

UNIVERSITATEA "LUCIAN BLAGA" DIN SIBIU  
FACULTATEA DE INGINERIE  
DEPARTAMENTUL DE CALCULATOARE ȘI INGINERIE ELECTRICĂ

# PROIECT DE DIPLOMĂ

Conducător științific:

Conf. dr. ing. Morariu Daniel

Absolvent: Burghelea Zaharia

Specializarea: Calculatoare

- Sibiu, 2021 -

UNIVERSITATEA "LUCIAN BLAGA" DIN SIBIU  
FACULTATEA DE INGINERIE  
DEPARTAMENTUL DE CALCULATOARE ȘI INGINERIE ELECTRICĂ

# **Studiu comparativ între diferiți algoritmi de clasificare**

Conducător științific:

Conf. dr. ing. Morariu Daniel

Absolvent: Burghelea Zaharia

Specializarea: Calculatoare

- Sibiu, 2021 -

# Cuprins

<b>1</b>	<b>Prezentarea temei</b>	<b>3</b>
<b>2</b>	<b>Considerații teoretice</b>	<b>5</b>
2.1	Algoritmi de clasificare . . . . .	5
2.1.1	Introducere . . . . .	5
2.1.2	Notatii matematice . . . . .	6
2.2	Rocchio . . . . .	8
2.3	Support Vector Machines . . . . .	10
2.3.1	Introducere . . . . .	10
2.3.2	Determinarea ecuației hiperplanului . . . . .	10
2.3.3	Trucul nucleu . . . . .	14
2.3.4	Introducerea unei măști de eroare . . . . .	15
2.3.5	Clasificarea mai multor clase . . . . .	17
2.3.6	Algoritmul Sequential Minimal Optimization . . . . .	18
2.4	Evaluarea algoritmilor de clasificare . . . . .	23
2.4.1	Împărțirea setului de date . . . . .	23
2.4.2	Metrici pentru evaluarea algoritmilor de clasificare . . . . .	23
<b>3</b>	<b>Rezolvarea temei de proiect</b>	<b>26</b>
3.1	Introducere . . . . .	26

3.2	Limbajul de programare ales . . . . .	27
3.2.1	De ce Racket? . . . . .	27
3.2.2	Scurt tutorial pentru Racket . . . . .	28
3.3	Arhitectura aplicației . . . . .	33
3.4	Citirea datelor din fișiere . . . . .	34
3.4.1	Setul de date generat aleator . . . . .	34
3.4.2	Setul de date extra pe baza unor documente Reuters . . . . .	37
3.5	Implementarea algoritmului Rocchio . . . . .	39
3.6	Îmbinarea algoritmului Rocchio cu SVM . . . . .	41
3.7	Implementarea algoritmului SVM . . . . .	46
3.8	Alegerea parametrilor pentru algoritmul SVM . . . . .	51
3.9	Implementarea evaluării algoritmilor . . . . .	53
3.10	Utilizarea aplicației . . . . .	54
3.11	Rezultate experimentale . . . . .	56
<b>4</b>	<b>Concluzii și dezvoltări ulterioare</b>	<b>59</b>
<b>5</b>	<b>Bibliografie</b>	<b>61</b>

# Capitolul 1

## Prezentarea temei

Un algoritm de clasificare are ca scop împărțirea în diferite categorii sau clase a unui set de date, sau simplu găsirea unei clase pentru un nou input după ce a învățat o funcție de decizie pe baza unor exemple deja clasificate anterior, în acest caz învățarea fiind supervizată.

Învățarea supervizată este o ramură a învățării automate ce a prins o popularitate uriașă în perioada recentă, în special datorită creșterii enorme a datelor și nevoii de automatizare în multe situații precum clasificarea de documente, filtrarea spam-urilor, împărțirea pe categorii a imaginilor și multe altele.

Lucrarea de față își propune înțelegerea și implementarea unor algoritmi de clasificare, iar mai apoi evaluarea și compararea lor. Există mulți algoritmi de clasificare la ora actuală, însă algoritmul Support Vector Machines iese imediat în evidență datorită particularitățile sale deosebite de a clasifica mapând datele într-o altă dimensiune, superioară față de cea originală.

Așadar această temă are ca scop principal implementarea algoritmului Support Vector Machines, dar mai ales de a înțelege cum funcționează clasificarea într-un spațiu de reprezentare diferit față de cel în care se află setul de date inițial.

Tot în acest proiect se va implementa și algoritmul Rocchio cu scopul de a putea studia compariv algoritmul Support Vector Machines. Iar mai apoi pentru a putea obține o intuiție mai bună despre cum funcționează vizual acești algoritmi, se dorește și implementarea unei interfețe grafice.

Este de dorit ca evaluarea acestor algoritmi să se facă pe un set de date cu aplicații practice în lumea reală și din acest motiv pentru testare se va utiliza un set de date extras pe baza unor documente obținute de pe urma știrilor publicate de către presa Reuters în 1996 timp de un an.

La ora actuală există multe librării ce au în componența lor algoritmul Support Vector Machines și mulți alți algoritmi, însă în tema de față se va dezvolta totul de la zero fără a utiliza librării sau framework-uri externe, chiar dacă aplicația rezultată va fi mai puțin performantă decât cele existente deja. Scopul fiind evident înțelegerea algoritmilor în detaliu nu doar folosirea lor.

Datorită faptului că aplicația nu este constrânsă de nimic fiind implementată de la zero, se va folosi această oportunitate pentru a încerca învățarea unei alte paradigme de programare, diferită de cele existente deja (precum cea procedurală sau orientată pe obiecte) și anume programarea funcțională. Pentru a facilita acest lucru din urmă, aplicația va fi dezvoltată integral folosind limbajul de programare Racket, un limbaj derivat din Lisp fiind și primul limbaj de programare care a introdus programarea funcțională (scopul fiind doar de familiarizare cu această paradigmă).

## Capitolul 2

# Considerații teoretice

### 2.1 Algoritmi de clasificare

#### 2.1.1 Introducere

Clasificarea formează o ramură a învățării supervizate, unde plecând de la un set de date în care clasele sunt cunoscute pentru fiecare exemplu se dorește a învăța o funcție ce asignează o clasă sau o categorie unui nou exemplu ce nu a făcut parte din setul de date inițial [1], [2].

Ca o analogie, acest tip de învățare este prezent și la oameni, denumită învățare de concepte, unde spre exemplu aflându-ne într-o anumită situație putem să deducem ce este mai probabil să se întâmple sau alternativ să putem clasifica anumite obiecte ca fiind de un tip sau de un altul bazându-ne pe experiențe anterioare în care deja am aflat urmările la situații similare, respectiv observând trăsături specifice a anumitor obiecte ce fac ca acestea să aparțină unei anumite clase [3].

În practică există numeroase aplicații ale acestor algoritmi. Clasificarea de documente, filtrarea spam-urilor, recunoasterea caracterelor scrise de mână sau clasificarea imaginilor fiind doar unele dintre ele. Există și mulți algoritmi de

clasificare (fiecare cu beneficiile și particularitățile lor) din care se poate alege pentru a putea fi aplicați acestor probleme. Unii dintre cei mai populari fiind: Naive Bayes, Neural Networks, k-Nearest Neighbours, Nearest Centroid Classifier (Rocchio), sau Support Vector Machines. În cele ce urmează (2.2, 2.3) vor fi prezentați în detaliu doi dintre acești algoritmi de clasificare (Rocchio respectiv Support Vector Machines) [1], [4], [12].

### 2.1.2 Notății matematice

Revenind la algoritmi de clasificare, pentru a învăța o funcție de decizie (ce decide cărei clase să aparțină un nou exemplu) este nevoie de un set de date format din mai multe exemple, unde un exemplu este descris matematic de un vector  $n$ -dimensional  $x \in \mathbb{R}^n$ ,  $x = (a_1, a_2, \dots, a_n)$  căruia i se atașează și o „etichetă”  $y$  ce reprezintă clasa exemplului. Această clasă la randul ei poate să fie un vector  $m$ -dimensional  $y \in \mathbb{R}^m$ ,  $y = (b_1, b_2, \dots, b_m)$ , un simplu obiect (un număr spre exemplu), sau chiar o valoare binară  $y \in \{-1, 1\}$ , unde valoarea 1 semnifică apartenența către o clasă, respectiv  $-1$  semnificând că acel exemplu nu aparține clasei în cauză, clasificarea în acest caz fiind una binară, întâlnită la algoritmi precum Support Vector Machines [4].

Vectorii din cadrul exemplelor setului de date sunt formați din numere reale, ponderi, ce reprezintă diferite trăsături ale exemplelor, adeseori însă vectorii sunt normalizați în intervalul  $[-1, 1]$  sau  $[0, 1]$ . Aceste trăsături în practică pot fi o gamă foarte largă de lucruri, spre exemplu frecvența de apariție a unui cuvânt într-un document text, culoarea dintr-o imagine și multe alte trăsături din diferite domenii [5].



Produsul scalar este o operație matematică ce apare adesea în cadrul acestor algoritmi, în special pentru Support Vector Machines. În literatura de specialitate sunt folosite mai multe notații, precum  $\langle x, y \rangle$ ,  $x \cdot y^t$  sau  $x \cdot y$ , ultima variantă fiind și cea care va fi utilizată pe parcursul acestei lucrări. Important este că vectorii  $x$  și  $y$  să aibă aceeași dimensiune, adică  $x, y \in R^n$ ,  $x = (a_1, a_2, \dots, a_n)$  și  $y = (b_1, b_2, \dots, b_n)$  [30].

$$x \cdot y = \sum_{i=1}^n a_i b_i$$

Pentru a calcula similaritatea dintre doi vectori, sunt folosite diferite metrici precum distanța Manhattan, similitudinea (asemănarea) cosinusului sau distanța Euclideană [29]. Ultima fiind folosită îndeosebi în această lucrare în cadrul algoritmului Rocchio pentru a putea decide care exemplu este mai apropiat de un centroid (acest lucru fiind detaliat ulterior). Păstrând notațiile utilizate anterior, adică  $x, y \in R^n$ , distanța Euclideană poate fi scrisă matematic ca:

$$|x - y| = \sqrt{(x - y)^2} = \sqrt{\sum_{i=1}^n (x_i - y_i)^2}$$

Un hiperplan reprezintă un subspațiu cu o dimensiune mai mică decât spațiul din care face parte. Matematic acesta este descris de o ecuație de forma:

$$x \cdot y + b = 0 \Leftrightarrow \sum_{i=1}^n x_i y_i + b = 0$$

Unde  $b$  este un număr real ce reprezintă un bias (prag). Geometric pentru cazul în care  $x, y \in R^2$  (vectorii sunt bidimensionali) acest hiperplan reprezintă o linie ce separă planul dat de cele două coordonate în două, similar în cazul în care vectorii sunt tridimensionali hiperplanul va deveni un plan ce separă spațiul tridimensional din care fac parte vectorii. Acest hiperplan este funcția de decizie pentru Support Vector Machines și este prezent chiar și la Rocchio, chiar dacă nu explicit [12].

## 2.2 Rocchio

Algoritmul Rocchio, cunoscut și sub denumirea de *Nearest centroid classifier* este o metodă de clasificare ce împarte spațiul vectorial în regiuni centrate pe centroizii aferenți fiecărei clase plecând de la un set de date de forma  $\{(x_1, y_1), (x_2, y_2), \dots, (x_N, y_N)\}$ , unde  $x_k \in \mathbb{R}^n$  este al  $k$ -lea exemplu căruia  $i$  se atașează și clasa sa:  $y_k$  [6], [16].

Un centroid este centrul de greutate al exemplurilor dintr-o clasă. Având  $k$  vectori  $x_i, i \in \overline{1, k}$  ce aparțin unei clase  $c$ , centroidul acestei clase poate fi obținut ca media aritmetică a vectorilor, adică:  $\mu_c = \frac{1}{|c|} \sum_{i=1}^k x_i$ , unde aici  $|c|$  semnifică numărul de elemente al clasei  $c$ . Procedeu se repetă pentru fiecare clasă, iar ulterior oricare nouă instanță neclasificată este asignată clasei celei mai similare (sau apropiate). Pentru calculul similarității Rocchio folosește ca metrică distanța Euclideană (dintre un nou exemplu neclasificat și un centroid), iar prin cea mai similară clasă se înțelege clasa a cărei centroid a produs distanța minimă [7].

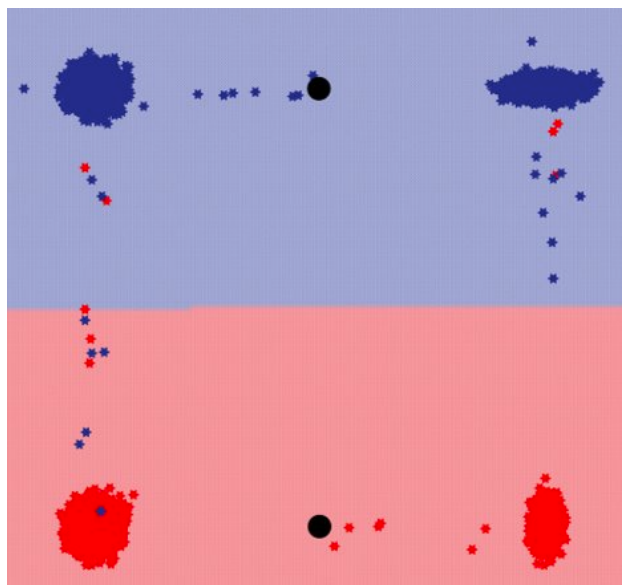


Fig. 2.2.1 Clasificarea Rocchio pentru două clase.

Vizual clasificarea cu algoritmul Rocchio se poate observa mai sus în [Fig. 2.2.1] unde având un set de date (fiecare exemplu este reprezentat ca un punct) aparținând la două clase (clasa albastră în partea de sus, respectiv clasa roșie în partea de jos), se observă că centroizii rezultați (cercurile negre) sunt chiar centrele de greutate al punctelor fiecăror clase.

Granița dintre cele două clase este o linie cand setul de date este bidimensional (ca în figură), iar acest lucru se generalizează și pentru orice dimensiune unde funcția de decizie devine un hiperplan (un subspațiu cu o dimensiune mai mică decât spațiul original). O problemă majoră ce rezultă imediat de pe urma aceasta este că pentru a obține corect centroizii aferenți fiecăror clase este nevoie ca datele să fie liniar separabile.

```

TRAINROCCHIO( $\mathbb{C}, \mathbb{D}$ )
1  for each  $c_j \in \mathbb{C}$ 
2  do  $D_j \leftarrow \{d : \langle d, c_j \rangle \in \mathbb{D}\}$ 
3      $\vec{\mu}_j \leftarrow \frac{1}{|D_j|} \sum_{d \in D_j} \vec{v}(d)$ 
4  return  $\{\vec{\mu}_1, \dots, \vec{\mu}_J\}$ 

APPLYROCCHIO( $\{\vec{\mu}_1, \dots, \vec{\mu}_J\}, d$ )
1  return  $\arg \min_j |\vec{\mu}_j - \vec{v}(d)|$ 

```

Fig. 2.2.2 Pseudocod pentru algoritmul Rocchio.

În [Fig. 2.2.2] sunt prezentate două componente principale ale algoritmului Rocchio scris sub formă de pseudocod, antrenarea (TrainRocchio) care pe baza setului de date ( $D$ ) și a claselor cunoscute ( $C$ ) returnează o listă cu centroizii aferenți fiecărei clase. În partea de testare (ApplyRocchio) este calculată distanța Euclideană dintre o nouă instanță  $d$  neclasificată și fiecare centroid, returnând clasa a cărei distanță a fost minimă [7].

## 2.3 Support Vector Machines

### 2.3.1 Introducere

Support Vector Machines (sau simplu SVM) este un algoritm de clasificare ce a fost introdus de catre Vladimir Vapnik în anii '90 și a început să prindă popularitate de atunci în special datorită faptului că prezintă abilitatea de a clasifica date care nu sunt lineir separabile mapându-le într-un alt spațiu, superior față de cel inițial, unde se poate găsi un hiperplan ce separă clasele [7], [13].

Inițial se pleacă de la un set de date de  $N$  elemente de forma  $\{(x_1, y_1), (x_2, y_2), \dots, (x_N, y_N)\}$ , unde  $x_k \in \mathbb{R}^d$  este un vector  $d$ -dimensional cu valori reale și  $y_k \in \{-1, 1\}$  fiind clasa căruia îi aparține exemplul. Datorită faptului că  $y_k$  poate lua doar două valori clasificarea este una binară, unde algoritmul poate decide doar apartenența sau neapartenența la o clasă. Ulterior se va prezenta și o metodă prin care acest obstacol este depășit prin a permite o clasificare și când este vorba despre mai multe clase, însă în continuare va fi prezentat algoritmul în cazul în care clasificarea este binară descris în mare parte în [9] și [12].

### 2.3.2 Determinarea ecuației hiperplanului

În urma procesului de antrenament, pe baza setului de date, algoritmul produce o funcție de decizie de forma:

$$f(x) = w \cdot x + b = \sum_{i=1}^d x_i w_i + b = 0 \quad (2.3.1)$$

Geometric această funcție reprezintă un hiperplan, iar pe baza acestei funcții instanțele noi ce apar pot fi asiginate uneia dintre cele două clase menționate mai sus:

$$\text{Dacă } f(x_n) \begin{cases} > 0 \\ < 0 \end{cases} \Rightarrow \begin{cases} y_n = 1 \Leftrightarrow x_n \in C_+ \\ y_n = -1 \Leftrightarrow x_n \in C_- \end{cases} \quad (2.3.2)$$

Pentru a reuși să obțină ecuația hiperplanului optim, algoritmul încearcă să determine vectorul  $w$  și bias-ul  $b$  astfel încat hiperplanul sa separe optim instanțele, adică distanța de la hiperplan la cele mai apropiate exemple (denumite și vectori suport) de pe ambele părți să fie maximizată [9], [15].

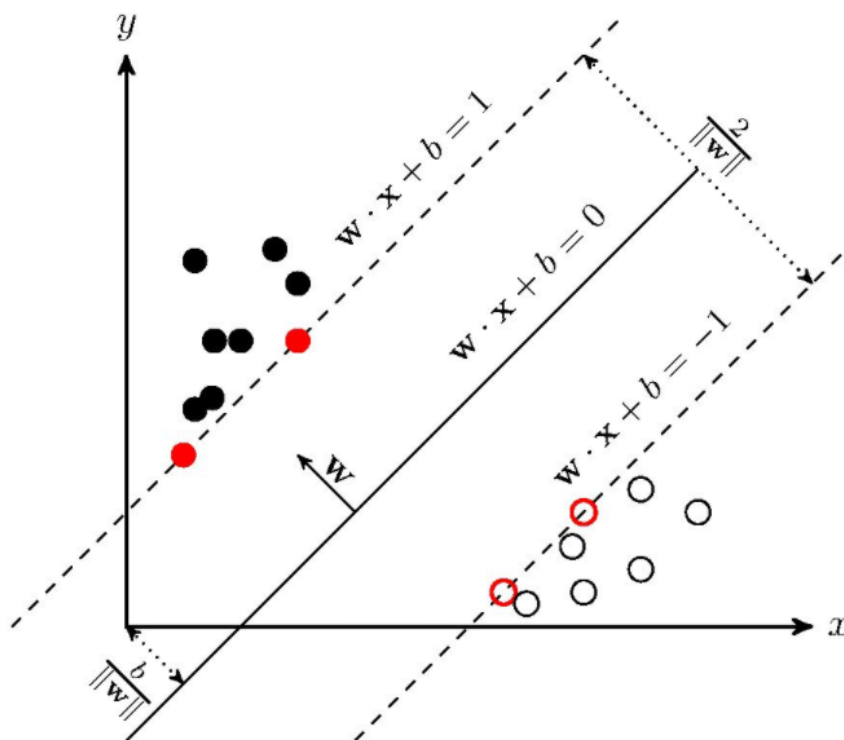


Fig. 2.3.1 Funcția de decizie a algoritmului SVM pentru două clase.

Vectorii suport sunt cei colorați cu roșu în [Fig. 2.3.1], iar matematic distanța ce dorim să o maximizăm este  $\frac{2}{\|w\|}$ , mai exact distanța de la hiperplanul cu clasa pozitivă,  $w \cdot x + b = 1$ , la hiperplanul cu clasa negativă,  $w \cdot x + b = -1$  (numită și marginea hiperplanului) [10], [15].

Pentru a separa corect toate exemplele trebuie ca sistemul de inecuații din

(2.3.2) să fie îndeplinit, iar acesta poate fi rescris după cum urmează:

$$\begin{cases} w \cdot x_n + b \leq 1 \\ w \cdot x_n + b \geq 1 \end{cases} \Rightarrow \begin{cases} y_n = 1 \\ y_n = -1 \end{cases} \Leftrightarrow y_n(w \cdot x + b) \geq 1 \quad (2.3.3)$$

Fără a denatura scopul găsirii acestui maxim, pentru a fi mai convenient matematic, vom încerca să obținem în schimb minimul inversului distanței dintre vectorii suport a celor două clase, adică  $\min \frac{\|w\|}{2} \Leftrightarrow \min \frac{\|w\|^2}{2}$  [10], [22]. Adicional trebuie să ținem cont că toate exemplele din setul de date să fie clasificate corect (adică să aparțină uneia dintre cele două clase sau interpretând geometric: să fie de o parte și de alta a marginilor respectând relația (2.3.3)):

$$\min \frac{\|w\|^2}{2}, y_n(w \cdot x_n + b) \geq 1, n \in \overline{1, N}. \quad (2.3.4)$$

Pentru a găsi acest minim vom introduce funcția Lagrange atașată problemei de mai sus, căreia îi vom încerca să îi găsim optimul, adică:

$$\begin{aligned} & \max_{\alpha} \left( \min_{w, b} \mathcal{L}(w, b, \alpha) \right) \\ & \mathcal{L}(w, b, \alpha) = \frac{1}{2} \|w\|^2 - \sum_{n=1}^N \alpha_n (y_n (w \cdot x_n + b) - 1) \end{aligned} \quad (2.3.5)$$

Unde  $\alpha_n \geq 0, n \in \overline{1, N}$  sunt multiplicatorii Lagrange condiționați să fie pozitivi. Partea de minimizare din (2.3.5) se poate obține din ecuația  $\nabla(\mathcal{L}(w, b, \alpha)) = 0$ , adică prin egalarea cu 0 a derivatelor parțiale în funcție de  $w$  și  $b$  ale Lagrangianului.

$$\frac{\partial \mathcal{L}}{\partial w} = 0 \Leftrightarrow w - \sum_{n=1}^N \alpha_n y_n x_n = 0 \Rightarrow w = \sum_{n=1}^N \alpha_n y_n x_n \quad (2.3.6)$$

$$\frac{\partial \mathcal{L}}{\partial b} = 0 \Leftrightarrow \sum_{n=1}^N \alpha_n y_n = 0 \quad (2.3.7)$$

Înlocuind (2.3.6) și (2.3.7) în (2.3.5) se obține:

$$\mathcal{L}(w, b, \alpha) = \frac{1}{2} \sum_{n=1}^N \alpha_n y_n x_n \sum_{m=1}^N \alpha_m y_m x_m - w \sum_{n=1}^N \alpha_n y_n x_n - b \sum_{n=1}^N \alpha_n y_n + \sum_{n=1}^N \alpha_n$$

$$\begin{aligned}
&= \sum_{n=1}^N \alpha_n - \frac{1}{2} \sum_{n=1}^N \alpha_n y_n x_n \sum_{m=1}^N \alpha_m y_m x_m \\
&= \sum_{n=1}^N \alpha_n - \frac{1}{2} \sum_{n=1}^N \sum_{m=1}^N \alpha_n \alpha_m y_n y_m (x_n \cdot x_m)
\end{aligned} \tag{2.3.8}$$

Mai departe, pentru a rezolva expresia din (2.3.5) mai trebuie doar să maximizăm (2.3.8) în funcție de multiplicatorii Lagrange, ținând cont adțional de (2.3.7):

$$\max_{\alpha} \left( \sum_{n=1}^N \alpha_n - \frac{1}{2} \sum_{n=1}^N \sum_{m=1}^N \alpha_n \alpha_m y_n y_m (x_n \cdot x_m) \right), \quad \sum_{n=1}^N \alpha_n y_n = 0 \tag{2.3.9}$$

Multiplicatorii Lagrange,  $\alpha_n$ , sunt egali cu 0 dacă exemplele din setul de date nu sunt vectori suport, respectiv strict pozitivi în caz că aceste exemple sunt chiar vectori suport, iar ecuația din (2.3.6) devine:

$$w = \sum_{n=1}^N \alpha_n y_n x_n = \sum_{\alpha_n > 0} \alpha_n y_n x_n \tag{2.3.10}$$

Practic datorită acestui lucru, în etapa de testare sau când apare un exemplu nou ce se dorește a fi clasificat nu este necesar a lua în calcul toate exemplele, ci doar acelea a căror multiplicatori Lagrange sunt strict pozitivi (adica vectorii suport). Cunoșcând multiplicatorii Lagrange se poate obține și bias-ul  $b$  de pe urma egalității  $y_n(w \cdot x_n + b) = 1$ , validă pentru orice vector suport:

$$w \cdot x_n + b = y_n \Rightarrow b = y_n - \sum_{\alpha_m \geq 0} \alpha_m y_m (x_n \cdot x_m) \tag{2.3.11}$$

În final orice nouă instanță  $x$  poate fi asignata uneia dintre clase folosind funcția de decizie din (2.3.2) modificată de către (2.3.10) și de (2.3.11).

$$f(x) = w \cdot x + b = \sum_{\alpha_n > 0} \alpha_n y_n (x_n \cdot x) + b \begin{cases} \geq 0 \\ < 0 \end{cases} \Rightarrow \begin{cases} y = 1 \\ y = -1 \end{cases} \tag{2.3.12}$$

### 2.3.3 Trucul nucleu

O problemă ce apare în cazul de mai sus, este că algoritmul converge și reușește să găsească hiperplanul optim numai dacă cele două clase din setul de date sunt liniar separabile (similar ca la algoritmul Rocchio). În acest caz, pentru a putea obține o funcție de decizie ce separă clasele nelinear separabile liniar se introduce așa numitul truc nucleu sau kernel, prin care instanțele din setul de date,  $x_n, n \in \overline{1, N}$  sunt mapate într-o dimensiune superioară [10], [13].

Spre exemplu având un vector de caractere bidimensional  $(a, b)$  acesta poate fi mapat în 3 dimensiuni sub forma  $(a^2, ab, b^2)$ , făcând acest lucru anumite seturi de date nelinear separabile liniar în două dimensiuni ar putea fi separabile în trei dimensiuni, făcând ca algoritmul să convergă.

Pentru acest algoritm însă nu este necesar ca această mapare să aibă loc deoarece instanțele ce apar în (2.3.9) (ce ne ajută să determinăm valorile multiplicatorilor Lagrange), respectiv a celor ce apar în (2.3.12) sunt sub formă de produs scalar:  $(x_n \cdot x_m)$  și se poate folosi o funcție kernel:  $k(x_n, x_m)$  ce primește ca parametrii doi vectori și returnează produsul scalar al celor doi vectori într-o altă dimensiune, fără a mapa explicit fiecare vector, ci doar aplicând funcția kernel asupra rezultatului dat de produsul scalar menționat mai sus (care este un simplu număr, scalar).

Un exemplu de astfel de kernel denumit și kernel-ul Gaussian (unde maparea se face într-o dimensiune de ordin infinit) este:

$$k(x_n, x_m) = e^{-\gamma |x_n - x_m|^2}$$

Unde  $\gamma$  este o variabilă reală pozitivă, ce urmează a fi modificată astfel încât să se găsească valoarea optimă a spațiului unde setul de date poate fi separabil liniar [8], [20]. Pe lângă acesta, adesea este utilizat și kernel-ul polinomial:

$$k(x_n, x_m) = (x_n \cdot x_m + a)^b$$



Adeseori însă se folosește kernel-ul polinomial omogen (cazul în care  $a = 0$ ), iar pentru flexibilitate este introdus și kernel-ul liniar  $k(x_n, x_m) = x_n \cdot x_m$  (cazul  $a = 0, b = 1$ ) ce ne ajută să rescriem mai general relațiile precedente, înlocuind produsul scalar cu funcția kernel. Multiplicatorii Lagrange în acest caz din (2.3.9) vor fi determinați din relația:

$$\max_{\alpha} \left( \sum_{n=1}^N \alpha_n - \frac{1}{2} \sum_{n=1}^N \sum_{m=1}^N \alpha_n \alpha_m y_n y_m k(x_n, x_m) \right), \sum_{n=1}^N \alpha_n y_n = 0 \quad (2.3.13)$$

Bias-ul  $b$ , din (2.3.11) se va calcula folosind:

$$b = y_n - \sum_{\alpha_m \geq 0} \alpha_m y_m k(x_n, x_m) \quad (2.3.14)$$

Iar în final, pentru a asigura o instanță nouă se va folosi o relație similară cu cea din (2.3.12) și anume:

$$f(x) = w \cdot x + b = \sum_{\alpha_n \geq 0} \alpha_n y_n k(x_n, x) + b \begin{cases} > 0 \\ < 0 \end{cases} \Rightarrow \begin{cases} x \in C_+ \\ x \in C_- \end{cases} \quad (2.3.15)$$

Motivul principal pentru care folosirea funcției kernel este foarte fezabilă acestui algoritm este că elementele setului de date apar sub formă de produs scalar peste tot, cum a fost menționat și mai sus, iar în plus există și o flexibilitate deoarece nici nu trebuie cunoscută explicit dimensiunea în care vor fi transformate instanțele setului de date.

## 2.3.4 Introducerea unei mărgi de eroare

Pe lângă problema faptului că algoritmul nu reușește să convergă în cazul în care datele nu sunt liniar separabile, o altă problemă este zgomotul care poate să apară (acele instanțe care seamănă foarte mult cu exemplele din clasa opusă, însă nu fac parte din ea - sau interpretând geometric: punctele dintr-o clasă ce se suprapun pe domeniul clasei opuse). Problema cu acest zgomot este că ar forța

maximizarea marginii dintre exemplele datasetului deși în practică ne dorim ca doar exemplele relevante să influențeze, nu și zgomotul [9], [12].

Pentru ca acest zgomot din setul de date să nu afecteze clasificarea putem ”relaxa” condiția din (2.3.3) introducând niște variabile  $\xi_n, n \in \overline{1, N}$ , ce au scopul de a da ignora cu o anumită măsură exemplele nerelavante:

$$y_n(w \cdot x + b) \geq 1 - \xi_n, n \in \overline{1, N} \quad (2.3.16)$$

Pentru a obține rezultate mai bune trebuie să ținem cont și de aceste variabile de eroare când încercăm să calculăm multiplicatorii Lagrange. Așadar problema din (2.3.4) devine acum:

$$\min \frac{\|w\|^2}{2} + C \sum_{n=1}^N \xi_n, y_n(w \cdot x_n + b) \geq 1 - \xi_n, n \in \overline{1, N}. \quad (2.3.17)$$

Unde  $C$  este un parametru ce trebuie determinat în funcție de cât de mult se dorește a evita zgomotul. Cu cât este mai mică valoarea acestui parametru, cu atât se permite ignorarea mai multor instanțe din setul de date cu scopul de maximiza marginea hiperplanului. În schimb pentru o valoare mai mare a lui  $C$  se încearcă clasificarea corectă pentru cât mai multe puncte, chiar dacă marginea hiperplanului va deveni mai mică. În practică trebuie modificat acest parametru astfel încât sa se ajungă la un optim, lucru ce depinde bineînțeles de setul de date [19], [21].

Relației (2.3.17) îi este atașată și funcția Lagrange de mai jos, (2.3.18):

$$\begin{aligned} & \max_{\alpha} \left( \min_{w, b} \mathcal{L}(w, b, \xi, \alpha, \beta) \right) \\ \mathcal{L}(w, b, \xi, \alpha, \beta) &= \frac{1}{2} \|w\|^2 + C \sum_{n=1}^N \xi_n - \sum_{n=1}^N \alpha_n (y_n (w \cdot x_n + b) + \xi_n - 1) + \sum_{n=1}^N \beta_n \xi_n \end{aligned}$$

Mai departe se procedează similar ca pentru relația din (2.3.5), adică  $\nabla \mathcal{L} = 0$ .

Din aceasta obținem:

$$\frac{\partial \mathcal{L}}{\partial w} = 0 \Rightarrow w = \sum_{n=1}^N \alpha_n y_n x_n \quad (2.3.19)$$

$$\frac{\partial \mathcal{L}}{\partial b} = 0 \Leftrightarrow \sum_{n=1}^N \alpha_n y_n = 0 \quad (2.3.20)$$

$$\frac{\partial \mathcal{L}}{\partial \xi} = 0 \Rightarrow C = \alpha_n + \beta_n \quad (2.3.21)$$

Înlocuind mai departe (2.3.19), (2.3.20) și (2.3.21) în (2.3.18), problema se reduce la a maximiza în funcție de  $\alpha$  funcția:

$$\begin{aligned} \mathcal{L}(\alpha) &= \sum_{n=1}^N \alpha_n - \frac{1}{2} \sum_{n=1}^N \sum_{m=1}^N \alpha_n \alpha_m y_n y_m (x_n \cdot x_m) + C \sum_{n=1}^N \xi_n - \sum_{n=1}^N \alpha_n \xi_n - \sum_{n=1}^N \beta_n \xi_n \\ &= \sum_{n=1}^N \alpha_n - \frac{1}{2} \sum_{n=1}^N \sum_{m=1}^N \alpha_n \alpha_m y_n y_m (x_n \cdot x_m) \end{aligned}$$

Ceea ce este identic cu ce a fost obținut în (2.3.9), însă adițional trebuie să ținem cont ca multiplicatorii Lagrange să fie mărginiți, adică  $\alpha_n \in [0, C]$ .

$$\begin{aligned} \max_{\alpha} \left( \sum_{n=1}^N \alpha_n - \frac{1}{2} \sum_{n=1}^N \sum_{m=1}^N \alpha_n \alpha_m y_n y_m (x_n \cdot x_m) \right) \\ 0 \leq \alpha_n \leq C, \sum_{n=1}^N \alpha_n y_n = 0 \end{aligned} \quad (2.3.22)$$

Se poate modifica și  $(x_n \cdot x_m)$  cu funcția kernel  $k(x_n, x_m)$  în (2.3.22), dacă se dorește folosirea unui kernel neliniar, ținând cont din nou ca multiplicatorii Lagrange să fie mărginiți:

$$\begin{aligned} \max_{\alpha} \left( \sum_{n=1}^N \alpha_n - \frac{1}{2} \sum_{n=1}^N \sum_{m=1}^N \alpha_n \alpha_m y_n y_m k(x_n, x_m) \right) \\ 0 \leq \alpha_n \leq C, \sum_{n=1}^N \alpha_n y_n = 0 \end{aligned}$$

Funcția de decizie prin care se calculează hiperplanul rămâne însă identică cu cea din (2.3.15), adică:

$$f(x) = w \cdot x + b = \sum_{\alpha_n > 0} \alpha_n y_n k(x_n, x) + b \begin{cases} \geq 0 \\ < 0 \end{cases} \Rightarrow \begin{cases} y = 1 \\ y = -1 \end{cases}$$

### 2.3.5 Clasificarea mai multor clase

Algoritmul SVM produce implicit o clasificare binară, cum a fost menționat mai sus clasa fiind  $y \in \{-1, 1\}$ , adică fie aparține unei clase, fie nu aparține,

neavând posibilitatea de a clasifica un set de date ce conține 3 sau mai multe clase [7], [9].

Pentru a putea clasifica mai multe clase:  $c_1, c_2, \dots, c_k$ , putem considera pe rând fiecare clasă ca să aibă „eticheta”  $y = 1$ , în timp ce păstrăm clasa  $y = -1$  pentru toate celălalte clase, procedeul repetându-se pentru toate clasele astfel încât într-un final să avem  $k$  funcții de decizie:  $f_1(x), f_2(x), \dots, f_k(x)$ . Unde  $f_i(x)$  a fost obținută păstrând  $y_i = 1$  și  $y_j = -1, j \neq i$ .

Într-un final, datorită faptului că fiecare funcție de decizie ar da o valoare pozitivă în cazul în care un nou exemplu aparține clasei respective, pentru a lua decizia finală acest exemplu va aparține clasei dată de:

$$\arg \max_c \{f_1(x), f_2(x), \dots, f_k(x)\}$$

Unde  $c \in \overline{1, k}$ . Pe scurt, clasa a cărei funcții de decizie produce valoarea maximă va fi clasa exemplui  $x$ .

### 2.3.6 Algoritmul Sequential Minimal Optimization

Pentru a implementa algoritmul Support Vector Machines în practică este mult prea costisitor, mai exact optimizarea ecuației din (2.3.22) este dificilă și neeficientă, deoarece pentru acest lucru ar trebui să folosim niște metode de programare quadratică astfel încât să optimizăm simultan toți multiplicatorii Lagrange,  $\alpha_n$ . Ca soluție, în practică se implementează un algoritm care ar produce aceleași rezultate și este mult mai eficient.

$$\begin{aligned} \max_{\alpha} \left( \sum_{n=1}^N \alpha_n - \frac{1}{2} \sum_{n=1}^N \sum_{m=1}^N \alpha_n \alpha_m y_n y_m k(x_n, x_m) \right) \\ 0 \leq \alpha_n \leq C, \sum_{n=1}^N \alpha_n y_n = 0 \end{aligned} \quad (2.3.22)$$

Algoritmul Sequential Minimal Optimization (SMO) a fost introdus de către John Platt în 1998. Acesta simplifică enorm antrenarea la Support Vector Machines, deoarece în loc să optimizeze simultan toți multiplicatorii Lagrange din

(2.3.22) optimizeaza pe rând tot câte doi multiplicatori, până când algoritmul converge.

Motivul pentru care nu a fost ales să se optimizeze doar câte un singur multiplicator la un moment dat se datorează condiției de liniaritate din (2.3.22) și anume  $\sum_{n=1}^N \alpha_n y_n = 0$  (dacă ar fi fost ales doar un singur  $\alpha$  la un moment dat atunci acesta ar fi trebuit să fie 0 penntu a îndeplini această condiție, secvența fiind liniar dependentă). Avantajul imediat a alegerii pentru optimizare doar a doi multiplicatori este și că se poate găsi acel maxim analitic matematic.

Adițional algoritmul optimizează câte doi multiplicatori Lagrange la fiecare pas, astfel încât ei să fie mărginiți, mai exact să respecte condițiile Karush-Kuhn-Tucker (KKT) ce reies din (2.3.16) și (2.3.21) (deliate însă și în [12]):

$$y_n(f(x_n) - y_n) = y_n E_n \begin{cases} > 0 \\ = 0 \\ < 0 \end{cases} \Leftrightarrow \begin{cases} \alpha = 0 \\ \alpha \in (0, C) \\ \alpha = C \end{cases} \quad (2.3.23)$$

Algoritmul SMO publicat în [14], are trei părți principale în componența sa, mai exact trei funcții ce formează întregul algoritmul.

```
main routine:
  numChanged = 0;
  examineAll = 1;
  while (numChanged > 0 | examineAll)
  {
    numChanged = 0;
    if (examineAll)
      loop I over all training examples
      numChanged += examineExample(I)
    else
      loop I over examples where alpha is not 0 & not C
      numChanged += examineExample(I)
    if (examineAll == 1)
      examineAll = 0
    else if (numChanged == 0)
      examineAll = 1
  }
```

Fig. 2.3.2 Pseudocod pentru procedura main din algoritmul SMO.

Procedura main observabilă în [Fig. 2.3.2] are ca scop alegerea primului multiplicator Lagrange. Se începe prin inițializarea a două variabile numChanged ce contorizează de câte ori au fost optimizați simultan doi multiplicatori, respectiv examineAll ce face să se parcurgă prin toate exemplele de antrenament când are valoarea 1, respectiv doar peste exemplele a căror multiplicatori Lagrange corespunzători nu se află la extreme (nu sunt egali cu 0 sau cu  $C$ ).

Pentru fiecare exemplu (în funcție de examineAll) se apelează funcția examineExample( $i$ ), unde  $i$  este indexul exemplului ales pentru examinare. Acest lucru se repetă până când ambele variabile, numChanged și examineAll sunt 0 (aceasta se întâmplă atunci când la o anumită iterație au fost parcurse doar exemplele a căror multiplicatori Lagrange nu au fost la valorile extreme și atunci nu a fost modificat niciun  $\alpha$ ).

```

procedure examineExample(i2)
    y2 = target[i2]
    alph2 = Lagrange multiplier for i2
    E2 = SVM output on point[i2] - y2 (check in error cache)
    r2 = E2*y2
    if ((r2 < -tol && alph2 < C) || (r2 > tol && alph2 > 0))
    {
        if (number of non-zero & non-C alpha > 1)
        {
            i1 = result of second choice heuristic (section 2.2)
            if takeStep(i1,i2)
                return 1
        }
        loop over all non-zero and non-C alpha, starting at a random point
        {
            i1 = identity of current alpha
            if takeStep(i1,i2)
                return 1
        }
        loop over all possible i1, starting at a random point
        {
            i1 = loop variable
            if (takeStep(i1,i2)
                return 1
            }
        }
    }
    return 0
endprocedure

```

Fig. 2.3.3 Pseudocod pentru procedura examineExample din algoritmul SMO.

În [Fig. 2.3.3] este prezentată funcția `examineExample` ce primește ca parametru indexul primului exemplu (dintre cei doi care vor fi selectați) ales pentru optimizare, acest index (i2) provine din procedura `main` după cum a fost explicat anterior. Prima dată este verificat dacă acest exemplu respectă condițiile KKT din (2.2.23) cu o anumită toleranță și dacă da atunci se caută al doilea exemplu cu ajutorul căruia să se optimizeze simultan multiplicatorii lor Lagrange corespunzători.

Alegerea celui de-al doilea exemplu se face după trei criterii. Primul criteriu este o euristică ce încearcă să aleagă acel multiplicator care produce cea mai mare optimizare (mai exact cel care maximizează  $|E_1 - E_2|$ , unde  $E_i = f(x_i) - y_i$ ). Dacă în acest mod nu s-a găsit niciun exemplu care să poată optimiza simultan cei doi multiplicatori corespunzători lor, atunci se încearcă căutarea celui de-al doilea exemplu printre exemplele a căror multiplicatori nu au valori extreme, adică  $\alpha \in (0, C)$ . Într-un final, dacă nici așa nu s-a reușit obține o posibilă optimizare atunci se alege un exemplu aleator din tot setul de date. În tot acest timp la fiecare criteriu pentru a observa dacă se obține vreo optimizare (sau chiar pentru a se optimiza) este apelată procedura `takeStep`.

Această procedură din urmă se poate observa mai jos în [Fig.2.3.4]. Pentru început se calculează  $E_1 = f(x_1) - y_1$  (pentru al doilea exemplu ales), urmat de  $L$  și  $H$  găsite în ecuațiile:

$$y_1 \neq y_2 \Leftrightarrow \begin{cases} L = \max(0, \alpha_2 - \alpha_1) \\ H = \min(C, C + \alpha_2 - \alpha_1) \end{cases} \quad (13)$$

$$y_1 = y_2 \Leftrightarrow \begin{cases} L = \max(0, \alpha_2 + \alpha_1 - C) \\ H = \min(C, \alpha_2 + \alpha_1) \end{cases} \quad (14)$$

$L$  și  $H$  asigură că condițiile relației din (2.3.22) sunt îndeplinite, în funcție se iese dacă aceste variabile sunt egale (practic nu sunt respectate condițiile menționate mai sus în (13) și (14)). Ulterior în algoritm se calculează  $\eta =$

$k(x_1, x_1) + k(x_2, x_2) - 2k(x_1, x_2)$  cu ajutorul căreia se determină noile valori pentru  $\alpha_2$  și  $\alpha_1$ , după formulele:

$$\alpha_2 = \alpha_2 \frac{y_2(E_1 - E_2)}{\eta}; \quad \alpha_1 = \alpha_1 + y_1 y_2 \Delta a_2$$

În plus  $\alpha_2$  se și normalizează, iar dacă modificarea este mult prea mică atunci se iese din funcție fără a se actualiza nimic. La final se actualizează noile valori pentru bias-ul  $b$  și multiplicatorii Lagrange. Pentru a obține o convergență într-un timp mai rapid algoritmul recomandă păstrarea într-un așa numit *error-cache* a erorilor pentru exemplele a căror multiplicatori Lagrange nu au valori la extreme.

```

procedure takeStep(i1,i2)
  if (i1 == i2) return 0
  alph1 = Lagrange multiplier for i1
  y1 = target[i1]
  E1 = SVM output on point[i1] - y1 (check in error cache)
  s = y1*y2
  Compute L, H via equations (13) and (14)
  if (L == H)
    return 0
  k11 = kernel(point[i1],point[i1])
  k12 = kernel(point[i1],point[i2])
  k22 = kernel(point[i2],point[i2])
  eta = k11+k22-2*k12
  if (eta > 0)
  {
    a2 = alph2 + y2*(E1-E2)/eta
    if (a2 < L) a2 = L
    else if (a2 > H) a2 = H
  }
  else
  {
    Lobj = objective function at a2=L
    Hobj = objective function at a2=H
    if (Lobj < Hobj-eps)
      a2 = L
    else if (Lobj > Hobj+eps)
      a2 = H
    else
      a2 = alph2
  }
  if (|a2-alph2| < eps*(a2+alph2+eps))
    return 0
  a1 = alph1+s*(alph2-a2)
  Update threshold to reflect change in Lagrange multipliers
  Update weight vector to reflect change in a1 & a2, if SVM is linear
  Update error cache using new Lagrange multipliers
  Store a1 in the alpha array
  Store a2 in the alpha array
  return 1
endprocedure

```

Fig. 2.3.4 Pseudocod pentru procedura takeStep din algoritmul SMO.



## **2.4 Evaluarea algoritmilor de clasificare**

### **2.4.1 Împărțirea setului de date**

Având un set de date și un algoritm putem obține o funcție de decizie de pe urma căreia să putem clasifica noi exemple, însă facând doar acest lucru nu avem de unde să știm dacă clasificarea a fost corectă sau dacă algoritmul a învățat bine. Pentru a avea garanția că algoritmul este eficient și că ne putem baza pe clasificarea dată de el vom folosi câteva metrici pentru a măsura eficiența algoritmului, însă mai întâi avem nevoie de un set de date pe care să putem face această măsurare (sau testare).

Putem obține acest lucru luând setul de date inițial și să-l împărțim în două seturi, un set de date de antrenament: pe baza căruia algoritmul învață și produce o funcție de decizie și un set de date de testare: pe baza căruia să măsurăm cât de eficient este algoritmul. Spre exemplu se poate păstra 70% din datele inițiale pentru antrenament (alese aleator) și restul de 30% pentru testare. Numărul exact nu are o importanță așa mare, important este ca datele alese pentru testare să nu se fi regăsit și în setul de antrenament, deoarece dorim să vedem cum se comportă algoritmul pe date noi și nu ne-ar ajuta deloc dacă algoritmul ar fi eficient doar pentru datele inițiale dacă el ar clasifica greșit exemplele noi.

### **2.4.2 Metrici pentru evaluarea algoritmilor de clasificare**

După ce facem acest lucru și algoritmul obține o funcție de decizie pe baza setului de date de antrenament, putem introduce cum a fost menționat mai sus niște metrici cu ajutorul cărora să evaluăm cât de bine a învățat algoritmul.

O metrică adesea utilizată este acuratețea, aceasta reprezintă raportul dintre numărul de predicții corecte și numărul total de predicții făcute. Adică plecând de la setul de date de test, se contorizează de câte ori algoritmul a clasificat exemplele

conform cu realitatea și se împarte acest număr la numărul exemplilor din setul de testare.

$$\text{Acurate\cetea} = \frac{\text{Num\c4rul de predic\c4ii corecte}}{\text{Num\c4rul de predic\c4ii f\c4cute}}$$

At\c4t timp c\c4t num\c4rul de exemple din datele de test sunt distribuite relativ egal la fiecare clas\c4 atunci ne putem baza pe aceast\c4 metric\c4, \c4ns\c4 \c4n cazul \c4n care predomin\c4 instan\c4e ce apar\c4tin unei singure clase \c4i foarte pu\c4tine instan\c4e r\c4m\c4n la cel\c4lalte clase atunci putem ob\c4tine o acurate\c4e foarte mare f\c4c\c4nd ca algoritmul s\c4 prezic\c4 numai clasa cu multe exemple. E clar \c4ns\c4 c\c4 dac\c4 am face din nou un test pe alt set de date ce are foarte pu\c4tine exemple \c4n clasa cu multe exemple men\c4ionate anterior atunci s-ar ob\c4tine o acurate\c4e mica \c4i s-ar putea observa c\c4 de fapt algoritmul era unul prost, chiar dac\c4 acurate\c4ea era ridicat\c4 [11].

Pentru a evita acest lucru putem introduce alte dou\c4 metrici: precizia \c4i sensibilitatea, ce se bazeaz\c4 pe matricea de eroare.

Mai jos \c4n [Fig. 2.4.1] este prezentat\c4 matricea de eroare pentru o clasificare binar\c4 (cazul \c4n care exemplele apar\c4tin sau nu unei clase) unde a fost notat:

TP = „True Positive” - num\c4rul de exemple predic\c4ionate ca apar\c4tin\c4nd clasei ce \c4i \c4n realitate apar\c4tineau clasei respective.

FP = „False Positive” - num\c4rul de exemple predic\c4ionate ca apar\c4tin\c4nd clasei, dar \c4n realitate nu apar\c4tineau clasei respective.

FN = „False Negative” - num\c4rul de exemple predic\c4ionate ca neapar\c4tin\c4nd clasei, dar \c4n realitate apar\c4tineau clasei respective.

TN = „True Negative” - num\c4rul de exemple predic\c4ionate ca neapar\c4tin\c4nd clasei ce \c4i \c4n realitate nu apar\c4tineau clasei respective [27].

\c4n cazul \c4n care \c4n setul de date sunt mai multe clase atunci se creeaz\c4 c\c4te o matrice de eroare pentru fiecare clas\c4, iar cele 4 contoare men\c4ionate mai sus sunt incrementate cu 1 pentru fiecare clas\c4, \c4n func\c4ie de cum a fost predic\c4ionat \c4i de cum a fost \c4n realitate exemplul.

Actual	Positive	TP	FN
	Negative	FP	TN
		Positive	Negative
		Predicted	

Fig. 2.4.1 Matricea de eroare

Având aceste lucruri putem determina și metricile menționate mai sus de pe urma formulelor:

$$\text{Precizia} = \frac{TP}{TP + FP}; \quad \text{Sensivitatea} = \frac{TP}{TP + FN}$$

Practic precizia reprezintă de câte ori algoritmul a predicționat corect, raportându-ne la numărul total de predicții date de algoritm ca aparținând clasei, pe când sensivitatea reprezintă de câte ori algoritmul a predicționat corect, dar raportându-ne în acest caz la numărul total în care algoritmul ar fi trebuit să prezică exemplele ca aparținând clasei. Aceste calcule făcându-se individual pentru fiecare matrice de eroare, (dacă este cazul) obținând precizia respectiv sensivitatea pentru fiecare clasă în parte. Ulterior pentru a obține precizia și sensivitatea algoritmului (nu numai pentru o clasă specifică) se poate face media aritmetică a metricilor din fiecare clasă.

Într-un final, chiar dacă aceste metrici ne spun că algoritmul ar fi eficient, este foarte important ca acest lucru să se întâmple într-un timp util, pentru că nu dorim să așteptăm câteva zile chiar dacă am obține niște rezultate mai bune. Așadar și timpul poate fi considerat o metrică foarte utilă.

## Capitolul 3

# Rezolvarea temei de proiect

### 3.1 Introducere

În continuare se va prezenta modul în care au fost dezvoltate în practică ce a fost propus la începutul lucrării, adică implementarea în cod a algoritmilor Rocchio și Support Vector Machines (detaliați în partea teoretică a lucrării), implementarea evaluării acestor algoritmi, dar și dezvoltarea unei interfețe grafice ce permite vizualizarea rezultatelor algoritmilor, în mare parte pentru date bidimensionale sau tridimensionale.

Aplicația a fost scrisă integral de la zero folosind doar tool-urile oferite de limbajul de programare ales, adică Racket, fără a folosi librării sau framework-uri externe (cu excepția funcțiilor din librăria de plot folosite pentru a crea interfața grafică). A fost folosită versiunea 8.1 pentru Racket în mediul de dezvoltare Dr-Racket, iar aplicația a fost testată pe sistemele de operare Windows 10 și Ubuntu 20.04. Motivul implementării de la zero a aplicației este că unul dintre scopurile acestei lucrări este de a înțelege funcționarea algoritmilor și a detaliilor din spatele acestora, nu de a prelua niște algoritmi deja implementați, chiar dacă poate s-ar obține ceva mai performant.

## 3.2 Limbajul de programare ales

### 3.2.1 De ce Racket?

Cum a fost menționat și mai sus, pentru a dezvolta această aplicație am ales să folosesc limbajul de programare Racket. În general pentru machine learning Python este de departe cel mai popular limbaj de programare folosit, asta și datorită numeroaselor librării ce sunt puse la dispoziție. Fiindcă dorința e de a implementa ceva de la zero, acest beneficiu devine inutil lăsând libertatea de a folosi absolut orice limbaj de programare.

Racket este un dialect de Lisp, un limbaj de programare multiparadigmă care încurajează programarea funcțională și care este orientat în special spre dezvoltarea de noi limbaje de programare, mai ales dezvoltarea de limbaje specializate pe anumite domenii (așa numitele DSL-uri: domain-specific language). Cu toate acestea această lucrare nu se va ocupa cu dezvoltarea de noi limbaje de de programare, ci doar cu implementarea algoritmilor de clasificare menționați.

Limbajul din care este derivat Racket, Lisp, a fost dezvoltat în 1958 de către John McCarthy și este al doilea cel mai vechi limbaj de programare, după Fortran. Este și primul limbaj de programare ce a introdus programarea funcțională, calculul lambda și a fost folosit îndeosebi pentru cercetare în domeniul inteligenței artificiale [28].

Programarea funcțională este o paradigmă de programare unde aplicațiile sunt dezvoltate apelând funcții și compuneri de funcții. Este un stil mai strict de programare unde funcțiile procesează ce primesc ca date de intrare și returnează un rezultat fără a folosi mutații adică a modifica valorile pe parcurs. Aplicațiile nu se bazează pe o stare sau un context extern, iar o expresie are o singură valoare pe tot parcursul programului, acesta fiind un motiv pentru care paralelizarea este foarte fezabilă în cadrul programării funcționale [24].

La ora actuală Lisp-ul propriu-zis nu mai există, ci doar limbaje derivate de-a lungul timpului din el, precum Common Lisp, Clojure, Racket și multe altele. Am ales totuși Racket dintre toate dialectele de Lisp existente fiindcă are posibilitatea de a lucra ușor cu o interfață grafică, are o documentație bogată și foarte utilă. Pe lângă acestea are în componența sa și un mediu de dezvoltare (DrRacket) foarte ușor de utilizat și interactiv.

Dorința de a deveni familiar cu programarea funcțională este și ea unul dintre motivele pentru care am ales un limbaj derivat din Lisp, chiar dacă aici este vorba de o programare funcțională impură (permite și altor paradigme de programare să coexiste). Racket este un limbaj cu mult diferit de limbaje uzuale precum C, Java, Matlab, Python sau multe altele și consider că mereu este util a învăța ceva nou. Însă binenteles motivul principal pentru care am ales Racket este că îmi place acest limbaj de programare.

### 3.2.2 Scurt tutorial pentru Racket

Racket este un limbaj „dynamic and strongly typed”, prin care se înțelege că tipul valorilor nu trebuie să fie menționat de către programator la declararea lor și nu se face cast automat la un alt tip de date. În plus evaluarea este aplicativă, adică parametrii sunt calculați înainte de aplicarea unei procedurii. Ca și Lisp-ul, Racket are o sintaxă bazată pe paranteze și folosește notația poloneză, așa numita formă prefixată [17], [18].

Pentru a pregăti mai bine „terenul” și a înțelege mai ușor ulterior algoritmi ce vor fi implementați, se va prezenta în continuare niște exemple de cod în Racket. De început pentru a observa funcționarea notației poloneze, vom scrie expresia matematică  $a \cdot e^b + \sin(a)$  în Racket, după cum urmează:

```
(+ (* 'a (exp 'b)) (sin 'a))
```

Din aceasta se poate observa că orice funcție (denumită și procedură) apare mereu înaintea operandilor, practic în Racket paranteza deschisă indică că va urma o apelare de funcție, urmată de parametrii ei (dacă este cazul bineînțeles, pentru că pot exista și funcții fără parametrii). Tot din secvența de mai sus se poate observa și noul tip de date: ' (apostroful) care are rolul de a preveni evaluarea simbolului (o valoare formată din numere sau caractere ce nu conțin spațiu) sau expresiei ce urmează după el.

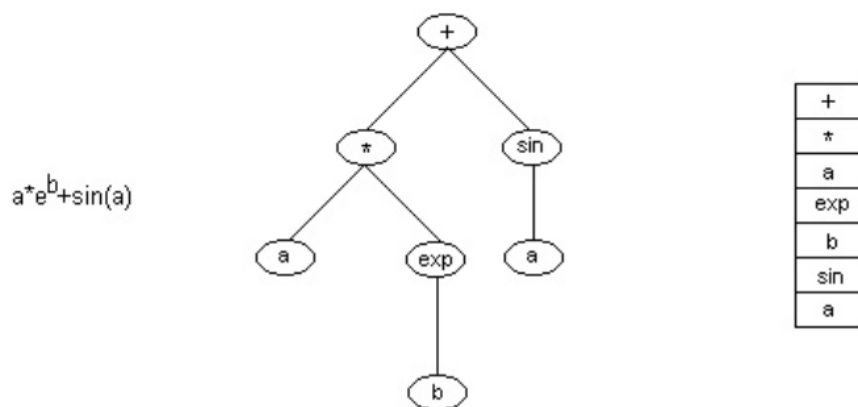


Fig. 3.1.1 Arborele și stiva aferenți unei expresii matematice

Această expresie poate fi parsată sub forma unui arbore pentru a putea fi înțeles mai ușor, ca în [Fig. 3.1.1], unde este prezentat și cum arată stiva aferentă acestei expresii matematice. Arborele din poză arată cum prioritatea operatorilor (funcțiilor) crește către frunzele arborelui (frunzele fiind parametrii în majoritatea cazurilor), urmând ca stiva să se umple printr-o parcurgere în preordine (rădăcină - stânga - dreapta). Acest lucru nu este neapărat necesar să fie cunoscut pentru a scrie cod în Racket, însă utilitatea acestei sintaxe apare în special în dezvoltarea de noi limbaje de programare, deoarece stilul acesta ierarhic a datelor este foarte convenabil de interpretat de către calculator, bineînțeles la prima vedere nu și pentru om deoarece este diferit de stilul uzual folosit.

Tipurile de date importante utilizate în Racket (și în acest proiect) sunt: numărul, simbolul (menționat deja mai sus la apostrof), șirul de caractere (string), lista (fiind structura de date fundamentală, chiar și denumirea de Lisp provine de la List Processing), iar pe lângă acestea mai sunt folosiți și vectorul și tabela de dispersie (hash-setul).

Listele în Racket sunt imutabile, adică odată create nu se pot modifica (dacă sunt asigurate unui identificator). Acestea pot fi create folosind procedura „list” sau alternativ cu un apostrof în față, diferența fiind că în ultimul caz nu se vor evalua elementele și vor rămâne la fel.

```
(list (+ 1 2) 0 1 2) ; se va evalua ca '(3 0 1 2)
'((+ 1 2) 0 1 2) ; nu se va evalua
```

Un identificator poate fi legat de o valoare folosind keyword-ul `define`. Diferența de limbajele de programare uzuale este că aceste valori nu se pot suprascrie în locații de memorie și doar permite referirea unei expresii cu ajutorul unui nume, spre exemplu mai jos se atribuie identificatorului „lista” o listă cu 6 numere de la 1 la 6:

```
(define lista (list 1 2 3 4 5 6))
```

Există o multitudine de funcții ce manipulează listele, câteva dintre ele prezentate succint mai jos:

```
(first lista) ;-> 1
(rest lista) ;-> '(2 3 4 5 6)
(last lista) ;-> 5
(length lista) ;-> 6
(reverse lista) ;-> '(6 5 4 3 2 1)
(cons 0 lista) ;-> '(0 1 2 3 4 5 6)
(take lista 3) ;-> '(1 2 3)
```



```

(drop lista 3) ;-> '(4 5 6)
(append (drop lista 4) (take lista 2)) ;-> '(5 6 1 2)
(map add1 lista) ;-> '(2 3 4 5 6 7)
(apply + lista) ;-> 21

```

Vectorii și tabelele de dispersii sunt mutabili (de aceasta programarea funcțională în Racket este una impură, permițând mutația), vom folosi totuși puțin ulterior în program aceste structuri din motive de performanță. Ca exemplu de utilizare a acestor două structuri de date se poate observa mai jos:

```

(define vectoras (vector 1 2 3)) ;-> #(1 2 3)
(vector-ref vectoras 0) ;-> 1
(vector-set! vectoras 0 5) ;-> #(5 2 3)

(define hashh (make-hash)) ;-> '#hash()
(hash-set! hashh 'unu 1) ;-> '#hash((unu . 1))
(hash-ref hashh 'unu) ;-> 1

```

În Racket toate procedurile sunt funcții lambda, însă acestea pot fi legate de un identificator, ca de exemplu:

```

(lambda(x) (* x 2) 3) ;-> 6
(define doubleaza (lambda(x) (* x 2)))
(doubleaza 3) ;-> 6
(define (simplificat-doubleaza x) (* x 2))
(simplificat-doubleaza 3) ;-> 6

```

Ultima variantă este varianta folosită în practică de obicei datorită simplității ei, însă în spate de fapt este tot o funcție lambda. O altă metodă similară pentru define, dar care se apelează implicit este let (și variantele sale).

Mai departe în [Fig. 3.2.1] vor fi prezentate și câteva structuri conditionale:

```
(define x 2)

(if (< x 3) ;evalueaza expresia
    "doi" ;cazul in care a fost adevarata
    "trei" ;cazul in care a fost falsa (else)

(when (= x 2) ;structura conditionata similara cu if
    "atata") ;diferenta este ca nu are partea de "else"

(cond ;tot o structura conditionata cu mai multe ramuri
  [(= x 1) "unu"]
  [(= x 3) "trei"]
  [else "altceva"])
```

Fig. 3.2.1 Expresii conditionale în Racket

În final, deși la fel de importante sunt și buclele de program, câteva observabile în [Fig. 3.2.2].

```
(for ([i (in-range 10)]) ;o varianta de for
  (print i)) ;aceasta itereaza in intervalul [0,10)
;si printeaza toate numerele din aceste interval

(for/list ([i (in-range 10)]) ;similar cu for-ul de mai sus
  (if (odd? i) i 0)) ;diferenta este ca rezultatele sunt puse
;intr-o lista, adica in acest caz numarul daca este impar
;sau 0 altfel, practic se returneaza '(0 1 0 3 0 5 0 7 0 9)

(for/fold ([sum 0]) ;o alta varianta de for
  ([i (in-range 10)]) ;stocheaza rezultatul
  (if (even? i) ;intr-o variabila sum, dar la fel itereaza
    (+ sum i) ;pe parcurs se adauga variabilei sum
    sum)) ;toate numerele pare, returnand sum in final
```

Fig. 3.2.2 Bucle de program în Racket

Bineînțeles aceste lucruri sunt departe de a cuprinde tot ce se poate obține folosind limbajul Racket, însă este un start bun pentru cele ce urmează, unde va fi descrisă aplicația propriu-zisă cu explicații adiționale unde este cazul.

### 3.3 Arhitectura aplicației

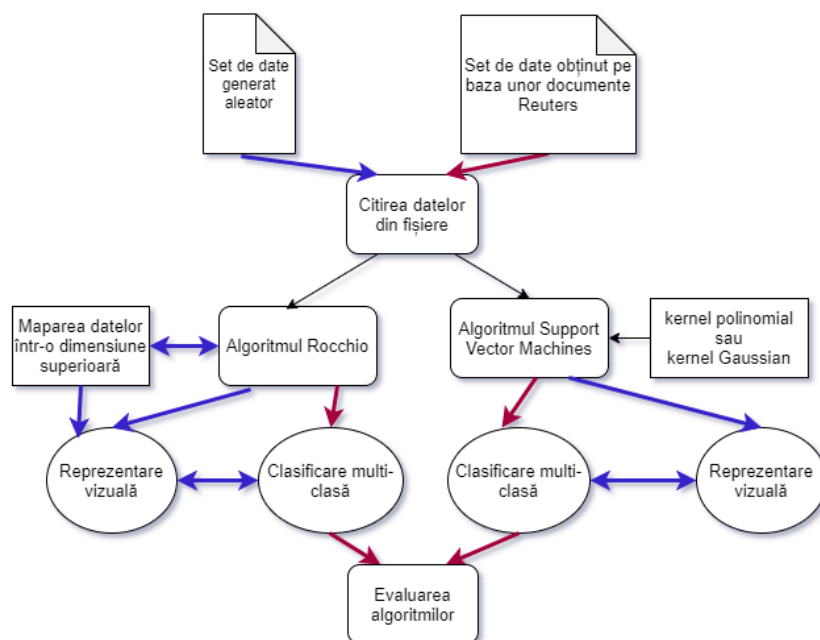


Fig. 3.3.1 Arhitectura aplicației

În [Fig. 3.3.1] este prezentată arhitectura acestei aplicații. Vom folosi două tipuri de date, primele generate aleator utilizate în special pentru a reprezenta vizual clasificarea algoritmilor (peste tot unde au fost folosite acestea săgeata este albastră), iar celălalte obținute pe baza unor documente Reuters folosite în special pentru a evalua algoritmi (peste tot unde au fost folosite acestea săgeata este roșie). După ce au fost citite datele se poate alege fie algoritmul Rocchio fie algoritmul SVM. Pentru cel dintâi a fost implementată posibilitatea de a mapa explicit datele într-o dimensiune superioară (mai exact din 2D în 3D pentru datele generate aleator), iar pentru algoritmul SVM datele nu sunt mapate explicit într-o altă dimensiune ci se folosește una dintre funcțiile kernel puse la dispoziție (polinomială respectiv Gaussiană). În practică sunt folosite mai multe scripturi ce formează întreaga aplicație.

## 3.4 Citirea datelor din fișiere

Primul lucru de care avem nevoie pentru a antrena un algoritm este un set de date, fiindcă bineînțeles nu putem face nimic fără date, chiar dacă am avea cei mai performanți algoritmi. Așadar vom începe „drumul” dezvoltării aplicației prezentând modul în care au fost obținute seturile de date.

Se vor arăta două modalități de citire din fișiere, deoarece am lucrat cu două tipuri de date, un set de date generat aleator unde exemplele sunt în două dimensiuni (cu scopul de a le utiliza pentru interfața grafică și a observa cum se comporta vizual algoritmi) și un alt set de date extras pe baza unor documente text din colecția Reuters utilizate în special pentru a evalua algoritmi, dorind să facem testarea pe ceva semnificativ, cu aplicații practice în lumea reală (precum clasificarea de documente) [12].

### 3.4.1 Setul de date generat aleator

Setul de date generat aleator are pattern-ul pentru un exemplu (fiecare linie reprezintă un exemplu) format din eticheta (clasa) exemplului urmat de vectorul de trăsături. Acest lucru este observabil în [Fig. 3.4.1] unde este vorba de exemple bidimensionale cu 3 clase (0, 1 și 2), iar vectorul de trăsături are domeniul în intervalul  $[-300, 300]$  (orice alt număr este în regulă a fi ales, însă în această aplicație a fost folosit 300).

```
0 173 233
1 -117 115
2 213 -150
2 211 -145
0 181 215
2 201 -107
0 189 215
```

Fig. 3.4.1 Formatul setului de date generat aleator

Pentru a obține aceste numere aleatoare în funcție de anumite clase au fost urmați pașii următori:

- se creează un număr de clase (poate fi și un număr random), unde o clasă conține media și dispersia fiecărei axe de coordonate (un exemplu bidimensional de astfel de clasă este  $[(m_1, \sigma_1), (m_2, \sigma_2)]$ ).
- se alege aleator una dintre clase și un număr  $x$  în intervalul  $[-300, 300]$ .
- se convertește numărul ales în intervalul  $(0,1)$  folosind  $m$  și  $\sigma$  generați la pasul anterior în funcția  $G(x) = e^{-\frac{(m-x)^2}{\sigma}}$ .
- dacă  $G(x)$  este mai mare decât un prag ales aleator în intervalul  $[0,1]$  atunci numărul este acceptat și se repetă procedeul de la al doilea pas pentru următoarea axa de coordonate, altfel se repetă tot procedeul de la al doilea pas, dar pentru aceeași axă de coordonate.

În Racket, partea principală de generare a acestor date este arătată în [Fig. 3.4.2], unde inițial este definită funcția  $G(x)$  menționată în al 3-lea pas al algoritmului, urmată de funcția ce generează aleator un număr conform pasului 4. Numărul random se alege în intervalul  $[0, 1000]$  pentru a obține mai mult zgomot, însă ulterior acest număr aleator este făcut subunitar deoarece și funcția  $G(x)$  produce un număr subunitar. În final folosind generate-list se vor genera un număr  $n$  de exemple, cu o distribuție relativ egală pentru fiecare clasă (deoarece fiecare zonă la început este aleasă random).

După ce am folosit acest algoritm și am obținut setul de date, a trebuit și să le scriem în fișiere text cu scopul de a putea repeta testele. Așadar am generat mai multe seturi de date pe care mai apoi le citim pentru aplicația propriu-zisă.

```

(define (Gauss m  $\sigma$  x)
  (exp (- (/ (sqr (- m x)) (* 2 (sqr  $\sigma$ ))))))

(define (generate M S)
  (define rnd (random (- N) N))
  (if (> (Gauss M S rnd) (/ (random 0 1001) 1000))
      rnd
      (generate M S)))

(define (generate-list n)
  (define rnd (random 0 (length zones)))
  (define random-zone (list-ref zones rnd))
  (if (= n 0)
      empty
      (cons (cons rnd
                  (for/list ([i (in-range DIMENSION)])
                    (generate (car (list-ref random-zone i))
                              (cadr (list-ref random-zone i)))))
            (generate-list (- n 1)))))

```

Fig. 3.4.2 Generarea aleator a setului de date

În [Fig. 3.4.3] este prezentată modalitatea de citire a datelor generate random. Inițial este creată o funcție ce primește ca parametru calea unui fișier și salvează datele citite din fișier într-o listă (fiecare linie, ce reprezintă o instanță într-o listă, iar toate aceste liste înglobate într-o listă mai mare). Mai apoi se apelează o funcție din Racket ce deschide fișierul și se parcurge fiecare linie din fișier (care este un string inițial), iar aceasta se salvează într-o listă, fiecare element fiind o coordonată

```

(define (file-lines->list path)
  (call-with-input-file path
    (lambda (file)
      (for/list ([line (in-lines file)])
        (map string->number (string-split line))))))

(define train-set (file-lines->list "training.txt"))
(define test-set (file-lines->list "testing.txt"))

```

Fig. 3.4.3 Citirea datelor generate aleator din fișiere

(în fișier fiind delimitată de spațiu), în final se convertește fiecare coordonată dintr-un string la număr și se returnează lista ce conține aceste liste mai mici. Cu această funcție sunt citite mai apoi două fișiere (de antrenament și de testare).

### 3.4.2 Setul de date extra pe baza unor documente Reuters

Acest set de date a fost obținut din [13]. Documentele provin de pe urma știrilor publicate de către presa Reuters din Iulie 1996 timp de un an, mai exact au fost extrase documentele ce aparțin sistemelor software. După ce aceste documente au fost reprezentate sub forma de vectori și au fost eliminate cele ce reprezintă prost clasa au fost obținute 7053 de exemple cu 24 clase principale. Aceste documente au fost împărțite ulterior în alte două seturi, mai exact 4702 de exemple pentru setul de antrenament respectiv 2351 pentru setul de test. Fiecare instanță din totalul de 7053 prezintă 1309 de trăsături (un vector de 1309 de elemente), iar ponderile sunt în intervalul  $[0, 70]$ .

```
2351
1309
24

0:1 1:15 2:1 3:3 4:2 5:2 6:3 7:9 8:2 9:2 10:2 11:3 12:1 13:2 14:3 15:2 16:2
17:1 18:4 19:1 20:2 21:1 22:1 23:2 24:1 25:1 26:2 27:1 28:1 29:1 30:4 31:1
32:1 33:2 34:15 35:1 36:1 37:1 38:1 39:1 40:1 41:1 42:1 43:1 44:2 45:1 46:1
47:2 48:1 49:1 50:2 51:1 52:1 53:1 54:1 55:3 56:2 57:1 58:4 59:2 60:1 61:4
62:1 63:3 64:2 65:3 66:3 67:1 68:2 69:1 70:1 71:1 72:4 73:1 74:1 75:1 76:1
77:1 78:1 79:1 80:1 81:1 82:2 83:1 84:1 85:1 86:1 87:1 88:1 89:1 90:1 91:1
92:1 93:1 94:1 95:1 96:1 97:1 98:2 99:1 100:1 101:1 102:1 103:1 104:1 105:1
106:1 107:1 108:1 109:1 110:1 111:1 112:1 113:1 114:1 115:1 116:1 117:1
118:2 119:1 120:1 121:1 122:1 123:1 124:1 125:1 126:1 # c18 c181
0:1 2:1 17:1 30:1 112:1 127:1 128:2 129:1 130:1 131:1 132:1 133:1 134:1
135:1 136:1 137:1 138:1 # c15 c152

0:1 1:7 3:1 4:1 5:1 7:3 9:4 10:2 11:1 13:1 15:1 17:8 19:1 20:1 21:3 26:1
28:1 33:1 34:4 39:1 45:2 47:1 49:1 50:4 51:3 52:1 55:5 60:1 61:1 72:2 83:1
86:1 89:2 90:2 100:2 101:1 104:1 105:1 106:1 107:1 108:1 109:1 124:1 139:1
140:6 141:3 142:1 143:1 144:1 145:1 146:1 147:1 148:1 149:1 150:2 151:1
152:2 153:1 154:1 155:1 156:3 157:1 158:1 159:1 160:1 161:1 162:3 163:1
164:1 165:1 166:2 167:2 168:2 169:2 170:1 171:1 172:1 173:1 174:1 175:1
176:1 177:1 178:1 179:1 180:1 181:1 182:1 183:1 184:1 185:1 186:1 # c18
c181
```

Fig. 3.4.4 Formatul setului de date bazat pe documente Reuters

În [Fig. 3.4.4] se poate observa formatul acestor fișiere, pe prima linie este indicat numărul de exemple, urmat de numărul de trăsături (dimensiunea spațiului vectorial) urmat de numărul de clase (numărul 24 reprezintă și clasele secundare, 14 fiind principale). Un exemplu constă doar din ponderile nenule (fiind inutil a reprezenta toate 1309 de numere dacă majoritatea sunt 0), mai exact este indicată poziția și valoarea ei (1:15 semnificând valoarea 15 pe a doua poziție, deoarece numerotarea începe de la 0). În final, după # se indică și clasele exemplului, în această aplicație va fi folosită doar prima clasă, adică cea principală.

```
(define trainset (file->lines "TrainingSVM1309.arff"))
(define testset (file->lines "TestingSVM1309.arff"))

;(define samples (string->number (first dataset)))
;(define dimension (string->number (second trainset)))
;(define classes (string->number (third dataset)))

(define (split-set dataset) (map (λ(x) (string-split x)) (drop dataset 3)))

(define (get-sample sample [constant 70.0])
  (let loop ([clone sample] [new-sample (make-hash)])
    (cond
      [(equal? "#" (first clone))
       (define cls (string->number (substring (second clone) 1)))
       (list (if (not cls) 0 cls) new-sample)]
      [else
       (define coordinate (map string->number (string-split (first clone) #rx"(:)"))))
       (hash-set! new-sample (first coordinate) (/ (second coordinate) constant))
       (loop (rest clone) new-sample)])))

(define (get-dataset fileset)
  (for/list ([i fileset])
    (get-sample i)))

(define train-set (get-dataset (split-set trainset)))
(define test-set (get-dataset (split-set testset)))
```

Fig. 3.4.5 Citirea setului de date bazat pe documente Reuters

Pentru a obține aceste date s-a procedat conform [Fig. 3.4.5] unde s-a citit linie cu linie fiecare exemplu (ignorând primele 3 linii ce reprezintă doar informații despre ce urmează), exemplele au fost salvate într-o listă (folosind funcția get-dataset). Funcția get-sample creează fiecare listă formată din clasa exemplului urmată de un hash-set cu vectorul de trăsături, mai exact pentru fiecare cheie ce reprezintă numărul axei de coordonate se atașează valoarea aferentă axei. În final s-au citit ambele seturi de date (de antrenament și de testare).



## 3.5 Implementarea algoritmului Rocchio

Algoritmul Rocchio a fost detaliat în subcapitolul 2.2, iar în continuare se va arăta direct cum a fost materializat în cod acesta, mai exact cum a fost implementat pseudocodul din [Fig. 2.2.2] în Racket.

```
(define (get-points lst clasa)
  (if (empty? lst)
      empty
      (if (= (caar lst) clasa)
          (cons (cdar lst) (get-points (cdr lst) clasa))
          (get-points (cdr lst) clasa))))

(define (train-rochio lst)
  (for/list ([i classes])
    (define class-points (get-points lst i))
    (map (λ(x) (decimal (/ x (length class-points))))
         (apply map + class-points)))))
```

Fig. 3.5.1 Antrenarea algoritmului Rocchio

În [Fig. 3.5.1] se observă partea corespunzătoare lui TrainRocchio din pseudocod, get-points este o funcție ce pe baza setului de date și a clasei extrage o listă ce conține toate exemplele ce aparțin clasei date ca parametru, iar mai apoi în funcția train-rochio se calculează pentru fiecare clasă centroidul ei, salvând toți centroizii într-o listă. Partea cu o importanță deosebită este expresia (apply map + class-points) care după ce a fost extrasă lista doar cu exemplele unei clase returnează o listă nouă ce conține suma ponderilor de pe pozițiile lor corespunzătoare. În final pentru lista rezultată, fiecare element este împărțit cu numărul total de elemente ce au aparținut clasei, obținând în acest fel centroidul clasei.

A doua parte a pseudocodului, ApplyRocchio, returnează clasa de care aparține un nou element, pe baza listei de centroizi obținută în urma antrenării. Aceasta este prezentată în [Fig.3.5.2], începând prin a defini două funcții ajutatoare, cea de calculare a distanței Euclidiene dintre două liste, respectiv o funcție arg-min ce returnează o listă sortată în funcție de al doilea element al listei. În apply-rochio inițial este formată câte o listă pentru fiecare centroid, ce conține

eticheta clasei urmată de distanța Euclideană dintre clasa și exemplul de test. Mai apoi se apelează funcția `arg-min` (motivul pentru care se sortează în funcție de al doilea element iese la iveală acum, scopul fiind obținerea distanței minime). În final se returnează clasa aferentă distanței Euclidiene minime folosind funcția `caar` (ce returnează primul element din prima listă).

```
(define (euclidean-distance p q)
  (sqrt (apply + (map (lambda (x y) (sqr (- x y))) p q))))

(define (arg-min lst)
  (sort lst < #:key second))

(define (apply-rocchio centroids new-element)
  (caar (arg-min
    (for/list ([i centroids] [j classes])
      (list j (euclidean-distance i new-element))))))
```

Fig. 3.5.2 Clasificarea unei instanțe folosind algoritmului Rocchio

Având acest algoritm, se poate clasifica un întreg set de date, apelând funcția `apply-rocchio` pentru fiecare element. Vizual această clasificare pentru niște date separabile liniar se poate observa în [Fig. 3.5.3].

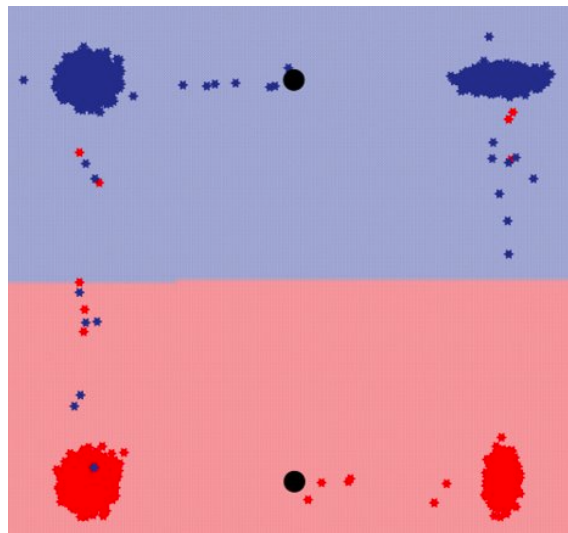


Fig. 3.5.3 Clasificarea Rocchio pentru un set de date separabil liniar.

### 3.6 Îmbinarea algoritmului Rocchio cu SVM

Cum a fost menționat și în subcapitolul 2.2, funcția de decizie a algoritmului Rocchio este un hiperplan sau mai exact spațiul vectorial este delimitat de câte un hiperplan pentru fiecare clasă. O problemă majoră care apare din cauza aceasta este că pentru a reuși să clasifice corect, algoritmul trebuie să lucreze cu date separabile liniar.

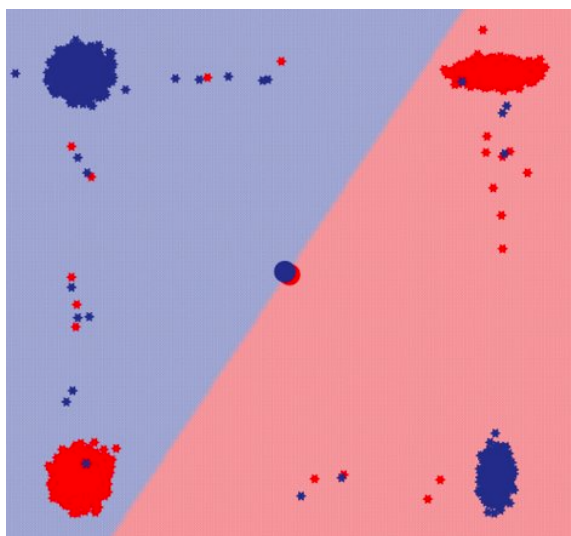


Fig. 3.6.1 Clasificarea Rocchio pentru un set de date neseparabil liniar.

În [Fig. 3.6.1] se poate observa cazul funcției XOR (mimicat prin setul de date), pentru care a folosit același set de date ca cel din [Fig. 3.5.3] cu diferența că clasele a două zone au fost inversate între ele, acest lucru făcând ca datele să nu mai poată fi separate liniar, iar clasificarea nu reușește să aibă loc decât pentru jumătate de puncte (din cauză că este imposibil a găsi un hiperplan ce separă corect punctele în acest caz).

Algoritmul Support Vector Machines utilizează așa numitul truc nucleu, prezentat în subcapitolul 2.3, prin care datele sunt mapate într-o dimensiune superioară astfel încât să se găsească un spațiu vectorial în care setul de date să poată

fi separabil liniar (în dimensiunea respectivă). În continuare în această lucrare se dorește a încerca aplicarea acestei idei și pentru algoritmul Rocchio pentru a observa vizual cum se comportă această transformare și a obține o intuiție mai bună despre cum funcționează în spate algoritmul Support Vector Machines când mapază datele într-o altă dimensiune și reușește să separe date neseperabile liniar în dimensiunea originală.

Pentru a reuși acest lucru, vom pleca de la setul de date bidimensional generat aleator unde un vector de trăsături poate fi scris la modul general ca  $(a, b)$ , deci inițial avem două axe de coordonate. Pe fiecare exemplu din setul de date îl vom transforma din două în trei dimensiuni având noua formă  $(a^2, \sqrt{2}ab, b^2)$ , obținând astfel trei axe de coordonate. Bineînțeles pe toată durata acestei transformări clasa exemplului se păstrează și după ce obținem tot setul de date, acum tridimensional, vom aplica algoritmul Rocchio.

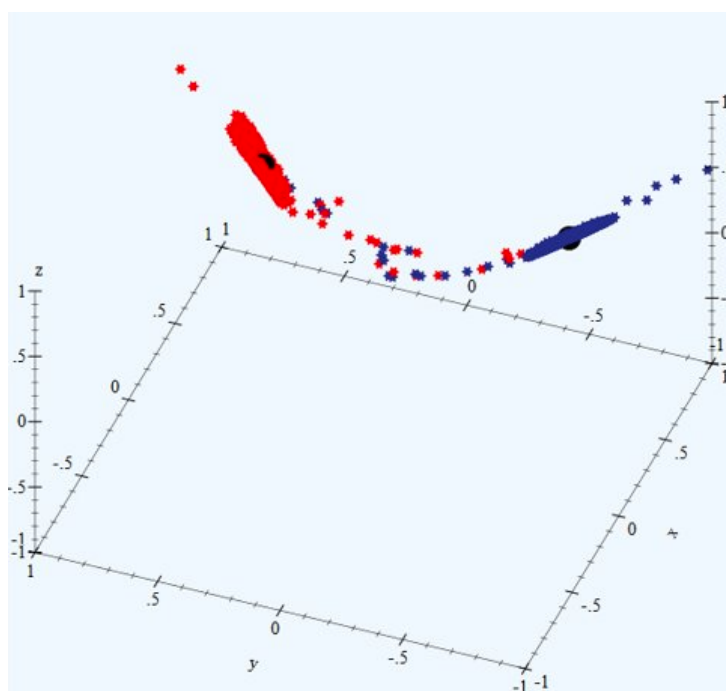


Fig. 3.6.2 Rezultatul transformării datelor din două în trei dimensiuni.

În [Fig. 3.6.2] se poate vizualiza cum arată aceleași puncte utilizate și în [Fig. 3.6.1], dar în acest caz mapate în trei dimensiuni folosind transformarea  $(a^2, \sqrt{2}ab, b^2)$  pentru exemplul inițial generic de forma  $(a, b)$ . Făcând acest lucru vedem cu ochiul liber că în acest spațiu tridimensional este relativ ușor a găsi un hiperplan ce separă cele două clase.

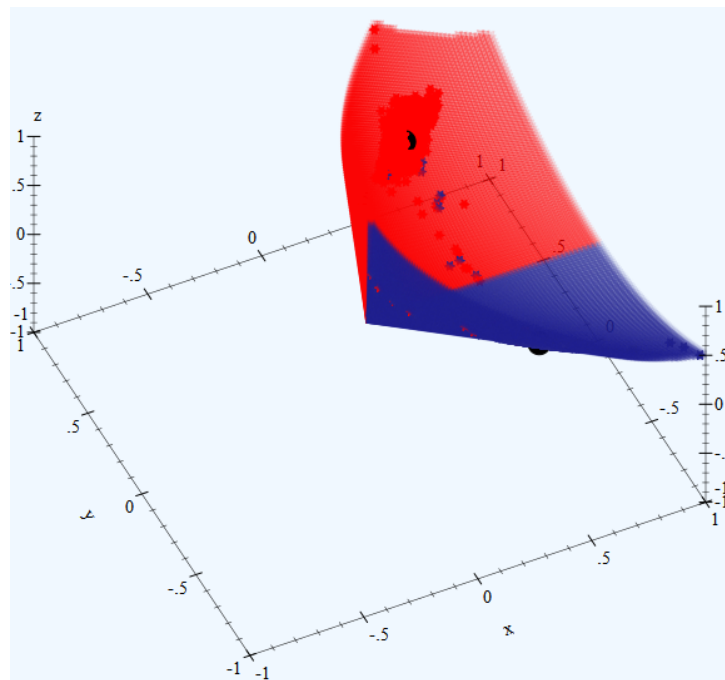


Fig. 3.6.3 Clasificarea Rocchio după ce a avut loc transformarea datelor.

Dupa ce a avut loc transformarea în trei dimensiuni, aplicarea algoritmului Rocchio va produce rezultatul vizibil în [Fig. 3.6.3], unde cum a fost și de așteptat a fost posibilă găsirea unui hiperplan ce separă datele acum tridimensionale.

În continuare rămâne un singur lucru de făcut și anume de a reveni înapoi în spațiul original cu două dimensiuni. După ce fiecare punct a fost clasificat cu algoritmul Rocchio în spațiul tridimensional, se preia eticheta asignată și se atribuie punctului original (cel cu două axe de coordonate), iar rezultatul acestei revenirii înapoi în spațiul bidimensional arată ca în [Fig. 3.6.4].

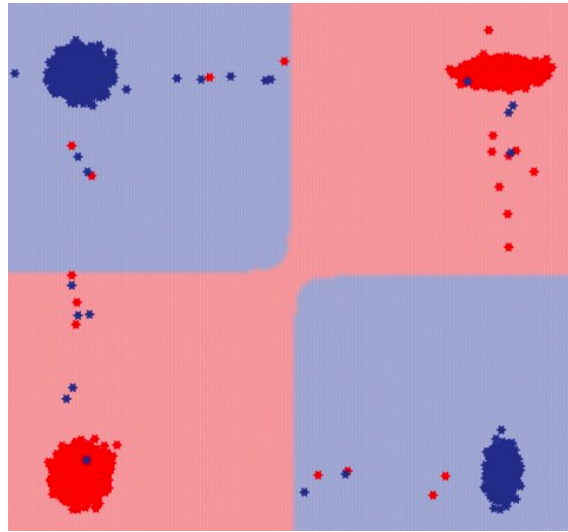


Fig. 3.6.4 Rezultatul clasificării Rocchio după revenirea în spațiul bidimensional.

Pentru a realiza acest lucru în Racket s-a procedat ca în [Fig. 3.6.5], mai întâi s-a creat o funcție, 2d-3d, ce așteaptă ca parametru o listă de forma  $(y, a, b)$  și returnează o altă listă ce păstrează clasa exemplului,  $y$ , dar cu 3 axe de coordonate, adică  $(y, a^2, \sqrt{2}ab, b^2)$ .

```
(define (2d-3d lst)
  (define x1 (second lst))
  (define x2 (third lst))
  (list (first lst) (sqr x1) (* (sqrt 2) x1 x2) (sqr x2)))

(define (change-class dataset prev after)
  (map (λ(x) (if (= prev (car x)) (cons after (cdr x)) x)) dataset))

(define (normalize lst)
  (cons (first lst) (map (λ(x) (/ x 300.0)) (rest lst))))

(define temp-train
  (map normalize (change-class (change-class (file-lines->list "train-xor.txt") 2 0) 3 1)))
(define temp-test
  (map normalize (change-class (change-class (file-lines->list "test-xor.txt") 2 0) 3 1)))

(define train-set (map 2d-3d temp-train))
(define test-set (map 2d-3d temp-test))
```

Fig. 3.6.5 Implementarea în Racket a transformării din 2d în 3d.

Tot aici a fost creată o funcție ce interschimbă clasa unei zone (pentru a reuși să se creeze funcția XOR din setul de date) și o altă funcție ce normalizează date în intervalul  $[-1, 1]$ , inițial fiind în intervalul  $[-300, 300]$ . Acest lucru a fost necesar

deoarce ridicând numerele la pătrat acestea ar fi devenit mult prea mari, haotice chiar pe tot domeniul numerelor reale, dar așa normalizate chiar și ridicate la pătrat sunt păstrate în intervalul  $[-1, 1]$ . În final, după ce au fost interschimbate clasele zonelor și normalizate datele a fost apelată funcția 2d-3d ce a mapat datele în trei dimensiuni.

Una dintre problemele ce apare pe urma acestor transformări este că nu este eficient a face acest lucru pentru date ce se află într-un spațiu de reprezentare deja ridicat, spre exemplu pentru setul de date Reuters ce se află într-o dimensiune de ordinul 1309 este foarte costisitor să mapăm datele într-o dimensiune superioară, în plus este dificil și de găsit un spațiu ideal unde să se facă clasificarea.

O altă problema este că nu putem lua un spațiu aleator și să ne așteptăm ca datele să poată fi clasificate corect. Spre exemplu în [Fig. 3.6.6] am folosit aceleași zone ca în [Fig. 3.5.3] și am aplicat aceeași transformare folosită anterior, totuși Rocchio nu reușește să clasifice punctele în acest spațiu tridimensional, chiar dacă a reușit foarte ușor în spațiul original fără a utiliza vreo transformare.

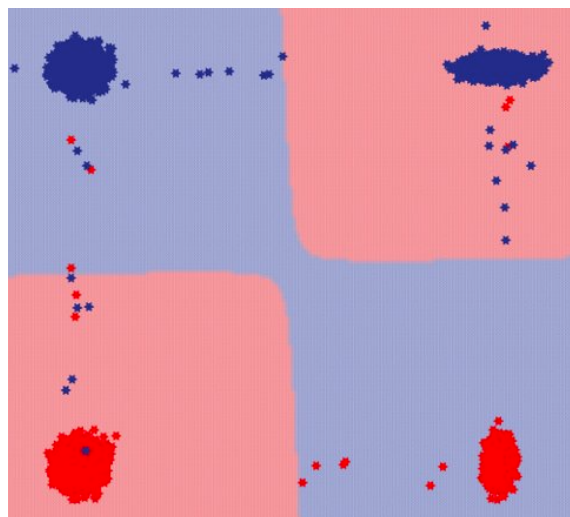


Fig. 3.6.6 Clasificarea Rocchio după ce a avut loc transformarea datelor.

### 3.7 Implementarea algoritmului SVM

Algoritmul Support Vector Machines depășește ambele probleme menționate anterior. În primul rând, acest algoritmul nu are nevoie să mapeze explicit setul de date într-o dimensiune superioară, deoarece exemplele din setul de date apar în algoritm doar sub formă de produs scalar și este de ajuns aplicarea unei transformări (numită și truc nucleu) asupra produsului scalar. În același timp putem folosi cu ușurință orice funcție pentru produsul scalar rezultat (care este un simplu număr) deci va fi mult mai ușor să căutăm un spațiu vectorial unde datele să poată fi separate liniar folosind algoritmul SVM.

În continuare se va arăta cum a fost implementat în Racket algoritmul SMO (prezentat în 2.3.5), mai exact materializarea în cod a celor 3 proceduri principale, însă mai întâi în [Fig. 3.7.1] se arată cum au fost atașați multiplicatorii Lagrange corespunzători fiecărui exemplu (inițializați cu 0 folosind funcția `vectorize`) și au fost salvați într-un vector nou numit `smo-set`. A fost scrisă și funcția `kernel` (sau mai exact două funcții `kernel` denumite `dot`, care implementează kernelul polinomial respectiv pe cel Gaussian, unde parametrul `gamma` poate fi modificat astfel încât să se caute optimul). De menționat aici este că datele au fost normalizate inițial (ca la algoritmul Rocchio), dar aici din intervalul  $[0, 70]$  în intervalul  $[0, 1]$  împărțind ponderile fiecărui exemplu cu 70 (valoarea maximă).

```
#|(define (dot a b [f sqr])
  (f (apply + (map * a b))))|#

(define (dot a b [gamma 1.25])
  (exp (* (- gamma) (apply + (map (lambda (x y) (sqr (- x y))) a b)))))

(define (vectorize set)
  (for/vector ([i set])
    (vector 0 i)))

(define smo-set (vectorize train-set))
```

Fig. 3.7.1 Funcțiile `kernel` și introducerea multiplicatorilor Lagrange.



Pentru implementarea procedurii main, singurul lucru diferit față de pseudocodul din [Fig. 2.3.2] a fost introducerea unei condiții suplimentare de ieșire din algoritm și anume o variabilă max-passes ce contorizează de câte ori după terminarea unei iterații nu s-a modificat niciun multiplicator Lagrange [25]. Din experimente am observat că nu este o mare diferență și este mult mai rapid a ieși atunci când s-a ajuns prima dată la o iterație în care nu s-a modificat nimic. Parametrul  $C$  este o constantă globală ce se poate modifica (din cod) cu scopul de a găsi valoarea mărgii de eroare optime. S-a schimbat stilul funcției, care inițial avea în centru ei o buclă while, în schimb funcția main în Racket a fost implementată într-un stil recursiv reapelând funcția main până când condiția de ieșire este îndeplinită, după cum se poate observa și în [Fig. 3.7.2].

```
(define (main [num-changed 0] [examine-all 1] [iteration 0])
  (printf "Iteration: ~a ~a Changed: ~a \n"
          iteration (vector-length smo-set) num-changed)
  (cond
    [(> max-passes 0) iteration]
    [(not (or (= examine-all 1) (> num-changed 0))) iteration]
    [else
     (set! N (vector-length smo-set))
     (set! num-changed 0)
     (if (= 1 examine-all)
         (for ([i N])
           (set! num-changed (+ num-changed (examine-example i))))
         (for ([i N])
           (define alphai (vfirst (vector-ref smo-set i)))
           (when (and (> alphai 0) (< alphai C))
             (set! num-changed (+ num-changed (examine-example i))))))
     (define temp empty)
     (for ([i smo-set])
       (unless (= (vfirst i) 0)
         (set! temp (cons i temp))))
     (set! smo-set (list->vector temp))
     (when (= num-changed 0)
       (set! max-passes (+ max-passes 1)))
     (if (= 1 examine-all)
         (set! examine-all 0)
         (when (= 0 num-changed)
           (set! examine-all 1)))
     (main num-changed examine-all (+ 1 iteration))]))
```

Fig. 3.7.2 Procedura main din SMO implementată în Racket.

În pseudocodul SMO se recomandă introducerea unei memorii cache de erori, dar deoarece introducerea lui a făcut ca algoritmul să funcționeze mult mai lent chiar și pentru seturile de date generate aleator, acesta nu a mai fost introdus (avea rolul doar de ajuta cu obținerea unei convergențe mai rapide în timp).

```
(define yi (first xyi))
(define Ei (- (f simplifier xi b) yi))

(when (or (and (< (* Ei yi) (- tol)) (< alphas C))
          (and (> (* Ei yi) tol) (> alphas 0))))

(define non-bound empty)
(for ([i smo-set] [index N])
  (unless (or (= (vfirst i) 0) (= (vfirst i) C))
    (set! non-bound (cons (list index i) non-bound))))

(when (= done 0)
  (let loop ([clone (shuffle non-bound)])
    (cond
      [(or (equal? empty clone) (= done 1)) done]
      [else
       (define j (caar clone))
       (define setj (cadar clone))
       (define alphaj (vfirst setj))
       (set! done (take-step i j (list alphas xi yi Ei)))
       (loop (rest clone)))])))
```

Fig. 3.7.3 Partea principală a funcției examineExample.

Partea principală a funcției examineExample este prezentată în [Fig. 3.7.3], unde se începe prin a calcula  $E_i$  folosind funcția de decizie  $f$  (din (2.3.15)) implementată în [Fig. 3.7.4] și se verifică dacă exemplul  $i$  (cel ce a fost ales din procedura main) respectă condițiile KKT din (2.3.23). Mai apoi se formează o listă cu exemplele a căror multiplicatori Lagrange nu au valori la extreme,  $\alpha \in (0, C)$ , iar aceste exemple sunt împrăștiate aleator (folosind funcția shuffle). După ce se alege un al doilea exemplu se apelează funcția takeStep, iar procedeul se repetă până când takeStep returnează 1 sau toate exemplele din această listă sunt testate. Dacă nu a fost obținută o optimizare în acest fel atunci se alege aleator un exemplu din întregul set de date.

```

(define (f set X [bias b])
  (for/fold ([sum bias])
    ([i set])
    (define xy (vsecond i))
    (+ sum (* (vfirst i) (first xy) (dot X (second xy))))))

```

Fig. 3.7.4 Implementarea funcției de decizie.

Într-un final, pentru funcția takeStep la fel este prezentată partea principală în [Fig. 3.7.5]. Dacă cele două exemple alese nu sunt identice și dacă condițiile de liniaritate sunt indeplinite prin  $L \neq H$  atunci se calculează  $\eta$  (care în cod a fost implementată cu semn negativ față de formula scrisă în algoritm, din acest motiv se trece mai departe în cod doar dacă  $\eta < 0$ ). Ulterior se calculează noua valoare a multiplicatorului Lagrange pentru al doilea exemplu ales, se normalizează și dacă modificarea este substanțială atunci se calculează și multiplicatorul Lagrange pentru primul exemplu și se salvează amândoi în vectorul inițial ce conține toți multiplicatorii Lagrange atașați exemplelor.

```

(unless (or (= i j) (= L H))
  (define eta (- (* 2 (dot xi xj)) (dot xi xi) (dot xj xj)))
  (unless (>= eta 0)
    (define aj (- alphaj (/ (* yj (- Ei Ej)) eta)))
    (when (> aj H)
      (set! aj H))
    (when (< aj L)
      (set! aj L))

    (unless (< (abs (- alphaj aj)) tol)
      (define ai (+ alphai (* yi yj (- alphaj aj))))
      (vector-set! smo-set i (vector ai (list yi xi)))
      (vector-set! smo-set j (vector aj xyj)))

```

Fig. 3.7.5 Partea principală a funcției takeStep.

Folosind un vector global pe urma căruia am modificat valorile multiplicatorilor Lagrange nu respectă absolut deloc principiul programării funcționale, însă am făcut totuși acest lucru pentru a obține o performanță mai ridicată (modificarea unei valori într-un vector având o complexitate constantă nu liniară).

Pentru a permite și clasificarea mai multor clase s-a urmat exact procedeul menționat la 2.3.5, formând câte un vector ce conține toate exemplele cu multiplicatorii Lagrange atașați, pentru toate clasele, unde pe rând s-a păstrat câte o clasă cu eticheta 1 în timp ce restul cu eticheta  $-1$  (lucru obținut de funcția `change-class` din [Fig. 3.7.6]).

```
(define (change-class cls dataset)
  (map (lambda (x) (if (= cls (first x)) (cons 1 (rest x)) (cons -1 (rest x)))) dataset))

(define smo-sets
  (for/vector ([i classes])
    (vectorize (change-class i train-set))))
```

Fig. 3.7.6 Crearea seturilor de date în cazul mai multor clase.

Trebuie menționat și că folosirea kernelului polinomial de ordin 2 (cum este și introdus în aplicație prin `[f sqr]` ca parametru pentru funcția `dot` din [Fig. 3.7.2]) este echivalent cu transformarea datelor făcută explicit în 3.6 pentru algoritmul Rocchio.

Matematic acest lucru poate fi demonstrat ca în relațiile de mai jos, unde plecând de la doi vectori  $(a_1, a_2), (b_1, b_2)$  și observând că rezultatul este același ori că se ridică la pătrat produsul scalar al acestor doi vectori, ori că se mapează din 2d în 3d vectorii și mai apoi se face produsul scalar.

$$\begin{aligned} [(a_1, a_2) \cdot (b_1, b_2)]^2 &= [a_1 b_1 + a_2 b_2]^2 \\ &= a_1^2 b_1^2 + 2a_1 b_1 a_2 b_2 + a_2^2 b_2^2 \\ &= (a_1^2, \sqrt{2}a_1 a_2, a_2^2) \cdot (b_1^2, \sqrt{2}b_1 b_2, b_2^2) \end{aligned}$$

A fost menționat și că funcția `kernel` se aplică direct produsului scalar, însă în cazul kernelului Gaussian, de fapt este un fel de pseudo produs scalar fiindcă de fapt se calculează distanța euclidiană dintre  $|x - y|$  și mai apoi se pasează acest lucru funcției  $e^{-\gamma|x-y|^2}$  (de aceasta a fost implementat și două funcții diferite `dot`).

### 3.8 Alegerea parametrilor pentru algoritmul SVM

Pe parcurs au fost menționați doi parametri importanți ce trebuie modificați pentru algoritmul SVM și anume  $C$  și  $\gamma$ .  $C$  este un parametru care determină cât de mult să se țină cont de zgomot, pentru o valoare mai mare un zgomot are o influență mai mare chiar dacă marginea hiperplanului devine mai mică în acest fel.  $\gamma$  este parametrul ce apare în cadrul kernelului Gaussian și reprezintă cât de mult să influențeze un exemplu clasa lui.

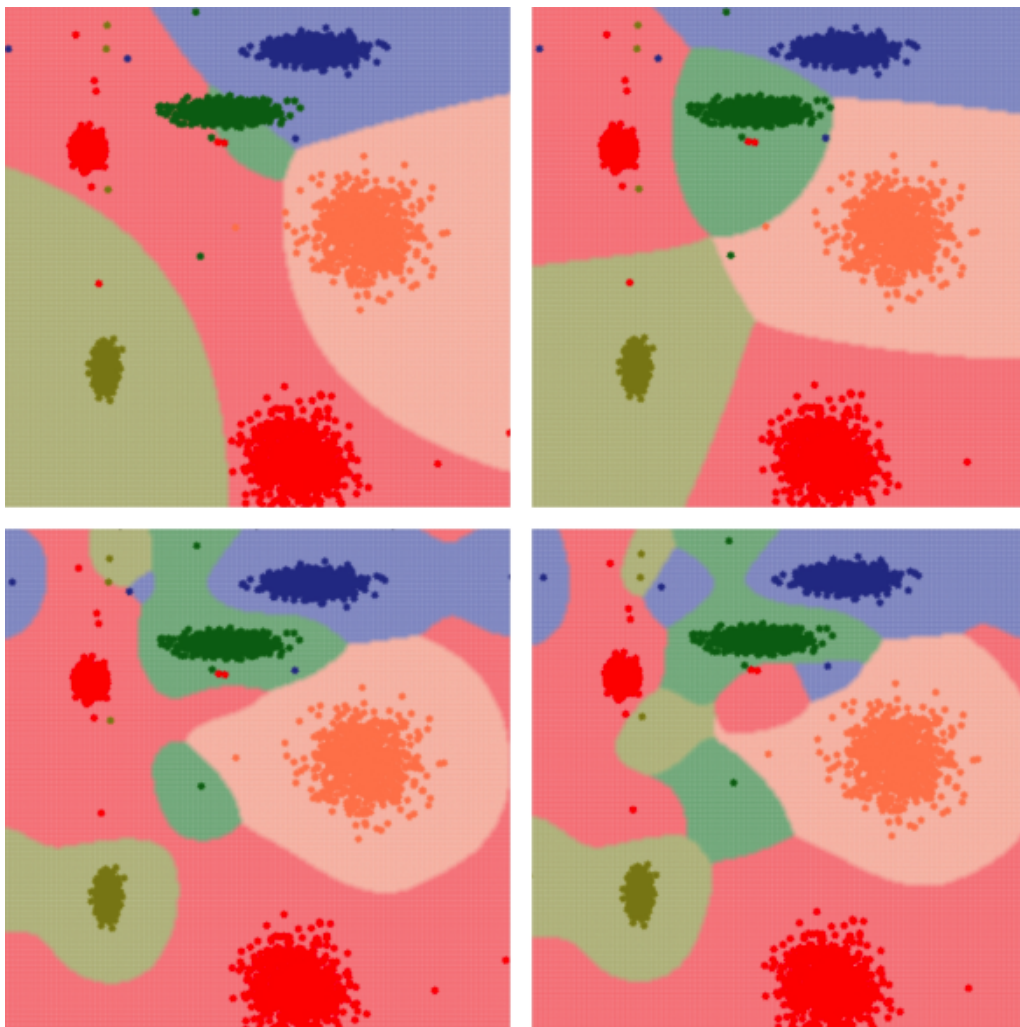


Fig. 3.8.1 Clasificarea SVM folosind diferite valori pentru  $C$  și  $\gamma$ .

În continuare se vor prezenta vizual câteva cazuri de alegere a acestor parametrii pentru a înțelege mai bine funcționarea lor. [Fig. 3.8.1] prezintă 4 situații:

- în stânga sus  $C = 1$  și  $\gamma = 0.05$
- în dreapta sus  $C = 1$  și  $\gamma = 1$
- în stânga jos  $C = 1$  și  $\gamma = 30$
- în dreapta jos  $C = 30$  și  $\gamma = 30$ .

În cazul în care  $C = 1$  și  $\gamma = 0.05$  se poate observa că algoritmul doar reușește să localizeze clasele, clasificarea însă lasă mult de dorit. Mai apoi în cazul  $C = 1$  și  $\gamma = 1$  clasificarea este una ce pare bună, punctele de zgomot sunt practic ignorate. În ambele cazuri când  $\gamma = 30$  algoritmul reușește să formeze niște zone chiar și pentru punctele de zgomot, mai ales pentru cazul în care  $C = 30$  când a reușit să formeze perfect câte o zonă pentru fiecare punct de zgomot (în contrast dacă  $C = 1$  câteva puncte totuși au fost omise).

Un lucru important în aceste simulări de mai sus este că au fost afișate punctele de antrenament, iar mai apoi a fost luat fiecare punct disponibil (la o distanță considerabilă) și astfel a fost determinate zonele date de clasificarea algoritmului. Dacă am lua totuși un alt set de date pentru a testa propriu-zis algoritmul atunci trebuie să ținem cont că clasificarea a fost una perfectă (în ultimul caz), dar pentru datele de antrenament, practic algoritmul a tocit (sau a făcut over-fitting). În primul caz însă nu a reușit să învețe, decât să localizeze zonele claselor (mai exact a făcut under-fitting). În practică ne-am dori să nu fim de nicio parte extremă, de aceasta pentru fiecare set de date trebuie să încercăm să găsim valorile optime ale acestor parametrii astfel încât să nu fie nici prea mari, dar nici prea mici.

### 3.9 Implementarea evaluării algoritmilor

Pentru a reuși să evaluăm algoritmi au fost introduse în cod 3 metrici (prezentate în 2.4.2) adică acuratețea, precizia și sensivitatea. Adicional s-a măsurat și timpul antrenării folosind funcția `time` din limbajul Racket. Aceste lucruri sunt prezentate în [Fig. 3.9.1] pentru algoritmul SVM (la Rocchio situația este foarte similară de aceea a fost aleasă varianta aceasta).

```
(define (update-matrix r p matrix)
  (define (index r p i)
    (cond
      [(= r p i) 0]
      [(and (= r i) (not (= r p))) 1]
      [(and (= p i) (not (= r p))) 2]
      [else 3]))
  (for/list ([classX matrix] [i classes])
    (list-update classX (index r p i) add1)))

(define (test-svm test-set)
  (let loop ([clone test-set]
             [error-matrix (make-list (length classes) (make-list 4 0))]
             [corrects 0])
    (cond
      [(equal? empty clone) (list error-matrix corrects)]
      [else
       (define reality (caar clone))
       (define element (cadar clone))

       (define prediction
         (caar (sort
                 (for/list ([i classes])
                   (list i (f (vector-ref smo-sets (index-of classes i))
                             element (vector-ref bs (index-of classes i))))
                 > #:key (lambda (x) (second x)))))

       (loop (rest clone) (update-matrix reality prediction error-matrix)
              (if (= reality prediction) (+ 1 corrects) corrects)))]))
```

Fig. 3.9.1 Implementarea metricilor pentru evaluarea algoritmului SVM.

Pentru început este creată o funcție `update-matrix` ce primește ca parametrii matricea de eroare împreună cu clasa reală (`r`), respectiv clasa predicționată (`p`) aferentă exemplului și returnează matricea de eroare actualizată după cum a fost descris în 2.4.2. De menționat este că matricea implementată în Racket este de fapt o listă de forma (TP, FN, FP, TN).

Funcția test-svm primește ca parametrul setul de date pentru testare din care ia pe rând (recursiv) fiecare exemplu de pe urma căruia se preia clasa reală (reality) și exemplul propriu-zis (element). Mai departe se face predicția ținând cont doar de exemplu fără clasa sa, și anume: se predicționează exemplul curent cu fiecare funcție de decizie (fiecare clasă a produs o funcție de decizie) și se returnează clasa a cărei funcție de decizie a produs cea mai mare valoare, aceasta fiind și predicția algoritmului. Mai apoi se pasează clasa reală împreună cu clasa predicționată funcției update-matrix, se actualizează matricea de eroare și se trece la exemplul următor.

Diferența la algoritmul Rocchio este că în loc de a funcția de decizie din algoritmul SVM se calculează pentru un exemplu care centroid se află la distanța cea mai mică față de el, clasa centroidul fiind și predicția ulterior.

### 3.10 Utilizarea aplicației

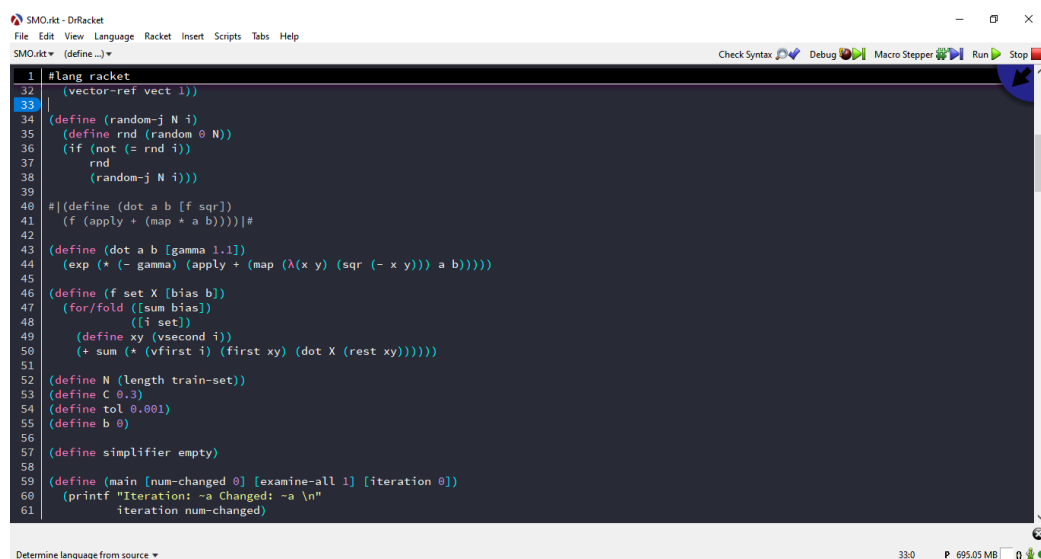
Bineînțeles pentru a putea utiliza aplicația mai întâi trebuie instalat limbajul de programare Racket folosind [31], acesta vine la pachet și cu mediul de programare DrRacket (se recomandă instalarea versiunii 8.1 (sau 8.0), folosită și pe parcursul dezvoltării acestei aplicații).

În starea actuală aplicația constă în mai multe fișiere cu denumiri sugestive, precum Rocchio\_Reuters respectiv SVM\_Reuters ce permit testarea celor doi algoritmi folosind seturile de date bazate pe documente Reuters, mai apoi există și varianta rulării algoritmului doar pentru seturile de date generate aleator cu scopul de a observa vizual cum se comportă aceștia în scripturile Rocchio\_MultiClass SVM\_MultiClass (acestea au implementată și o interfață grafică). Motivul pentru care au fost folosite mai multe scripturi este că citirea este diferită de la un tip de fișier la altul, iar pentru primele variante menționate nu a fost posibilă imple-



mentarea unei interfețe vizuale (datele fiind într-o dimensiune de ordinul 1309). Adicional a fost adăugat și codul (Generate\_Dataset) ce generează aleator primul tip de date și scriptul Rocchio\_2d3d pentru a putea observa vizual maparea datelor din 2d în 3d detaliat în 3.6). Însă nu este necesară generarea datelor manual (decât dacă se dorește obținerea unor date noi), deoarece sunt incluse deja fișierele cu seturile de date în aplicație.

La pornirea aplicației veți fi întâmpinați în DrRacket cu o fereastră similară ca în [Fig. 3.10.1] (posibil cu o altă temă), iar tot ce trebuie făcut pentru a rula este de a apăsa butonul verde Run din colțul din dreapta sus. Adicional se poate consulta și [17] pentru mai multe detalii despre cum funcționează acest mediu de programare. Pentru a varia parametrii algoritmului SVM (în special  $C$  și  $\gamma$ ) se poate modifica în cod variabilele cu aceleași denumiri.



```

1 #lang racket
2
3 (vector-ref vect i))
4
5 (define (random-j N i)
6   (define rnd (random 0 N))
7   (if (not (= rnd i))
8       rnd
9       (random-j N i)))
10
11 #|(define (dot a b [f sqr])
12   (f (apply + (map * a b))))|#
13
14 (define (dot a b [gamma 1.1])
15   (exp (* (- gamma) (apply + (map (lambda (x y) (sqr (- x y))) a b)))))
16
17 (define (f set X [bias b])
18   (for/fold ([sum bias])
19     ([i set])
20     (define xy (vsecond i))
21     (+ sum (* (vfirst i) (first xy) (dot X (rest xy))))))
22
23 (define N (length train-set))
24 (define C 0.3)
25 (define tol 0.001)
26 (define b 0)
27
28 (define simplifier empty)
29
30 (define (main [num-changed 0] [examine-all 1] [iteration 0])
31   (printf "Iteration: ~a Changed: ~a ~n"
32     iteration num-changed)
33 )

```

Fig. 3.10.1 Mediul de dezvoltare DrRacket.

### 3.11 Rezultate experimentale

Rezultatele prezentate în cele ce urmează au fost obținute testând algoritmi Rocchio respectiv SVM pentru setul de date bazat pe niște documente Reuters, descris și în 3.4.2. Antrenarea a avut loc pe 4702 de exemple, iar testarea pe 2305 de exemple, toate având o dimensiune de ordinul 1309 (vectori cu 1309 de elemente) și în total au fost 14 clase.

Clasa	Nr. elemente	TP (r p)	FN (r !p)	FP (!r p)	Precizia(%)	Sensitivitatea(%)	Acuratețea(%)
0	14	3	11	34	8.11	21.43	21.43
11	196	22	174	47	31.88	11.22	11.22
12	89	32	57	21	60.38	35.96	35.96
13	55	21	34	10	67.74	38.18	38.18
14	54	29	25	39	42.65	53.70	53.70
15	1192	604	588	8	98.69	50.67	50.67
17	94	77	17	427	15.28	81.91	81.91
18	162	44	118	53	45.36	27.16	27.16
21	59	4	55	48	7.69	6.78	6.78
22	171	48	123	76	38.71	28.07	28.07
23	11	1	10	45	2.17	9.09	9.09
31	62	18	44	146	10.98	29.03	29.03
33	84	70	14	306	18.62	83.33	83.33
41	108	95	13	23	80.51	87.96	87.96
Media	167.93	76.29	91.64	91.64	<b>37.77</b>	<b>40.32</b>	<b>45.43</b>

Fig. 3.11.1 Rezultatele algoritmului Rocchio.

Clasa	Nr. elemente	TP (r p)	FN (r !p)	FP (!r p)	Precizia(%)	Sensitivitatea(%)	Acuratețea(%)
0	14	6	8	9	40.00	42.86	42.86
11	196	92	104	88	51.11	46.94	46.94
12	89	77	12	13	85.56	86.52	86.52
13	55	30	25	31	49.18	54.55	54.55
14	54	31	23	25	55.36	57.41	57.41
15	1192	1135	57	117	90.65	95.22	95.22
17	94	50	44	18	73.53	53.19	53.19
18	162	123	39	45	73.21	75.93	75.93
21	59	15	44	21	41.67	25.42	25.42
22	171	105	66	64	62.13	61.40	61.40
23	11	0	11	4	0.00	0.00	0.00
31	62	18	44	24	42.86	29.03	29.03
33	84	57	27	44	56.44	67.86	67.86
41	108	97	11	12	88.99	89.81	89.81
Media	167.93	131.14	36.79	36.79	<b>57.91</b>	<b>56.15</b>	<b>78.09</b>

Fig. 3.11.2 Rezultatele algoritmului SVM în cazul cel mai favorabil.

În [Fig. 3.11.1] și [Fig. 3.11.2] sunt prezentate rezultate obținute pentru cei doi algoritmi, cazul favorabil la SVM este reprezentat de parametrii  $C = 10$  și  $\gamma = 1.25$  (va fi explicat totuși ulterior cum s-a ajuns la acest lucru).

Pe coloane se observă numărul de elemente pentru fiecare clasă, urmat de contoarele TP, FN și FP (descrise în 2.4.2 la matricea de eroare). Numărul elementelor corect predicționate pentru fiecare clasă este reprezentat de (TP), urmat de FN care reprezintă de câte ori a fost prezisă greșit clasa deși ar fi trebuit prezisă corect, respectiv FP care reprezintă de câte ori a fost prezisă clasa deși nu trebuia.

Ultimele 3 coloane reprezintă metricile implementate, din care se observă mai ales la algoritmul SVM că Precizia și Sensivitatea au o valoare relativ mică comparativ cu acuratețea, mai ales pentru algoritmul SVM. Acest lucru se datorează faptului că unele clase conțin mult prea puține exemple și algoritmul nu reușește să le învețe. Trebuie reamintit că precizia =  $\frac{TP}{TP+FP}$ , iar sensivitatea =  $\frac{TP}{TP+FN}$ , din care se observă că pentru a calcula aceste două metrici se ține cont și de exemplele altor clase (prin FN și FP), pe când la acuratețe se contorizează simplu câte predicții corecte au fost făcute și se împarte la numărul de elemente. Totuși putem observa o îmbunătățire semnificativă (mai ales în cazul acurateții, din cauza menționată anterior) față de algoritmul Rocchio observabilă în [Fig. 3.11.3].

Motivul pentru care în această comparație algoritmul SVM se comportă mult mai bine este din cauza obstacolului principal întâmpinat de către Rocchio și anume că nu poate clasificata date neseparabile liniar, în schimb folosind kernelul Gaussian SVM reușește să depășească acest obstacol. Au fost încercate și alte tipuri de kerneluri precum cel polinomial sau un kernel sigmoidal  $k(x_n, x_m) = \frac{1}{1+e^{-x_n \cdot x_m}}$ , însă rezultatele au fost extrem de slabe, nefiind posibilă găsirea unui spațiu de reprezentare în care să poată fi separate datele, de aceasta majoritatea simulărilor sunt prezentate folosind kernelul Gaussian.

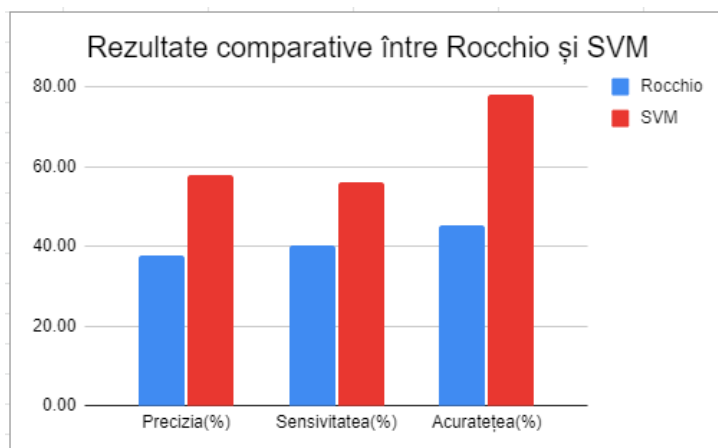


Fig. 3.11.3 Rezultate comparative: Rocchio vs SVM.

Pentru a găsi cazul cel mai favorabil al parametriilor din cadrul SVM a fost realizat un mini grid-search, prin care am modificat valorile celor doi parametri și am măsurat acuratețea până când aceasta a început să scadă reprezentând că deja mai încolo se face over-fitting, acest lucru a fost prezentat în [Fig. 3.11.4].

C	Gamma	Acuratețea(%)
1	0.5	74.73
3	0.5	76.18
5	0.5	74.95
5	0.75	76.69
10	1	77.29
<b>10</b>	<b>1.25</b>	<b>78.09</b>
12.5	1	78.05
12.5	1.25	77.75
15	15	77.16

Fig. 3.11.4 Acuratețea algoritmului SVM pentru difeți parametri.

A fost măsurat și timpul pentru algoritmul SVM, dar acesta s-a comportat destul de diferit chiar și pentru aceeași parametri (motivul principal fiind alegerea aleatoare a multiplicatorilor Lagrange), însă în medie o simulare întreagă pentru SVM a durat între 100 și 200 de minute. La Rocchio simularea a durat 20 de secunde (dar și rezultate au fost slabe după cum s-a văzut).

## Capitolul 4

### Concluzii și dezvoltări ulterioare

Obiectivele propuse în această temă au fost în mare parte îndeplinite, însă este loc mult de îmbunătățiri. În primul rând au fost implementați cei doi algoritmi de clasificare, Support Vector Machines în special cu scopul de a înțelege și observa cum este posibilă clasificarea datelor neseparabile liniar transformându-le într-o altă dimensiune și Rocchio cu scopul de a putea studia comparativ cei doi algoritmi.

A fost dezvoltată o interfață grafică pentru a putea observa vizual rezultate algoritmilor, iar în plus a fost implementată și o tehnică preluată de la algoritmul Support Vector Machines în algoritmul Rocchio pentru a putea obține o bună intuiție despre cum poate clasifica un algoritm datele într-un alt spațiu de reprezentare chiar dacă în spațiul original nu reușește acest lucru, mai exact a fost implementată o interfață ce face posibilă vizualizarea mapării dintr-un spațiu bi-dimensional într-unul tridimensional. Mai apoi pentru a putea evalua algoritmi și de a face o comparație între ei au fost implementate și niște metrice cu scopul de a măsura eficiența acestor algoritmi.

În cele din urmă au fost dobândite și cunoștințe noi despre programarea funcțională și despre limbajul de programare Racket. Însă pentru realizarea aplicației nu au fost

folosite doar tehnici aparținând programării funcționale din motive de performanță după cum a fost și menționat la implementarea algoritmului SVM. Mai exact a fost mult mai eficient a folosi vectori în schimbul listelor și de a modifica direct acești vectori ce utilizează setul de date cu multiplicatorii Lagrange atașați. Însă această experiență nu a fost una negativă deoarece este bine să putem decide când o paradigmă de programare este utilă sau nu și mai ales când este vorba despre obținerea unei performanțe mai bune.

Algoritmul SVM a fost destul de lent, după cum a fost și menționat o simulare a durat în medie între 100 și 200 de minute, depinzând bineînțeles și de parametrii, din această cauză tema implementată are aplicabilitate mai mult didactică și de înțelege mai bine funcționarea algoritmilor. Totuși în viitor se propune optimizarea algoritmului prin analizarea mai în detaliu a complexității (în special de timp) și de a îmbunătății unde este cazul cu scopul de a obține adevăratul potențial al acestui algoritm, ce are multe aplicabilități în practică precum clasificarea de documente, filtrarea spam-urilor și altele menționate în partea introductivă.

Tot în legătură cu performanța în timp se propune și paralelizarea algoritmului SVM, prin antrenarea simultană a diferitelor bucăți din setul de date, practic un fel de Batch Gradient Descent pentru rețelele neuronale detaliat în [1].

O altă problemă ce s-a văzut la algoritmul SVM este alegerea parametrilor  $C$  și  $\gamma$ , pentru a rezolva acest lucru se dorește implementarea unor tehnici adaptive de actualizare pe parcursul simulării a acestor parametrii, după cum a și fost implementat în [13].

# Capitolul 5

## Bibliografie

1. Tom Mitchell, Machine Learning, McGraw Hill, 1997.
2. Stuart Russell, Peter Norvig, Artificial Intelligence: A modern approach, 3rd edition, 2009.
3. [https://psychology.wikia.org/wiki/Concept\\_learning](https://psychology.wikia.org/wiki/Concept_learning)
4. [https://en.wikipedia.org/wiki/Supervised\\_learning](https://en.wikipedia.org/wiki/Supervised_learning)
5. <https://brilliant.org/wiki/feature-vector>
6. [https://en.wikipedia.org/wiki/Nearest\\_centroid\\_classifier](https://en.wikipedia.org/wiki/Nearest_centroid_classifier)
7. Christopher D. Manning et al, Introduction to Information Retrieval, Cambridge University Press, 2008.
8. Gareth James et al, An Introduction to Statistical Learning, Springer, 2015.
9. <http://fourier.eng.hmc.edu/e176/lectures/ch9/node5.html>
10. <https://stats.stackexchange.com/questions/23391/how-does-a-support-vector-machine-svm-work>
11. <https://towardsdatascience.com/metrics-to-evaluate-your-machine-learning-algorithm-f10ba6e38234>
12. Bernhard Scholkopf, Alexander Smola, Learning with Kernels, Support Vector Machines, MIT Press, 2002.

13. Daniel Morariu, PhD thesis: Contribution to Automatic Knowledge Extraction from Unstructured Data, 2007.
14. John Platt, Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines, 1998.
15. Kristin Benett, Erin Bredensteiner, Duality and Geometry in SVM Classifiers, 2000.
16. J.J. Rocchio, Relevance Feedback in Information Retrieval, 1965.
17. <https://docs.racket-lang.org>
18. <https://elf.cs.pub.ro/pp/20/laboratoare/racket/intro>
19. <https://stats.stackexchange.com/questions/31066/what-is-the-influence-of-c-in-svms-with-linear-kernel>
20. <https://www.quora.com/What-are-C-and-gamma-with-regards-to-a-support-vector-machine>
21. <https://www.youtube.com/watch?v=efR1C6CvbmE>
22. [https://www.youtube.com/watch?v=\\_PwhiWxHK8o](https://www.youtube.com/watch?v=_PwhiWxHK8o)
23. <https://beautifulracket.com/explainer/functions.html>
24. <https://beautifulracket.com/appendix/why-racket-why-lisp.html>
25. <http://cs229.stanford.edu/materials/smo.pdf>
26. <https://stackoverflow.com/questions/67831104/squared-euclidean-distance-in-racket-using-a-hash-set>
27. <https://towardsdatascience.com/confusion-matrix-for-your-multi-class-machine-learning-model-ff9aa3bf7826>
28. [https://en.wikipedia.org/wiki/Racket\\_\(programming\\_language\)](https://en.wikipedia.org/wiki/Racket_(programming_language))
29. [https://en.wikipedia.org/wiki/Euclidean\\_distance](https://en.wikipedia.org/wiki/Euclidean_distance)
30. [https://en.wikipedia.org/wiki/Dot\\_product](https://en.wikipedia.org/wiki/Dot_product)
31. <https://download.racket-lang.org><sup>1</sup>

---

<sup>1</sup>toate link-urile au fost accesate și verificate o ultimă dată în 22.06.2021