UNIVERSITATEA ”ALEXANDRU-IOAN CUZA” DIN IAȘI

FACULTATEA DE INFORMATICĂ

LUCRARE DE LICENȚĂ

UFL: Unsupervised feature learning using (α, β)-$Lloyds$++ algorithm

propusă de

Mihai-Ștefan Strugari

Sesiunea: iulie, 2019

Coordonator științific

Conf. Dr. Liviu Ciortuz

UNIVERSITATEA ”ALEXANDRU-IOAN CUZA” DIN IAȘI

FACULTATEA DE INFORMATICĂ

UFL: Unsupervised feature learning using (α, β)-$Lloyds$++ algorithm

Mihai-Ștefan Strugari

Sesiunea: iulie, 2019

Coordonator științific

Conf. Dr. Liviu Ciortuz

Avizat,

Îndrumător lucrare de licență,

Conf. Dr. Liviu Ciortuz.

Data: ..... Semnătura: .....

Declarație privind originalitatea conținutului lucrării de licență

Subsemnatul **Strugari Mihai-Ștefan** domiciliat în **România, jud. Suceava, mun. Suceava, blvd. G. Tudoraș, nr. 21, bl. B6, sc. A, et. 4, ap. 19**, născut la data de **31 octombrie 1997**, identificat prin CNP **1971031330231**, absolvent al Facultății de informatică, **Facultatea de informatică** specializarea **informatică**, promoția 2019, declar pe propria răspundere cunoscând consecințele falsului în declarații în sensul art. 326 din Noul Cod Penal și dispozițiile Legii Educației Naționale nr. 1/2011 art. 143 al. 4 și 5 referitoare la plagiat, că lucrarea de licență cu titlul **UFL: Unsupervised feature learning using** (α, β)**-**$Lloyds$**++ algorithm** elaborată sub îndrumarea domnului **Conf. Dr. Liviu Ciortuz**, pe care urmează să o susțin în fața comisiei este originală, îmi aparține și îmi asum conținutul său în întregime.

De asemenea, declar că sunt de acord ca lucrarea mea de licență să fie verificată prin orice modalitate legală pentru confirmarea originalității, consimțind inclusiv la introducerea conținutului ei într-o bază de date în acest scop.

Am luat la cunoștință despre faptul că este interzisă comercializarea de lucrări științifice în vederea facilitării falsificării de către cumpărător a calității de autor al unei lucrări de licență, de diplomă sau de disertație și în acest sens, declar pe proprie răspundere că lucrarea de față nu a fost copiată ci reprezintă rodul cercetării pe care am întreprins-o.

Data: ..... Semnătura: .....

Declarație de consimțământ

Prin prezenta declar că sunt de acord ca lucrarea de licență cu titlul **UFL: Unsupervised feature learning using** (α, β)**-**$Lloyds$**++ algorithm**, codul sursă al programelor și celelalte conținuturi (grafice, multimedia, date de test, etc.) care însoțesc această lucrare să fie utilizate în cadrul Facultății de informatică.

De asemenea, sunt de acord ca Facultatea de informatică de la Universitatea "Alexandru-Ioan Cuza" din Iași, să utilizeze, modifice, reproducă și să distribuie în scopuri necomerciale programele-calculator, format executabil și sursă, realizate de mine în cadrul prezentei lucrări de licență.

Absolvent **Mihai-Ștefan Strugari**

Data: ..... Semnătura: .....

Cuprins

Motivație

Recunoșterea unui obiect într-o imagine nu constituie o problemă prea dificilă pentru noi, oamenii. Ca orice altă problemă, a fost dată înspre rezolvare programelor care, prin algoritmii lor tradiționali, determiniști, nu duc la bun sfârșit acest sarcină, virtualmente imposibilă. Au apărut totuși alte modalități, bazate pe inteligență artificială sau pe, subdomeniul său, învățare automată, care aproximează soluția acestei probleme și reuște să obțină rezultate din ce in ce mai bune pe parcursul trecerii timpului.

Introducere

Acestă lucrare își propune să expună o modalitate prin care se poate construi un *framework*, dedicat implementării unor modele pentru clasificare, rezolvând probleme de învățare automată ca *object recognition* și *computer vision*. Conținutul este împărțit în două capitole. Primul capitol se preocupă de elementul conceptual al temei,iar celălalt pune în evidență dezvoltarea unei interfețe la linia de comandă (*command line interface*), prin care utilizatorul să poată crea propriile modele pentru clasificare, pornind de la *dataset*-uri personalizate.

Ideea a pornit de la articolul *An Analysis of Single-Layer Networks in Unsupervised Feature Learning* [1], în care se prezintă acest framework - cu etape de preprocesare a imaginilor, extragere de atribute (*features*) din imagini și folosirea acestora în constituirea unui model static - toate acestea fiind aplicate pe niște *dataset*-uri de referiță, cum ar fi CIFAR-10, NORB și STL-10. Autorii au evidențiat multe aspecte importante, cu precădere folosirea algoritmului *k-means*, pentru faza de preantrenare, care a obținut cele mai bune rezultate experimentale dintre toți algoritmii utilizați.

Urmează încercarea de reproducere a rezultatelor obținute de aceștia, specifice algoritmului *k-means*, și aplicarea unei versiuni generalizate a algoritmului, propuse în articolul *Data-Driven Clustering via Parameterized Lloyd's Families* [2].

Elementul de noutate constă în parametrizarea lui *k-means*, prin parametrii care influențează initilizarea și căutarea locală. Această extindere pune la dispoziție această familie de algoritmi care produc rezultate - clusterizări - diferite, descoperind astfel că pentru o mulțime de date fixată putem obține clusterizări mai bune față de varianta clasică, dacă folosim parametrii potriviți.

Capitolul 1

Conceptualizare

1.1 Formalizarea problemei

Imaginea $i \in D$ poate fi reprezentată digital în forma unui *array* tridimensional, prima dimensiune fiind lățimea w (*width*), a doua dimensiune fiind înălțimea h (*height*) și a treia dimensiune fiind dată de numărul de canale d (*channels*) prin care se reprezintă un pixel. În general un pixel este reprezentat prin 3 canale, fiecare canal $d_i \in \{0, 1, ..., 256\}$ cu $i \in \{1, 2, 3\}$ semnificând proporția culorilor RGB, roșu, verde și respectiv albastru, care combinate produc culoarea pixelului.

Învățarea automată rezolvă o serie de sarcini, printre care se află și clasificarea. Din [4], clasificarea presupune ca programul nostru să precizeze în ce categorie se află o intrare specifică. Pentru aceasta, algoritmul de învățare produce o funcție care mapează mulțimea intrărilor la mulțimea categoriilor.

Definim funcția *target* $g\colon D \rightarrow L$, cu L mulțime finită de etichete. Plecând de la o mulțime de imagini $V = \{i^{(1)}, i^{(2)}, ..., i^{(m)}\}$ din D^m și o mulțime de etichete $Y = \{y^{(1)}, y^{(2)}, ..., y^{(m)}\}$ din L^m, astfel încât $g(i^{(i)}) = y^{(i)}, \forall i \in \{1...m\}$, caută acea o ipoteză $h \in H$ care aproximează funcția *target* g. Obținem astfel un sistem inductiv, pe baza căruia putem prezice eticheta imaginii i ca fiind $y = h(i)$.

Algoritmii care rezolvă acest tip de problemă se încadrează în categoria învățării supervizate și au nevoie de o măsură de performanță. Mai specific, în cazul clasificării, putem vorbi de acuratețea modelului, care este determinată prin numărarea instațelor clasificate corect de model. Aceași informație o putem obține dacă calculăm rata erorii, de data aceasta prin numărarea instanțelor etichetate în mod eronat. De obicei, avem de-a face cu două tipuri de erore (sau acurateță), și anume eroare la antrenare și

eroare la validare. Modelul este construit pe baza unei mulțimi de instanțe de antrenare, iar eroarea la antrenare este dată de submulțimea instațelor clasificate greșit după antrenare. Analog se calculează și eroarea la validare, diferența constând în faptul că mulțimea de instanțe nu a luat parte la antrenarea modelui. Mai mult ne interesează eroarea la validare, deoarece vrem ca modelul creat să nu fie influențat (*biased*) prea tare de instațele de antrenare. Așa ne asigurăm că modelul ar putea funcționa bine și pe instanțe pe care nu le-a văzut până acum. Vrem totuși să avem o eroare cât mai mică la antrenare, adică modelul să nu manifeste *underfitting*, dar să avem și o difență mică între cele două erori, adică modelul să nu manifeste fenomenul de *overfitting*.

Utilizând o metodă tradițional specifică învățării automate, ar trebui să ne fixăm mulțimea de atribute de intrare pentru imaginile de clasificat și să lasăm algoritmul să găsească maparea de la aceste atribute la atributul de ieșire (mulțimea de etichete). Problema intervine în modul de reprezentare al unei imagini. Nu putem delimita clar ce atribute trebuie să extragem din imagini.

O soluție la această problemă este recurgerea la *reprezentation learning*, un subdomeniu al învățarii automate care presupune existența unei etape automate, care nu are nevoie de intervetia programatorului, de extragere a reprezentării datelor (Figura 1.1). Din capitolul *Introduction* [4], *deep learning* rezolvă problema centrală a învățarii reprezentării datelor, prin exprimarea reprezentărilor în funcție de alte reprezentări mai simple. În cazul imaginilor, putem vorbi de exprimarea acesteia în funcție de concepte mult mai simple cum ar fi colțuri, contururi, muchii etc.

În articolul [1], se pun în evidență mai mulți algoritmi din această sferă, și anume *sparse auto-encoders*, *sparse restricted Boltzmann machines*, clusterizare *K-means* și mixturi de distribuții gaussiune. Toți acești algoritmi de învățare nesupervizată produc o funcție care poate primi ca input o zonă patrată din imagine și returnează k numere reale care ar trebui să reprezinte acestă zonă introdusă. Această funcție joacă rolul unui extractor de atribute pentru imaginile noastre.

Posibilii pași în construirea unui model complet pentru clasificarea imaginilor [1] sunt următorii:

1. Extrage zone la întâmplare din mulțimea de imagini de antrenare.
2. Aplică o etapă de preprocesare a zonelor extrase.
3. Învață funcția de mapare f, folosind un algoritm de învățare nesupervizată.

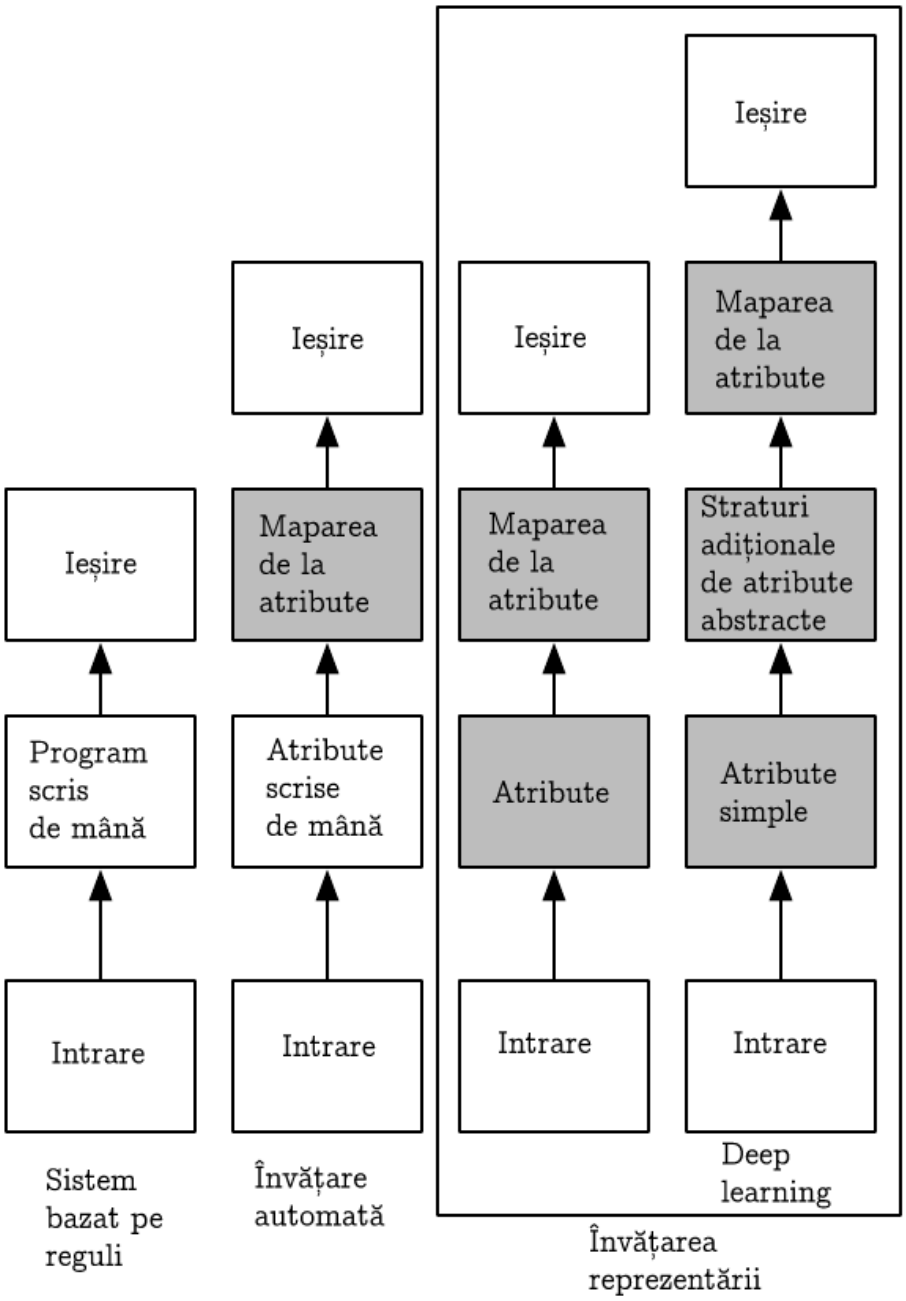


Figura 1.1: O ilustrare a difențelor dintre subdomenii ale inteligenței artificiale. Căsuțele gri reprezintă componente care pot învăța din date. [4]

4. Extrage atributele din zone echidistante care acoperă imaginea.

5. Aplică o etapă de *pooling* prin care să reduci numarul de atribute obținute până acum.

6. Atrenează un clasificator linear care să prezică eticheta unei imagini pe baza vectorilor de atribute.

În secțiunile care urmează vom discuta mai în amănunt chestiuni legate de unele dintre etapele menționate mai sus.

1.2 Preprocesarea

După extragerea în mod aleatoriu a zonelor din imaginile de antrenare, putem apela la niște stagii de preprocesare cum ar fi:

- normalizarea;
- transformarea de tip *whitening* (*sphering transformation*);
- transformarea de tip standardizare (*standardization transformation*).

Aceste etape de preprocesare pot avea un efect semnificativ asupra performanței clasificatorului final [1, 3].

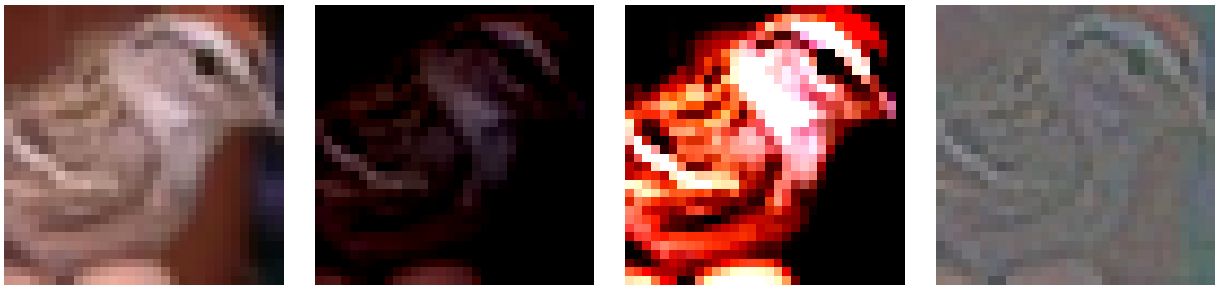
Realizăm normalizarea prin trecerea valorilor pixelilor din $\{0, ..., 256\}$ în valori din $[0, 1]$ și prin scăderea mediei intesității pixelilor tuturor zonelor extrase.

$$\tilde{x} = x - mean(x)$$

Etapa de standardizare [5] se referă la normalizarea *brighness*-ului și a contrastului zonelor extrase. Pentru orice zonă $x^{(i)}$ din mulțimea zonelor $X = \{x^{(1)}, x^{(2)}, ..., x^{(n)}\}$, scădem media intensității pixelilor și împărțim prin deviația standard. Pentru cazurile în care deviația standard ia valoarea 0, scădem un ϵ pozitiv.

$$\tilde{x}^{(i)} = \frac{x^{(i)} - mean(x^{(i)})}{\sqrt{var(x^{(i)}) - \epsilon}}$$

Transformarea de tip *whitening* [6] presupune transformarea mulțimii de zone, a cărei matrici de covariație este cunoscută, într-o mulțime necorelată, ce are variație 1



(a) Imaginea originlă (b) După normalizare (c) După standardizare (d) După *whitening*

Figura 1.2: Efectele aplicării diferitelor tipuri de transformări peste un eșantion de 2000 de instanțe din *dataset*-ul CIFAR10. O imagine din cele eșantionate în toate cele patru contexte.

(cu matricea de covariație egală cu matricea identitate I_n). Una din metole de *whitening* este transformarea ZCA:

$$[V, D] = eig(cov(x))$$

$$\tilde{x}^{(i)} = V(D + \epsilon_{zca} I)^{-\frac{1}{2}} V^T x^{(i)}$$

,unde $[V, D]$ reprezintă descompunerea după valori proprii a matricii de covariație a tuturor zonelor din mulțimea X, ϵ_{zca} este o constantă pozitivă apropiată de 0.

În cazul în care au fost extrase foarte multe zone, este destul de constisitor de folosit transformarea de tip *whitening* întrucât memorarea matricii de covariație este de ordinul lui $\mathcal{O}(n^2)$, iar descompunerea după valori singulare are o complexitate timp de $\mathcal{O}(n^3)$. Dacă nu se observă o difență prea mare în performanța celor două metode, se poate recurge doar la transformarea prin standardizare, care nu influențează corelația dar setează variția 1 și care nu prezintă o complexitate atât de mare din punct de vedere spațiu și timp.

1.3 Extragerea atributelor

Orice zonă patrată este caracterizată de lungimea (*receptive field size*) l și numărul de canaluri d, deci are dimensiunea $N = l \times l \times d$. Definim funcția $f : \mathbb{R}^N \to \mathbb{R}^K$. Asemănător rețelelor neuronale convolutive, pe o imagine de dimensiune $w \times h \times d$, aplicăm funcția f peste toate zonele de dimesiune N, obținând o reprezentare de un singur *layer* de $(w-l+1) \times (h-l+1) \times K$ (Figura 1.3).Acestă reprezentare este la rândul

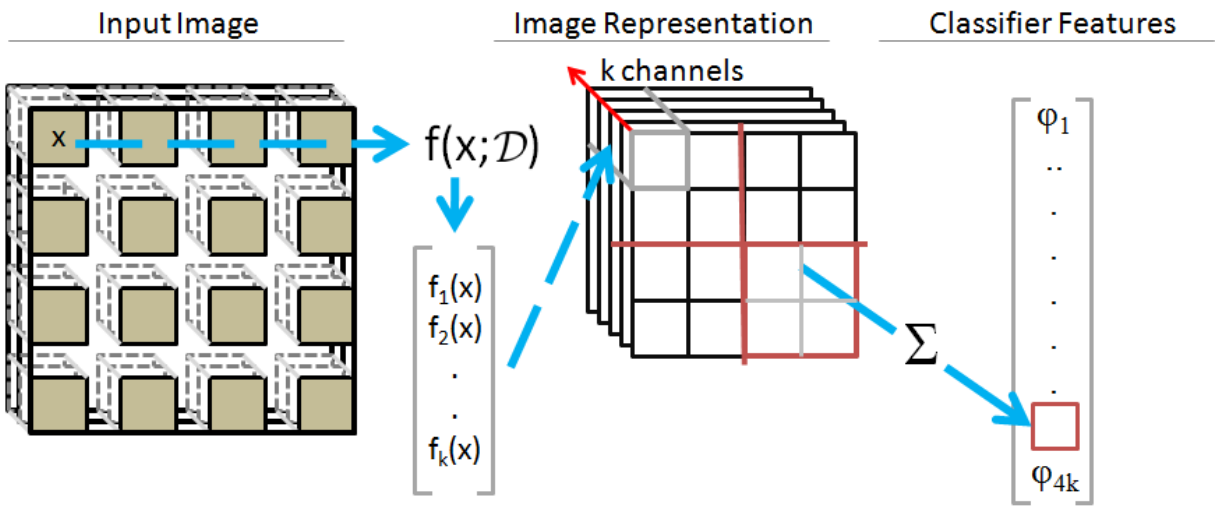


Figura 1.3: O ilustrare a procesului de extragere de atribute din imagine. [3]

ei transformată printr-un proces numit *sum pooling*, proces prin care reducem dimensionalitatea reprezentării prin sumarea elementelor din cele 4 cadrane din secțiunile $(w-l+1)\times(h-l+1)$ ale tuturor celor K *channel*-uri. Reprezentarea, în forma $4\times K$, va servi ca mulțime finală de atribute extrase pentru construirea clasificatorului.

Toate zonele, de dimensiune N, extrase la pasul precedent sunt folosite acum ca input pentru un algoritm de clusterizare, *K-means*. Centroizii obținuți $C=\{c^{(1)},c^{(2)},...,c^{(k)}\}$ pot fi folosiți în construirea funcției de extragere (codare) de atribute f, definită mai sus, în următoarele două moduri:

- *hard*, reprezentare printr-u vector rar, cu mulți de 0. Considerăm doar centroidul cel mai apropiat de zona x.

$$f_i(x)=\begin{cases}1, & \text{dacă } i=\text{argmin}_j||c^{(j)}-x||_2^2\\ 0, & \text{altfel}\end{cases}$$

- *triangle*, reprezentare mai *soft*, prin care fac diferența și ceilalți centroizi, cu precădere cei mai apropiați, în medie, de zona x.

$$f_i(x)=\max\{0,\mu(z)-z_i\},$$

, unde $z_i=||x-c^{(i)}||_2$ și $\mu(z)$ este media elementelor z_i.

Ambele variante au fost propuse în articolul [1], iar varianta *triangle* a dus la producerea celor mai bune rezultate experimentale dintre toți algorimii utilizați, pe

dataset-uri ca CIFAR10 și NORB. În secțiunea care urmează vom discuta despre o varintă extinsă a algoritmului *K-means*, în vederea realizării unor modele cu o mai bună acuratețe decât cele realizate de autori.

1.4 K-means parametrizat

Clusterizarea reprezintă gruparea unor elemente astfel încât elementele din același cluster să fie mai asemănătore (după un anumit criteriu de asemănare) față de elemente din celelalte clustere. *K-means* este unul dintre cei mai populari algorimi care încearcă să rezolve problema clusterizării, problemă care intră în clasa problemelor NP-hard. Acest algoritm poate fi privit ca un algoritm de optimizare, care își propune găsirea unei clusterizări cât mai apropiate de cea optimă. Este un algoritm *greedy*, deci se mulțumește cu găsirea unui optim local, rezultat care este de cele mai multe ori suficient.

Algorithm 1 Calculează o clusterizare cu *K-means*

function KMEANSCLUSTERING(X, k, d)
  Inițializează mulțimea de centroizi C cu k instanțe alese aleator din X.
  $t = 0$
  while $t <$ TMAX sau nu s-a îndeplinit criteriul de convergență **do**
    $C^{(i)} = \emptyset,\ \forall i = \overline{1..k}$
    for x din X **do**
      $i = \text{argmin}_j ||c^{(j)} - x||_2^2$
      $C^{(i)} = C^{(i)} \cup \{x\}$
    end for
    for $i = \overline{1..k}$ **do**
      $c^{(i)} = \mu(C^{(i)})$
    end for
    $t = t + 1$
  end while
  return $\mathcal{C}$, C
end function

Pentru o mulțime de instanțe $X = \{x^{(1)}, x^{(2)}, ..., x^{(n)}\}$, cu $x^{(i)} \in \mathbb{R}^d$, un $k \in \mathbb{N}$, și un criteriu de asemănare dat de metrica de distanță euclidiană, *K-means* returnează

o k-partiționare $\mathcal{C} = \{C^{(1)}, C^{(2)}, ..., C^{(k)}\}$, formată pe baza mulțimii finale de centroizi $C = \{c^{(1)}, c^{(2)}, ..., c^{(k)}\}$ (1). La fiecare iterație t, minimizăm suma distanțelor de la fiecare instanță din clusterul $C_t^{(i)}$ la centroidul $c_t^{(i)}$.

În articolul [2], se studiază o variantă generalizată a algoritmului *K-means*, $(\alpha, \beta) - lloyds++$, o familie de algorimi definită de cei doi parametrii:

- $\alpha \in [0, \infty]$, parametru corespunzător unei proceduri de inițializare a centroizilor.
- $\beta \in [1, \infty]$, parametru care determină funcția obiectiv în procedura căutării locale.

Anumite valori ale parametrilor α și β corespund unor algoritmi standard. Posibilitățile defășurării fazei de inițializare (Figura 1.4) se întind de la inițializarea la intâmplare a centroizilor (pentru $\alpha = 0$) până la algorimul *farthest-first traversal* (pentru $\alpha = \infty$). Și în cazul căutării locale avem de a face cu algorimi cunoscuți ca *K-median* (pentru $\beta = 1$), *K-means* (pentru $\beta = 2$ și *K-centers* (pentru $\beta = \infty$).

Notăm cu $d_i^\alpha = \min_{c \in C} d(x^{(i)}, c)^\alpha$ distanța minimă de la instanța $x^{(i)}$ la mulțimea de centroizi C, iar cu $D_j(\alpha) = \sum_{l=1}^{j} d_l^\alpha$ suma cumulată a acestor distanțe. Funcția p_α are următoarea definiție:

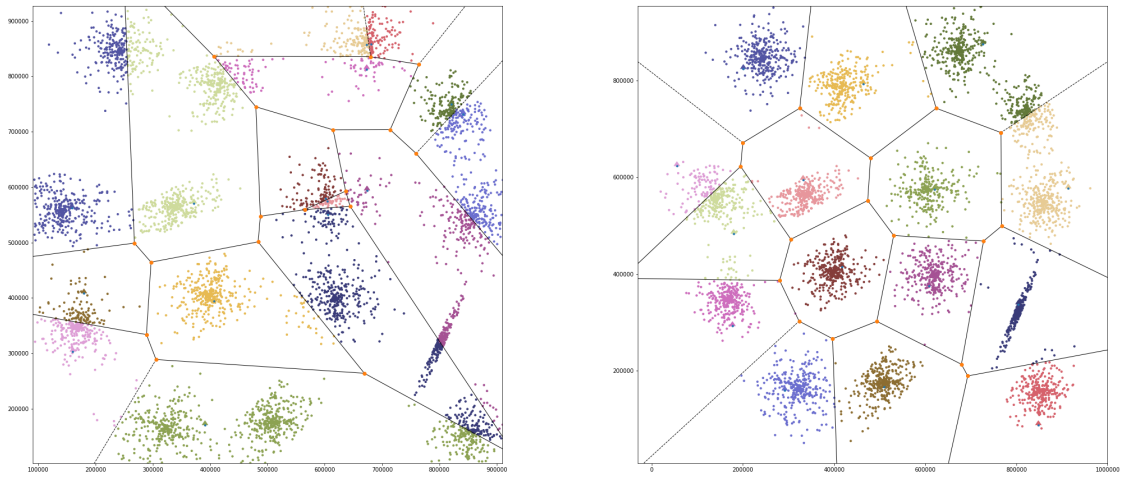
$$p_\alpha \colon X \times 2^X \to [0, 1]$$

,

$$p_\alpha(x^{(i)}, C) = \frac{d_i^\alpha}{D_n(\alpha)}$$

Algorithm 2 Alege centroizii inițiali pentru algoritmul de clusterizarea *K-means*

1: **function** INITIALIZATION(X, k, d, α)
2: $C = \emptyset$.
3: Extrage vectorul $\overrightarrow{Z} \in [0, 1]^k$, aleatoriu și uniform.
4: **for** $t = \overline{1..k}$ **do**
5: Calculăm toate valorile d_i^α.
6: Calculăm sumele cumulate $D_i(\alpha)$.
7: Împărțim $[0, 1]$ în n subintervale $I_{x^{(i)}} = [\frac{D_{i-1}(\alpha)}{D_n(\alpha)}, \frac{D_i(\alpha)}{D_n(\alpha)}]$.
8: Adăugăm la C acea instanță $x^{(i)}$ pentru care $z^{(t)} \in I_{x^{(i)}}$.
9: **end for**
10: return C
11: **end function**



(a) Inițializare la întâmplare

(b) Inițializare prin metoda *farthest-first traversal*

Figura 1.4: Diagramele Voronoi ale unei instanțe de clusterizat din $\mathbb{R}^2$, după aplicarea metodei de inițializare.

Procedura de inițializare (2) este un proces iterativ, începe cu o mulțime vidă C, iar la fiecare iterație $t = \overline{1..k}$ eșantionăm o instanță $x^{(i)}$ din X cu o probabilitate egală cu $p_\alpha(x, C)$ și o adaugăm la mulțimea de centroizi. Cu ajutorul unui vector $\overrightarrow{Z}$, generat aleatoriu dintr-o distriubție uniformă, simulăm selectarea valorilor din variabila aleatoare determinată de funcția de probabilitate p_α. În esență, pe măsură ce folosim valori din ce în ce mai mari ale parametrului α, algoritmul tinde să aleagă drept centroizi inițiali acele instanțe care sunt cât mai puțin similare (din punctul de vedere al metricii de distanță d) una față de cealaltă. Procedeul exprimat prin pașii 5-8 ai funcției de inițializare este referit de către autorii articolului sub denumirea d^α-$sampling$.

Tot în articolul [2], se pune problema existenței unui meta algoritm care are în vedere configurarea dinamică a parametrului α. Date, o instanță de clusterizat $\mathcal{V} = (X, k, d)$, un vector aleatoriu $\overrightarrow{Z}$, o limită superioară α_h a intervalului de căutare și o eroare ϵ , algoritmul face o parcurgere în lățime a arborelui de execuție al procedurii de inițializare (2). Nodurile sunt reprezentate de tuple (C, A) (Figura 1.5 d), fiecare nod are ca noduri fii (C_j, A_j). La fiecare pas al parcurgerii avem de a face cu mulțimea de centroizi C, intervalul A și $t = |C|$, scalar care ar fi reprezentat iterația la care se afla procedura de inițializare. Pentru a determina acele instanțe din X care ar putea fi alese, mai întâi trebuie să ordonăm X descrescător după distanța de la $x^{(i)}$ la cel mai

apropiat centroid din C, iar mai apoi aplicăm d^{α}-$sampling$ pentru fiecare din capetele intervalului A. Așa obținem două seturi de subintervale $\underline{I_{x^{(i)}}}$, corespunzător capătului inferior din A, și $\overline{I_{x^{(i)}}}$, corespunzător capătului superior din A, ambele ordonate descrescător după lungimea intervalelor. Vom lua în considerare ca fiind elemente din U acea secvență de elemente $x^{(i)}$ care începe cu acel $x^{(i)}$ pentru care $z_t \in \overline{I_{x^{(i)}}}$ și se termină cu acel $x^{(i)}$ pentru care $z_t \in \underline{I_{x^{(i)}}}$ (Figura 1.5 c). Pentru fiecare pereche de elemente u_j, u_{j+1} consecutive din secvența obținută U, efectuăm o căutare binară până găsim $\hat{\alpha}$ pentru care capătul superior al Intervalului I_{u_j} rezultat prin $d^{\hat{\alpha}}$-$sampling$ este la distața cel mult ϵ față de z_t. Acest $\hat{\alpha}$ este referit sub denumirea de *breakpoint* de către autorii articolului și reprezintă limita care determină alegerea a doi centroizi diferiți. Acest breakpoint va constitui limită inferioară A_j și limită inferioară pentru A_{j+1}. Pentru cazurile de bază, pentru primul și ultimul element din U, limita superioară este chiar capătul superior al intervalului A ,și respectiv, limita inferioară este capătul inferior al intervalului A.

Algorithm 3 Configurarea dinamică a parametrului α [2]

function DYNAMICCONFIGURATION(X, k, d, $\overrightarrow{Z}$, α_h, ϵ)
    coada Q vidă.
    $Q.push((\{\}, [0, \alpha_h]))$
    while Q este nevidă **do**
        $(C, A) = Q.pop()$.
        $U = [x \in X | x$ poate fi ales drept centroid$]$
        for u_j din U **do**
            Determină subintervalele A_j pentru care u_i este singurul ales.
            $C_j = C \cup \{u_j\}$
            if $|C_j| < k$ **then**
                $Q.push((C_j, A_j))$
            else
                $yield\ (C_j, A_j)$
            end if
        end for
    end while
end function

În final, algoritmul 3 generează toate tuplele de forma (C_j, A_j), unde C_j repre-

zintă o posibilă mulțime de centroizi, unică printre toate celelalte generate, iar A_j reprezintă subintervalul pentru care s-au obținut acești centroizi. Capetele acestor intervale sunt chiar *breakpoint*-urile despre care am discutat în paragraful anterior. Având în vedere că subintervalele sunt disjuncte, reuninea lor produce intervalul de căutare principal $[0, \alpha_h]$, și mai ales că procedura de căutare locală este o componentă deterministă din punct de vedere algoritmic, putem evalua indepent performața fiecărei clusterizări obținute cu inițializarea centroizilor C_j și putem selecta α din subintervalul corespunzător.

În cazul în care dorim să recurgem la o configurare dinamică a parametrului α, vom dezvolta un sistem care determină în mod empiric valoarea potrivită, descris prin următorii pași:

1. Patiționează aleatoriu mulțimea de zone extrase X în m partiții a câte r instanțe.

2. Pentru fiecare partiție aplică algoritmul de configurare dinamică (3) și produ o mulțime de *breakpoint*-uri formată din reuniunea mulțimilor tuturor *breakpoint*-urilor obținute de pe urma aplicării algoritmului.

3. Pentru fiecare α din mulțime și pentru fiecare partiție execută procedura de inițializare. Pornind de la centroizii obținuți execută o etapă de căutare locală.

4. Evaluează performața fiecărei clusterizări pe baza unei funcții de cost sau a unei funcții de *fitness*. Selectează α pentru care am obținut, în medie, cea mai bună performață.

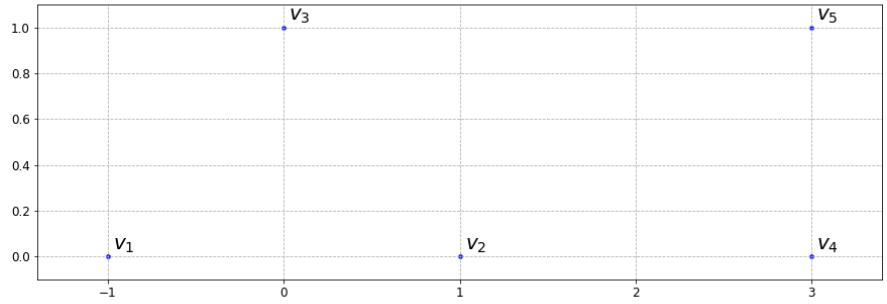
Pentru pasul 4 al sistemului, trebuie să definim o funcție $cost$, care returnează un număr real pozitiv, constituind procedura de evaluare a performanței unei clusterizări. Cea mai evidentă variantă este chiar funcția obicetiv, de minimizat, din familia ℓ_2, a algoritmului de clusterizare *K-means* și poate fi expimată matematic în felul următor:

$$cost(C^{(1)}, ..., C^{(k)}, c^{(1)}, ..., c^{(k)}) = (\sum_{i=1}^{k} \sum_{x \in C^{(i)}} d(x, c^{(i)})^2)^{\frac{1}{2}}$$

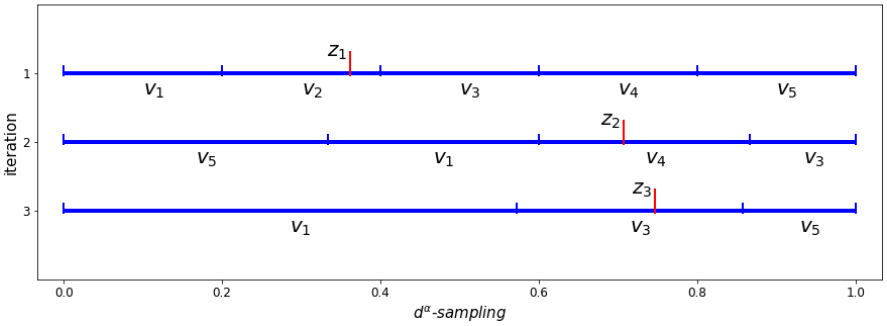
O altă variantă de folosit este indexul *Calinski Harabasz* (CH) [7]. Acest index se obține prin calcularea a două componente:

- *trace* B, urma matricii de împrăștiere interclustere,

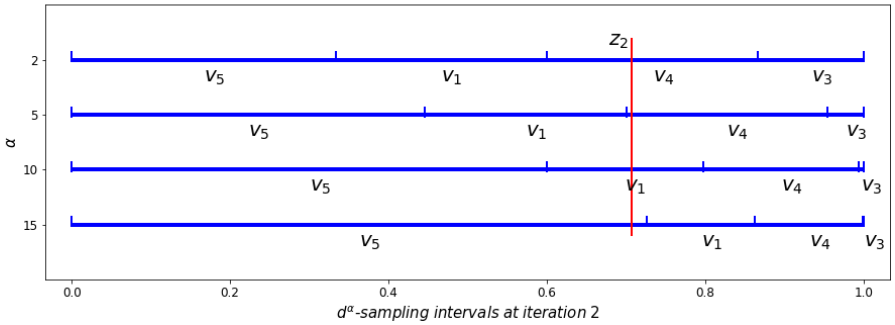
$$\textit{trace } B = \sum_{i=1}^{k} n_i ||c^{(i)} - c||_2^2$$



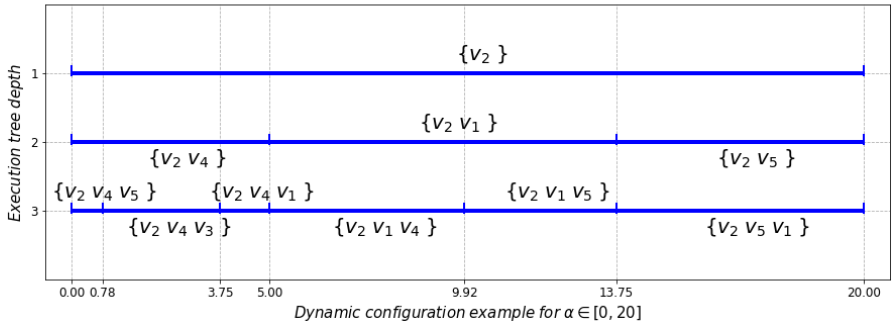
(a) Instanța



(b) Alegerea centroizilor inițiali



(c) Selecția centroizilor diferinți în funcție de valoarea parametrului α la a doua iterație a algoritmului de inițializare



(d) Evidențierea *breakpoint*-urilor create de algortimul de configurare dinamică

Figura 1.5: Exemplu din $\mathbb{R}^2$ care ilustrează etapele importante care țin de inițializarea centroizilor, pentru $n = 5, k = 3$.

- *trace* W, urma matricii de împrăștiere intraclustere,

$$\textit{trace } W = \sum_{i=1}^{k} \sum_{j=1}^{n_i} ||x^{(j)} - c^{(i)}||_2^2$$

unde $n_i = |C^{(i)}|, \forall i = \overline{1..k}$, iar c este centroidul întregii mulțimi de instanțe de clusterizat X. În acest caz, funcția va fi exprimată așa:

$$cost(C^{(1)}, ..., C^{(k)}, c^{(1)}, ..., c^{(k)}) = \frac{[\textit{trace } B/(k-1)]}{[\textit{trace } W/(n-k)]}$$

Costul mai mare indică o clusterizare mai bună. Urmărim ca instanțele din fiecare cluster să fie cât mai coezive - costul este în raport de inversă proporționalitate cu *trace* W - iar clusterele să fie cât mai separate - costul este în raport de directă proporționalitate cu *trace* B.

În această secțiune am studiat varianta parametrizată a algoritmului de clusterizare *K-means*, punând mai mult accent pe parametrul de inițializare 2. Am evidențiat un algoritm de configurare dinamică (3), iar la final am propus un sistem care determină experimental parametrul potrivit.

1.5 Rezultate experimentale

În acestă secțiune pun în evidență performanța unor serii de modele, măsurată prin acuratețe la validare, eșantionate din date reale, a câte 2000 de instanțe pentru antrenare și 400 de instanțe pentru validare. Submulțimile sunt selectate aletoriu din următoarele *dataset*-uri de referință:

- CIFAR10 - conține imagini, cu obiecte, cu dimensiunea de 32×32 pixeli, reprezentați pe 3 *channel*-uri. Fiecare imagine are asignată câte o etichetă din *airplane, car, bird, cat, deer, dog, frog, horse, ship, truck*.

- MNIST - conține imagini, cu simboluri, cu dimensiunea de 28×28 pixeli, reprezentați pe un singur *channel*. Fiecare imagine are asignată câte o etichetă din mulțimea cifrelor arabice $\{0, 1, ..., 9\}$.

- NORB - conține imagini, cu obiecte, cu dimensiunea de 96×96 pixeli, reprezentați pe un singur *channel*. Fiecare imagine are asignată câte o etichetă din *four-legged animal, human figure, airplane, truck, car*.

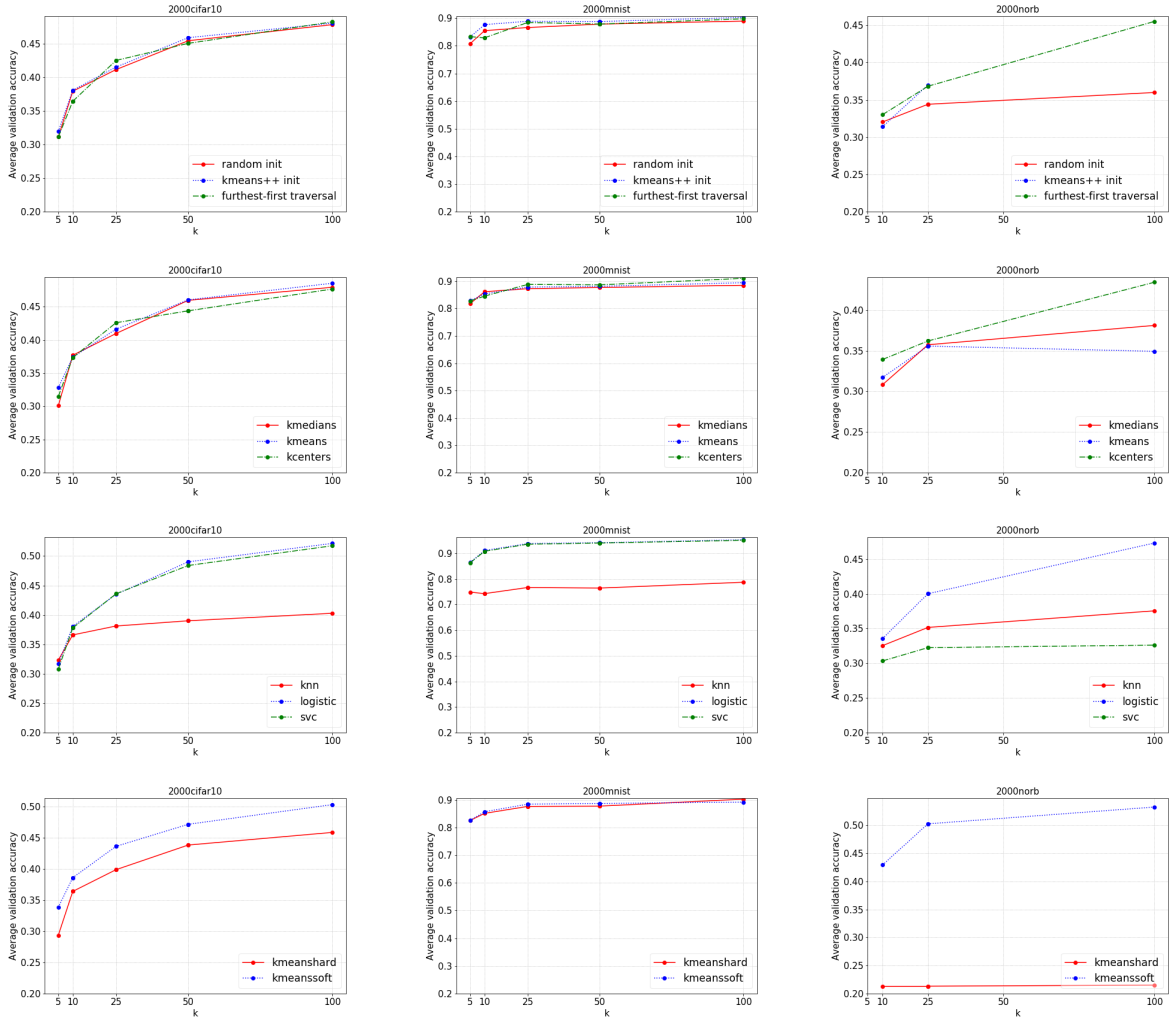


Figura 1.6: Grid care ilustrează acuratețea medie a modelelor, la validare, în funcție de k, pentru fiecare dintre ceilalți patru parametrii (α, β, metoda de activare și algoritmul de clasificare).

Pentru fiecare eșantion am obținut modele în funcție de valorile mai multor parametri, cum ar fi $k \in \{5, 10, 25, 50, 100\}$, $\alpha \in \{0, 2, 20\}$, $\beta \in \{1, 2, 10\}$, funcțiile de activare, *hard* și *triangle*, și strategiile de clasificare, *multiclass support-vector machines*, *logistic regression* și *k-nearest neighbors*.

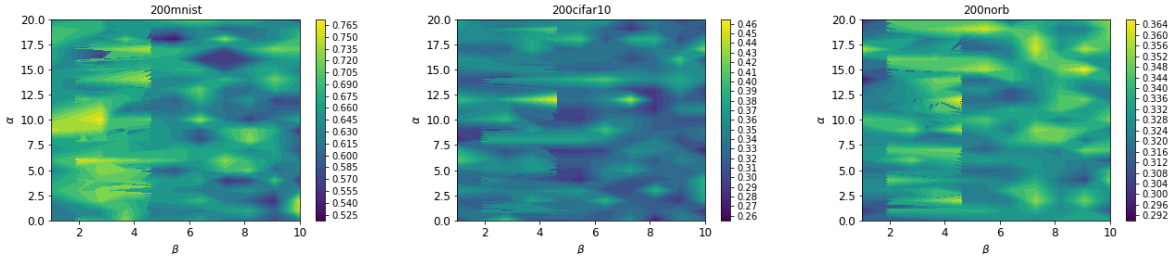
Parametrul k, care determină numărul de atribute prin care sunt reprezentate imaginile, rămâne unul din parametrii care influențează mult performața modelelor. Un alt parametru important este reprezentat de funcția de activare. Varianta neliniară *triangle* duce la obținerea, în medie, a unor rezultate mult mai bune decât varianta *hard* (secțiunea 1.3), cel puțin pentru eșanioanele selectate din CIFAR10 și NORB (Figura 1.6). Din cei trei clasificatori, *kNN* tinde să producă rezultate mai slabe în comparație cu ceilalți doi clasificatori liniari. Creșterea numărului de instațe reprezintă un alt factor

Dataset	alpha	beta	Classifier	Activation	k	Training	Validation
2000mnist	2.0	10.0	svc	kmeanshard	100	99.8%	97.5%
200mnist	20.0	10.0	logisticRegression	kmeanshard	100	100%	95%
2000cifar10	0.0	2.0	svc	kmeanssoft	100	78%	57.25%
200cifar10	20.0	10.0	svc	kmeanshard	50	100%	57.5%
2000norb	20.0	10.0	logisticRegression	kmeanssoft	100	92.45%	71.5%
200norb	20.0	1.0	logisticRegression	kmeanssoft	100	93.5%	60%

Tabela 1.1: Cele mai bune rezultate din punct de vedere al acurateții la validare pentru cele trei *dataset*-uri, pentru eșantioane de 2000 și 200 de instanțe.

semnificativ, duce la o performață mai bună a modelelor, reducând și diferența dintre acuratețea la antrenare și acuratețea la validare.

Pentru cele trei mulțimi de date, pentru un eșantion de 200 de imagini de antrenare și 40 de testare, $k = 5$, am efectuat o discretizare a intervalelor de căutare $\alpha \in [0, 20]$, cu 21 de valori, și $\beta \in [1, 10]$, cu 11 valori, obținând o anumită performață medie în funcție de valorile celor doi parametri (Figura 1.7). Rezultatele arată că parametrii corespunzători etapelor de inițializare și căutare locală influențează într-o oarecare măsură perfomanța sistemului. De exemplu, în cazul MNIST, acuratețea la validarea variază între 52.5% ($\alpha = 17$, $\beta = 7$) și 76.5% ($\alpha = 10$, $\beta = 2$).



(a) MNIST (b) CIFAR10 (c) NORB

Figura 1.7: Performanța unor modele funcție de parametri α și β.

Mărimea imaginilor face ca extragerea atributelor să se desfășoare pe o durată mai îndelungată de timp. În cazul de față, dintre cele trei *dataset*-uri, NORB este cel mai costisitor din punct de vedere al timpului de execuție necesar construirii modelelor derivate din acesta. Pentru același eșantion de 2000 de instanțe prezentat mai sus, am redus dimensiunile imaginilor din de la 96×96 pixeli la 28×28 pixeli și am

construit un model cu aceași parametri (α, β, funcția de activare, numărul de atribute, metoda de clasificare) care au fost folosiți pentru construirea modelului care a obținut cea mai bună performanță pentru eșantionul cu imagini mari (Tabele 1.1). Surprinzător modelul construit manifestă o mai bună performață și reduce diferența dintre erorile la antrenare și testare, înregistând acuratețe la validare de 79.5%, iar la antrenare 90.45%.

Capitolul 2

Implementare

Programul `ufl` reprezintă o interfață la linia de comandă (CLI), pentru platforma *Windows*, prin care utilizatorul poate obține modele pentru clasificarea *dataset*-urilor sale personalizate.

2.1 Funcționalități

Utilizatorul dezvoltă modele folosind de următoarele comenzi:

```
> ufl -c loaddata -xtp /xtrain/path/ -xtep /xtest/path/ -ytp /ytrain/path/
    file.csv -ytep /ytest/path/file.csv -lp /labels/path/file.csv
```

Creează un nou *dataset*. În directoarele `/xtrain/path/` și `/xtest/path/` se află mulțimile de imagini pentru antrenare, respectiv validare. În fișierul `/labels/path/file.csv` se află mulțimea de etichete ca *string*-uri. În fișierele `/ytrain/path/file.csv` și `/ytest/path/file.csv` se află etichetele, exprimate de această dată ca întregi, corespunzătoare imaginilor de antrenare, respectiv testare. Utilizatorul alege un nume pentru *dataset*-ul pe care vrea să-l creeze și dimensiunile unei imagini (lățime și înălțime în pixeli). Toate imaginile, aduse la aceste dimensiuni, împreună cu etichetele corespunzătoare, sunt serializate într-un fișier care este adăugat în subdirectorul `/datasets` din directorul curent.

```
> ufl -c preprocess
```

Extrage un nouă mulțime de zone, din mulțimea imaginilor de antrenare, care urmează să fie folosite pentru faza de preantrenare. Utilizatorul poate selecta un *dataset* creat anterior, alege dimensinile *receptive field size* și *stride*, exprima dorința

în legătură cu aplicarea unei etape intermediare de *whitening* și seta proporția zonelor extrase din totalul zonelor posibile. Este creat un fișier cu zonele extrase din *dataset*-ul ales de utilizator în subdirectorul `/patches` din directorul curent.

```
> ufl -c pretrain
```

Calculează centroizii care urmează să fie folosiți în cadrul etapei de extragere de atribute. Utilizatorul selectează o mulțime de zone obținute anterior, setează numărul de centroizi, setează parametri α și β, corespunzători fazei de inițializare, respectiv fazei de căutare locală, are opțiunea de a configura în mod dinamic parametrul pentru inițializare. Zonele extrase sunt clusterizate prin aplicarea algoritmului *Kmeans* parametrizat, iar centroizii returnați sunt serializați într-un fișier din subdirectorul `/centroids` a directorului curent.

```
> ufl -c extractfeatures
```

Extrage atributele din imaginile dedicate antrenării și testării, pentru un anumit *dataset* și o anumită mulțime de centroizi. Utilizatorul mai alege între cele două funcții de activare posibile: *hard* și *soft*. Atributele corespunzătoare imaginilor pentru antrenare sunt salvate în subdirectorul `/repr_train`, iar cele corespunzătoare celor pentru testare în subdirectorul `/repr_test` a directorului curent.

```
> ufl -c trainmodel
```

Efectuează antrenarea unor modele, pe baza mulțimilor de atribute și a unor algoritmi selectați dintr-o serie de clasificatori liniari, cum ar fi *multiclass support-vector machines*, *logistic regression* și *k-nearest neighbors*. Este creat un obiect de tip `Model`. Acest obiect conține clasificatorul antrenat, care a obținut cea mai bună acuratețe la *cross*-validare, împreună cu funcția de activare, și va fi folosit pentru a prezice etichetele altor imagini. Modelul va fi serializat într-un fișier din subdirectorul `/models` a directorului curent.

```
> ufl -c predict -p /to/predict/path/
```

Utilizatorul alege un model dintre cele create anterior, iar modelul face o predicție despre etichetele tuturor imaginilor din directorul `/to/predict/path/`

```
> ufl -c mnist
> ufl -c cifar10
```

Crearea unui *dataset* care consistă în imagini eșantionate aleatoriu din CIFAR10 sau MNIST. Utilizatorul selectează dimensiunea mulțimii pentru antrenare și dimensiunea mulțimii pentru validare. `/to/predict/path/`

```
> ufl -c getbyaccuracy
```

Pentru un *dataset* selectat, comanda produce afisarea modelor asociate in ordinea descrescătoare a acurateții la validare și antrenare.

```
> ufl -c remove
```

Pentru un *dataset* selectat, comanda produce ștergerea *dataset*-ului împreună cu toate elementele stocate, ce țin de zone extrase, centroizi, atribute extrase, modele, asociate acestuia.

2.2 Modulele aplicației

Programul este împărțit în mai multe module:

- `utils.py` - modul care conține metode utilitare, folosite în cadrul multor alte module. Metoda care se ocupă cu persitența datelor poate salva și încărca date în formatul *npz*. Salvarea presupune serializarea unui dicționar ce conține date, împreună cu o listă de argumente adiționale care ar putea avea legătură cu datele principale. Fișerul creat este situat într-un subdirector, a cărei cale e dată ca argument, și este identificat în mod unic prin numele acestuia. Încărcarea se face tot pe baza unei căi de director și *basename* de fișier. O altă metodă produce schimbarea formei în care sunt reprezentate imaginile. O colecție de imagini poate fi reprezentată ca un tensor *numpy* 4D, fiecare imagine cu lățime, înălțime și *channel*-uri. În unele cazuri avem nevoie de o reprezentare pe doar două dimensiuni și deci trebuie să transformăm ultimele 3 dimensiuni într-o singură dimensiune. Acest proces poartă denumirea de *flattening*.

- `load_images.py` - modul pentru încărcarea *dataset*-urilor. Conține metode de citire a imaginilor dintr-un director, de redimensionare a imaginilor, de citire a fișirelor, în formatul *csv*, ce conțin etichetele imaginilor.

- `d_sampling.py` - modul care conține metodele dedicate implementării algoritmului de eșantionare descris în secțiunea 1.4.

- `ab_lloyds.py` - modul care conține o implementare a variantei generalizate a algoritmului *Kmeans* - procedura de inițializare și procedura care se preocupă de căutarea locală.

- `dynamic_configuration.py` - modul care implementeză algoritmul pe care se bazează sistemul de configurare dinamică a parametrului corespunzător procedurii de inițializare.

- `preprocessing.py` - modul care conține metode ce țin de etapa de extragere de zone din imagini, zone care sunt mai apoi folosite la faza de preantrenare. Zonele extrase trec acum printr-o etapă opțională de preprocesare, costând în aplicarea unor algoritmi descriși în secțiunea 1.2.

- `pretraining.py` - modul joacă rol de *wrapper* pentru metodele din modulul `ab_lloyds.py`, implementează sitemul de configurare dinamică a parametruilui α și conține două metode de evaluare a performației algoritmului de clusterizare, funcția obiectiv de minimizat, caracteristică algoritmului *K-means*, și respectiv indexul CH. Aceste metode au fost descrise la sfârșitul secțiunii 1.4.

- `feature_learner.py` - modul ce implementează strategiile de codare a zonelor din interiorul imaginilor - modurile *hard* și *triangle*, descrise la sfârșitul secțiunii 1.3.

- `feature_extraction.py` - modul care conține metodele prin care sunt extrase atributele din imagini. Un element de noutate ar fi paralelizarea acestei sarcini:

```
class FeatureExtractor:

    def __call__(self, images, feature_learner, k,
        receptive_field_size, stride):
         self.feature_learner = feature_learner
         self.k = k
         self.receptive_field_size = receptive_field_size
         self.stride = stride

         processes = []
         n = len(images)
```

```
N = int(os.environ['NUMBER_OF_PROCESSORS'])
manager = Manager()
return_dict = manager.dict()

for i in range(N):
    p = Process(target = self.__get_images_reprezentation__,
        args = (images[i*n//N:(i+1)*n//N], i, return_dict))
    processes.append(p)
    p.start()

results = []
for i in range(N):
    processes[i].join()
    results.append(return_dict[i])

images_reprezentation = np.concatenate(tuple(results), axis =
    0)
return images_reprezentation
```

Mulțimea imaginilor este împărțită în N partiții, corespunzător numărului de *core*-uri virtuale existente pe mașina de pe care se execută programul. Sunt create N procese, fiecare proces ocupându-se de extragerea atributelor - după cum este descris în secțiunea 1.3 - din partiția sa. În procesul părinte sunt așteptați vectorii de rezultate ale tuturor proceselor aflate în execuție, iar la final este returnată concatenarea acestora. Această construcție sporește viteza de procesare a imaginilor, față de varianta clasică în care întreaga mulțime de imagini este tratată în cadrul aceluiași proces.

- `classifer.py` - modul care conține obiecte ce implementeză interfața `IClassifier`, cu o metodă care primește reprezentările imaginilor și etichetele, câte un set pentru antrenare și validare, construiește un model pentru clasificare și returnează acuratețea modelului, și o altă metodă care primește o mulțime de reprezentări ale imaginilor și returnează mulțimea corespunzătoare de etichete, predicția modelului creat anterior. Acest șablon facilitează adăugarea altor strategii fără a avea de-a face cu prea multe modificări. Până acum sunt construite trei tipuri de clasificatori, și anume *multiclass support-vector machines*, *logistic regression* și *k-nearest neighbors*. Acești clasificatori folosesc pe implmentările de clasificatori din biblioteca *sklearn*.

- `model.py` - modulul conține clasa `Model`. Obiectele de acest tip încapsulează toate datele ce țin de etapele de preprocesare, preantrenare, extragere de atribute și clasificare, care au dus la obținerea acestui model, în plus referințe către obiecte de tip `IFeatureLearner` și `IClassifier`, care vor fi folosite pentru obținerea reprezentării imaginlor, și respectiv utilizarea acestor reprezentări pentru prezicerea etichetelor.

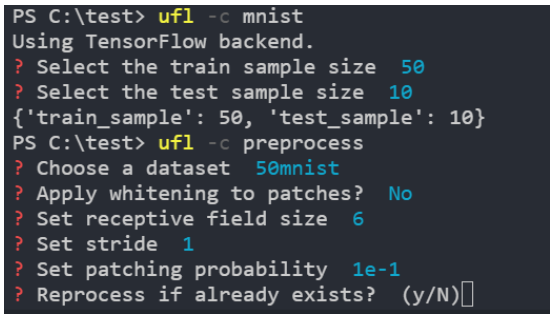
```
class Model:
    def __init__(self, classifier, feature_learner,
       train_reprezentation, test_reprezentation, x_train, y_train,
       x_test, y_test, k, receptive_field_size, stride, labels):
        self.classifier = classifier
        self.feature_learner = feature_learner
        self.train_reprezentation = train_reprezentation
        self.test_reprezentation = test_reprezentation
        self.x_train = x_train
        self.y_train = y_train\textit{sklearn}
        self.y_test = y_test
        self.x_test = x_test
        self.receptive_field_size = receptive_field_size
        self.stride = stride\textit{sklearn}
        self.k = k
        self.labels = labels

    def predict(self, images):
        img_repr = self.__get_images_reprezentation__(images)
        pred = []
        for i in range(len(img_repr)):
            pred.append(self.classifier.predict(img_repr[i:i+1]))
        return pred
    ...
```

Toate modelele sunt serializate în fișiere din directorul `/models`, iar mai apoi pot fi folosite programatic, în limbajul *Python*, pentru a prezice etichetele unei mulțimi de imagini neetichetate. Codul de mai jos reprezintă un exemplu de funcție care face acest lucru:

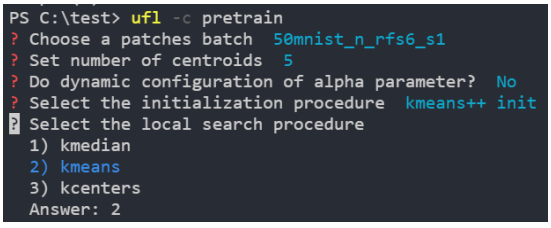
```
def load_model(model_path, images_dir_path):
    f = np.load(model_path)
    try:
        model = f['data'].item()
```

```
except ValueError:
    model = f['data']
arguments = f['arguments'].item()

print ("Acuratete la antrenare: ", model.classifer.train_score)
print ("Acuratete la validare: ", model.classifer.accuracy)

images = []
for image_path in os.listdir(images_dir_path):
    if image_path.lower().endswith(('.png', '.jpg', '.jpeg')):
        img = imread(os.path.join(dirpath, image_path) , as_gray=
            False)
        images.append(img)
images = np.array(images)

y_pred = model.predict(images)
predictions = []
for i, _ in enumerate(images):
    predictions.append(model.labels[y_pred[i]])

return predictions
```

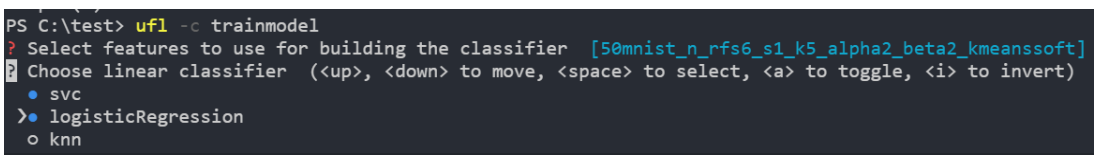
- `ufl.py` - modulul are rol de punct de intrare, prin care utilizatorul interacționează cu aplicația.



(a) Preprocesare



(b) Extragerea centroizilor



(c) Antrenarea modelului pentru clasificare

Figura 2.1: Exemple de interacțiune cu programul `ufl`.

Utilizează bibliotecile *click*, pentru parsarea argumentelelor de la linia de comandă, și *PyInquirer*, pentru chestionarea utilizatorului în comformitate cu comanda executată de acesta la un moment dat. Raspunzând la întrebări, utilizatorul setează parametrii sau își exprimă alegerea folosirii doar a unor anumiți algoritmi specifici.

Concluzii

Aplicația implementează *framework*-ul propus în [1], generează o infinitate opțiuni noi pentru etapa de preantrenare prin folosirea unei noi versiuni de *K-means* și oferă utilizatorului posibilitatea de a antrena modele pentru clasificarea imaginilor, fără să cunoască detaliile construirii unui astfel de model, neavând nevoie de cunoștințe de învățare automată. Aceste modele pot fi integrate programatic de către utilizator în propriile sisteme dezvoltate de acesta. Eventualele îmbunătățiri ar putea consta în:

- expunerea posibilității de dezvoltarea a unor modele dinamice, care să se poată actualiza, continuu, în intervale relativ scurte de timp, în contrast cu modele statice care odată construite nu mai pot fi modificate.
- introducerea posibilității execuției în paralel și în cazul algoritmului *K-means* [8], sporind viteza de calcul a centroizilor în cadrul etapei de preantrenare.
- refactorizarea sistemului, cu precădere în cazul infrastructurii datelor generate. Datele să nu fie reprezentate doar ca niște simple fișiere, relațiile dintre fișiere să fie mai clare, iar utilizatorul să nu dețină acces complet la datele obținute în etapele intermediare, apelând de data aceasta la stocarea acestora în baze de date.

Din punct de vedere experimental, nu am reușit să determin o îmbunătățire semnificativă la trecerea de la algoritmul *k-means* clasic la algoritmul extins (α, β)-$Lloyds$++ [2], când vine vorba de extragerea de atribute pentru modelele de clasificare a imaginilor. Numărul mare de atribute rămâne unul din cele mai importante aspecte care dermină creșterea performnței modelelor și, din păcate, face ca algoritmul de configurare dinamică să fie mult prea costisitor în raport cu beneficiile aduse.

Bibliografie

[1] Adam Coates, Honglak Lee, and Andrew Y. Ng.
An Analysis of Single-Layer Networks in Unsupervised Feature Learning. 14th International Conference on Artificial Intelligence and Statistics (AISTATS) 2011, Fort Lauderdale, FL, USA. Volume 15 of JMLR: WCP 15. Copyright 2011 by the authors.

[2] Maria-Florina Balcan, Travis Dick, and Colin White.
Data-Driven Clustering via Parameterized Lloyd's Families. Advances in Neural Information Processing Systems 31 (NIPS 2018).

[3] Adam Coates, and Andrew Y. Ng.
Learning Feature Representations with K-means. Montavon G., Orr G.B., Müller KR. (eds) Neural Networks: Tricks of the Trade. Lecture Notes in Computer Science, vol 7700. Springer, Berlin, Heidelberg. 2012.

[4] Ian Goodfellow, Yoshua Bengio, and Aaron Courville.
Deep Learning. An MIT Press book, 2016.

[5] Sergey Ioffe and Christian Szegedy
Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. 2015.

[6] Agnan Kessy, Alex Lewin, and Korbinian Strimmer
Optimal Whitening and Decorrelation. The American Statistician, 72:4, 309-314. 2015.

[7] Ujjwal Maulik, and Sanghamitra Bandyopadhyay
Performance Evaluation of Some Clustering Algorithms and Validity Indices. IEEE Transactions on Pattern Analysis and Machine Intelligence (Volume: 24 , Issue: 12) 2002.

[8] Shikai Jin, Yuxuan Cui, and Chunli Yu
A New Parallelization Method for K-means. 2016.