr 1 : Clustery jsou vypsány postupně - nejdříve souřadnice středu clusteru, pak jeho data, pak další cluster.

4.2 Add random data

Program si udržuje jeden seznam vstupních bodů.
Zde jsou možnosti pro přidávání vstupů do seznamu. Existují dvě volby pro distribuci náhodnosti rozložení dat.
Uniformní rozdělení - Vygeneruje zadaný počet (Count) vstupů v hyperkrychli zadané souřadnicemi Bottom left a Top Right. Počet čísel určuje dimenzi. Ta musí být stejná jako dimenze dat v paměti, určí se tedy podle první dávky přidaných dat.
Normální rozdělení - přidá shluk dat s normálním rozdělením kolem středu Mean (počet souřadnic opět určuje dimenzi) s rozptylem Variability.

4.3 Plot Data

Zde jsou možnosti pro grafické vykreslení dat a struktur algoritmů. Vykresluje se 2D. Data s nižší dimenzí se nevykreslí, u dat s větší se vykreslí pouze první dvě dimenze.
Možnosti jsou tři:
Plot data - vykreslí pouze vstupní body
Plot clusters - vykreslí vstupní body a centroidy. Body jsou barevně označeny

podle příslušnosti k centroidům. Využívá se u K-means, Fuzzy K-means a hierarchického clusterování (single linkage).
Plot grid - vykreslí data a síť Kohonenovy mapy.

Je-li okno otevřeno po inicializaci algoritmu, budou již struktury správně zobrazeny. Je-li okno otevřeno, obrázek se bude aktualizovat při krokování algoritmu (Step). Pokud se u algoritmu spustí Run, zobrazí se až konečný výsledek.

Celé vykreslování provádí třída Plot, která je však stále nad VCL a není tím pádem přenositelná. Od třídy DataHolder si pouze bere potřebná data.

4.4 Run algorithm

V této části se ovládají samotné algoritmy. Před spuštěním algoritmu je nutné algoritmus inicializovat, toto se podaří pouze jsou-li k dispozici nějaká vstupní data. Dokud není zavolána další inicializace, pracuje se pořád se stejným algoritmem.

První je možnost volby algoritmu, podle ní se upraví další možnosti.

K-means

Inicializace: Jediným parametrem je počet středů. Tyto středy se nainicializují náhodně (uniformě) v hyperkrychli, jejíž hranice v jednotlivých dimenzích jsou souřadnice nejvzdálenějších bodů v daném směru.

Krok (step): Jeden krok algoritmu je přiřazení nových dat středům a přesun středů to těžiště těchto dat.

Běh (Run): Algoritmus proběhne až do splnění některé koncové podmínky. K ohodnocení výsledku bude použit index zvolený v End conditon. Dostupné indexy jsou Basic energetic function - základní energetická funkce a Dunn index. Je-li zadána Limit value pod ním nerovnající se -1, bude dosažení této hodnoty použito jako jedna z koncových podmínek.
Další možností je omezení na počet kroků.
Třetí možnost je Stability difference - rozdíl mezi dvěma kroky, při kterém

se řešení považuje za dostatečně stabilní a program může skončit. K-means funguje diskrétně a 0 je zde obvyklé nastavení.

Fuzzy K-means

Inicializace: Parametry jsou počet středů a exponent, který ovlivňuje běh algoritmu a konvergenci (viz. sekce 2.2.2)

Krok (step): Jeden krok algoritmu je přepočítání náležitosti dat ke středům a přesun středů do těžišť.

Běh (Run): Algoritmus proběhne až do splnění některé koncové podmínky. K ohodnocení výsledku bude použit index zvolený v End conditon. Dostupné indexy jsou Fuzzy energetic function - základní energetická funkce pro fuzzy reprezentaci a dále indexy pro k-means (Basic energetic function a Dunn index), pro tyto je však nutné data vždy převádět z fuzzy reprezentace na diskrétní. Je-li zadána Limit value pod ním nerovnající se -1, bude dosažení této hodnoty použita jako jedna z koncových podmínek
Další možností je omezení na počet kroků.
Třetí možnost je Stability difference - rozdíl mezi dvěma kroky, při kterém se řešení považuje za dostatečně stabilní a program může skončit. Zde je vhodné zadat nějaké malé kladné číslo.

Hierarchické clusterování - Single Linkage

Tento algoritmus má jediný parametr a tím je počet kroků. Maximální možný počet se rovná počtu vstupů, pokud je tedy zadaný počet nižší, algoritmus se zastaví dříve.
Step umožňuje provádět slučování clusterů po jednom.

Tento algoritmus bohužel nezapadá do použitého systému vykreslování, který původně nepočítal s průběžně se měnícím počtem clusterů, takže se v každém kroku mění rozložení barev.

Kohonenovy mapy

Tento algoritmus má značně rozdílné parametry od předchozích.

Inicializace:
Size (Rozměry mřížky): Umožněny jsou jedna a dvě dimenze, tzn. budou zadána 1-2 čísla. Například 5 5 zadává mřížku 5x5 bodů.
Topology - topologie sítě, zde se zadává systém propojení mřížky. K dispozici je čtvercová topologie.
Interaction function. V kroku algoritmu se aktualizuje (přesouvá směrem ke vstupnímu bodu) vždy vítězný neuron a jeho sousedé v síti až do zadané hloubky. Nejvíce se přesune vítěz, sousedé už méně. Tato funkce určuje míru tohoto posunu v závislosti na vzdálenosti (počtu hran v síti) od vítěze.
Propagation depth - maximální hloubka, do jaké se sousedé aktualizují.
Max steps - maximální počet kroků. Toto je jediná ukončující podmínka a algoritmus probíhá různě v závislosti na ní. Vigilance - parametr určující počáteční rychlost učení sítě. Parametr klesá průběžně až k nule, té dosáhne v posledním (maximálním) kroku.

Běh, krok:
U tohoto algoritmu se vše nastavuje při inicializaci, takže zde se již pouze spustí.

Rostoucí Kohonenovy mapy - GSOM

Zadávané parametry se jen mírně liší od Kohonenových map.

Inicializace:
Velikost sítě se nezadává, při inicializaci je vytvořena základní buňka.
Steps in iteration - maximální povolený počet kroků jedné růstové iterace. Jakmile je v nějaké iteraci nedojde k růstu a je dosaženo tohoto počtu kroků, algoritmus končí.
Běh, krok:
V kroku provedena aktualizace jako v Kohonenových mapách, navíc se může síť rozrůst.

4.5 Index

Zde je možné zadat vypočítání jiného indexu, než hlavního, který byl použit při běhu algoritmu.

Kapitola 5

Závěr

Dá se říci, že hlavní cíle práce byly splněny. Program je můj vlastní, lze tedy snadno upravovat a rozšiřovat, navíc jsem algoritmům značně porozuměl a naučil se něco z jejich programování. Co se týče rychlosti, MatLab byl někdy překonán, někdy ne. Kvalita prostředí a nástrojů je samozřejmě nesrovnatelná, ale výstupy a ovladatelnost mého programu jsou poměrně použitelné. Díky různým možnostem ukládání a nahrávání dat je možné práci programů propojit a dosáhnout tak ideálního výsledku.

Literatura

[1] I.Gath, A. B. Geva: *Unsupervised Optimal Fuzzy Clustering*, IEEE Transactions on Pattern Analysis and Machine Intelligence, 1989, 11(7), 773-781

[2] N. Bolshakova, F. Azuaje: *Cluster validation techniques for genome expression data*,Department of Computer Science, Trinity College Dublin

[3] Various: *Numerical Recipes*, Cambridge University Press, 1988-1992, ISBN 0-521-43108-5

[4] M. Ramze Rezaee, B. P. F. Lelieveldt, J. H. C. Reiber: *A new cluster validity index for the fuzzy c-mean*, Leiden University Medical Center, 1997, DOI doi:10.1016/j.physletb.2003.10.071

[5] David Wishart *Clustering messy social data with clustangraphics*, University of St. Andrews, speech on RC33 2004

[6] Ferenc Kovács, Csaba Legány, Attila Babos: *Cluster Validity Measurement Techniques*,Department of Automation and Applied Informatics, Budapest University of Technology and Economics

[7] Christopher D. Manning, Prabhakar Raghavan and Hinrich Schütze: *Introduction to Information Retrieval*, Cambridge University Press, 2008 (http://www-csli.stanford.edu/ hinrich/information-retrieval-book.html)

[8] Arthur L. Hsu, Sen-Lin Tang, Saman K. Halgamuge: *An unsupervised hierarchical dynamic self-organizing approach to cancer class discovery and marker gene identification in microarray data*, University of Melbourne, Australia. 2003, Bioinformatics 19(16): pp 2131-2140

[9] T. Kohonen, J. Hynninen, J. Kangas, J. Laaksonen, K. Torkkola: *The Learning Vector Quantization Program Package*, Helsinki University of Technology, 1995

[10] R.O. Duda, P.E. Hart, D.G. Stork: *Pattern Classification* , John Wiley & sons, Inc. 2001, ISBN 0-471-05669-3

[11] T. Kohonen: *Self-Organizing Maps*, 3rd ed. Springer 2001, ISBN 3-540-67921-9

[12] : Yu Zheng Zhai, Arthur Hsu, and Saman K. Halgamuge *Scalable Dynamic Self-Organising Maps for Mining Massive Textual Data*,Lecture Notes in Computer Science, 4234/2006, p. 260-267, Springer Berlin / Heidelberg

[13] Anne Lorette, Xavier Descombes, Josiane Zerubia : *Fully Unsupervised Fuzzy Clustering with Entropy Criterion*, 15th International Conference on Pattern Recognition (ICPR'00) - Volume 3 p. 3998

[14] Various: *Wikipedia*, http://wikipedia.org