

Architetture e Programmazione dei Sistemi di Elaborazione
Progetto a.a. 2020/21

“Polynomial Regression and Stochastic Gradient Descent”
in linguaggio assembly x86-32+SSE, x86-64+AVX e openMP

Fabrizio Angiulli Fabio Fassetti

Gruppo 5

Calabrò Michele (224521)

Colantonio Viviana (224473)

Salatino Francesco (227543)

Salerno Giuseppe (224525)

1. Introduzione: Algoritmo di Regressione polinomiale e discesa stocastica del gradiente

Dato un insieme di osservazioni riguardanti le variabili x e y e fissato un insieme di funzioni $f(x, \Theta)$ da usare per approssimare la relazione tra x e y , ossia:

$$y \sim f(x, \Theta)$$

il problema consiste nel trovare i parametri theta della funzione f per cui sia minimizzato l'errore di approssimazione. Nel caso di regressione polinomiale f è un polinomio di grado *degree*.

La discesa stocastica del gradiente è un metodo iterativo che, ad ogni iterazione, sostituisce il valore esatto del gradiente della funzione di costo con una stima ottenuta valutando il gradiente solo su un sottoinsieme di addendi. Più in dettaglio, l'algoritmo stima i parametri Θ che minimizzano una funzione di costo $C(\Theta, x, y)$, aggiornando iterativamente Θ e, in particolare, Θ_{t+1} viene ottenuto sottraendo a Θ_t il gradiente della funzione di costo C calcolato rispetto a Θ , dove Θ_t indica il valore dei parametri all'interazione t . Nel caso particolare di funzione di costo pari all'errore quadratico medio, $\nabla \Theta C(\Theta_t, x_i) = (y_i - f(x_i, \Theta)) \cdot x_i^*$.

x^* indica il vettore i cui elementi x_j^* sono ottenuti come:

$$\forall h \in [0, \deg] \quad \forall J \in [d]^h \quad x_j^* = \prod_{i=1}^h x_{J_i}.$$

2. Funzione dell'algoritmo e utilizzo nel caso specifico

L'algoritmo di Regressione Polinomiale può essere suddiviso in due parti indipendenti:

- Conversione del Dataset
- SGD nelle sue due varianti (Batch e AdaGrad)

La prima problematica riguarda il calcolo del dataset convertito a partire da un insieme di osservazioni X .

La funzione che esprime tale conversione è la seguente:

$$X, d \rightarrow X^*$$

Dove X corrisponde a un vettore d dimensionale nella forma $[x_1, \dots, x_d] \in \mathbb{R}^d$ e X^* corrisponde al vettore dimensionale X esploso nelle sue potenze.

Esempio:

$$X \in \mathbb{R}, d=2$$

$$X_1 \in [1, X_1, X_1^2]$$

Oppure ancora:

$$X = [X_1, X_2], d=2$$

$$X^* = [1, X_1, X_2, X_1^2, X_1 X_2, X_2^2, X_1^3, X_1^2 X_2, X_1 X_2^2, X_2^3, X_1^4, X_1^3 X_2, X_1^2 X_2^2, X_1 X_2^3, X_2^4].$$

La conversione del dataset è essenziale per l'algoritmo della discesa stocastica del gradiente (poiché fa uso dell'errore quadratico medio come misura dell'errore di stima) e ne costituisce un input.

3. Implementazione e Linee guida generali

Dopo uno studio approfondito dello pseudocodice riportato nella traccia, per prima cosa è stato realizzato il codice in C.

Il metodo che implementa l'algoritmo di conversione del dataset è

```
void convert_data(params* input)
```

A partire dallo pseudocodice riportato nella traccia del progetto, per prima cosa è stato realizzato il codice C dell'algoritmo SGDBATCH e la variante che utilizza l'algoritmo *AdaGrad* come acceleratore.

Il metodo che implementa l'algoritmo è

```
void sgd(params* input)
```

il quale prende in input la struttura dati *params* in modo da avere a disposizione tutti i parametri di input. Il controllo sulla richiesta o meno dell'accelerazione *AdaGrad* viene effettuato tramite un if esterno sul parametro *adagrad*, in modo da permettere l'esecuzione esclusiva dell'algoritmo SGDBATCH con accelerazione o meno.

3.1 Implementazione in C e prima ottimizzazioni

Per prima cosa il metodo si preoccupa di calcolare il numero esatto di colonne che avrà il dataset convertito, cioè la matrice *x_ast*.

Questo numero corrisponde esattamente:

$$\sum_{h=0}^{\text{degree}} Cr(d, h)$$

Dove *Cr* sta per Combinazioni con ripetizione di *d* elementi a gruppi di *h*. Dove *d* indica il numero di variabili in input.

Abbiamo tradotto la sommatoria con un for che chiama *degree* volte il metodo “**combinazioni**”.

Una sua prima implementazione è la seguente:

```
int combinazioni(int n,int k){  
    return fattoriale(n+k-1)/(fattoriale(k)*fattoriale(n-1));  
}
```

Che è esattamente la formula delle combinazioni con ripetizione.

Questa prima versione andava assolutamente ottimizzata perché calcola tre fattoriali il cui costo è non banale.

La versione finale del metodo **combinazioni** è la seguente:

```

void combinazioni(int *v, int n, int k){

    int num= n+k-1;
    int den=n-1;
    int div=k;

    if(den<div){
        div= den;
        den=k;
    }

    int parziale=1;
    int fact=1;

    for(int i=num;i>den;i--){
        parziale*=i;
        fact*=div;
        div--;
    }

    v[k]=parziale/fact;
}

```

Si è cercato di semplificare al massimo il calcolo del fattoriale.

Il metodo esegue esattamente la “semplificazione carta e penna” e successivamente in un solo ciclo calcola in *parziale* il dividendo (ovvero la moltiplicazione di ciò che è rimasto al numeratore dopo le semplificazioni), e in *fact* il divisore. Il numero di iterazioni è stato limitato.

Il risultato viene memorizzato in un vettore, che ci tornerà utile in seguito.

Il numero di colonne della matrice *x_ast* è dato dalla somma dei valori calcolati alle varie iterazioni del metodo *combinazioni*.

Viene dunque allocata la matrice *x_ast*, che conterrà il dataset convertito.

Come primo step, per ogni osservazione viene scritta la prima colonna del dataset convertito.

Prima di vedere l’ultima parte dell’algoritmo, bisogna fare una premessa sulla matrice *J*. Essa conterrà alla generica iterazione *h* tante righe quante sono le possibili combinazioni con ripetizione di *d* elementi a gruppi di *h*. Esempio per *d*=2, *h*=3:

$$J \in \{(1, 1, 1), (1, 1, 2), (1, 2, 1), (1, 2, 2), (2, 1, 1), (2, 1, 2), (2, 2, 1), (2, 2, 2)\}$$

L’ultima parte dell’algoritmo è quella che popola il dataset. Per ogni *h* t.c. $1 \leq h \leq \text{degree}$ si popolano tante colonne di *x_ast* quanti sono i termini di grado *h* che possono essere generati su *d* variabili. Il numero di colonne da popolare per ogni iterazione è stato già calcolato in precedenza e memorizzato nell’array *vettore_combinazioni*. Alla generica iterazione del ciclo:

- 1 Nella variabile *tuple* viene inserito il numero di colonne da popolare, che corrisponde al numero di righe in J.
- 2 Si effettua una realloc su J, allocando lo spazio di memoria necessario per l'iterazione corrente.
- 3 Si chiama il metodo ***calcolaJ***, che popola la matrice J.
- 4 Si chiama il metodo ***popola*** che scrive le colonne di x_ast (il numero di colonne scritte è pari al numero di righe in J, cioè “tuple”)

Per tenere traccia della prossima colonna da riempire in x_ast viene usata la variabile *coldest*, inizializzata a 1 e incrementata di *tuple* ad ogni iterazione del ciclo.

Una particolare menzione va riservata al metodo ***calcolaJ***.

```
void calcolaJ(int* J, int d, int h)
```

Esso si basa su un procedimento iterativo che viene così descritto:

Ad ogni elemento d viene associato un numero $1 \leq n \leq d$. Il risultato corrisponde a un insieme di n-pla di h elementi.

- 1 Viene inizializzato il primo vettore riga con tutti “1”.
- 2 Attraverso una variabile memorizzo il riferimento a tale vettore.
- 3 Memorizzo nella variabile k l'indice della posizione più a destra (h-1)
- 4 Finché ci sono ancora numeri da incrementare ($k >= 0$):
 - 4.1 Creo un nuovo vettore riga copia di quello puntato.
 - 4.2 Mi posiziono sull'elemento k-esimo.
 - 4.3 Provo ad incrementare -> v:
 - 4.3.1 Se il numero v $\leq d$:
 - 4.3.1.1 Sovrascrivo l'elemento k-esimo con v.
 - 4.3.1.2 Copio tale elemento in tutte le posizioni comprese tra [k, h-1].
 - 4.3.1.3 Sposto il puntatore riga sull'attuale riga.
 - 4.3.1.4 Pongo k= h-1
 - 4.3.1.5 Vado al passo 5
 - 4.3.2 Se il numero v $> d$:
 - 4.3.2.1 Pongo k= k-1
 - 4.3.2.2 Vado al passo 5.2

Come si può notare, ogni vettore riga mantiene un ordine crescente.

Tale procedimento iterativo viene eseguito esattamente un numero di volte pari alle combinazioni con ripetizione di d elementi a gruppi di h, e scarta automaticamente i termini duplicati.

Nelle immagini sono rappresentati i tempi di conversione del dataset fornito per i testing, con degree pari a 4 e successivamente a 10.

```
pep@pep-X580VD ~/Scrivania/Progetto $ ./runregression32 ./Test/datasets/test_2000x4_4f -batch 2 -degree 4 -eta 0.001 -iter 1000
Input data name: './Test/datasets/test_2000x4_4f.data'
Input label name: './Test/datasets/test_2000x4_4f.labels'
Data set size [n]: 2000
Number of dimensions [d]: 4
Batch dimension: 2
Degree: 4
Eta: 0.001000
Adagrad disabled
Conversion time = 0.007 secs
```

```

pep@pep-X580VD ~/Scrivania/Progetto $ ./runregression32 ./Test/datasets/test_2000x4_4f -batch 2 -degree 10 -eta 0.001 -iter 1000
Input data name: './Test/datasets/test_2000x4_4f.data'
Input label name: './Test/datasets/test_2000x4_4f.labels'
Data set size [n]: 2000
Number of dimensions [d]: 4
Batch dimension: 2
Degree: 10
Eta: 0.001000
Adagrad disabled
Conversion time = 0.055 secs

```

Per semplificare l'implementazione in codice assembly, l'algoritmo SGD è stato scomposto in procedure più semplici. Tra queste troviamo i metodi:

- *azzeramentoDaSottrarre(daSottrarre,size)* , che effettua l'inizializzazione del vettore temporaneo *daSottrarre* utilizzato per il calcolo di theta e che ad ogni iterazione viene aggiornato con il risultato delle seguenti operazioni:

$$\sum_{j=i}^{i+\nu-1} (\langle \Theta, \mathbf{x}_j^* \rangle - y_j) \cdot \mathbf{x}_j^*$$

nella versione SGDBATCH

$$; \sum_{j=i}^{i+\nu-1} \frac{\eta}{\sqrt{G_j + \epsilon}} g_j$$

nella versione AdaGrad

- *prodottoScalare(&cost_molt, theta, xast, j*size, size)*, che memorizza all'interno della variabile *cost_molt* di tipo *type* il risultato di $\langle \theta, \mathbf{x}_j^* \rangle$
- *moltiplicazionePerScalare(g, xast, j*size, size, cost_molt)*, che effettua il prodotto

$$(\langle \Theta, \mathbf{x}_j^* \rangle - y_j) \cdot \mathbf{x}_j^*$$

memorizzandolo all'interno del vettore *g*.

- *quadratoVettore(g, riga*size, G, size)*, che effettua il calcolo di G_j nella versione AdaGrad come:

$$G_j := \sum_{t=1}^{it} g_j^2$$

- *faiSommatoria(daSottrarre, size, g, G, riga*size, epsilon)*, che effettua all'interno di un ciclo le operazioni seguenti, in modo da aggiornare il vettore temporaneo *daSottrarre*

$$; \sum_{j=i}^{i+\nu-1} \frac{\eta}{\sqrt{G_j + \epsilon}} g_j$$

- *calcoloTheta(daSottrarre, theta, eta/v, size)*, che in entrambe le versioni aggiorna il vettore theta sottraendo a quest'ultimo il risultato delle operazioni precedenti.
- *sommaVettori(daSottrarre, temp, size)*, che nella versione SGDBATCH effettua l'aggiornamento del vettore *daSottrarre*, sommando quest'ultimo al contenuto dell'iterazione precedente.

In una prima implementazione in linguaggio C, era stato scelto di allocare il gradiente *g* come matrice memorizzata per righe di dimensione pari al prodotto della dimensione del batch passato in input (parametro *k*) e della dimensione del vettore *theta* (ovvero il parametro *t*, indicato con *size* nel nostro metodo).

Successivamente, il codice è stato ottimizzato allocando g come vettore di dimensione $size$, evitando così per il calcolo di theta di implementare un ulteriore ciclo. Non era infatti necessario memorizzare tutti i vettori g_j , in quanto all'iterazione j -esima il vettore g_{j-1} può essere tranquillamente sovrascritto.

```
if(adagrad){
    type epsilon = 1E-8;
    int riga = 0;
    MATRIX g=(MATRIX) malloc(size*k*sizeof(type));
    MATRIX G =(MATRIX) malloc(size*k*sizeof(type));

    for(int m = 0; m<k; m++){
        for(int n=0; n<size; n++){
            g[m*size + n]=0;
            G[m*size + n]=0;
        }
    }

    while(t<iter){
        for(=0; <n;){
            //calcolo del minimo
            tr((n-l)*k) v=n-l;
            else v=k;

            //inizializzazione vettore da sottrarre
            for(int index=0; index<size; index++) daSottrarre[index]=0;

            for(j=0; j<v; j++){
                riga = j % k;

                //calcolo riga g_j
                prodottoScalare(&cost_molt, theta, xast, j, size);
                cost_molt cost_molt - y[j];
                moltiplicazionePerScalareADAGRAD(g,riga, xast, j, size, cost_molt);

                // G_j += g_j^2
                quadratoVettore(g, riga, G, size);
                faSomma(g, daSottrarre, size, g, G, riga, epsilon);
            }

            for(int h = 0; h<size; h++) {
                daSottrarre[h] *= eta/v;
                theta[h] = theta[h] - daSottrarre[h];
            }

            i=i+v;
        }
        it++;
    }
}
```

Versione originale



```
if(adagrad){
    type epsilon = 1E-8;
    int riga = 0;
    VECTOR g=(VECTOR) malloc(size*sizeof(type));
    MATRIX G =(MATRIX) malloc(size*k*sizeof(type));

    for(int m = 0; m<k*size; m++)
        G[m]=0;

    while(it<iter){
        for(i=0; i<n;){
            //calcolo del minimo
            if((n-i)<k) v=n-l;
            else v=k;

            //inizializzazione vettore da sottrarre
            for(int index=0; index<size; index++) daSottrarre[index]=0;

            for(j=i; j<i+v; j++){
                riga = j % k;

                //calcolo riga g_j
                prodottoScalare(&cost_molt, theta, xast, j, size);
                cost_molt cost_molt - y[j];
                moltiplicazionePerScalareADAGRAD(g,riga, xast, j, size, cost_molt);

                // G_j += g_j^2
                quadratoVettore(g, riga, G, size);
                faSomma(g, daSottrarre, size, g, G, riga, epsilon);
            }

            for(int h = 0; h<size; h++) {
                daSottrarre[h] *= eta/v;
                theta[h] = theta[h] - daSottrarre[h];
            }

            i=i+v;
        }
        it++;
    }
}
```

Versione ottimizzata

Questa strategia ha permesso di ottenere un primo miglioramento delle prestazioni dell'algoritmo, constatando sperimentalmente un miglioramento di circa 0,3 secondi nella versione a 64 bit.

64 ADAGRAD
64;30;4;3;3000;ADA;5790;0.002;**5.786**;0;2;0.790583

Versione originale

64 ADAGRAD
64;30;4;3;3000;ADA;5497;0.002;**5.491**;0;2;0.790583

Versione ottimizzata

3.2 Tecniche di ottimizzazione utilizzate

- LOOP UNROLLING

La tecnica rientra nella categoria del **parallelismo implicito** e sfrutta tecniche **ILP-based** (Instruction Level Parallelism). È una forma di parallelismo trasparente al programmatore, che non ne ha un pieno comando. Il programmatore scrive un programma costituito da istruzioni sequenziali. Poiché nel calcolatore ci sono più ALU, in maniera trasparente al programmatore si prende un blocco di istruzioni indipendenti e si eseguono in contemporanea nelle unità di calcolo, riducendo i tempi di esecuzione.

- LOOP VECTORIZATION

La seguente tecnica di ottimizzazione rientra nella categoria del parallelismo esplicito e permette di scrivere il codice in maniera tale che, quando si deve gestire una grossa quantità di dati, l'algoritmo riesca ad operare su più porzioni di dati in parallelo. Per implementare tale tipo di tecnica, è necessaria la presenza di registri che possano ospitare vettori di dati. Grazie ai repertori SSE e AVX, con una singola istruzione è possibile lavorare su dati multipli: in particolare, ogni registro XMM può contenere 4 float, mentre ogni registro YMM 4 double. Su di essi sarà possibile effettuare operazioni in parallelo.

Tutte le funzioni che sfruttano la tecnica di parallelismo implicito sono state strutturate operando prima sulla porzione di dati di dimensione multipla di 4 (per la versione a 32bit) o di 8 (per la versione a 64 bit), utilizzando le istruzioni con suffisso *packed*, e poi iterando in maniera scalare, ovvero un *float* per volta oppure un *double* per volta, tramite le operazioni con suffisso *scalar*.

Sfruttando la vettorizzazione, se ogni ciclo doveva essere eseguito n volte, in questo modo il costo verrà ridotto di n/4 operazioni.

3.3 Scelta delle parti da ottimizzare in assembly e confronto tra le varie scelte (pro e contro).

La parte sicuramente più dispendiosa dell'algoritmo è quella che popola la matrice *x_ast*. Questo compito è svolto dal metodo *popola*, citato nel paragrafo precedente.

```
void popola(int tuple, int n, int coldest, int h, int d, int* j, int col, params* input){
    type num;
    int i, oss, k;
    for(i=0;i<tuple;i++) {

        for(oss=0;oss<n;oss++) {
            num=1;
            for(k=0;k<h;k++) {
                num=num* input->x[oss+(j[i*h+k]-1)*n];
            }
            input->xast[oss+coldest*n]=num;
        }

        coldest++; //andiamo a scrivere la colonna coldest successiva di xast
    }
}
```

Il ciclo più esterno itera sulle righe della matrice J. Ogni riga è usata per il calcolo della corrispondente colonna in *x_ast*.

Per ogni riga della matrice *x_ast* si inizializza *num=1* per calcolare il prodotto del generico termine di grado *h*, il cui calcolo viene effettuato nel ciclo più interno. Si noti che, di volta in volta, è la matrice J a dettare quale debba essere l'indice di colonna della variabile *x* da moltiplicare.

Per quanto riguarda l'algoritmo SGD, si è scelto di implementare in assembly tutti i metodi interni alla procedura *sgd*, in quanto utilizzano grandi blocchi di dati e le tecniche di ottimizzazione viste durante il corso hanno pertanto permesso di ridurre notevolmente le tempistiche.

3.4 Implementazione dei metodi in assembly

Per quanto riguarda l'implementazione del metodo *popola* precedentemente descritto, in prima analisi è stata pensata una scrittura della matrice *x_ast* per righe, con un algoritmo che segue tale procedimento:

Per ogni osservazione (e dunque vettore riga della matrice x) viene completata la conversione, per intero, della relativa riga nella matrice *x_ast*.

Il calcolo della conversione di una generica osservazione richiede la conoscenza preliminare dell'intera matrice J, che per sua natura, è una matrice altamente sparsa. La sparsità della matrice ci ha posti davanti ad un problema: appurato che ogni riga di tale matrice ha dimensione *h*, quanti di questi *h* elementi sono elementi validi? Ricordiamo che ogni riga di J rappresenta logicamente un generico termine del polinomio ed esso può avere lunghezza variabile.

Una prima soluzione è stata quella di frammentare logicamente la matrice J in h sottomatrici ciascuna con un numero h_i di colonne “valide”, dove h_i per ogni $1 < i < h-1$ rappresenta il numero di combinazioni con ripetizione di d elementi su i posizioni.

Anche se questa soluzione sembra funzionare, è difficile da tradurre in assembly perché richiederebbe ricordare il numero di sottomatrici, il numero di partenza, il numero di righe e di colonne per ogni sottomatrice. Non è possibile nemmeno applicare Loop Vectorization, poiché, anche volendo ottimizzare considerando la conversione in parallelo di un gruppo di osservazioni, le righe della matrice J andrebbero comunque prese una alla volta (due righe successive potrebbero non avere un numero uguale di colonne valide), e per ognuna di esse i singoli elementi andrebbero considerati sempre una alla volta poiché usati come indici di colonna sulla matrice x.

Il vero problema di questa soluzione consiste soprattutto nella memorizzazione della matrice J. Essendo una matrice dipendente in modo esponenziale dal parametro degree, la sua dimensione cresce in modo spropositato all'aumentare di degree e ciò ha come conseguenza la crescita del numero dei parametri da memorizzare e il numero degli elementi non validi.

Per tutti questi motivi è stata effettuata la scelta di salvare sia la matrice dati e sia la matrice che rappresenta il dataset convertito come una matrice memorizzata per colonne (In realtà la matrice x_ast verrà riconvertita in una matrice salvata per righe prima dell'esecuzione dell'algoritmo SGD) per sfruttare al meglio le tecniche di ottimizzazione *Loop Vectorization* e poi *Loop unrolling* che permettono di usare la potenza del calcolatore sul quale verrà eseguito il codice. Tale soluzione segue uno schema diverso da quello precedente. Intrinsecamente, già in C, l'algoritmo popola per ogni i , un'intera colonna della matrice x_ast.

Tramite Loop Vectorization si è pensato di parallelizzare il numero di righe popolate ad ogni iterazione.

Viene di seguito riportato lo pseudo-codice in C:

```
for(i=0;i<tuple;i++) {
    for(oss=0;oss<n;oss=oss+P) {
        XMM0 = [1,1,1,1]
        for(k=0;k<h;k++) {
            int x_i = j[i*h+k]
            XMM0 = XMM0 * ->x[(oss ... oss+p-1) + x_i*n]
        }
        input->xast[(oss ... oss+p-1)+coldest*n]=XMM0
    }
    coldest++;
}
```

Nello pseudo-codice il “vettore moltiplicazione” è indicato con XMM0 per esplicitare il fatto che vengono eseguite 4 moltiplicazioni per volta in parallelo.

Di seguito vengono riportati i tempi ottenuti dopo questa prima ottimizzazione (con degree 4 e 10 rispettivamente).

```
pep@pep-X580VD ~/Scrivania/Progetto $ ./runregression32 ./Test/datasets/test_2000x4_4f -batch 2 -degree 4 -eta 0.001 -iter 1000
Input data name: './Test/datasets/test_2000x4_4f.data'
Input label name: './Test/datasets/test_2000x4_4f.labels'
Data set size [n]: 2000
Number of dimensions [d]: 4
Batch dimension: 2
Degree: 4
Eta: 0.001000
Adagrad disabled
Conversion time = 0.0003 secs
```

```
pep@pep-X580VD ~/Scrivania/Progetto $ ./runregression32 ./Test/datasets/test_2000x4_4f -batch 2 -degree 10 -eta 0.001 -iter 1000
Input data name: './Test/datasets/test_2000x4_4f.data'
Input label name: './Test/datasets/test_2000x4_4f.labels'
Data set size [n]: 2000
Number of dimensions [d]: 4
Batch dimension: 2
Degree: 10
Eta: 0.001000
Adagrad disabled
Conversion time = 0.0080 secs
```

Come si può notare, sono stati ottenuti miglioramenti di un fattore 8-10 %.

Dato che il calcolo di ogni riga di x_{ast} è indipendente dalle altre, si è pensato di sfruttare anche la tecnica di loop unrolling. Di seguito pseudo-codice in c:

```
;for(i=0;i<tuple;i++) {

    for(oss=0;oss<n;oss=oss+p*unroll) {
        XMM0 = [1,1,1,1]
        XMM2 = [1,1,1,1]
        XMM4 = [1,1,1,1]
        XMM6 = [1,1,1,1]
        for(k=0;k<h;k++) {
            int x_i = j[i*h+k]
            XMM0 = XMM0 * ->x[(oss ... oss+p-1) + x_i*n]
            XMM2 = XMM2 * ->x[(oss+p ... oss+2p-1) + x_i*n]
            XMM4 = XMM4 * ->x[(oss+2p ... oss+3p-1) + x_i*n]
            XMM6 = XMM6 * ->x[(oss+3p ... oss+4p-1) + x_i*n]
        }
        input->xast[(oss ... oss+p-1)+coldest*n]=XMM0
        input->xast[(oss ... oss+p-1)+coldest*n]=XMM2
        input->xast[(oss ... oss+p-1)+coldest*n]=XMM4
        input->xast[(oss ... oss+p-1)+coldest*n]=XMM6
    }
    coldest++; //andiamo a scrivere la colonna coldest successiva di xast
}
```

Con quest'ultima ottimizzazione è stato possibile ottenere un ulteriore miglioramento.

Nelle immagini sono descritti i tempi ottenuti con loop vectorization e unrolling, rispettivamente con degree 10 e 4.

```
pep@pep-X580VD ~/Scrivania/Progetto $ ./runregression32 ./Test/datasets/test_2000x4_4f -batch 2 -degree 10 -eta 0.001 -iter 1000
Input data name: './Test/datasets/test_2000x4_4f.data'
Input label name: './Test/datasets/test_2000x4_4f.labels'
Data set size [n]: 2000
Number of dimensions [d]: 4
Batch dimension: 2
Degree: 10
Eta: 0.001000
Adagrad disabled
Conversion time = 0.0045 secs

done!
pep@pep-X580VD ~/Scrivania/Progetto $ ./runregression32 ./Test/datasets/test_2000x4_4f -batch 2 -degree 4 -eta 0.001 -iter 1000
Input data name: './Test/datasets/test_2000x4_4f.data'
Input label name: './Test/datasets/test_2000x4_4f.labels'
Data set size [n]: 2000
Number of dimensions [d]: 4
Batch dimension: 2
Degree: 4
Eta: 0.001000
Adagrad disabled
Conversion time = 0.0002 secs
```

Di seguito un grafico riassuntivo che evidenzia l'andamento al variare del grado dei tempi del codice puramente in C (senza ottimizzazioni) e l'implementazione assembly ottimizzata.

Degree	Versione C	Vers. Assembly Ottimizzata
4	0,00337	0,00030
5	0,00444	0,00055
6	0,00958	0,00149
7	0,01794	0,00155
8	0,02523	0,00315
9	0,04412	0,00533
10	0,06017	0,00630
11	0,09609	0,00854
12	0,13051	0,01403
13	0,17251	0,01777
14	0,2394	0,03155
15	0,32541	0,03464
16	0,44228	0,03730
17	0,56952	0,04596
18	0,71341	0,05957
19	0,92632	0,07006
20	1,188	0,09033
22	1,836	0,12522
24	2,806	0,18964
26	4,009	0,24856
28	5,711	0,33234
30	7,976	0,44563
35	16,835	0,95547



- Implementazione SGD con Loop Vectorization

Utilizzando l'operazione HADDPS (nel caso a 32 bit), il repertorio SSE permette di effettuare la somma di due registri con un numero di passi pari a $\log_2(n)$, se n è il numero di operandi. L'istruzione è stata utilizzata nella procedura *prodottoScalare*.

Utilizzando il repertorio istruzioni **x86-64+AVX** e i registri **YMM**-, è stato possibile parallelizzare operazioni aritmetiche e di accesso in memoria, riducendo i tempi di calcolo mediamente dell'80%.

```
64 SGD
64;30;4;0.001;3000;SGD;3125;0.002;3.120;0;2;1.069638
64 ADAGRAD
64;30;4;3;3000;ADA;5497;0.002;5.491;0;2;0.790583
```



```
64 SGD
64;30;4;0.001;3000;SGD;501;0.002;0.497;0;2;1.069638
64 ADAGRAD
64;30;4;3;3000;ADA;1197;0.002;1.193;0;2;0.790583
```

Versione C

Versione assembly-LV

Nella versione a 32 bit, invece, i miglioramenti sono stati i seguenti:

```
32 SGD
32;30;4;0.001;3000;SGD;2996;0.003;2.988;0;2;1.069638
32 ADAGRAD
32;30;4;3;3000;ADA;12351;0.002;12.346;0;2;0.790585
```



```
32 SGD
32;30;4;0.001;3000;SGD;707;0.003;0.702;0;2;1.005867
32 ADAGRAD
32;30;4;3;3000;ADA;715;0.002;0.710;0;2;0.790585
```

Versione C

Versione assembly - LV

È opportuno segnalare l'utilizzo dell'istruzione **AVX VFMADD231PD** che, per ogni elemento, esegue il calcolo **MUL** e **ADD** con 3 operandi e imposta il risultato sul primo operando

Per la funzione *prodottoScalare*, risulta al quanto utile adoperare tale istruzione.

```

global prodottoScalare
prodottoScalare:
    push    rbp
    mov     rbp, rsp
    pushaq
    ; salva il Base Pointer
    ; il Base Pointer punta al Record di Attivazione corrente
    ; salva i registri generali

    ;[RDI] cost
    ;[RSI] theta
    ;[RDX] xast
    ;RCX jsize
    ;R8 size

    MOV    RAX, 0
    MOV    R9, R8
    SHR   R8, 2
    SHL   RCX, 3
    VXORPD YMM0, YMM0
    for4: CMP   RAX, R8
    JNL   fine4
    MOV    R10, RAX
    SHL   R10, 5
    VMOVUPD YMM1, [RSI+R10]
    ADD   R10, RCX
    VFMADD231PD YMM0, YMM1, [RDX+R10]
    ADD   RAX, 1
    JMP   for4
    fine4: SHL  R8, 2
    VXORPD XMM3, XMM3
    fors4: CMP  R8, R9
    JNL   return4
    MOV    R10, R8
    SHL   R10, 3
    VMOVSQ XMM1, [RSI+R10]
    ADD   R10, RCX
    VFMADD231SD XMM3, XMM1, [RDX+R10]
    ADD   R8, 1
    JMP   fors4
    return4: VHADDPD YMM0, YMM0
    VPERMPD YMM0, YMM0, 11011000
    VHADDPD YMM0, YMM0
    VADDSQ XMM3, XMM0
    VMOVSQ [RDI], XMM3
    popaq
    mov    rsp, rbp
    pop    rbp
    ret
    ; ripristina i registri generali
    ; ripristina lo Stack Pointer
    ; ripristina il Base Pointer
    ; torna alla funzione C chiamante

```

Esempio di utilizzo dell’istruzione **VFMADD231PD** nel calcolo del prodotto scalare

- Implementazione SGD con Loop Vectorization e Loop Unrolling

Nel progetto il fattore di unrolling scelto, sia nella versione a 64 bit che a 32 bit, è pari a 4. Ciò ha permesso di ridurre l’esecuzione dei cicli dello stesso fattore. E’ stato constatato sperimentalmente che, nel caso specifico, aumentando il fattore, e portandolo ad esempio ad 8, le tempistiche anziché migliorare tendevano al peggioramento.

Riguardo alla versione a 64 bit, rispetto alla versione precedentemente discussa, si ottiene mediamente un miglioramento di circa 0.07 secondi.

```

64 SGD
64;30;4;0.001;3000;SGD;501;0.002;0.497;0;2;1.069638
64 ADAGRAD
64;30;4;3;3000;ADA;1197;0.002;1.193;0;2;0.790583

```

Versione assembly - LV

```

64 SGD
64;30;4;0.001;3000;SGD;474;0.002;0.471;0;2;1.069638
64 ADAGRAD
64;30;4;3;3000;ADA;1085;0.002;1.080;0;2;0.790583

```

Versione assembly - LV ed LU

Nella versione a 32 bit, invece, il miglioramento è stato di circa 0,05 secondi per SGD e di 0,1 secondo per la versione AdaGrad.

```

32 SGD
32;30;4;0.001;3000;SGD;707;0.003;0.702;0;2;1.005867
32 ADAGRAD
32;30;4;3;3000;ADA;715;0.002;0.710;0;2;0.790583

```

Versione assembly - LV

```

32 SGD
32;30;4;0.001;3000;SGD;653;0.002;0.649;0;2;0.965715
32 ADAGRAD
32;30;4;3;3000;ADA;606;0.002;0.601;0;2;0.787015

```

Versione assembly - LV ed LU

Si sottolinea che la tecnica del loop unrolling risulta essere efficiente quando le operazioni da parallelizzare lavorano su dati già “pronti”, ovvero già reperiti dalla memoria, e quando tra tali operazioni non sono presenti istruzioni condizionali. Per questo motivo, si è deciso di non utilizzare tale tecnica nei metodi *faiSommatoria* e *azzeramentoDaSottrarre*. Nella versione a 32 bit, invece, nonostante sia stato implementato

nella versione a 64 bit, si è notato che i tempi di esecuzione peggioravano parallelizzando il metodo *prodottoScalare*. Pertanto, si è deciso di implementare su quest'ultimo soltanto la tecnica di *loop vectorization*.

3.5 Implementazione in OPEN-MP e motivazioni

Nell'algoritmo SGD, come si può vedere dai file sorgenti, non è stata utilizzata la parallelizzazione con OpenMP. Ciò è dovuto al fatto che l'algoritmo SGD in entrambe le varianti è intrinsecamente iterativo: ogni iterazione del while esterno dipende dai valori calcolati nella precedente, quindi lanciare dei thread che lavorano contemporaneamente su di esse porterebbe a risultati scorretti.

Le funzioni richiamate internamente dalla procedura SGD potevano invece essere parallelizzate ma, poiché si tratta dei cicli più interni, ciò avrebbe comportato l'allocazione e deallocazione di troppi thread hardware, peggiorando di gran lunga le tempistiche dell'algoritmo.

Ben più semplice è stata la scelta dei punti da ottimizzare nell'algoritmo **convert_data**.

La scelta è ricaduta su due cicli che per loro natura in ogni iterazione lavorano su dati indipendenti :

- Il ciclo che calcola il numero di combinazioni con ripetizione (eseguito h volte);
- Il ciclo che popola le colonne della matrice x_ast

Nel secondo caso, rispetto alla versione non OpenMP, la matrice J viene inizializzata ogni volta.

Nell'algoritmo SGD, come si può vedere dai file sorgenti, non è stata utilizzata la parallelizzazione con OpenMP. Ciò è dovuto al fatto che l'algoritmo SGD in entrambe le varianti è intrinsecamente iterativo: ogni iterazione dipende dai valori calcolati nella precedente, quindi lanciare dei thread che lavorano contemporaneamente su di esse porterebbe a risultati scorretti.

Le funzioni richiamate internamente dalla procedura SGD potevano invece essere parallelizzate ma, poiché si tratta dei cicli più interni, ciò avrebbe comportato l'allocazione e deallocazione di troppi thread hardware, peggiorando di gran lunga le tempistiche dell'algoritmo.