Assignment 2

Contents

1. Il metodo del gradiente (Gradient Descent) per il modello di machine learning Support Vector Machine (SVM)

Prima di implementare il metodo del gradiente è opportuno specificare la funzione di perdita impiegata

$$h(w) = 1/n \sum_{i=1}^{n} [1/2||w||^2 + C * max(0, 1 - y_i(w_0 x_i))] \tag{1}$$

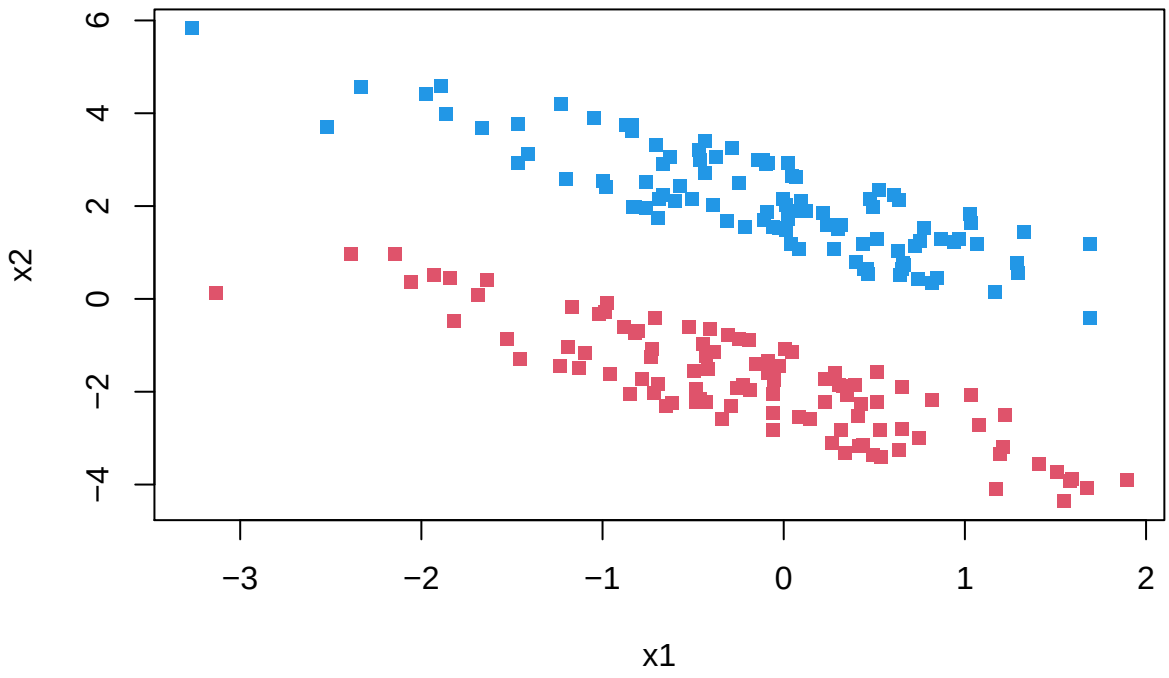
Questa funzione di perdita è composta da due elementi perincipali. La prima parte permette di massimizzare la distanza tra i due margini costruiti sui vettori di supporto; così facendo si può ottenere l'iperpiano equidistante da entrambi, ovvero il miglior iperpiano per separare le oservazioni. La seconda parte è una funzione di hinge loss che permette di attribuire flessibilità all'algoritmo nella fase di separazione delle osservazioni. Infatti, al contrario di una support vector machine a margine inflessibile, che non permette classificazioni errate, questa in esame lascia la possibilità all'algoritmo di classificare erroneamente delle osservazioni, punendo però tale comportamneto con una funzione lineare della distanza dell'osservazione dal margine. Per ottenere i coefficienti (w) di una support vector machine a soft-margin si deve quindi minimizzare la funzione di perdita che, nel caso della hinge loss, è conosciuta essere convessa e derivabile.

Calcolando il gradiente della funzione rispetto ai pesi w per una singola iterazione si ottiene

$$\frac{df}{dw_j} = \begin{cases} -x_{ij} y_i & y_i(w^T x_i + w_0) \ < 1 \\ 0 & y_i(w^T x_i + w_0) \geq 1 \end{cases} \tag{2}$$

In generale come gradiente su tutte le osservazioni X si considera la sommatoria dei risultati minori di uno, che dovrà quindi essere minimizzata.

Come primo passaggio per risolvere il problema si crea un dataset contenente 200 osservazioni distribuite in maniera utile all'implementazione di una support vector machine.

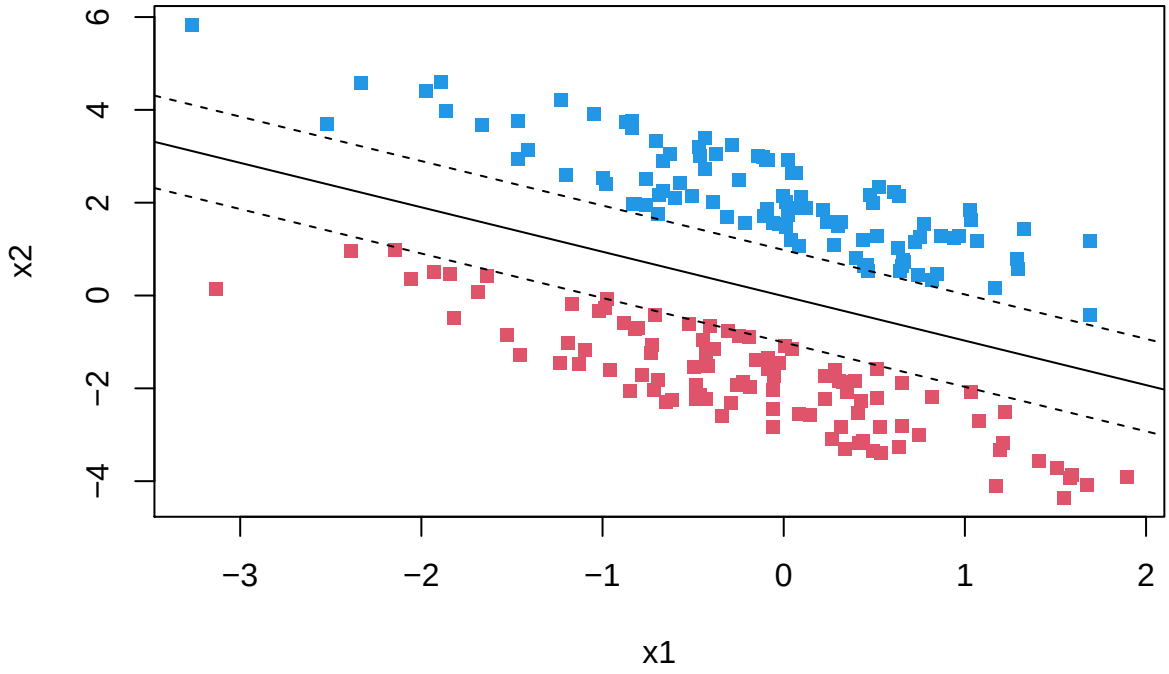


Creato il dataset delle osservazioni si procede ad implementare l'algoritmo per la stima dei coefficienti della SVM.

Il metodo del gradient descent classico viene implementato partendo da un punto iniziale specificato precedentemente, poi, per ogni punto successivo, calcola il gradiente della funzione di pedita in quel punto e lo utilizza come direzione lungo cui muoversi. Il passo compiuto da quel punto lungo la direzione del gradiente ed il numero delle iterazioni sono fornite anticipatamente. In questo esempio, dovendo minimizare la funzione, si è preso il negativo del gradiente in ogni punto come direzione, mentre il learning rate si è tenuto costante per ogni processo di stima. Come criterio di stop della procedura si è utilizzato il numero di iterazioni, posto uguale a 1000, anche se molte volte la convergenza è stata raggiunta prima. Di seguito l'implementazione dell'algoritmo di gradient descent classico

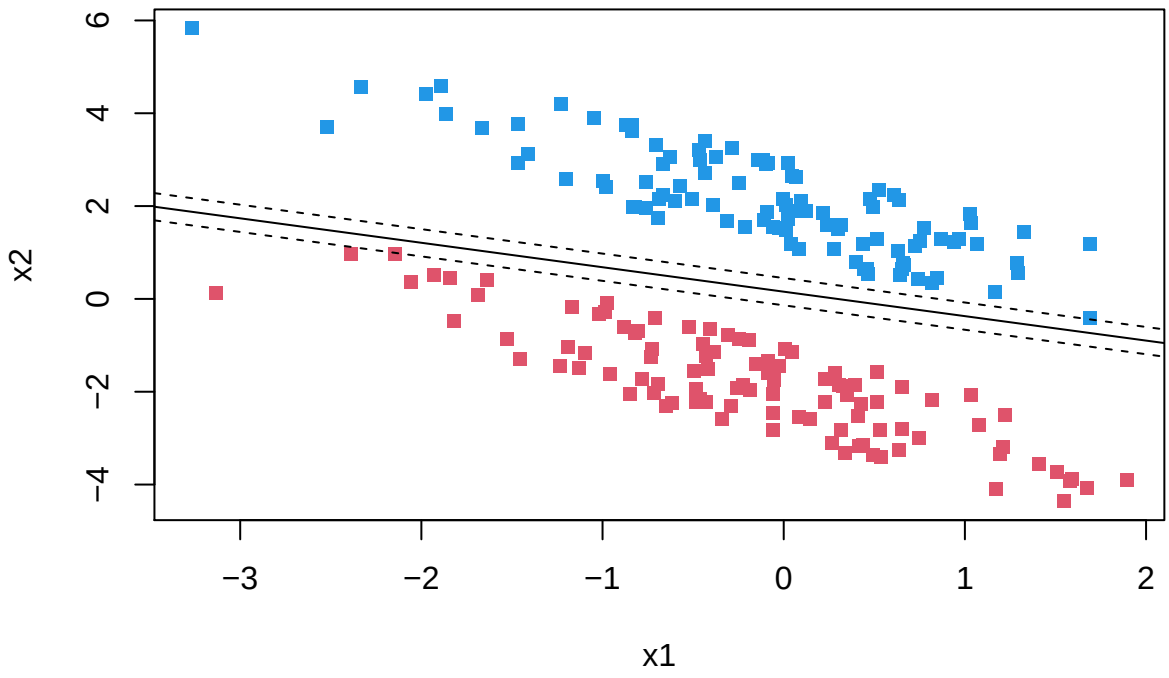
```
grd<- function(x,lr,iterazioni=1000, coef_iniziali){
  X<- cbind(1,x)
  n <- nrow(X)
  p <- ncol(X)
  w_intial <- coef_iniziali
  W <- matrix(w_intial ,nrow = iterazioni+1,ncol = p,byrow = T)
  for(i in 1:iterazioni){
    for(j in 1:p)
    {
      W[i+1,j]<- W[i,j]+lr*sum(((y*(X%*%W[i,]))<1)*1 * y * X[,j] )
    }
  }
  return(W)
}
```

La stima dei coefficienti è stata effettuata diverse volte, variando qualche parametro dell'algoritmo. Per primo si è utilizzato un learning rate costante di 0.001, ma si sono scelti due punti diversi di partenza. Inizialmente il valore dei coefficienti di partenza è stato posto a B0(0,0,0), così facendo si è ottenuta una convergenza in 110 iterazioni, con dei valori per i coefficienti stimati finali di B(0.002, 0.962, 1.004). Poi si è proceduto ripetendo il processo di stima, ma utilizzando come valori di partenza B(1,2,3), in questo caso la convergenza è avvenuta in 91 iterazioni su valori sostanzialmente uguali ai precedenti. I coefficienti così stimati che minimizzano la funzione obiettivo definiscono una la seguente support vector machine:



Il processo di stima ha ottenuto quindi dei coefficienti che forniscono una buona rappresentazione di una SVM.

Successivamente si è provato ad alterare il learning rate, alzandolo da 0.001 a 0.1, questo ha portato ad una stima dei coefficienti pari a B(0.400, 1.962, 3.528). Come previsto aumentando il learning rate l'algoritmo non riesce a convergere sull'ottimo, ma continua a rimbalzare nel suo intorno. Al contrario impostando un learning rate troppo basso, esso impiegherebbe una quantità troppo elevata di iterazioni per convergere finalmente all'ottimo della funzione. La rappresentazione grafica della support vector machine con coefficienti stimati attraverso un learning rate di 0.01 è la seguente:



Chiaramente meno ottimale della prima.

La rappresentazione grafica mostra chiaramente una support vector machine più sbilanciata della precedente.

Infine, come mezzo di confronto, si è utilizzato il pacchetto e1071 di R per stimare automaticamente i coefficinti della support vector machine, con questa procedura i valori ottenuti sono B(-0.024, 0.972, 0.978), molto simili a quelli ottimali stimati inizialmente. La differenza è da attribuirsi al fatto che il pacchetto preconfezionato di R utilizza la modificazione stocastica del metodo del gradiente, impiegando solo alcune delle osservazione per la valutazione del modello di passaggio in passaggio.

2. Basandosi sullo stesso modello utilizzato al punto 1, implementare il metodo del Mini-Batch Stochastic Gradient Descent

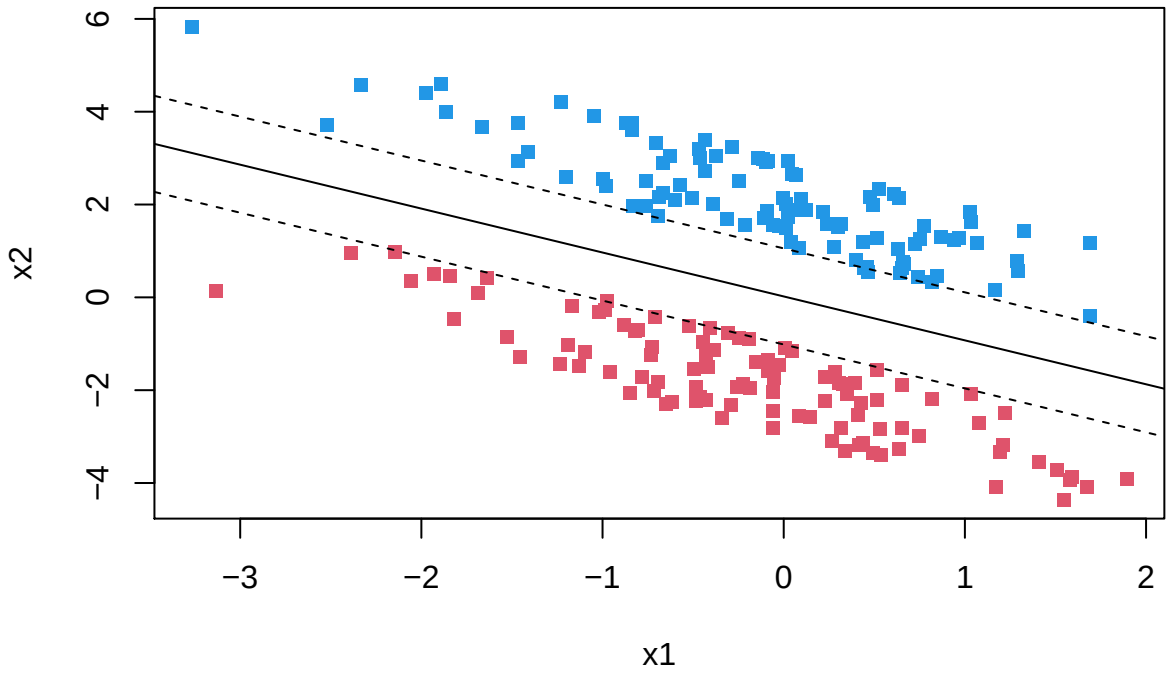
Questa modificazione dell'algoritmo principale consiste nel valutare i coefficienti per ogni passaggio non sulla totalità delle osservaioni nel dataset, ma solo su mini-batch da 10 osservazioni l'uno estratti casualmente ad ogni aggiornamento dei coefficienti. Ciò permette di alleggerire la necessità computazionale dell'algoritmo. La sua implementazione è la seguente:

```
grd_mini_batch<- function(x,lr,iterazioni=1000, coef_iniziali){
  X<- cbind(1,x)
  n <- nrow(X)
  p <- ncol(X)
 w_intial <- coef_iniziali
  W_iter <- matrix(w_intial ,nrow = iterazioni+1,ncol = p,byrow = T)
  for(i in 1:iterazioni){
    xysamp <-as.matrix( data[sample (nrow(data), 10, replace = TRUE), ] )
```

```
    xsamp <- cbind(1,matrix(ncol=2, cbind(as.numeric(xysamp[,1]), as.numeric(xysamp[,2]))))
    ysamp <- matrix(as.numeric(xysamp[,3]))
      for(j in 1:p)
      {
        W_iter[i+1,j]<- W_iter[i,j]+lr*sum(((ysamp*(xsamp%*%W_iter[i,]))<1)*1 * ysamp * xsamp[,j] )
      }
  }
  return(W_iter)
}
```

Il vantaggio di questo metodo è che consente un'analisi più veloce e meno dispendiosa sotto un profilo computazionale, inoltre è anche la giusta via di mezzo tra il metodo del gradiente classico, molto preciso, ma costoso, e la sua variante stocastica, più veloce, ma anche più imprecisa. Rispetto al metodo completo risulta comunuqe leggermento meno peciso, infatti, applicandolo al dataset si è ottenuto come risultato B(-0.023, 0.911, 0.960). Il learning rate è stato impostato pari a 0.001.

La support vector machine implementata risulta comunque ottima:



3. Risolvere il problema al punto 2 implementando un algoritmo metaeuristico

L'algoritmo metaeuristico scelto per questo problema è l'Hill Climbing. Questo algoritmo, al contario di quelli visti fino ad ora, non necessita del calcolo del gradiente per essere implementato, ma della sola funzione di perdita. Esso si basa infatti su un processo iterativo che inizia da un punto casuale che viene scelto come soluzione migliore iniziale. Poi procede nell'eseguire una copia di questo punto e a modificarla leggermente, fatto ciò la nuova soluzione modificata viene valutata e paragonata alla miglior soluzione. Se

è ritenuta migliorativa, nel caso specifico se minimizza la funzione di perdita, viene impostata come nuova soluzione migliore ed il processo è ripetuto fino a che non è più possibile trovare soluzioni che apportino un miglioramento o fino allo scadere delle iterazioni concesse.

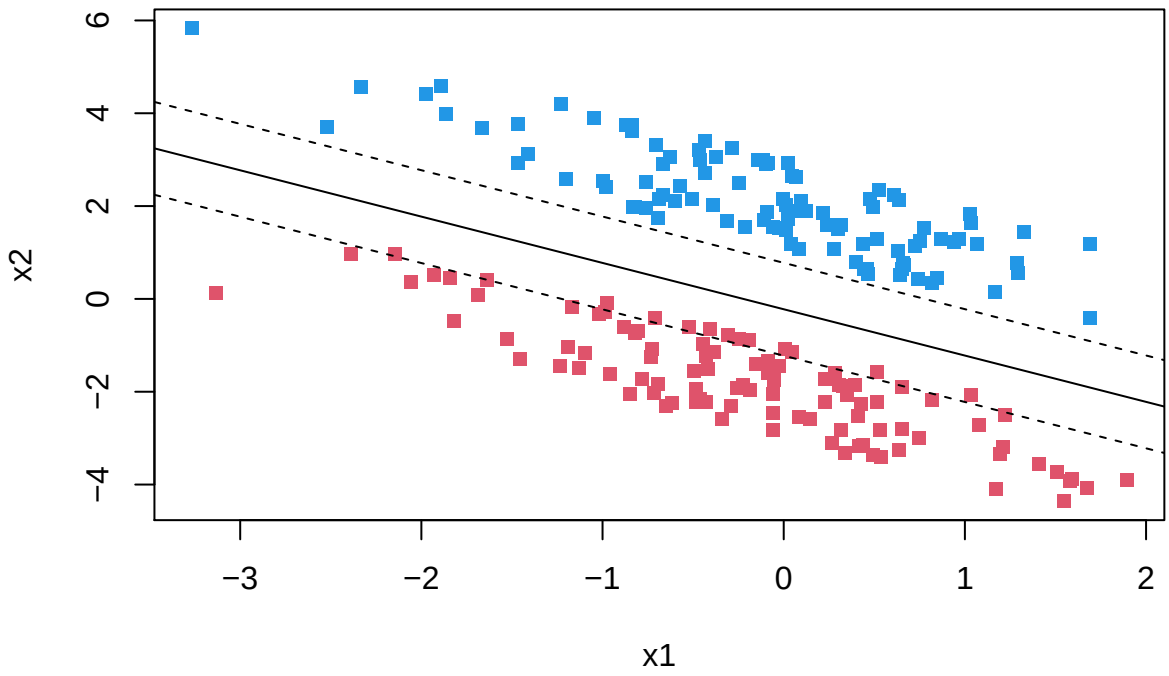
A livello concettuale l'algoritmo segue una struttura del tipo seguente.

```
# dopo aver impostato la fnzione obiettivo desiderata
for (i in iterazioni) {
    gold_coeff<- candidate_sol[i,] #inizialmente posta pari a B(1,1,1)
    gold_state<- obj_func(x, w=gold_coeff)
    new_coeff<-candidate_sol[i,]+candidate_sol[i,]*tweaks # dove tweaks è un modo per alterare la copia
    new_state= obj_func(x, w=new_coeff)
    if (new_state < gold_state){
          gold_state<-new_state
          gold_coeff<-new_coeff
    }
}
  candidate_sol[i+1,]<-new_coeff
```

Nell'implementazione effettiva si è utilizzato un limite di 1000 iterazioni, un coefficiente C nella funzione obiettivo pari a 100 e, come modifica rispetto alla soluzione precedente, si è consierata la somma di 0.001 unità. Impostato un punto di partenza pari a B0(0,0,0) l'algoritmo ha ottenuto una convergenza alla soluzione B(0.120, 0.998, 1.00), valore molto simile a quelli precedentemente stimati con l'approcio del gradiente.

```
hill_climbing<- function(x,iterazioni=1000){
  X<- cbind(1,x)
  n <- nrow(X)
  p <- ncol(X)
  intial <- rep(0,p)
  candidate_sol<- matrix(intial ,nrow = iterazioni+1,ncol = p,byrow = T)
  new_sol<- matrix(rep(0,p) ,nrow = iterazioni+1,ncol = p,byrow = T)
  for(i in 1:iterazioni){
    for(j in 1:p){
      new_sol[i,j]<- candidate_sol[i,j]+0.001
      obj_func<-function (x, w){
        ris<-(1/n)*sum(1/2*(sqrt(sum(w[i,j]^2)))+100*max(0,1-y*(X%*%w[i,])))
        return(ris)
      }
      if (obj_func(x, w=new_sol)<obj_func(x, w=candidate_sol)) {
        candidate_sol[i+1,j]<-new_sol[i,j]
      }
    }
  }
  candidate_sol[,1:2]<-candidate_sol[,2:1]
  return(candidate_sol)
}
```

La rappresentazione grafica della support vector machine ottenua con questi coefficienti è la seguente


x2
6
4
2
0
−2
−4
−3
−2
−1
0
1
2
x1