UNIVERSITÀ DEGLI STUDI DI PERUGIA

Corso di laurea magistrale in Ingegneria Informatica e Robotica
Dipartimento d'Ingegneria

Signal Processing and Optimization for Big Data

Sparse SVM: SVM lineare con regolarizzazione L1

Matteo Rinalduzzi

Matricola 327493

A. A. 2020/2021

INDICE

SUPPORT VECTOR MACHINE 3

ALGORITMO CENTRALIZZATO 3
ALGORITMO DISTRIBUITO 4

IMPLEMENTAZIONE IN PYTHON 7

PARTE 1 – VALUTAZIONE EFFETTO DELLA NORMA L1 7
PARTE 2 – CONFRONTO VERSIONE CENTRALIZZATA E VERSIONE DISTRIBUITA 9
PARTE 3 – APPLICAZIONE A UN DATASET REALE 12

Support Vector Machine

In questa sezione viene brevemente descritto quale problema va a risolvere il classificatore SVM lineare in versione centralizzata. Segue poi una implementazione distribuita dell'algoritmo in cui si fa splitting tra i dati. Come termine di regolarizzazione si utilizza la norma L1.

Algoritmo centralizzato

SVM è un algoritmo di classificazione binaria che si pone l'obiettivo di trovare i coefficienti dell'iperpiano H che meglio separa i dati.

$$H = \left\{ \underline{x} : \underline{\beta}^T \underline{x} + \beta_0 = 0 \right\}$$

Una prima soluzione del problema, elaborata da Rosenblatt, va a cercare i coefficienti dell'iperpiano che minimizzano una funzione costo in cui compare la somma delle distanze (dall'iperpiano) dei punti classificati male.

$$\underset{(\underline{\beta},\beta_0)}{argmin} \left(-\sum_{i=1}^{n} \underline{y}_i \left(\underline{\beta}^T \underline{x}_i + \beta_0 \right) u_{-1} \left(\underline{y}_i \left(\underline{\beta}^T \underline{x}_i + \beta_0 \right) \right) \right)$$

Tale formulazione del problema non ammette soluzione in forma chiusa. Inoltre, nel caso di dati non linearmente separabili non funziona molto bene e nel caso fortunato di dati linearmente separabili la soluzione non è unica. Per risolvere questo inconveniente e per prevenire overfitting si può effettuare una modifica e aggiungere un termine di regolarizzazione.

$$\underset{(\underline{\beta},\beta_0)}{argmin} \left(\frac{1}{n} \sum_{i=1}^{n} max \left\{ 0, 1 - \underline{y}_i \left(\underline{\beta}^T \underline{x}_i + \beta_0 \right) \right\} + \lambda \left\| \underline{\beta} \right\|_2^2 \right)$$

L'obiettivo di questa tesina è studiare il problema utilizzando la norma L1 invece che la norma L2, andando a implementare la versione centralizzata dell'algoritmo e una sua versione distribuita ottenuta sfruttando ADMM nel caso di splitting tra i dati.

$$\underset{(\underline{\beta},\beta_0)}{argmin} \left(\frac{1}{n} \sum_{i=1}^{n} max \left\{ 0, 1 - \underline{y}_i \left(\underline{\beta}^T \underline{x}_i + \beta_0 \right) \right\} + \lambda \left\| \underline{\beta} \right\|_1 \right)$$

La funzione obiettivo è convessa in quanto somma di funzioni convesse:

- il primo termine è la somma di massimi tra funzioni convesse e per le proprietà di pointwise maximum e per la proprietà di somma di funzioni convesse è convesso
- il secondo termine è la regolarizzazione L1, quindi convesso

Algoritmo distribuito

SVM nella versione descritta precedentemente si presta bene ad essere risolto in maniera distribuita poiché la hinge loss è scrivibile nella forma:

$$\boldsymbol{\ell}(A\underline{x} - \underline{b}) = \sum_{\mathrm{i}=1}^{\mathrm{m}} \boldsymbol{\ell}_i(\underline{a}_i{}^T\underline{x} - \underline{b}_i)$$

Si può dunque applicare il framework più generale dell'ottimizzazione al consenso in cui si fa splitting tra i dati. Questo acquista di significato nel caso in cui si ha a che fare con dataset di grandi dimensioni. Data una rete di N agenti, si crea una copia locale della variabile di ottimizzazione globale su ogni agente e poi si impone il consenso tra le variabili. Ogni agente si prende carico di una porzione del dataset.

Definendo $\underline{x} = \begin{pmatrix}\beta\\ \beta_0\end{pmatrix}$, $\underline{a}_i = y_i\begin{pmatrix}\underline{x}_i\\ 1\end{pmatrix}$, $C = \begin{bmatrix}I & 0\\ 0 & 0\end{bmatrix}$ si ottiene la formulazione equivalente:

$$\min_{\underline{x}}\left(\frac{1}{n}\sum_{i=1}^{n} max\{0, 1 - \underline{a}_i{}^T\underline{x}\} + \lambda\left\|C\underline{x}\right\|_1\right)$$

La matrice $A \in R^{m*n}$ può essere suddivisa in sottomatrici $A_i \in R^{m_i*n}$ in modo che risulti $A = \begin{bmatrix}A_1\\ \vdots\\ A_N\end{bmatrix}$. A_i rappresenta la porzione di dati assegnata all'i-esimo agente.

Con questa notazione si può riformulare il problema come un problema di ottimizzazione al consenso in una rete con N agenti, risolvibile con ADMM:

$$\min_{\underline{z},\{\underline{x}_i\}_{i=1,..,N}}\left(\frac{1}{n}\sum_{i=1}^{N}\underline{1}_{m_i}{}^T max\{\underline{0}, \underline{1}_{m_i} - A_i\underline{x}_i\} + \lambda\left\|C\underline{z}\right\|_1\right) \quad s.t. \quad \underline{x}_i - \underline{z} = 0, i = 1, \dots, N$$

$$L_\rho\left(\{\underline{x}_i\}_{i=1,\dots,N}, \underline{z}, \{\underline{u}_i\}_{i=1,\dots,N}\right) = \frac{1}{n}\sum_{i=1}^{N}\underline{1}_{m_i}{}^T max\{\underline{0}, \underline{1}_{m_i} - A_i\underline{x}_i\} + \lambda\left\|C\underline{z}\right\|_1 + \frac{\rho}{2}\sum_{i=1}^{N}\left\|\underline{x}_i - \underline{z} + \underline{u}_i\right\|_2^2$$

Il lagrangiano aumentato associato al problema può essere visto come la somma di N lagrangiani aumentati che condividono la stessa variabile globale $\underline{z}$, ma che hanno variabili primali $\underline{x}_i$ e variabili duali $\underline{u}_i$ separate.

$$L_\rho\left(\{\underline{x}_i\}_{i=1,\dots,N}, \underline{z}, \{\underline{u}_i\}_{i=1,\dots,N}\right) = \sum_{i=1}^{N} L_\rho(\underline{x}_i, \underline{z}, \underline{u}_i) + \lambda\left\|C\underline{z}\right\|_1$$

ADMM può aggiornare separatamente su ogni processore $\underline{x}_i$ e $\underline{u}_i$, ma a meno che non si voglia far girare un algoritmo di consenso ad ogni step di ADMM, si ha bisogno di un fusion center per calcolare $\underline{z}$.

Iterazione k+1 - (STEP 1)

$$\underline{x}_i^{(k+1)} = \underset{\underline{x}_i}{argmin}\left(\frac{1}{n}\underline{1}_{m_i}{}^T max\{\underline{0}, \underline{1}_{m_i} - A_i\underline{x}_i\} + \frac{\rho}{2}\left\|\underline{x}_i - \underline{z}^{(k)} + \underline{u}_i^{(k)}\right\|_2^2\right)$$

Scrivendo il massimo in forma scalare si ottiene:

$$\underline{x}_i^{(k+1)} = \underset{\underline{x}_i}{argmin}\left(\frac{1}{n}\sum_{i=1}^{m_j} max\{0, 1 - \underline{a}_{ij}{}^T\underline{x}_i\} + \frac{\rho}{2}\left\|\underline{x}_i - \underline{z}^{(k)} + \underline{u}_i^{(k)}\right\|_2^2\right), i = 1, \dots, N$$

Questo problema non è risolvibile in forma chiusa, ma la funzione costo è convessa. Si può utilizzare un solver per problemi di ottimizzazione convessa da applicare ad ogni iterazione.

Iterazione k+1 - (STEP 2)

$$\underline{z}^{(k+1)} = \underset{\underline{z}}{argmin}\left(\lambda\left\|C\underline{z}\right\|_1 + + \frac{\rho}{2}\sum_{i=1}^{N}\left\|\underline{x}_i^{(k+1)} - \underline{z} + \underline{u}_i^{(k)}\right\|_2^2\right)$$

Con un po' di semplici conti si può dimostrare che il problema può essere riformulato in modo da includere la sommatoria all'interno della norma L2.

$$\underline{z}^{(k+1)} = \underset{\underline{z}}{argmin}\left(\lambda\sum_{i=1}^{N-1}|z_i| + \frac{N\rho}{2}\left\|\overline{\underline{x}}^{(k+1)} - \underline{z} + \overline{\underline{u}}^{(k)}\right\|_2^2\right)$$

$$dove \quad \overline{\underline{x}}^{(k+1)} = \frac{1}{N}\sum_{i=1}^{N}\underline{x}_i^{(k+1)} \quad e \quad \overline{\underline{u}}^{(k)} = \frac{1}{N}\sum_{i=1}^{N}\underline{u}_i^{(k)}.$$

Questo secondo problema ammette soluzione in forma chiusa, anche se la funzione costo non è differenziabile. E' sufficiente applicare il subgradiente a ogni componente:

$$\frac{\partial J(\underline{z})}{\partial z_i} = \begin{cases} \lambda + N\rho\left(\overline{x}_i{}^{(k+1)} - z_i + \overline{u}_i{}^{(k)}\right), & z_i > 0 \\ -\lambda + N\rho\left(\overline{x}_i{}^{(k+1)} - z_i + \overline{u}_i{}^{(k)}\right), & z_i < 0 \\ 0, & z_i = 0 \end{cases}, \qquad i = 1, \dots, N-1$$

$$\frac{\partial J(\underline{z})}{\partial z_N} = N\rho\left(\overline{x}_i{}^{(k+1)} - z_i + \overline{u}_i{}^{(k)}\right)$$

e imporlo uguale a zero:

$$\frac{\partial J(\underline{z})}{\partial z_i} = 0, \qquad i = 1, \dots, N$$

La soluzione è data dall'operatore di soft thresholding, il quale induce sparsità mettendo a zero tutte le componenti che sono più piccole in modulo di $\frac{\lambda}{N\rho}$.

$$z_i = S_{\lambda/N\rho}\left(\overline{x}_i^{(k+1)} + \overline{u}_i^{(k)}\right) = \begin{cases} \overline{x}_i^{(k+1)} + \overline{u}_i^{(k)} - \dfrac{\lambda}{N\rho}, & \overline{x}_i^{(k+1)} + \overline{u}_i^{(k)} > \dfrac{\lambda}{N\rho} \\ \overline{x}_i^{(k+1)} + \overline{u}_i^{(k)} + \dfrac{\lambda}{N\rho}, & \overline{x}_i^{(k+1)} + \overline{u}_i^{(k)} < -\dfrac{\lambda}{N\rho} \\ 0, & \left|\overline{x}_i^{(k+1)} + \overline{u}_i^{(k)}\right| < \dfrac{\lambda}{N\rho} \end{cases}, \qquad i = 1, \dots, N-1$$

$$z_N = \overline{x}_N^{(k+1)} + \overline{u}_N^{(k)}$$

Iterazione k+1 – (STEP 3)

$$\underline{u}_i^{(k+1)} = \underline{u}_i^{(k)} + \underline{x}_i^{(k+1)} - \underline{z}^{(k+1)}, \qquad i = 1, \dots, N-1$$

Allo step i-esimo:

- l'agente j-esimo (j=1,…,N) esegue dei calcoli locali e calcola $\underline{x}_j^{(k+1)}$, poi lo comunica al fusion center
- il fusion center mette insieme i risultati ricevuti dai vari agenti e calcola $\underline{z}^{(k+1)}$, poi lo comunica indietro agli N agenti
- l'agente j-esimo (j=1,…,N) sulla base del calcolo locale effettuato precedentemente e sulla base del risultato ricevuto dal fusion center calcola $\underline{u}_j^{(k+1)}$, poi lo comunica al fusion center. Quest'ultimo passo è molto semplice e potrebbe essere anche eseguito dal fusion center in persona, il quale dovrebbe poi comunicare il risultato j-esimo al j-esimo agente.

Implementazione in python

Le operazione eseguite lato codice hanno riguardato 3 aspetti:

- *implementazione della versione centralizzata di SVM per valutare l'effetto della norma L1 al variare di lambda*
- *confronto della versione centralizzata con la versione distribuita generando fittiziamente un insieme di dati linearmente separabili da un iperpiano e verifica del fatto che i due algoritmi convergono alla stessa soluzione*
- *applicazione dell'algoritmo distribuito su di un semplice dataset reale di dati quasi linearmente separabili nel piano e verifica grafica del fatto che l'algoritmo restituisce i parametri della retta che meglio separa i dati*

Parte 1 – Valutazione effetto della norma L1

L'obiettivo di questa prima parte è di verificare che la norma L1 ha l'effetto di forzare alcune delle stime dei coefficienti, quelle con un contributo minore al modello, ad essere zero.

Generazione dataset:

```
import numpy as np
np.random.seed(1)
n = 20 # n. di features
m = 1000 # n. di esempi
beta_true = np.random.randn(n,1) # coefficienti dell'iperpiano (beta)
offset = np.random.randn(1) # intercetta (beta_0)

# Genero dati linearmente separati dall'iperpiano identificato da (beta_true, offset)
X = np.random.normal(0, 5, size=(m,n))
Y = np.sign(X.dot(beta_true) + offset)
```

Impostazione problema centralizzato con CVXPY:

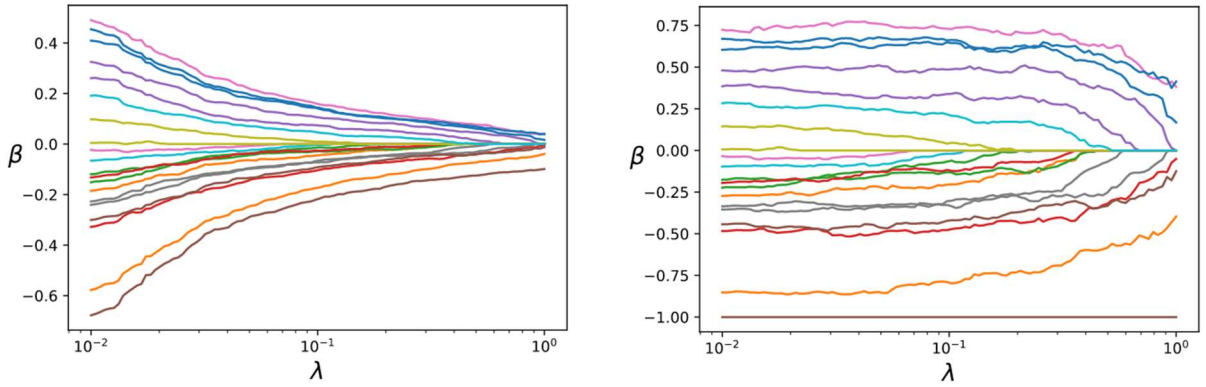
```
import cvxpy as cp
beta = cp.Variable((n,1)) # var. di ottimizzazione
v = cp.Variable()         # intercetta
loss = cp.sum(cp.pos(1 - cp.multiply(Y, X @ beta + v)))
reg = cp.norm(beta, 1)                  # regolarizzazione con norma L1
lambd = cp.Parameter(nonneg=True) # parametro che incide sul peso della regolarizzazione
prob = cp.Problem(cp.Minimize(loss/m + lambd*reg))
```

Risoluzione del problema per diversi valori di lambda:

```
TRIALS = 100
lambda_vals = np.logspace(-2, 0, TRIALS) # valori di lambda da 0.01 a 1
beta_vals = []
for i in range(TRIALS):
   lambd.value = lambda_vals[i]
```

```
prob.solve()
beta_vals.append(beta.value)
```

Nei seguenti grafici viene riportato l'andamento del valore dei coefficienti dell'iperpiano al variare di lambda. Nel grafico di destra i coefficienti sono normalizzati per il valore del coefficiente più piccolo, nel grafico di sinistra vengono riportati così come sono forniti in output dall'algoritmo.



Lambda è il peso che viene dato alla parte di regolarizzazione in:

$$\underset{(\underline{\beta},\beta_0)}{argmin}\left(\frac{1}{n}\sum_{i=1}^{n} max\left\{0, 1-\underline{y}_i\left(\underline{\beta}^T\underline{x}_i+\beta_0\right)\right\}+\lambda\left\|\underline{\beta}\right\|_1\right)$$

Il grafico di destra conferma ciò che ci viene detto dalla teoria. Per lambda molto piccoli il termine di regolarizzazione influisce poco e le stime dei coefficienti sono quasi tutte diverse da zero, mentre all'aumentare di lambda sempre più coefficienti vengono messi a zero. Restano solo i coefficienti che danno un contributo maggiore al modello.

Parte 2 – Confronto versione centralizzata e versione distribuita

L'obiettivo di questa sezione è di implementare la versione distribuita dell'algoritmo e verificare che converge alla stessa soluzione ottenuta con l'algoritmo centralizzato. Per confrontare gli algoritmi è stato generato un insieme di dati linearmente separabili da un iperpiano e sono stati fissati a priori i parametri rho e lambda.

Formulazione centralizzata:

```
# Generazione dei dati
n = 20 # n. di features
m = 1000 # n. di esempi
beta_true = np.random.randn(n,1)
offset = np.random.randn(1)
beta = np.append(beta_true, offset).reshape(21) # coefficienti dell'iperpiano in un unico vettore
X = np.random.normal(0, 5, size=(m,n))
Y = np.sign(X.dot(beta_true) + offset)

# Formulazione centralizzata
import cvxpy as cp
A = np.hstack((X*Y,Y))
n_features = n +1
# Parametri
rho = 1
lamda = 0.5
C = np.identity(n_features)
C[n_features-1,n_features-1] = 0

beta = cp.Variable((n_features,1))     # var. di ottimizzazione
loss = cp.sum(cp.pos(1 - A @ beta )) # loss
reg = cp.norm(C@beta, 1)               # regolarizzazione
prob = cp.Problem(cp.Minimize(loss/m + lamda*reg))

# Risoluzione con CVXPY
prob.solve()
centralized_solution = beta.value

print(centralized_solution.reshape((centralized_solution.shape[0],))
```

Formulazione distribuita:

```
N = 20 # n. di processori, ogni processore ha un pezzetto di dataset
n_iter = 500 # n. di iterazioni
n_samples = math.floor(A.shape[0] / N) # suddividione del dataset tra i 20 processori

X = np.zeros((n_iter, N, n_features)) # NOTAZIONE: X[k,i,:] corrisponde al vettore x_i alla k-esima iterazione
Z = np.zeros((n_iter, n_features))      # NOTAZIONE: Z[k,:] corrisponde al vettore z alla k-esima iterazione
U = np.zeros((n_iter, N, n_features)) # NOTAZIONE: U[k,i,:] corrisponde al vettore u_i alla k-esima iter.
LOSS_1 = np.zeros(n_iter) # contiene l'andamento della LOSS relativo allo step 1 per gli N processori
```

```
for k in range(0,n_iter-1,1):
    # --------- STEP 1 ----------
    count = 0
    for i in range(N):
        x_cp = cp.Variable(n_features)
        loss = cp.sum(cp.pos(np.ones(n_samples) - A[count:count+n_samples,:] @ x_cp))
        reg = cp.sum_squares(x_cp - Z[k,:] + U[k,i,:])
        aug_lagr = loss/m + (rho/2)*reg
        prob = cp.Problem(cp.Minimize(aug_lagr))
        prob.solve(solver=cp.ECOS)#verbose=True, adaptive_rho = False,
        X[k+1,i,:] = x_cp.value

        #Calcolo LOSS
        for j in range(n_samples):
            cost = 1 - np.inner(A[count+j,:], X[k+1,i,:])
            if cost >0:
                LOSS_1[k+1] += cost
            LOSS_1[k+1] += rho/2 * np.linalg.norm(X[k+1,i,:] - Z[k,:] + U[k,i,:])**2

        count += n_samples


    # --------- STEP 2 ----------
    mean_X = np.zeros(n_features)
    mean_U = np.zeros(n_features)
    for i in range(N):
        mean_X += X[k+1,i,:]
        mean_U += U[k,i,:]
    mean_X = 1/N * mean_X
    mean_U = 1/N * mean_U

    for i in range(n_features-1):
        if mean_X[i] + mean_U[i] > lamda/(N*rho):
            Z[k+1,i] = mean_X[i] + mean_U[i] - lamda/(N*rho)
        elif mean_X[i] + mean_U[i] < - lamda/(N*rho):
            Z[k+1,i] = mean_X[i] + mean_U[i] + lamda/(N*rho)
        else:
            Z[k+1,i] = 0
    Z[k+1,n_features-1] = mean_X[n_features-1] + mean_U[n_features-1] #l'ultima è un caso particolare


    # --------- STEP 3 ----------
    for i in range(N):
        U[k+1,i,:] = U[k,i,:] + X[k+1,i,:] - Z[k+1,:]

print(Z[n_iter-1,:])#/Z[n_iter-1,0]
```

Andando a controllare numericamente le soluzioni ottenute possiamo affermare che i due algoritmi convergono alla stessa soluzione.

Coefficienti algoritmo centralizzato:

0.059509, 0, 0, -0.041129, 0.013422, -0.125327, 0.077078, -0.029868, 0, 0, 0.067933, -0.072551, 0, 0, 0.046497, -0.042021, 0, -0.008500, 0, 0.001006, -0.171363

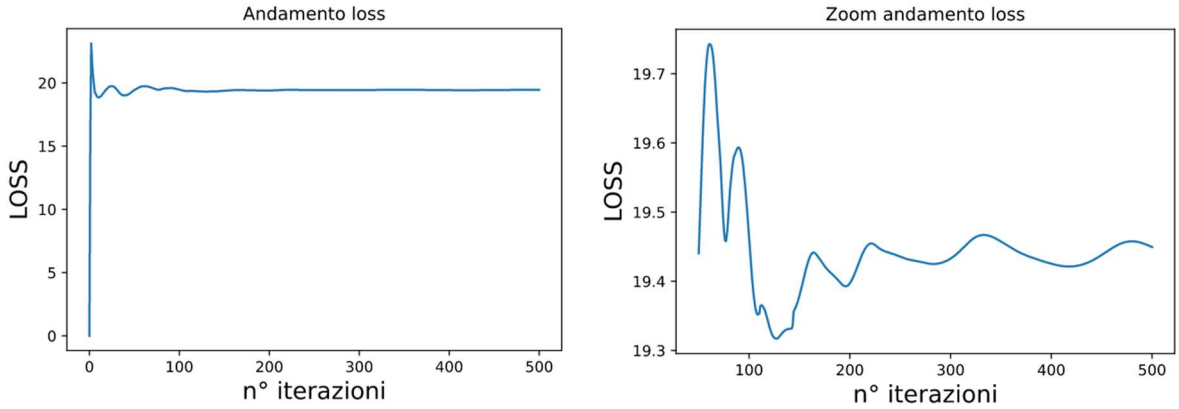
Coefficienti algoritmo distribuito:

0.059721, 0, 0, -0.040628, 0.013971, -0.125124, 0.077057, -0.030355, 0, 0, 0.068140, -0.072568, 0, 0, 0.046229, -0.041881, 0, -0.008448, 0, 0.001258, -0.170985

Può avere senso andare a graficare la LOSS relativa allo step 1 per uno degli N agenti coinvolto, per capire dopo quanto tempo l'algoritmo arriva in convergenza.

La loss function in questo caso è la funzione che l'i-esimo processore si trova a dover minimizzare:

$$\underline{x}_i^{(k+1)} = \underset{\underline{x}_i}{argmin}\left(\frac{1}{n}\sum_{i=1}^{n_j} max\{0, 1 - \underline{a}_{ij}{}^T \underline{x}_i\} + \frac{\rho}{2}\left\|\underline{x}_i - \underline{z}^{(k)} + \underline{u}_i^{(k)}\right\|_2^2\right)$$

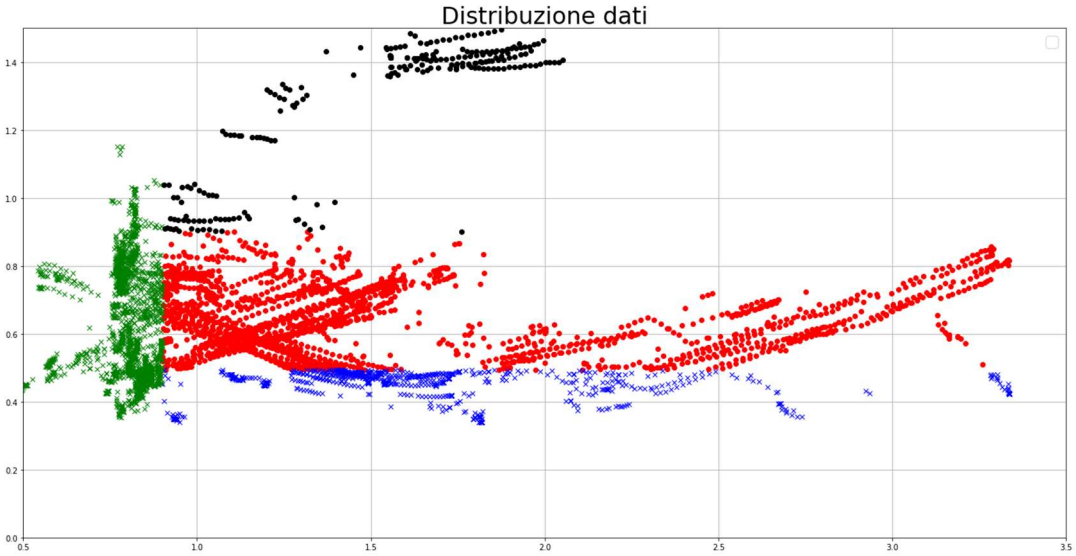


Dai grafici si evince che dopo 200 iterazioni si è praticamente arrivati in convergenza.

Parte 3 – Applicazione a un dataset reale

Come ultimo step si è deciso di testare l'algoritmo distribuito con un dataset reale molto semplice. Il dataset contiene i dati grezzi raccolti da 2 sensori disposti sulla vita di un robot che si muove in senso antiorario in una stanza seguendo il muro. I sensori misurano la minima distanza che il robot percepisce alla sua sinistra e la minima distanza che percepisce di fronte a sé. In base alla misura effettuata il robot prende una tra 4 decisioni: "vai avanti", "ruota leggermente a destra", "ruota a destra" o "ruota leggermente a sinistra".

La distribuzione delle quattro classi rispetto alle due features è la seguente:



I dati sono quasi linearmente separabili, quindi SVM dovrebbe funzionare bene.

Il problema che ci si trova ad affrontare è quello della classificazione multiclasse, si deve predirre una tra 4 possibili uscite. SVM nasce per risolvere il problema della classificazione binaria ma può essere facilmente esteso al caso multiclasse con uno dei seguenti approcci: OvA o OvO. Se c è il numero di classi, con l'approccio One Versus One si vanno a costruire $\frac{c*(c-1)}{2}$ classificatori binari (uno per ciascuna coppia di classi) e si addestrano utilizzando solo i dati relativi alle due classi considerate. Nel nostro caso i classificatori da costruire sono 6.

Per prima cosa il dataset è stato suddiviso in due parti, una per addestrare i classificatori e una per testarne le prestazioni. Poiché il dataset descrive le decisioni che un robot prende mentre fa 5 volte il giro di una stanza, si è deciso di assegnare al training una porzione che corrisponde all'80% del dataset totale (circa 4 giri).

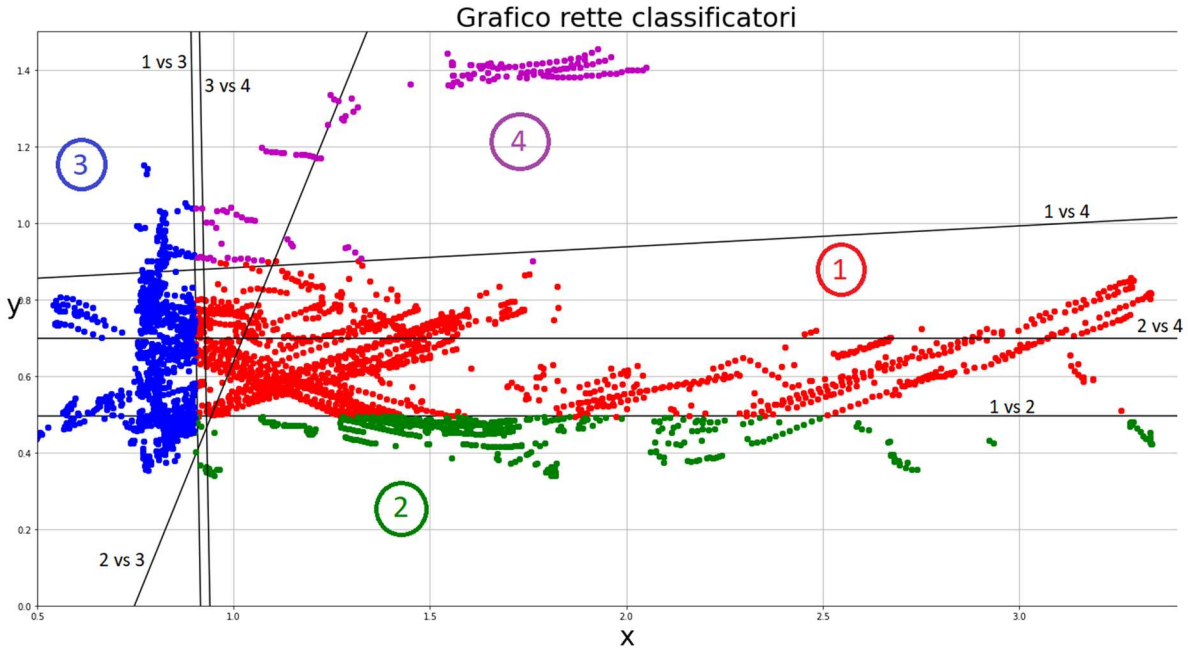
```
df = pd.read_csv("../input/wall-following-robot/sensor_readings_2.csv")
df.columns = ['SD_front', 'SD_left', 'Label']
class_names = ['Move-Forward', 'Slight-Right-Turn', 'Sharp-Right-Turn', 'Slight-Left-Turn']
output_dictionary = {'Move-Forward': 1, 'Slight-Right-Turn': 2, 'Sharp-Right-Turn': 3, 'Slight-Left-Turn': 4}

x1 = df['SD_front'].to_numpy() #prima feature
x2 = df['SD_left'].to_numpy() #seconda feature
y  = df['Label'].replace(output_dictionary).to_numpy()  #classe
train_samples = np.int(np.around(x1.shape[0]*0.8))
n_iter = 500
```

```
x1_train = x1[0:train_samples]
x2_train = x2[0:train_samples]
y_train = y[0:train_samples]

x1_test = x1[train_samples:y.size]
x2_test = x2[train_samples:y.size]
y_test = y[train_samples:y.size]
```

L'algoritmo distribuito descritto precedentemente è stato riportato all'interno di una funzione chiamata con il nome *svm* che prende in input le etichette delle due classi su cui deve fare classificazione binaria e restituisce i coefficienti della retta ottenuti. Le seguenti righe di codice sono per l'addestramento dei 6 classificatori.

```
# ---------------------------- TRAIN ----------------------------
beta_tilde_1 = svm(1,2)
beta_tilde_2 = svm(1,3)
beta_tilde_3 = svm(1,4)
beta_tilde_4 = svm(2,3)
beta_tilde_5 = svm(2,4)
beta_tilde_6 = svm(3,4)

plot_train(beta_tilde_1, 1,2,'-r','-g') # 1 vs 2
plot_train(beta_tilde_2, 1,3,'-r','-b') # 1 vs 3
plot_train(beta_tilde_3, 1,4,'-r','-m') # 1 vs 4
plot_train(beta_tilde_4, 2,3,'-g','-b') # 2 vs 3
plot_train(beta_tilde_5, 2,4,'-g','-m') # 2 vs 4
plot_train(beta_tilde_6, 3,4,'-b','-m') # 3 vs 4
```

Le rette ottenute sono mostrate in figura seguente. Accanto ad ogni retta sono riportate le etichette delle classi che la retta va a separare.



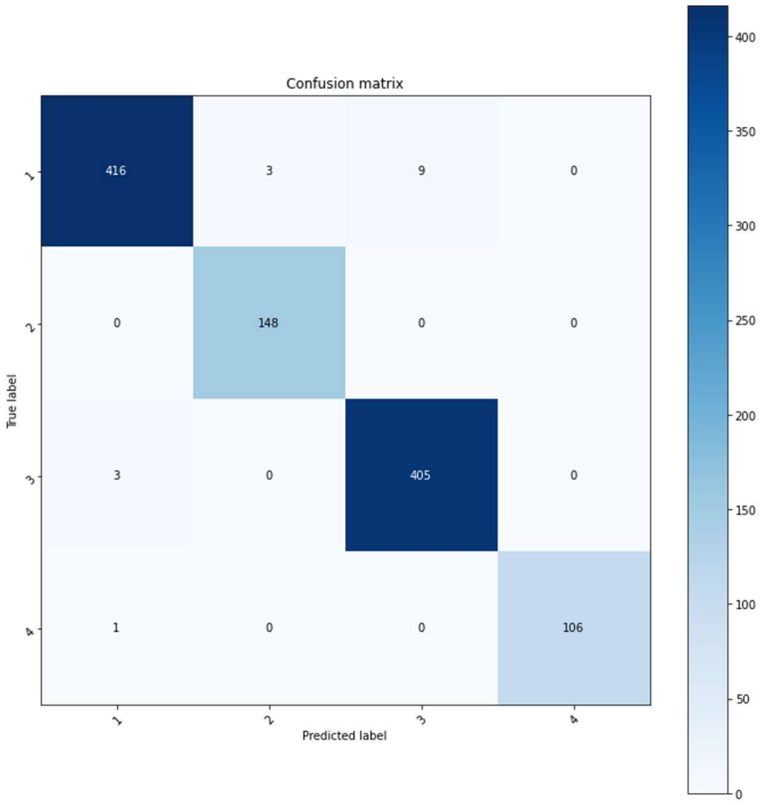
Nella fase di test, per decidere quale etichetta assegnare a un dato, si utilizza un meccanismo di maggioranza: si applica il dato a tutti e 6 i classificatori e gli si assegna la classe che risulta vincente un maggior numero di volte. Il codice per valutare le prestazioni sul test set viene riportato qui di seguito.

```
# ------------------------------------------  TEST  ------------------------------------------
y_pred = np.zeros(y_test.size,dtype='int') # vettore che contiene la predizione
for i in range(y_test.size):
    pred_count = np.zeros(7) # la cella i identifica il classificatore i,
    #Classificatore 1
    a = beta_tilde_1[0]
    b = beta_tilde_1[1]
    c = beta_tilde_1[2]
    if x1_test[i]*a + x2_test[i]*b + c > 0: # prodotto scalare
        pred_count[1] += 1
    else:
        pred_count[2] += 1

    #Classificatore 2
    a = beta_tilde_2[0]
    b = beta_tilde_2[1]
    c = beta_tilde_2[2]
    if x1_test[i]*a + x2_test[i]*b + c > 0:
        pred_count[1] += 1
    else:
        pred_count[3] += 1

    #Classificatore 3
    a = beta_tilde_3[0]
    b = beta_tilde_3[1]
    c = beta_tilde_3[2]
    if x1_test[i]*a + x2_test[i]*b + c > 0:
        pred_count[1] += 1
    else:
        pred_count[4] += 1

    #Classificatore 4
    a = beta_tilde_4[0]
    b = beta_tilde_4[1]
    c = beta_tilde_4[2]
    if x1_test[i]*a + x2_test[i]*b + c > 0:
        pred_count[2] += 1
    else:
        pred_count[3] += 1

    #Classificatore 5
    a = beta_tilde_5[0]
    b = beta_tilde_5[1]
    c = beta_tilde_5[2]
```

```
    if x1_test[i]*a + x2_test[i]*b + c > 0:
        pred_count[2] += 1
    else:
        pred_count[4] += 1

    #Classificatore 6
    a = beta_tilde_6[0]
    b = beta_tilde_6[1]
    c = beta_tilde_6[2]
    if x1_test[i]*a + x2_test[i]*b + c > 0:
        pred_count[3] += 1
    else:
        pred_count[4] += 1

    y_pred[i] = np.argmax(pred_count) # maggioranza (l'indice con più count è la classe assegnata in output)
```

Per andare a comprendere numericamente quanti errori di classificazione si vanno a commettere si può calcolare la matrice di confusione.

```
plot_confusion_matrix(y_test, y_pred, 'Confusion matrix', normalize=False)
```



Su un totale di 1091 esempi presenti nel test set, solo 16 vengono classificati male.