Proiect

Tehnici de optimizare

COBUZ Ionut-Alexandru
MACOVEI Bogdan
OPARIUC Rares-Petru

Grupa 344

Facultatea de Matematica si Informatica
Universitatea din Bucuresti
Aprilie 2019

Contents

1 Introducere in implementarea proiectului

1.1 Descrierea modului de lucru

Vom utiliza pentru implementarea proiectului un dataset ce contine informatii despre anumiti predictori medicali, in scopul de a determina daca o anumita persoana sufera sau nu de diabet in urma valorilor pe care le detinem despre aceasta. Feature-urile de care dispunem sunt

- *Pregnancies*, numarul de sarcini;
- *Glucose*, concentratia plasmatica a glucozei timp de doua ore intr-un test de toleranta la glucoza pe cale orala;
- *BloodPresure*, tensiunea arteriala diastolica (masurata in mmHg);
- *SkinThickness*, grosimea indoirii tricepsului (masurata in mm);
- *Insulin*, insulina serica timp de doua ore;
- *BMI*, indicele de masa corporala;
- *DiabetesPedigreeFunction*;
- *Age*;
- variabila de clasa *Outcome*, cea pe care incercam sa o prezicem, care ia valori in $\{0, 1\}$.

Pentru rezolvarea task-ului vom folosi ca algoritm de clasificare *Logistic Regression* cu diferite tipuri de optimizare. Pentru fiecare algoritm in parte vom oferi o descriere matematica si o implementare in Matlab.

1.2 Prelucrarea initiala a datelor

Vom stoca datele in variabila *df*, acestea fiind preluate din fisierul *diabetes.csv*.

Populam setul de date:

```
function [df, cols] = getData(filename)
    df = csvread(filename, 1, 0);
    cols = ["Pregnancies" "Glucose" "BloodPressure" "SkinThickness"...
        "Insulin" "BMI" "DiabetesPedigreeFunction" "Age" "Outcome"];
end

[df, cols] = getData("diabetes.csv");
```

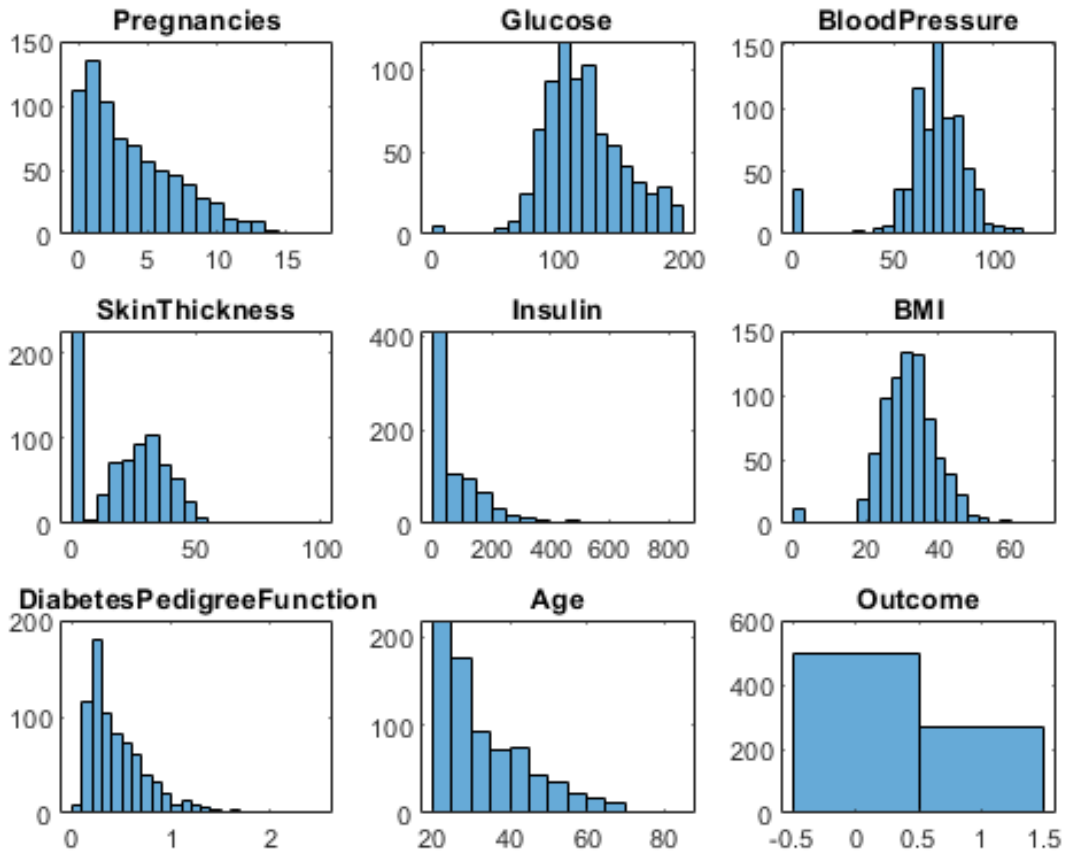
Analizam setul de date si plotam atat repartitia fiecarei clase, cat si perechi de ploturi (o clasa versus alta clasa).

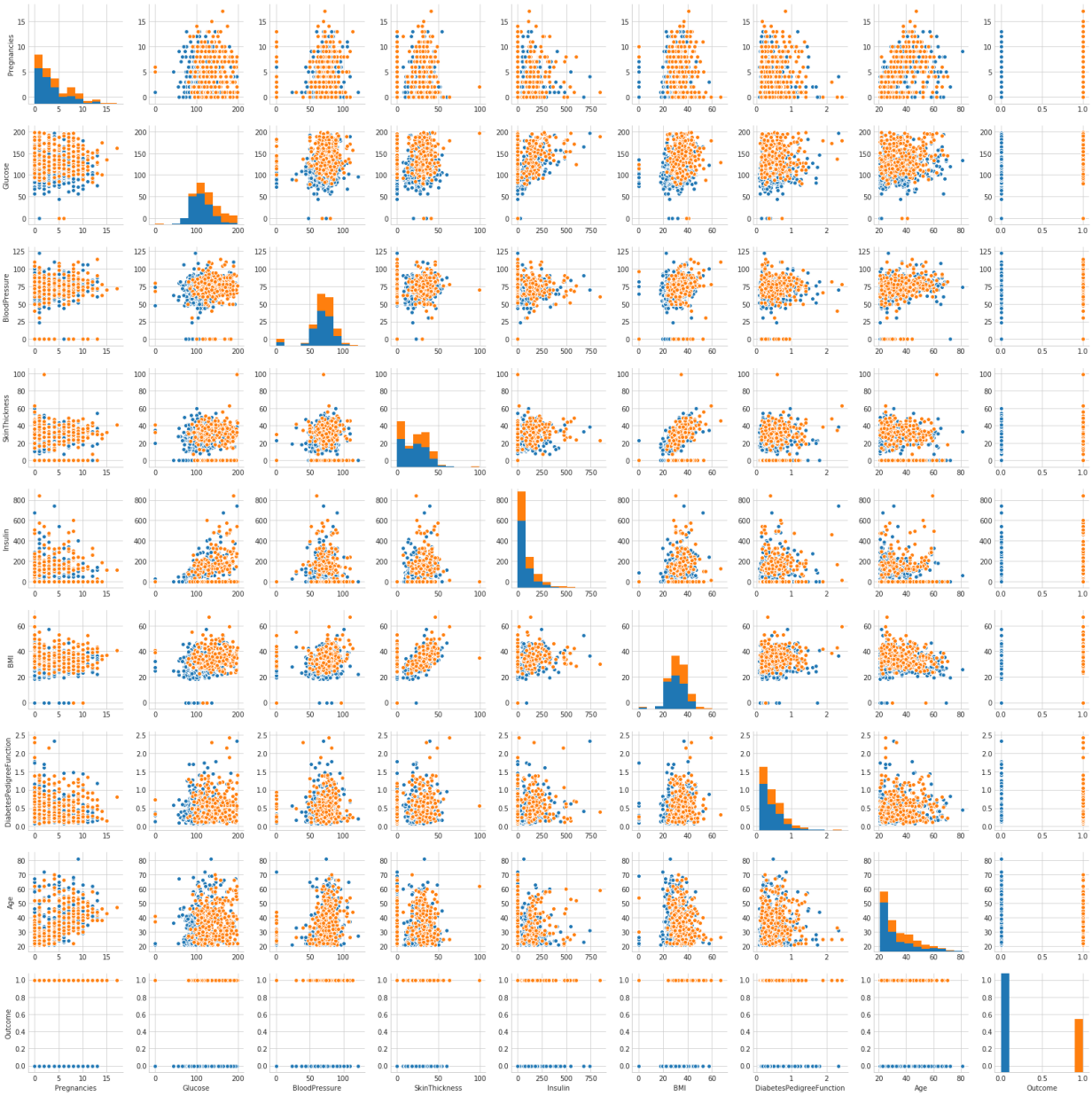
Functia pentru repartitia fiecarei clase:

```
function [] = viewRepartitions(df, cols)
    for i = 1 : 9
        subplot(3, 3, i);
        histogram(df(:, i));
        title(cols(i));
    end
end
```

Functia pentru perechea de ploturi:

```
function [] = viewPairPlot(df)
    [df0, df1] = splitDataInTwoClasses(df);
    for i = 1 : 9
        for j = 1 : 9
            subplot(9, 9, 9 * (i - 1) + j);
            if i ~= j
                plot(df0(:, i), df0(:, j), '*');
                hold on;
                grid on;
                plot(df1(:, i), df1(:, j), '*');
            else
                histogram(df0(:, i));
                hold on;
                grid on;
                histogram(df1(:, j));
            end
        end
    end
end
```





1.2.1 Impartirea datelor

Vrem sa folosim datele atat pentru antrenare, cat si pentru testare, astfel ca vom construi o functie trainTestSplit care va primi dataframe-ul initial, dimensiunea datelor pe care se face testarea, si returneaza perechile corespunzatoare de (train, test).

```
function [X_train, X_test, y_train, y_test] = trainTestSplit(df, test_size)
    random = randperm(length(df), floor((1 - test_size) * length(df)));
    X_train = df(random, 1:8);
    X_test = df(setdiff(1:length(df), random), 1:8);
    y_test = df(setdiff(1:length(df), random), 9);
    y_train = df(random, 9);
end

[X_train, X_test, y_train, y_test] = trainTestSplit(df, 0.3);
```

1.2.2 Evaluarea modelului

Vom construi o functie care sa ne ajute sa stabilim performanta modelului in functie de rezultatele reale pe care trebuia sa le ofere. Vom calcula masurile Precision, Recall si Accuracy, definite astfel, in functie de True negative (am prezis 0 si trebuia 0), True positive (am prezit 1 si trebuia 1), False positive (am prezis 1 si trebuia 0) si False negative (am prezis 0 si trebuia 1):

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

```
function [precision, recall, accuracy] = ModelEvaluation(prediction, y_test
    )
    TP = 0; TN = 0; FP = 0; FN = 0;
    for i = 1 : length(prediction)
        if prediction(i) == 0 && y_test(i) == 0
            TN = TN + 1;
        elseif prediction(i) == 1 && y_test(i) == 1
            TP = TP + 1;
        elseif prediction(i) == 1 && y_test(i) == 0
            FP = FP + 1;
        elseif prediction(i) == 0 && y_test(i) == 1
            FN = FN + 1;
        end
    end

    precision = TP / (TP + FP);
    recall = TP / (TP + FN);
    accuracy = (TP + TN) / (TP + TN + FP + FN);
end

[precision, recall, accuracy] = ModelEvaluation(prediction, y_test);
fprintf("Accuracy: %f\nPrecision: %f\nRecall: %f\n", accuracy, precision,
    recall);
```

2 Implementarea modelului

2.1 Metoda standard cu Gradient Descent

Primul model pe care il vom aplica este *Logistic Regression*. Vom incerca mai multe variante, de la cea standard, in care datele sunt neprelucrate, cu o optimizare Gradient Descent, la o varianta optimizata pentru care vom aplica si metode de feature engineering.

2.1.1 Descrierea modelului matematic

Metoda de regresie logistica este o metoda de clasificare pentru care se foloseste o functie logistica,

$$\sigma(x) = \frac{L}{1 + e^{-k(x-x_0)}}$$

(un caz special al functiei sigmoid). Pentru modelul care urmeaza, utilizam functia standard cu $L = 1$, $k = 1$ si $x_0 = 0$:

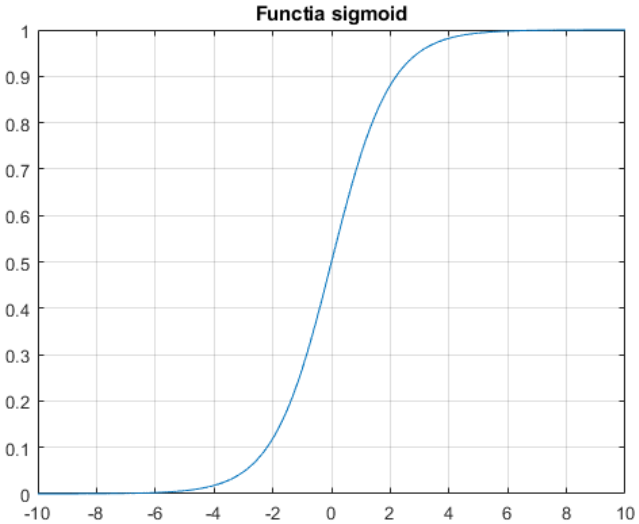
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Cum $0 < \sigma(x) < 1$, functia poate fi interpretata ca o probabilitate.

Putem vizualiza functia in Matlab:

```
function [sgm] = sigmoid(x)
    sgm = zeros(size(x));
    sgm = 1./(1 + exp(-x));
end

x = linspace(-10, 10, 5000);
plot(x, sigmoid(x));
grid on;
title("Functia sigmoid");
```



Utilizand aceasta functie logistica si un feature vector $x \in \mathbf{R}^n$, obtinem ca $\hat{y} = \sigma(w_0 + \sum_{i=1}^{n} w_i x_i)$, i.e.

$$\hat{y} = \frac{1}{1 + e^{-w^T x}}$$

ceea ce inseamna ca vom aplica o functie logistica pe un model liniar.

Predictia $\hat{y}$ va fi interpretata ca o probabilitate de forma $\hat{y} = P(y = 1|x, w)$.

Pentru a putea utiliza modelul, trebuie sa calculam vectorul w (weights), iar acesta va fi obtinut minimizand loss function-ul. In cazul regresiei logistice, utilizam

$$L = -\sum_i [y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log (1 - \hat{y}^{(i)})] + \lambda ||w||_2^2$$

Vrem sa determinam $w^* = \arg\min_w L(w)$. Pentru aceasta vom utiliza, in prima faza a implementarii, metoda Gradient Descent, ulterior aplicand Conjugate Gradient, si Metode Newton, alaturi de o descriere matematica a algoritmului L-BFGS.

$$\begin{aligned}
\arg\min_w \{L(w)\} = \\
\arg\min_w \Big\{ -\sum_i [y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log (1 - \hat{y}^{(i)})] + \lambda ||w||_2^2 \Big\} = \\
\arg\min_w \Big\{ -\frac{1}{\lambda} \sum_i [y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log (1 - \hat{y}^{(i)})] + ||w||_2^2 \Big\} = \\
\arg\min_w \Big\{ -C \sum_i [y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log (1 - \hat{y}^{(i)})] + ||w||_2^2 \Big\}
\end{aligned}$$

Ultima varianta este cea mai comuna, deoarece este usor de interpretat C-ul ca fiind costul pentru o greseala de antrenare.

Alte formulari ale loss function-ului sunt date de interpretarea diferita a variabilei de prezis: daca vom considera ca $y^{(i)} \in \{-1, 1\}\ \forall i$, atunci avem

$$P(y^{(i)}|x^{(i)}, w) = \frac{1}{1 + e^{-y^{(i)} \cdot w^T x^{(i)}}}$$

Se defineste, astfel, Log-likelihood:

$$\log \big(\prod_{i=1}^{m} \frac{1}{1 + e^{-y^{(i)} \cdot w^T x^{(i)}}} \big) = -\sum_{i=1}^{m} \log \big(1 + e^{-y^{(i)} \cdot w^T x^{(i)}} \big)$$

Vrem sa maximizam log-likelihood, astfel ca problema este echivalenta cu minimizarea (cu regularizare):

$$\min_w C \sum_{i=1}^{m} \log \big(1 + e^{-y^{(i)} \cdot w^T x^{(i)}} \big) + ||w||_2^2$$

2.1.2 Descriere minimala pentru Gradient Descent

Utilizam forma generala de aplicare pentru Gradient Descent, utilizand notatiile problemei pe care incercam sa o rezolvam,

$$w_j = w_j - \alpha \frac{\partial}{\partial w_j} L(w)$$

Avem de calculat

$$\alpha \frac{\partial}{\partial w_j} L(w) = \frac{\partial}{\partial w_j} \big[-\alpha C \sum_i [y^{(i)} \log \sigma_w(x^{(i)}) + (1 - y^{(i)}) \log 1 - \sigma_w(x^{(i)})] + ||w||_2^2 \big]$$

$$= -\alpha C \sum_{i=1}^{m} \big(\sigma_w(x^{(i)}) - y^{(i)} \big) x_j^{(i)} = -\frac{\alpha}{m} \sum_{i=1}^{m} \big(\sigma_w(x^{(i)}) - y^{(i)} \big) x_j^{(i)}$$

Utilizand o descriere vectoriala, obtinem ca

$$w := w - \frac{\alpha}{m} X^T(\sigma(X \cdot w) - y)$$

2.1.3 Implementare in cazul problemei curente

Vom implementa o functie LogisticRegressionFit, care va primi ca intrare un set de date de antrenare si va returna vectorul w (weights). Conditiile de oprire vor fi date de obtinerea unei norme suficient de mici pentru w, sau de obtinerea la un pas i a unei norme vectoriale pentru w mai mari decat norma de la pasul $i-1$.

```
function [weights] = LogisticRegressionFit(X_train, y_train)
    m = length(X_train); alpha = 1e-4;
    coef = alpha / m;
    oldW = rand(size(X_train, 2) + 1, 1);
    localX_train = [ones(size(X_train, 1), 1) X_train];
    while 1
        newW = oldW - coef * localX_train' * ...
            (sigmoid(localX_train * oldW) - y_train);

        if norm(newW) > norm(oldW)
            break;
        end

        oldW = newW;
    end
    weights = newW;
end
```

2.1.4 Functia de predictie

```
function [prediction] = LogisticRegressionPrediction(X_test, weights)
    localX_test = [ones(size(X_test, 1), 1) X_test];
    prediction = round(sigmoid(localX_test * weights));
end
```

2.1.5 Modelul final

```
[df, cols] = getData("diabetes.csv");
[X_train, X_test, y_train, y_test] = trainTestSplit(df, 0.25);

weights = LogisticRegressionFit(X_train, y_train);
prediction = LogisticRegressionPrediction(X_test, weights);

[precision, recall, accuracy] = ModelEvaluation(prediction, y_test);

fprintf("Accuracy: %f\nPrecision: %f\nRecall: %f\n",...
    accuracy, precision, recall);
```

Obtinem un accuracy de aproximativ 0.7.

2.1.6 Gradient prin metoda backtracking

Vom alege o constanta $c_1 \in (0, 1)$ si vom utiliza prima conditie Wolfe,

$$f(x_k - \alpha_k \nabla f(x_k)) \leq f(x_k) - c_1 \alpha_k \nabla f(x_k)^T \nabla f(x_k)$$

pentru a determina acel pas α pentru care functia are o descrestere "suficienta". Alegem $\rho \in (0, 1]$, $\alpha_0 > 0$ si aplicam iteratia

$$\alpha_{k+1} = \rho \alpha_k$$

Algoritmul in Matlab este, bazat pe problema pe care o rezolvam:

```
function [alpha] = GradientDescentWolfe(X_train, y_train)
    oldAlpha = 0.6; rho = 0.8; c1 = 0.2;
    k = 1;
    while 1
        [cost, gradient] = CostAndGradientFunction(X_train, y_train, X_train
            (k, :)');
        arg = X_train(k, :)' - oldAlpha * gradient;
        [cost2, ~] = CostAndGradientFunction(X_train, y_train, arg);
        md = cost - c1 * oldAlpha * (gradient') * gradient;

        newAlpha = rho * oldAlpha;
        oldAlpha = newAlpha;

        if cost2 <= md && k >= 50
            break;
        end


        k = k + 1;
    end
    alpha = newAlpha;
end
```

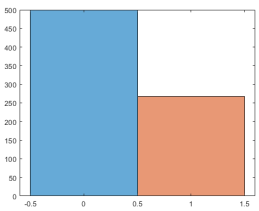
Observatie: pentru algoritmul de mai sus, am impus manual o oprire dupa 50 de pasi, chiar daca este satisfacuta conditia Wolfe. Constanta de oprire este special aleasa pentru a rezolva problema de fata.

2.1.7 Comparatie initiala intre cele doua metode

Vom compara performanta obtinuta de algoritmul de regresie logistica in cazul celor doua metode de optimizare prezentate pana acum. Pentru a elimina din erorile de masurare, vom normaliza datele din dataset-ul initial.

```
function [dfN] = DataNormalization(df)
    dfN = zeros(size(df));
    for i = 1 : 9
        dfN(:, i) = (df(:, i) - min(df(:, i))) / (max(df(:, i)) - min(df(:,
            i)));
    end
end
```

Reamintim faptul ca cele doua clase nu erau echilibrate, astfel ca invatarea va fi ingreunata si va fi supusa erorilor.



Pentru analiza performantelor ne vom folosi de marimile Precision, Recall, Accuracy si de numarul de pasi pe care ii executa algoritmul pentru a obtine o anumita performanta.

Marime evaluata	G. D. standard	G. D. Backtracking
Accuracy	0.549827	0.544372
Precision	0.423859	0.423159
Recall	0.807349	0.797638
Iteratii	25328	566

2.2 Alte metode pentru Regresia Logistica

2.2.1 Descrierea modelului

Vom alege, pentru urmatoarele optimizari, modelul in care variabila de prezis, y, ia valori in multimea $\{0, 1\}$. Descriem acest fapt prin

$$P(y = \pm 1|x, w) = \sigma(yw^T x) = \frac{1}{1 + e^{-yw^T x}}$$

Avand un dataset $(X, y) = \{(x^{(i)}, y^{(i)})\}\ \forall i$ vrem sa maximizam, conform descrierilor anterioare, functia

$$L(w) = -\sum_i \log\left(1 + e^{-y_i w^T x_i}\right) - \frac{\lambda}{2} w^T w$$

unde, pentru $\lambda > 0$ obtinem o estimare "regularizata" a lui w ce ne ofera o performanta foarte buna pentru model.

Vom calcula gradientul $g(w) = \nabla_w L(w)$:

$$\begin{aligned}
\nabla_w L(w) &= \nabla_w \big[-\sum_i \log\left(1 + e^{-y_i w^T x_i}\right)\big] - \frac{\lambda}{2} \nabla_w \big[w^T w\big] \\
&= -\sum_i \nabla_w \big[\log\left(1 + e^{-y_i w^T x_i}\right)\big] - \frac{\lambda}{2} 2w \\
&= -\sum_i \frac{1}{1 + e^{-y_i w^T x_i}} \nabla_w [1 + e^{-y_i w^T x_i}] - \lambda w \\
&= -\sum_i \frac{1}{1 + e^{-y_i w^T x_i}} \cdot e^{-y_i w^T x_i} \cdot \nabla_w [-y_i w^T x_i] - \lambda w \\
&= -\sum_i \frac{1 + e^{-y_i w^T x_i} - 1}{1 + e^{-y_i w^T x_i}} - y_i x_i - \lambda w \\
&= -\sum_i (1 - \sigma(y_i w^T x_i))(-y_i x_i) - \lambda w \\
&= \sum_i (1 - \sigma(y_i w^T x_i)) y_i x_i - \lambda w
\end{aligned}$$

Avem, astfel, ca $g(w) = \sum_i (1 - \sigma(y_i w^T x_i)) y_i x_i - \lambda w$. Procedand analog, obtinem expresia Hessianei:

$$H(w) = \frac{\partial^2 L(w)}{\partial w \partial w^T} = -\sum_i \sigma(w^T x_i)(1 - \sigma(w^T x_i)) x_i x_i^T - \lambda I$$

Notand cu $a_{ii} := \sigma(w^T x_i)(1 - \sigma(w^T x_i))$, obtinem expresia in forma matriceala

$$H = -XAX^T - \lambda I$$

2.2.2 Metoda Newton

Un pas in metoda Newton este

$$\begin{aligned}
w_{new} &= w_{old} + (XAX^T + \lambda I)^{-1} \Big(\sum_i (1 - \sigma(y_i w^T x_i)) y_i x_i - \lambda w \Big) \\
&= (XAX^T + \lambda I)^{-1} XA \Big(X^T w_{old} + \Big[\frac{1 - \sigma(y_i w^T x_i)) y_i}{a_{ii}} \Big] \Big)
\end{aligned}$$

Vom implementa o functie care sa ne returneze valoarea functiilor de cost si gradient in intrarea w curenta:

```
function [cost, gradient] = CostAndGradientFunction(X, y, w)
    gradient = zeros(size(w));
    m = size(X, 1);
    h = sigmoid(X * w);
    cost = -(1 / m) * sum((y .* log(h)) + (1 - y) .* log(1 - h));

    for i = 1 : size(w, 1)
        gradient(i) = (1 / m) * sum((h - y) .* X(:, i));
    end
end
```

O implementare pentru functia de fit in modelul logistic va fi:

```
function [weights] = LogisticRegressionFitMN(X_train, y_train, lambda)
    [m, n] = size(X_train);
    X = [ones(m, 1) X_train];
    oldW = zeros(n + 1, 1);
    oldCost = CostAndGradientFunction(X, y_train, oldW);
    while 1
        a = zeros(1, m);
        for j = 1 : m
            sgm = sigmoid(oldW' * X(j, :)');
            a(j) = sgm * (1 - sgm);
        end
        A = diag(a);

        z = zeros(1, m);
        for j = 1 : m
            z(j) = X(j, :) * oldW + (1 - sigmoid(y_train(j) * oldW' ...
                * X(j, :)') * y_train(j)) / A(j, j);
        end
        z = z';

        newW = inv(X' * A * X + lambda * eye(n + 1, n + 1)) * X' * A * z;
        newCost = CostAndGradientFunction(X, y_train, newW);

        if abs(oldCost - newCost) <= 1e-2
            break;
        end

        oldW = newW;
        oldCost = newCost;
    end
    weights = newW;
end
```

Rezultatele obtinute sunt mai slabe decat in cazul modelului Gradient Descent, acuratetea fiind in jurul valorii 0.4.

2.2.3 Gradientul conjugat

Pastrand aceleasi expresii pentru Hessiana si gradient, un pas este dat de actualizarea

$$w_{new} = w_{old} - \frac{g^T u}{u^T H u} u$$

unde u este o directie arbitrara. Pentru o implementare eficienta, utilizam

$$u^T H u = -\lambda u^T u - \sum_i a_{ii} (u^T x_i)^2$$

Actualizarea vectorului u se face dupa formula

$$u_{new} = g - u_{old} \frac{g^T H (w_{new} - w_{old})}{u_{old}^T H (w_{new} - w_{old})}$$

Implementarea in Matlab este:

```
function [weights] = ConjugateGradient(X_train, y_train, lambda)
    [m, n] = size(X_train);
    X = [ones(m, 1) X_train];
    oldW = 2.*rand(n + 1,1) - 1;
    oldU = 2.*rand(n + 1,1) - 1;
    for i = 1 : m
        Xi = X(i, :)';
        [~, g] = CostAndGradientFunction(X, y_train, Xi);
        a = zeros(1, m);
        for j = 1 : m
            sgm = sigmoid(oldW' * X(j, :)');
            a(j) = sgm * (1 - sgm);
        end
        A = diag(a);

        H = -X'*A*X - lambda * eye(n + 1, n + 1);

        newU = g - oldU * ((g') * H * oldU) / ((oldU') * H * oldU);
        newW = oldW - newU * ((g') * newU) / ((newU') * H * newU);
        oldU = newU;
        oldW = newW;
    end
    weights = newW;
end
```

Pentru valori mare ale lui λ modelul se comporta bine pe date normalizate, accuracy-ul ajungand in jurul valorii 0.68.

2.2.4 Metoda Newton cu Hessiana fixata

Pentru a inversa o singura data Hessiana, alegem expresia

$$\hat{H} = -\frac{1}{4} X X^T$$

care se generalizeaza

$$\hat{H} = -\frac{1}{4} X X^T - \lambda I$$

astfel ca actualizarea se va face iterativ cu pasul

$$w_{new} = w_{old} - \hat{H}^{-1} g$$

Implementarea in Matlab:

```
function [weights] = NewtonFixedHessian(X_train, y_train, lambda)
    [m, n] = size(X_train);
    X = [ones(m, 1) X_train];
    oldW = 2.*rand(n + 1,1) - 1;
    H = -1/4 * (X') * X - lambda * eye(n + 1, n + 1);
    for i = 1 : n
        Xi = X(i, :)';
        [~, g] = CostAndGradientFunction(X, y_train, Xi);
        newW = oldW - H\g;

        if abs(norm(newW) - norm(oldW)) < 1e-5
            break;
        end
    end
    weights = newW;
end
```

2.2.5 Modelul Perceptron

Pentru modelul Perceptron, optimizarea se realizeaza actualizand vectorul de weights w la fiecare pas,

$$w_{new} = w_{old} + y^{(i)} x^{(i)}$$

Cum valorile pe care le prezicem iau valori in multimea $\{-1, 1\}$, folosindu-ne doar de feature-ul curent, vom avea

$$\hat{y} = sgn(w^T x^{(i)})$$

Vom opri iteratiile dupa un numar setat de pasi (epoci de antrenare), sau dupa ce nu vom mai avea nicio eroare la un anumit pas. Definim eroarea

$$\varepsilon^{(i+1)} = \varepsilon^{(i)} + 1$$

daca $y^{(i)} \hat{y}^{(i)} < 0$.

Implementarea in Matlab a modelului este

```
function [weights, bias] = perceptron(X_train, y_train)
    [m, n] = size(X_train);
    weights = zeros(n, 1);
    bias = zeros(1, 1);
    max_iter = 5000;
    for iter = 1 : max_iter
        errors = 0;
        for i = 1 : m
            xi = X_train(i, :)';
            predicted = sign(weights' * xi + bias);
            if y_train(i) * predicted <= 0
                weights = weights + y_train(i) * xi;
                bias = bias + y_train(i);
                errors = errors + 1;
            end
```

```
16            end
17            if (errors == 0)
18                break;
19            end
20        end
21    end
```

Accuracy-ul obtinut pe datele normalizate este de aproximativ 0.68, numarul de pasi fiind egal cu numarul pe care il setam, pe cazul de mai sus acesta fiind 5000.

2.2.6 Metoda Quasi-Newton. BFGS. L-BFGS

Cel mai utilizat algoritm in practica este L-BFGS, a carei descriere matematica o vom prezenta mai jos, fara a avea o implementare.

In loc sa calculam totul in functie de Hessiana fixata, putem lasa marimile sa varieze. Initial setam $\hat{H}^{-1} = I$, iar la fiecare pas calculam $w_{new} = w + \Delta w$, provocand o schimbare si in gradient: $\nabla g = g_{new} - g$. Pentru functii patratice avem

$$\nabla w = -H^{-1}\nabla g$$

In algoritmul BFGS, update-ul se face folosind relatiile

$$b = 1 + \frac{\nabla g^T \hat{H}^{-1} \nabla g}{\nabla w^T \Delta g}$$

$$\hat{H}^{-1}_{new} = \hat{H}^{-1} + \frac{1}{\Delta w^T \Delta g}\left(b\Delta w \Delta w^T - \Delta w \Delta g^T \hat{H}^{-1} - \hat{H}^{-1}\Delta g \Delta w^T\right)$$

O varianta a acestui algoritm este L-BFGS ("limited-memory" BFGS), unde $\hat{H}^{-1}$ nu este stocata, dar se asuma ca este I dupa fiecare update. Obtinem relatiile

$$b = 1 + \frac{\Delta g^T \Delta g}{\Delta w^t \Delta g}$$

$$a_g = \frac{\Delta w^T g}{\Delta w^T \Delta g}$$

$$a_w = \frac{\Delta g^T g}{\Delta w^T \Delta g} - b a_g$$

$$u = -g + a_w \Delta w + a_g \Delta g$$

2.3 Comparatii intre modele

Urmatorul tabel descrie rezultatele obtinute de fiecare algoritm privind metricile specifice clasificarii (precision, recall, accuracy) si numarul de pasi necesari pentru finalizare.

Algoritm	Precision	Recall	Accuracy	Iteratii
Gradient Descent	0.423859	0.807349	0.549827	25328
Gradient Descent Backtracking	0.423159	0.797638	0.544372	566
Metoda Newton	0.367965	1.000000	0.367965	537
Gradient conjugat	0.600000	0.068966	0.632035	537
Newton cu Hessiana fixata	0.541667	0.313253	0.658009	537
Modelul Perceptron	0.591304	0.860759	0.748918	5000

Observam ca, pe modelul de fata, cel mai rau se comporta Metoda Newton, iar cel mai bine se comporta algoritmii Gradient conjugat, Newton cu Hessiana fixata si modelul Perceptron; performanta este afectata si de setul de date neechilibrat, modelul invatand mai bine o clasa in defavoarea alteia (de aici diferentele semnificative intre Precision si Recall).

O comparatie privind unii dintre cei mai importanti algoritmi utilizati in implementarea regresiei logistice: