Universitatea Tehnică "Gheorghe Asachi" din Iași
Facultatea de Automatică și Calculatoare
Domeniul Calculatoare și Tehnologia Informației
Specializarea Tehnologia Informației

INTELIGENȚĂ ARTIFICIALĂ

Detecția site-urilor folosite în atacuri de tip phishing, folosind mașini cu vectori suport optimizate cu ajutorul unui algoritm evolutiv

Studenți,
Murariu-Tănăsache Iulian - 1411B
Enache Ștefan - 1411B

An universitar 2022-2023

1. Descrierea problemei considerate

Un atac de tip phishing este o modalitate de manipulare și decepție pentru a obține date confidențiale, cum ar fi date de acces pentru aplicații de tip bancar, aplicații de comerț electronic sau informații referitoare la carduri de credit. În astfel de atacuri, de multe ori infractorii se folosesc de site-uri care apar ca fiind legitime pentru a convinge utilizatorii să-și dezvălui datele personale.

Pentru a evita accesarea acestori site-turi, vom antrena un algoritm bazat pe mașini cu vectori suport pe un set mare de date[1] pentru a putea fi folosit în detecția site-urilor malițioase.

2. Aspecte teoretice privind algoritmul

Mașinile cu vectori suport (engl. "Support Vector Machines", SVM) reprezintă o metodă de învăţare de nouă generaţie (Shawe-Taylor & Cristianini, 2000), cu fundamentare matematică riguroasă, bazată pe conceptul de maximizare a „marginii" care separă instanţele din două clase diferite, iar maximizarea este rezolvată analitic, nu empiric.

Datorită acestei fundamentări, maşinile cu vectori suport au demonstrat performanţe foarte bune pentru probleme reale cum ar fi clasificarea textelor, recunoaşterea caracterelor scrise de mână, clasificarea imaginilor etc. În general, sunt considerate unele dintre cele mai bune metode de clasificare cunoscute la ora actuală.

Marginea este distanţa dintre dreptele paralele cu dreapta de separare care ating cel puţin una din instanţele fiecărei clase. Pentru a formaliza această descriere, să considerăm că o suprafaţă de separare este descrisă de următoarea ecuaţie:

$$(w \cdot x) + b = 0 \qquad (1)$$

unde x este vectorul ce reprezintă caracteristicile obiectului ce trebuie clasificat, w este un vector de *"greutăţi"* care ajută la clasificare, iar b este un scalar numit *bias*.

Rezolvarea problemei de clasificare cu SVM se reduce la determinarea parametrilor *w* și *b* astfel încât marginea să fie maximă pentru toate instanţele din mulţimea de antrenare. Apoi, o instanţă nouă *z* va fi clasificată în una din cele două clase (–1 sau 1) cu ajutorul funcţiei discriminant:

$$f_d(z) = \mathrm{sgn}((w \cdot z) + b) \qquad (2)$$

Marginea poate fi exprimată cu ajutorul expresiei: $m = 2 / ||w||$. Scopul este de a maximiza această expresie, însă pentru a ne folosi de conceptul de problemă duală, putem în schimb minimiza expresia $\frac{1}{2} ||w||^2$, astfel că problema primară ce trebuie rezolvată este:

$$\min_{w,b} \frac{1}{2} ||w||^2 \qquad (3)$$

respectând constrângerile:

[1] Link către dataset: https://archive.ics.uci.edu/ml/datasets/Website+Phishing

$$y_i((w \cdot x_i) + b) \geq 1, i = 1 \ldots N, \quad (4)$$

unde N este numărul de instanţe din mulţimea de antrenare, iar y_i este rezultatul aşteptat pentru o intrare x_i .

Pe baza acestor observaţii se poate ajunge la următorul lagrangian:

$$L(w,b,\alpha) = \frac{1}{2} ||w||^2 - \sum_{i=1}^{N} \alpha_i (y_i((w \cdot x_i) + b) - 1) \quad (5)$$

În final, după ce se aplică condiţiile Karush-Kuhn-Tucker şi se fac înlocuiri în expresia problemei duale, se ajunge la următoarea formulare pentru o maşină cu vectori suport liniară:

$$\max_{\alpha_i} \left(\sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j (\mathbf{x}_i \cdot \mathbf{x}_j) \right) \quad (6)$$

respectând constrângerile:

$$\alpha_i \geq 0, i = 1..N, \quad (7)$$

$$\sum_{i=1}^{N} \alpha_i y_i = 0. \quad (8)$$

După rezolvarea problemei duale şi calcularea multiplicatorilor Lagrange şi al bias-ului, funcţia de ieşire a algoritmului poate fi scrisă în felul următor:

$$f(x) = \text{sign} \left(\sum_{i=1}^{N} \alpha_i y_i x \cdot x_i + b \right) \quad (9)$$

3. Modalitatea de rezolvare

Pentru determinarea multiplicatorilor Lagrange şi a bias-ului, s-a folosit un algoritm genetic, care urmează arhitectura din figura 1.

Cromozomii reprezintă un vector de lungime *N,* unde fiecare genă reprezintă un multiplicator Lagrange. Se începe cu o populaţie în care fiecare cromozom este generat aleatoriu şi se aplică algoritmul iterativ până se atinge un număr propus de iteraţii(epoci) sau cel mai adaptat individ al populaţiei nu se schimbă în 3 iteraţii. Elitismul asigură faptul că la fiecare epocă reţinem cel mai bun individ din populaţie, iar restul sunt aleşi prin turnir, astfel că

rămânem cu jumătate din populația inițială. Cealaltă jumătate se completează cu copii obținuți prin încrucișare aritmetică. Asupra copiilor se aplică un operator de mutație cu o șansă de 2% care interschimbă două gene între ele.

Funcția de adaptare propusă este aceeași cu ecuația *(6),* însă a fost rescrisă pentru a fi minimizată:

$$F(\boldsymbol{\alpha}) = -\sum_{i=1}^{N} \alpha_i + \tfrac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j x_i \cdot x_j \quad (10)$$

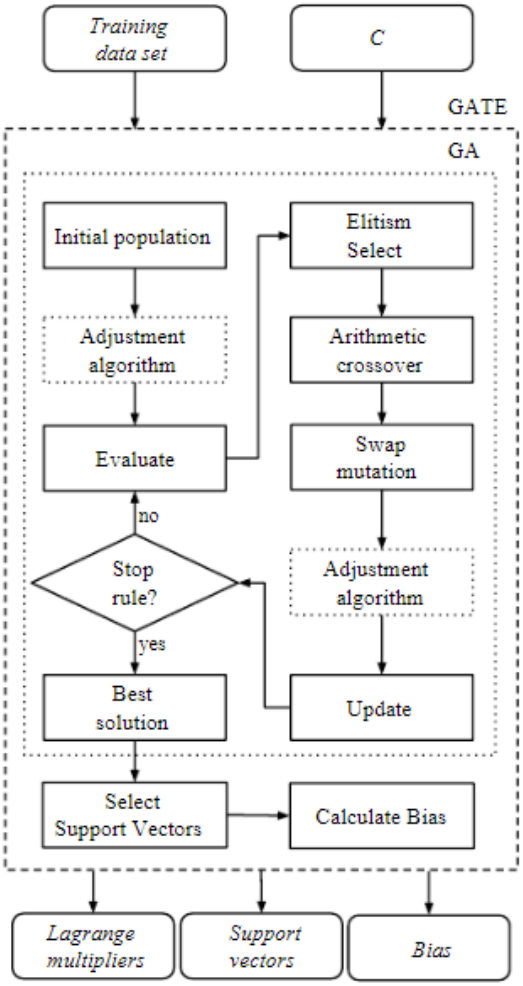

Figura 1. Structura algoritmului genetic

La fiecare nouă populație trebuie asigurat faptul că fiecare individ respectă constrângerile (7) și (8). Ecuația (7) poate fi ușor asigurată de algoritmul genetic prin modul în care se inițializează prima oară populația, însă pentru constrângerea (8) s-a adăugat un pas suplimentar după generarea fiecărei noi populații. Acest pas este constituit de un algoritm de ajustare.

După obținerea multiplicatorilor Lagrange (reprezentând individul cel mai adaptat la finalul algoritmului genetic), bias-ul poate fi calculat conform ecuației:

$$b = 1/N \sum_{i=1}^{N} (y_i - \sum_{j=1}^{N} y_j \alpha_j \, x_i \cdot x_j) \quad (11)$$

4. Explicarea părților semnificative de cod

Implementarea am ales să o facem în python3. Am implementat algoritmul genetic GATE[2] , cu mici modificări, pentru a obține multiplicatorii Lagrange și bias-ul, iar pe baza lor și a setului de antrenare, SVM-ul poate decide dacă un site nou este periculos sau nu. La final am evaluat algoritmul pe un set de testare ca să observăm acuratețea, precizia și scorul F1.

Datele noastre de intrare au constituit dintr-un fișier de tip arf, unde fiecare linie reprezenta o intrare, un site cu caracteristicile sale, iar ultimul număr era verdictul dacă acel site era malițios sau nu. Caracteristicile[3] sunt prezentate în document și apar în ordine.

[2] Algoritmul este descris în acest articol: https://www.researchgate.net/publication/309565420_Evolutionary_Support_Vector_Machines_A_Dual_Approach

[3] Caracteristicile sunt explicate mai în detaliu în acest document: https://archive.ics.uci.edu/ml/machine-learning-databases/00327/Phishing%20Websites%20Features.docx

```
@relation phishing

@attribute having_IP_Address  { -1,1 }
@attribute URL_Length   { 1,0,-1 }
@attribute Shortining_Service { 1,-1 }
@attribute having_At_Symbol   { 1,-1 }
@attribute double_slash_redirecting { -1,1 }
@attribute Prefix_Suffix  { -1,1 }
@attribute having_Sub_Domain  { -1,0,1 }
@attribute SSLfinal_State  { -1,1,0 }
@attribute Domain_registeration_length { -1,1 }
@attribute Favicon { 1,-1 }
@attribute port { 1,-1 }
@attribute HTTPS_token { -1,1 }
@attribute Request_URL  { 1,-1 }
@attribute URL_of_Anchor { -1,0,1 }
@attribute Links_in_tags { 1,-1,0 }
@attribute SFH  { -1,1,0 }
@attribute Submitting_to_email { -1,1 }
@attribute Abnormal_URL { -1,1 }
@attribute Redirect  { 0,1 }
@attribute on_mouseover  { 1,-1 }
@attribute RightClick  { 1,-1 }
@attribute popUpWidnow  { 1,-1 }
@attribute Iframe { 1,-1 }
@attribute age_of_domain  { -1,1 }
@attribute DNSRecord   { -1,1 }
@attribute web_traffic  { -1,0,1 }
@attribute Page_Rank { -1,1 }
@attribute Google_Index { 1,-1 }
@attribute Links_pointing_to_page { 1,0,-1 }
@attribute Statistical_report { -1,1 }
@attribute Result  { -1,1 }


@data
-1,1,1,1,-1,-1,-1,-1,-1,1,1,-1,1,-1,1,-1,-1,-1,0,1,1,1,1,-1,-1,-1,-1,1,1,-1,-1
1,1,1,1,1,-1,0,1,-1,1,1,-1,1,0,-1,-1,1,1,0,1,1,1,1,-1,-1,0,-1,1,1,1,-1
1,0,1,1,1,-1,-1,-1,-1,1,1,-1,1,0,-1,-1,-1,-1,0,1,1,1,1,1,-1,1,-1,1,0,-1,-1
1,0,1,1,1,-1,-1,-1,1,1,1,-1,-1,0,0,-1,1,1,0,1,1,1,1,-1,-1,1,-1,1,-1,1,-1
1,0,-1,1,1,-1,1,1,-1,1,1,1,1,0,0,-1,1,1,0,-1,1,-1,1,-1,-1,0,-1,1,1,1,1
```

Figura 2. Structura fișierului de intrare

Aceste date au fost parsate și pe baza lor s-a construit o listă de tuple de forma:

([atribut1, atribut2, …, atribut30], rezultat),

unde prima listă constă în cele 30 de atribute despre site, iar rezultatul este verdictul despre legitimitatea lui.

```
def parsare(data):
    list = []
    list2 = []
    list3 = []
    for i in data[0]:
        linie = ""
        for j in i:
            linie = linie + str(j)
            linie.split(")")
        list.append(linie)
    for line in list:
        list2.append(line.split("b'"))
    for lin in list2:
        for i in lin:
            list3.append(re.findall(r'[+-]?\d+', i))
    count = 0
    new_list = []
    vector_date = []
    for i in range(1, len(list3)):
        if list3[i] == ' ':
            list3[i] = int(list3[i + 1])
        if count < 31:
            new_list.append(int(list3[i][0]))
            count += 1
        elif count == 31:
            # print(new_list)
            #vector_date.append(new_list)
            vector_date.append((new_list[:len(new_list) - 1], new_list[len(new_list) - 1]))
            new_list = []
            count = 0
    #print(vector_date)
    return vector_date
```

Figura 3. Funcția de parsare a datelor de intrare

Algoritmul genetic GATE a fost implementat conform *Figura 1*. Clasa *GATE* are definiți toți pașii ca metode și o metodă numită *run_algorithm()* care execută pașii în ordine timp de un număr de epoci primit ca argument in constructor. Algoritmul se oprește ori când s-a atins numărul de epoci sau când cel mai adaptat individ nu se schimbă. La final multiplicatorii Lagrange sunt luați ca fiind genele celui mai adaptat individ și se calculează bias-ul conform ecuației (11)

```
class GATE:
    def __init__(self, dataset, epochs, mutation_rate, pop_size):
        self.x = [np.asarray(x[0]) for x in dataset]
        self.y = [x[1] for x in dataset]
        self.dataset = dataset
        self.population_size = pop_size
        self.length = len(dataset)
        self.alpha_pop = [[random.random() for y in range(0,self.length)] for x in range(0, self.population_size)]
        self.b = random.random()
        self.mutation_rate = mutation_rate
        self.epochs = epochs
        self.early_stop = 3
        self.error = 0.001
```

Figura 4. Clasa GATE și constructorul acesteia

```
def run_algorithm(self):
    curr_fitness = 0
    no_change = 0
    #adjustment here
    new_pop = self.adjustment(self.alpha_pop)
    for ep in range(0, self.epochs):
        best_pop = self.elitism(new_pop)
        best_pop_fitness = self.fitness(best_pop)
        if np.abs(curr_fitness - self.error) <= best_pop_fitness <= np.abs(curr_fitness + self.error):
            no_change += 1
            if no_change > self.early_stop:
                break
        new_pop = self.tournir_selection(new_pop)
        new_pop = self.crossover(new_pop)
        new_pop.append(best_pop)
        new_pop = self.mutation(new_pop)
        #adjustment here
        new_pop = self.adjustment(new_pop)
        self.alpha_pop = new_pop
    lagrange_mul = self.elitism(self.alpha_pop)
    return lagrange_mul, self.bias_computation(lagrange_mul)
```

Figura 5. Pașii algoritmului genetic GATE

Algoritmul de ajustare este necesar pentru a asigura respectarea constrângerii (8). Strategia este de a calcula suma la fiecare iterație. Dacă suma este pozitivă, atunci se scade o genă aleasă aleatoriu care să contribui pozitiv la sumă (se scade dintr-un α_i, unde $y_i = +1$). Dacă suma este negativă, se scade dintr-o genă ce contribuie negativ la sumă (se scade dintr-un α_i, unde $y_i = -1$). Suma finală ar trebui să fie 0, însă pentru a face procesul mai rapid am ales să acceptăm o sumă care să aparțină intervalului (-1, 1).

```
def ajustare(alpha, vector_intrare):
    new_alpha = [x for x in alpha]
    suma = suma_alfa_y(alpha, vector_intrare)

    while not(1 > suma > -1):
        s_pozitiv = 0
        s_negativ = 0
        for j in range(0, len(new_alpha)):
            delta_pozitiv = alegere_delta_pozitiv(vector_intrare[j][1])
            delta_negativ = alegere_delta_negativ(vector_intrare[j][1])
            s_pozitiv += new_alpha[j] * vector_intrare[j][1] * delta_pozitiv
            s_negativ += new_alpha[j] * vector_intrare[j][1] * delta_negativ

        k = random.randint(0, len(vector_intrare) - 1)
        if s_pozitiv > s_negativ:
            while vector_intrare[k][1] != 1:
                k = random.randint(0, len(vector_intrare) - 1)
        else:
            while vector_intrare[k][1] != -1:
                k = random.randint(0, len(vector_intrare) - 1)
        if new_alpha[k] > suma:
            new_alpha[k] = new_alpha[k] - suma
        else:
            new_alpha[k] = 0

        suma = suma_alfa_y(new_alpha, vector_intrare)

    return new_alpha
```

Figura 6. Funcția de ajustare

```
def alegere_delta_pozitiv(rezultat):
    return 1 if rezultat == 1 else 0


def alegere_delta_negativ(rezultat):
    return 1 if rezultat == -1 else 0


def suma_alfa_y(alpha, vector_intrare):
    suma = 0
    for i in range(0, len(alpha)):
        suma = suma + alpha[i] * vector_intrare[i][1]

    return suma
```

Figura 7. Metode folosite în pasul de ajustare

Clasa *SVM* definește primește ca argumente la constructor setul de date de antrenare și cel de test. Aceasta definește patru metode:

- *salvare_model(path)* care salvează multiplicatorii Lagrange ți bias-ul într-un fișier la *path*-ul indicat ca argument
- *incarcare_model(path)* încarcă multiplicatorii Lagrange ți bias-ul dintr-un fișier de la *path*-ul primit ca argument
- *antrenare()* rulează algoritmul genetic GATE pentru a calcula multiplicatorii Lagrange și bias-ul
- *prezicere(z)* returnează predicția SVM-ului pentru o intrare *z* primită ca argument. Predicția este calculată conform formulei (9)

```
class SVM:
    def __init__(self, date_antrenare, date_test):
        self.date_antrenare = date_antrenare
        self.date_test = date_test
        self.alfa = None
        self.bias = None
        self.GATE = GATE(date_antrenare, 20, 0.02, 100)

    def salvare_model(self, path):
        with open(path, 'w') as file:
            file.write('')
            file.write(','.join([str(a) for a in self.alfa]))
            file.write('\n')
            file.write(str(self.bias))

    def incarcare_model(self, path):
        with open(path, 'r') as file:
            lines  = file.readlines()
            self.alfa = [float(a) for a in lines[0].replace('\n','').split(',')]
            self.bias = float(lines[1].replace('\n',''))

    def antrenare(self):
        self.alfa, self.bias = self.GATE.run_algorithm()

    def prezicere(self, z):
        return functie_activare(z, self.date_antrenare, self.alfa, self.bias)
```

Figura 8. Clasa SVM

Pentru a evalua performanțele algoritmului SVM implementat, am calculat predicțiile pentru setul de date de testare și am calculat precizia, acuratețea și scorul F1 acestuia.

```
def evaluate(svm, test_data):
    y_true = []
    y_pred = []
    for entry in test_data:
        y_true.append(entry[1])
        y_pred.append(svm.prezicere(entry[0]))

    print('accuracy: ' + str(metrics.accuracy_score(y_true, y_pred)))
    print('precision: ' + str(metrics.precision_score(y_true, y_pred)))
    print('f1 score: ' + str(metrics.f1_score(y_true, y_pred)))
```

Figura 9. Funcția de evaluare a algoritmului

5. Rezultatele obținute

Am considerat 1.000 de intrări, care au fost împărțite în 2 seturi: 70% din date fac parte din setul de antrenare, iar restul de 30% din setul de testare. Algoritmul genetic GATE a rulat pentru 20 de epoci, cu o populație inițială de 100 de indivizi. Antrenarea a durat în jur de 4 ore.

SVM-ul a obținut o acuratețe de 57%, o precizie de 55% și un scor F1 de 69%, rezultate care nu impresionează, dar care sunt promițătoare. Setul de date inițial are peste 10.000 de intrări, astfel că antrenarea pe întreg setul de date ar îmbunătății considerabil rezultatele.

```
accuracy: 0.5719063545150501
precision: 0.556390977443609
f1 score: 0.6981132075471698
```

Figura 10. Rezultatele obținute

6. Concluzii

Mașinile cu vectori suport au un suport matematic riguros în spate pentru a putea rezolva probleme de clasificare cât mai precis. O parte semnificativă a acestei metode este calculului marginei maxime dintre planele formate de clasele considerate. Noi am explorat folosirea unui algoritm genetic pentru acest lucru.

Algoritmi genetici sunt o categorie de algoritmi, inspirați din procesul de selecție și evoluție observat în natură pentru a converge spre rezultatul unei probleme dificil de rezolvat prin metode formale.

Evitarea rezolvării de ecuații matematice complexe este un avantaj major adus de această abordare și suntem siguri că se pot obține rezultate satisfăcătoare, comparabile cu alte metode de clasificare. Cel mai mare dezavantaj pentru noi a fost complexitatea timp a algoritmului, care poate fi optimizat sau antrenat pe un suport hardware mai capabil.

7. Bibliografie

- Suport curs Inteligență Artificială, Florin Leon
- https://www.researchgate.net/publication/309565420_Evolutionary_Support_Vector_Machines_A_Dual_Approach
- https://archive.ics.uci.edu/ml/datasets/Website+Phishing
- https://www.sciencedirect.com/science/article/abs/pii/S0957417414001481
- https://ro.wikipedia.org/wiki/%C3%8En%C8%99el%C4%83ciune_electronic%C4%83
- https://en.wikipedia.org/wiki/Support_vector_machine
- https://en.wikipedia.org/wiki/Genetic_algorithm

8. Contribuția membrilor din echipă

- Enache Ștefan:
  - Parsarea și pre-procesarea datelor de intrare
  - Pregătirea și împărțirea dataset-ului
  - Implementarea funcției de activare SVM
  - Implementarea pasului de ajustare din algoritmul genetic
- Murariu-Tănăsache Iulian:
  - Implementarea algoritmului genetic pentru calculul multiplicatorilor Lagrange
  - Evaluare performanțe algoritm
  - Integrare
  - Documentație