Riduzione della dimensione di un Dataset

Adriano Tumino, Pietro Morichetti

Giorno/11/2018

Indice

1 Introduzione **5**
1.1 Processo di Data Mining 5

2 Libreria **7**
2.1 Vettori e Matrici 7
2.2 Permutazione ed Inversione 8
2.3 Algebra Lineare 9
2.4 Autovalori ed Autovettori 9
2.5 Probabilitá e Statistica 10

3 Multi-Thread **11**

4 Programma **13**
4.1 Strutture 13
4.2 Lettura 15
4.3 Colonne 17
4.4 Ottieni 18
4.5 Trasponi 19
4.6 Media 19
4.7 Covarianza 20
4.8 Eigen 22
4.9 Ordinamento 22
4.10 Sottrazione_matriciale 23
4.11 Prodotto_matriciale 24
4.12 Thread_function 29
4.13 Inversione_operazione 30
4.14 Inversione_matriciale 31
4.15 Addizione_matriciale 33
4.16 Stampa 34
4.17 Main 35

5 Conclusioni **39**

Capitolo 1

Introduzione

Il problema applicativo è la descrizione di un evento fisico con il minor numero di dati possibili. I motivi possono essere svariati ma tra i piú popolari sicuramente troviamo:

- Data Compression: Comprimere l'insieme dei dati;
- Data Mining: Ricerca di qualche struttura nei dati;

Avendo numerose quantitá di dati allora è possibile ricavare una struttura dei dati solo quando questi sono limitati.

1.1 Processo di Data Mining

Consideriamo dei dati discretizzati e presentati sotto forma matriciale, in cui gli assi sono definiti dalle colonne e le righe dalle misure delle variabili. L'obbiettivo è quindi quello di rappresentare tali misure con un numero di variabili inferiori, atti a considerare solo le variazioni sui dati piú significativi; per poter far ció bisogna ridefinire il sistema di riferimento, secondo i cosiddetti Principal Components, un nuovo set di assi mutualmente ortogonali e che considerano la massima variazione di dati, nel piú piccolo errore di riduzione della dimensionalitá possibile.

Capitolo 2

Libreria

Per utilizzare numerosi comandi è stata utilizzata la libreria gratuita *GSL - GNU Scientific Library*. Questa è una libreria per C e C++ liberamente consultabile su: *Gun.org*.
La libreria offre una vasta gamma di routine matematiche. Ci sono oltre 1000 funzioni in totale con un vario assortimento di test, in particolare presenteremo quelle che sono state utilizzate per la realizzazione del progetto.

```
#include <sys/types.h> // per utilizzare variabili di tipo pthread_t
#include <stdlib.h> // utilità generale (allocazione, controllo processi,
#include <pthread.h> // per utilizzo dei thread
#include <string.h> // per elaborazione su stringhe
#include <math.h>
#include <gsl/gsl_vector.h> // per usare i vettori
#include <gsl/gsl_matrix.h> // per usare le matrici
#include <gsl/gsl_eigen.h> // per calcolare autovalori e autovettori
#include <gsl/gsl_statistics.h> // per usare funzioni di probabilità
```

2.1 Vettori e Matrici

Premettiamo che oggetti come vettori e matrici sono pensati sotto forma di strutture in cui si definiscono: dimensioni, passo e dati contenuti; inoltre le seguenti funzioni non possono essere richiamate senza l'apposita libreria, <gsl_vector.h> e <gsl_matrix.h> rispettivamente.
Vettori:

- *gsl_vector * gsl_vector_alloc(size_t n):* La funzione crea un array di dimensione n e ritorna un puntatore al vettore.

- *gsl_ vector *gsl_ vector_ calloc(size_ t n):* La funzione crea un array di dimensione n e la inizializza con valori pari a 0 in ogni cella, ritorna poi un puntatore al vettore.

- *void gsl_ vector_ free(gsl_ vector * v):* La funzione de-alloca lo spazio dedicato al vettore v.

- *double gsl_ vector_ get(const gsl_ vector * v, const size_ t i):* La funzione preleva il valore memorizzato nella cella i del vettore v e lo restituisce al chiamante.

- *void gsl_ vector_ set(gsl_ vector * v, const size_ t i, double x):* La funzione inserisce il parametro x nella cella i del vettore v, non restituisce nulla.

Matrici:

- *gsl_ matrix * gsl_ matrix_ alloc(size_ t n1, size_ t n2):* La funzione alloca una matrice di dimensioni n1 x n2 e ritorna un puntatore al vettore bidimensionale.

- *void gsl_ matrix_ free(gsl_ matrix * m):* La funzione libera lo spazio in memoria dedicato alla matrice m, non ritorna nulla alla funzione chiamante.

- *double gsl_ matrix_ get(const gsl_ matrix * m, const size_ t i, const size_ t j):* La funzione restituisce il valore memorizzato nella cella (i, j) della matrice m.

- *void gsl_ matrix_ set(gsl_ matrix * m, const size_ t i, const size_ t j, double x):* La funzione inserisce il parametro x nella cella (i, j) della matrice m, non restituisce nulla.

- *int gsl_ matrix_ swap_ columns(gsl_ matrix * m, size_ t i, size_ t j):* La funzione scambia fra di loro le colonne i e j della matrice m, restituisce un parametro di controllo.

2.2 Permutazione ed Inversione

Una permutazione p è rappresentata come un array di n interi, dove ogni valore della permutazione appare una ed una sola volta, e la chiamata a tali funzioni è possibile richiamando la libreria <gsl_permutation.h>.

- *gsl_permutation * gsl_permutation_calloc(size_t n):* La funzione alloca uno spazio in memoria pari a n celle, la funzione ritorna un puntatore alla memoria.

- *void gsl_permutation_free(gsl_permutation * p):* La funzione libera lo spazio prealloccato per l'array p, permutazione; la funzione non ritorna nulla.

2.3 Algebra Lineare

Inserendo fra gli header la chiamata alla libreria <gsl_linalg.h> è possibile usare tutti quegli strumenti atti a risolvere problemi di algebra lineare; solitamente il primo passo è quello di eseguire una decomposizione della matrice per poterne determinare i parametri più significativi.

- *int gsl_linalg_LU_decomp(gsl_matrix * A, gsl_permutation * p, int * signum):* La funzione esegue una decomposizione LU sulla matrice A, secondo una specifica permutazione ed un segno; la funzione ritorna un valore di controllo.

- *int gsl_linalg_LU_invert(const gsl_matrix * LU, const gsl_permutation * p, gsl_matrix * inverse):* La funzione esegue l'operazione d'inversione matriciale della LU, secondo i vincoli della permutazione; la funzione salva la matrice inversa nel vettore bidimensionale "inverse" e ritorna un valore di controllo.

2.4 Autovalori ed Autovettori

Il presente paragrafo descrive le funzioni utilizzate per il calcolo di autovalori ed autovettori di una matrice, ma solo inserendo fra gli header la chiamata alla libreria <gsl_eigen.h>.

- *gsl_eigen_symmv_workspace * gsl_eigen_symmv_alloc(const size_t n):* La funzione allocca uno spazio di dimensione n come workspace, ovvero un foglio di lavoro in cui altre funzioni possono svolgere calcoli per la determinazione degli autovalori; la funzione ritorna un puntatore al workspace.

- *void gsl_eigen_symmv_free(gsl_eigen_symmv_workspace * w):* La funzione libera lo spazio dedicato al workspace w, e non ritorna nulla.

- *int gsl_ eigen_ symmv(gsl_ matrix * A, gsl_ vector * eval, gsl_ matrix * evec, gsl_ eigen_ symmv_ workspace * w):* Questa è la funzione che si occupa di determinare gli autovalori ed autovettori della matrice A, salvati nei vettori eval ed evec rispettivamente; si osservi come tale funzione faccia uso di un workspace che deve essere stato precedentemente alloccato, inoltre è bene osservare che è presente una referenza fra l'indice di cella dell'array eval e l'indice di colonna della matrice evec (una corrispondenza fra autovalore ed autovettore). La funzione ritorna un parametro di controllo.

2.5 Probabilitá e Statistica

La libreria *GSL GNU Scientific Library* mette a disposizione molti strumenti per il calcolo probabilistico, ma per la realizzazione del progetto si è fatto uso di una sola funzione per la determinazione del parametro di covarianza fra due vettori; in ogni caso c'è la necessitá di richiamare la libreria <gsl_statistics.h>.

- *double gsl_ stats_ covariance(const double data1[], const size_ t stride1, const double data2[], constsize_ t stride2, const size_ t n):* La funzione determina il valore della covarianza fra il vettore data1 ed il vettore data2, con passo stride1 e stride2 per la modalitá di estrazione dei valori per cella, fra i due vettori; si osservi come è necessario che i due vettori presentino la stessa dimensionalità n. La funzione restituisce la covarianza.

Capitolo 3

Multi-Thread

Il multithreading indica il supporto hardware da parte di un processore di eseguire piú thread. Questo meccanismo migliora le prestazioni dei programmi solamente quando questi sono stati sviluppati suddividendo il carico di lavoro su piú thread che possono essere eseguiti in apparenza in parallelo. Mentre i sistemi multiprocessore sono dotati di piú unitá di calcolo indipendenti per le quali l'esecuzione è effettivamente parallela, un sistema multithread invece è dotato di una singola unitá di calcolo che si cerca di utilizzare al meglio eseguendo piú thread nella stessa unitá di calcolo.

Capitolo 4

Programma

Per il corretto funzionamento di questo programma abbiamo deciso di utilizzare numerose funzioni, adibite a scopi diversi, richiamabili tra di loro; nella fatti specie, tali funzioni verranno presentati secondo un ordine sequenziale alla procedura dell'algoritmo implementato.
Il codice è costellato di stampe e controlli per monitorare il corretto funzionamento del programma, esse non saranno citate durante le spiegazioni del contenuto delle singole funzioni.

4.1 Strutture

Vengono utilizzate tre strutture per aver il corretto funzionamento del programma, e questi sono:

```
struct gsl{
        gsl_vector *vector;
        gsl_matrix *matrix;
} puntatori_gsl;

struct registro{
        double *mtx_1, *mtx_2;
        int *indici;
} catalogo;

struct gsl eigen(double **covarianza){
        int i=0, j=0;
        gsl_matrix *matrice, *evec;
        gsl_vector *evel;
        gsl_eigen_symmv_workspace *campo;
```

```
struct gsl_pointer_gsl;
matrice=gsl_matrix_alloc(dimensione, dimensione);
if(matrice==NULL){
        printf("ERRORE - eigen: matrice non alloccato");
        exit(0);
}
evec=gsl_matrix_alloc(dimensione, dimensione);
if(evec==NULL){
        printf("ERRORE - eigen: evec non alloccato");
        exit(0);
}
evel=gsl_vector_calloc(dimensione);
if(evel==NULL){
        printf("ERRORE - eigen: evel non alloccato");
        exit(0);
}
campo=gsl_eigen_symmv_alloc(dimensione);
if(campo==NULL){
        printf("ERRORE - eigen: campo non alloccato");
        exit(0);
}
for(i=0; i<dimensione; i++){
        for(j=0; j<dimensione; j++){
                if(debug==1){
                        printf("covarianza[%d][%d] = %2.2f\
                                i, j, covarianza[i][j]);
                }
                gsl_matrix_set(matrice, i, j,
                        covarianza[i][j]);
                if(debug==1){
                        printf("matrice[%d][%d] = %2.2f\n",
                         i, j,
                         gsl_matrix_get(matrice, i, j));
                }
        }
}
controllo=gsl_eigen_symmv(matrice, evel, evec, campo);
if(controllo!=0){
        printf("ERRORE - eigen:
                gsl_eigen_symmv fallito.\n");
        exit(0);
```

```
        }
        pointer_gsl.vector=evel;
        pointer_gsl.matrix=evec;
        gsl_matrix_free(matrice);
        gsl_eigen_symmv_free(campo);
        return pointer_gsl;
}
```

4.2 Lettura

La funzione Lettura è una funzione di tipo int la quale non ha nessuna variabile in ingresso.
Lo scopo di questa funzione è la lettura del database da analizzare.
Viene richiesto di inserire da tastiera il pathname del file il quale deve essere di tipo ".txt".
Preso il file, allora questo viene letto riga per riga e poi ne estrae tutti i valori salvandoli nelle celle corrispondenti. Per far ció viene utilizzato il comando fscanf il quale termina non appena incontra il carattere separatore. Questo viene preceduto dal fseek con il quale settiamo il puntatore al valore successivo da leggere. Servono altre funzioni per far ció che questo avvenga. Queste funzioni sono:

- Colonne
- Ottieni
- Trasponi

Infine questa funzione ritorna il numero di righe.

```
int lettura(){
        FILE *fp;
        char pathname[1000];
        char buf[2000];
        char *s;
        int i=0, nrighe=0, ncolonne=1;
        int j=0;
        printf("Inserisci l'indirizzo del file: \n");
        scanf("%s", pathname);
        if( (fp = fopen(pathname, "r")) == NULL ){
                printf("ERRORE - lettura: apertura file %s
```

```
                fallito.\n", pathname);
                exit(0);
        }
        while(1){
                stpcpy(buf, "");
                s = fgets(buf, sizeof(buf), fp);
                if(s == NULL) break;
                nrighe++;
                if(j == 0){
                        ncolonne=colonne(buf);
                        j++;
                }
        }
        data_set_origine=malloc(nrighe*sizeof(double*));
        data_set=malloc(ncolonne*sizeof(double*));
        if(data_set == NULL){
                printf("ERRORE - lettura: data_set
                 non alloccato.\n");
                exit(0);
        }
        if(data_set_origine == NULL){
                printf("ERRORE - lettura: data_set
                 non alloccato.\n");
                exit(0);
        }
        for(i = 0; i < nrighe; i++){
                data_set_origine[i]=malloc(ncolonne*sizeof(double))
                data_set[i]=malloc(nrighe*sizeof(double));
                if(data_set[i] == NULL){
                        printf("ERRORE - lettura: data_set[%d]
                         non alloccato.\n", i);
                        exit(0);
                }
                if(data_set_origine[i] == NULL){
                        printf("ERRORE - lettura: data_set_origine[%
                         non alloccato.\n", i);
                        exit(0);
                }
        }
        rewind(fp);
        for(i = 0; i < nrighe; i++){
```

```
                stpcpy(buf, "");
                s = fgets(buf, sizeof(buf), fp);
                ottieni(buf, i);
        }
        fclose(fp);
        trasponi(nrighe, ncolonne);
        dimensione = nrighe;
        return ncolonne;
}
```

4.3 Colonne

Questa è una funziona adibita per contare il numero di colonne del dataset. Prende in ingresso un array di tipo char dalla quale conta il numero di colonne.
Questo avviene esaminando carattere per carattere finché non viene trovato il carattere separatore. Esaminata tutta la stringa allora viene ritornato il numero di caratteri.

```
int colonne(char stringa[]){
        int set=0, last_set=0;
        int i, j=0;
        for(i = 0; i < strlen(stringa); i++){
                if( (((int)stringa[i]) == 43)||(((int)stringa[i]) == 45)
                ||(((int)stringa[i]) == 46)||((((int)stringa[i]) > 47)
                &&(((int)stringa[i]) < 58))){
                        set=1;
                }else{
                        set=0;
                }
                if((last_set == 1 && set == 0)||
                (set == 1 && i == (strlen(stringa) -1))){
                        j++;
                }
                last_set=set;
        }
        return j;
}
```

4.4 Ottieni

Questa funzione ha lo scopo di trovare tutti i valori di una riga presa in ingresso e salvarli nella corrispettiva cella della matrice.
Utilizza un procedimento simile a quello della funziona colonne, infatti controlla carattere per carattere la stringa, salva tutti i valori necessari e quando incontra il carattere separatore allora tutti i caratteri incontrati precedentemente li converte in tipo float per poi salvarli nel dataset.
Questo procedimento viene fatto per ogni riga del dataset.

```
void ottieni(char stringa[], int riga){
        int set=0, last_set=0;
        int i, j=0, k;
        char buf[100];
        double a;
        k=0;
        for(i = 0; i < strlen(stringa); i++){
                if(((int)stringa[i] == 43)||((int)stringa[i] == 45)
                ||(((int)stringa[i] == 46)||(((int)stringa[i] > 47)
                &&((int)stringa[i] < 58))){
                        set=1;
                        buf[j]=stringa[i];
                        j++;
                }else{
                        set=0;
                }
                if((set==0 && last_set==1)||
                (set==1 && i == (strlen(stringa)-1))){
                        a=atof(buf);
                        data_set_origine[riga][k]=a;
                        k++;
                        strcpy(buf, "");
                        j=0;
                }
                last_set=set;
        }
}
```

4.5 Trasponi

Trasponi è una funzione che preso ha il solo compito di trasporre il dataset. Supponiamo una matrice MxN allora questa funzione la fa diventare una matrice NxM, senza peró andare ad intaccare la matrice precedente.

```
void trasponi(int nr, int nc){
        int i, j;
        j = 0;
        for(j = 0; j < nc; j++){
                for(i = 0; i < nr; i++){
                        data_set[j][i] = data_set_origine[i][j];
                }
        }
}
```

4.6 Media

La funzione necessita del solo parametr in ingresso "nrighe", parametro che si riferisce al numero di righe del Data Set ed ha il solo scopo di eseguire una media sulle righe della matrice "data_set", memorizzate all'interno dell'array "media", in corrispondenza dell'indice di riga considerata; la funzione ritorna il puntatore a "media".

```
double *media(int nrighe){
        int i, j = 0;
        double somma = 0;
        double *media;
        media=malloc(nrighe*sizeof(double));
        if(media==NULL){
                printf("ERRORE: media non alloccata.\n");
        }
        for(i=0; i<nrighe; i++){
                for(j=0; j<dimensione; j++){
                        somma = somma+data_set[i][j];
                        if(debug==1){
                                printf("somma = %2.2f\n", somma);
                        }
                }
                media[i]=somma/dimensione;
```

```
                somma=0;
                if(debug){
                        printf("Media: %2.2f \n", media[i]);
                }
        }
        return media;
}
```

4.7 Covarianza

Il suo compito è quello di determinare la matrice di covarianza della matrice "data_set", ma per far ció ha bisogno di riceve in ingresso l'array "media" per poter applicare la formula della covarianza fra due variabili equidimensionali

$$Cov(A,B) = \frac{1}{n-1} \cdot \sum (a_i - m_A) \cdot (b_i - m_B)$$

Il calcolo va quindi iterato per ogni coppia di colonne del "data_set". Infatti la funzione consta di quattro cicli for annidati, vediamone il funzionamento, a partire da quello più esterno:

- 1° for: il ciclo piú lento e va a considerare la prima colonna del "data_set" (riga 0);
- 2° for: copia la colonna iesima nell'array "tmp1", un array ausiliario, e rimarrá inalterato per tutta la durata dei cicli successivi;
- 3° for: dato che la prima variabile è stata fissata, adesso andremo a considerare, una per volta, tutte le variabili in gioco e poi a calcolare la covarianza per mezzo della funzione *gsl_stats_covariance*;
- 4° for: copia la colonna jesima nell'array "tmp2", un array ausiliario e seconda variabile della formula di covarianza;

L'iterazione del blocco di cicli for andrá a costruire la matrice di covarianza per colonna. La funzione conclude restituendo il puntatore alla matrice "covarianza".

```
double **covarianza(double *media, int nrighe){
        int i, j, k = 0;
        double **covarianza;
        double  *tmp1, *tmp2;
        tmp1=malloc(nrighe*sizeof(double));
        if(tmp1==NULL){
```

```
                printf("ERRORE - covarianza: tmp1 non alloccato.\n");
                exit(0);
        }
        tmp2=malloc(nrighe*sizeof(double));
        if(tmp2==NULL){
                printf("ERRORE - covarianza: tmp2 non alloccato.\n");
                exit(0);
        }
        covarianza=malloc(dimensione*sizeof(double*));
        if(covarianza==NULL){
                printf("ERRORE - covarianza: covarianza non alloccato.\n"
                exit(0);
        }
        for(i=0; i<dimensione; i++){
                covarianza[i]=malloc(dimensione*sizeof(double));
                if(covarianza[i]==NULL){
                        printf("ERRORE - covarianza: covarianza[%d]
                         non alloccato.\n", i);
                        exit(0);
                }
        }
        for(i=0; i<dimensione; i++){
                for(k=0; k<nrighe; k++){
                        tmp1[k]=data_set[k][i];
                        if(debug==1){
                                stampa(6, NULL, tmp1, "tmp1", 0, 0);
                        }
                }
                for(j=0; j<dimensione; j++){
                        for(k=0; k<nrighe; k++){
                                tmp2[k]=data_set[k][j];
                                        if(debug==1){
                                                stampa(6, NULL,
                                                  tmp2, "tmp2",
                                                  0, 0);
                                        }
                        }
                        covarianza[i][j]=gsl_stats_covariance(tmp1,
                                          1, tmp2, 1, nrighe);
                }
                if(debug==1){
```

```
                        stampa(7, covarianza, NULL,
                                "Covarianza", 0, 0);
                }
        }
        free(tmp1);
        free(tmp2);
        return covarianza;
}
```

4.8 Eigen

La funzione riceve in ingresso la matrice "covarianza" e ne determina i suoi autovalori ed autovettori, per mezzo della libreria *GSL GNU Scientific Library*; infatti, al di lá di un paio di variabili di servizio, si fa uso di variabili di tipo gsl e chiamate a funzioni di libreria. La prima parte della funzione è dedicata all'allocamento ed al settaggio, in particolare la matrice "covarianza" viene copiata nella matrice gsl "matrice", tramite la chiamata alla funzione *gsl_matrix_set*. La seconda parte è incentrata sul calcolo degli autovalori e autovettori, per mezzo della funzione *gsl_eigen_symmv*, e inseriti nell'array "evel" e nella matrice "evec" rispettivamente; i quali puntatori vengono salvati nella struttura "gsl". La funzione si conclude restituendo la variabile di tipo struttura gsl.

4.9 Ordinamento

Ordina ha il solo scopo di ordinare gli autovalori calcolati in ordine decrescente. Questo viene effettuato con il classico ordinamento effettuato con un doppio for e un controllo, il quale se ha successo scambia di posto sia gli autovalori e sia gli autovettori corrispondenti.

```
void ordinamento(gsl_matrix *vettori){
        int i,j = 0;
        double tmp = 0;
        for(i=0; i<dimensione; i++){
                if(autovalori[i]<0){
                        autovalori[i]=autovalori[i]*(-1);
                }
        }
        for(i = 0; i < dimensione; i++){
                for(j = 0; j < dimensione; j++){
```

```
                        if(debug==1){
                                printf("autovalori[%d] = %2.2f,
                                autovalori[%d] = %2.2f\n", j,
                                autovalori[j], i, autovalori[i]);
                        }
                        if(autovalori[j] < autovalori[i]){
                                tmp=autovalori[i];
                                if(debug==1){
                                        printf("tmp = %2.2f\n", tmp);
                                }
                        autovalori[i]=autovalori[j];
                        autovalori[j]=tmp;
                        if(debug==1){
                                printf("autovalori[%d] = %2.2f ,
                                 autovalori[%d]=%2.2f\n", j,
                                 autovalori[j], i, autovalori[i]);
                         }
                        gsl_matrix_swap_columns(vettori, i, j);
                        if(controllo!=0){
                        printf("ERRORE - ordinamento:
                         gsl_matrix_swap_columns fallito.\n");
                        exit(0);
                        }
                }
        }
        tmp=0;
}
```

4.10 Sottrazione_matriciale

La funzione riceve in ingresso l'intero "nrighe" e l'array "media" e si limita ad eseguire una operazione di sottrazione matriciale fra: la matrice "data_set" ed una matrice fittizia (su ogni riga iesima c'è il valore nella cella iesima dell'array "media"). La funzione restituisce la matrice "sottrazione", alloccata all'inizio del blocco.

```
double **sottrazione_matriciale(int nrighe, double *media){
        int i, j = 0;
        double **sottrazione;
        sottrazione=malloc(nrighe*sizeof(double*));
```

```
        if(sottrazione==NULL){
                printf("ERRORE - sottrazione_matriciale:
                 sottrazione non alloccato.\n");
                exit(0);
        }
        for(i=0; i<nrighe; i++){
                sottrazione[i]=malloc(dimensione*sizeof(double));
                if(sottrazione[i]==NULL){
                        printf("ERRORE - sottrazione_matriciale:
                    sottrazione[%d] non alloccato.\n", i);
                        exit(0);
                }
        }
        for(i=0; i<nrighe; i++){
                for(j=0; j<dimensione; j++){
                        sottrazione[i][j]=data_set[i][j]-media[i];
                        if(debug==1){
                                printf("data_set[%d][%d] = %2.2f,
                                 media[%d] = %2.2f,
                                 sottrazione[%d][%d] = %2.2f\n",
                                 i, j, data_set[i][j], i, media[i],
                                  i,j, sottrazione[i][j]);
                        }
                }
        }
        return sottrazione;
}
```

4.11 Prodotto_matriciale

Il cuore della tesina è costituito dall'implementare il prodotto matriciale tramite multithread, quindi è giunto il momento di andare ad esplorare la parte principale di questo codice.
La funzione è stata pensata e realizzata nel caso generale, ovvero quando si hanno in ingresso due matrici di dimensioni diverse (ma che rispettano i vincoli del prodotto matriciale); per questo motivo la funzione richiede in ingresso, non solo le matrici in questione ma anche le loro dimensioni.
Prima d'iniziare a trattare il contenuto del blocco è bene chiarire un paio di concetti:

- Definiamo come "prodotto", il prodotto dei valori (in stessa posizione) fra due array, e successiva somma dei singoli prodotti;

- Definiamo come "singola iterazione" o "singola procedura" il prodotto fra la riga i-esima della matrice A per la colonna j-esima della matrice B, al variare di i e j.

Con questa premessa possiamo iniziare l'analisi. La prima parte del blocco prevede il settaggio dei parametri da inviare ai thread, ovvero la riga di A, la colonna di B e gli indici di posizione in cui salvare il risultato della singola procedura del prodotto matriciale; il tutto conservato in una cella dell'array di struttura di tipo "registro". La seconda parte del blocco inizia con l'alloccamento dinamico di un array di thread ed un successivo ciclo for in cui si esegue la chiamata del singolo thread, dell'array di thread, che riceve come argomento la singola struttura di parametri, precedentemente preparati.
Osserviamo che la funzione chiama in ripetizione tutti i thread, ma attende solo l'ultimo thread prima di proseguire il suo codice. La funzione torna un parametro di controllo, mentre è la "thread_function" ha memorizzare il risultato.

```
int prodotto_matriciale(int righe_1, int colonne_1, int righe_2, int colon
 double **matrice_1, double **matrice_2){
        int i, i_1, j_1, contatore, fine = 0;
        int *indici;
        double *mtx_1, *mtx_2;
        pthread_t *thread;
        struct registro *enciclopedia;
        if(colonne_1 != righe_2){
                printf("ERRORE - prodotto_matriciale:
                        non è soddisfatta la condizione del
                        prodotto matriciale.\n");
                return -1;
        }
        enciclopedia=
                malloc(righe_1*colonne_2*sizeof(struct registro));
        if(enciclopedia==NULL){
                printf("ERRORE - prodotto_matriciale:
                        enciclopedia non alloccata.\n");
                         exit(0);
        }
        contatore=0;
```

```
i_1=0;
j_1=0;
while(fine==0){
        mtx_1=malloc(colonne_1*sizeof(double));
        if(mtx_1==NULL){
                printf("ERRORE - prodotto_matriciale:
                                mtx_1 non alloccato.\n");
                exit(0);
        }
        mtx_2=malloc(righe_2*sizeof(double*));
        if(mtx_2==NULL){
                printf("ERRORE - prodotto_matriciale:
                        mtx_2 non alloccato.\n");
                exit(0);
        }
        indici=malloc(2*sizeof(int));
        if(indici==NULL){
                printf("ERRORE - prodotto_matriciale:
                        indici non alloccato.\n");
                exit(0);
        }
        if(debug==1){
                printf("contatore = %d, colonne_2=%d,
                        righe_1=%d, i_1=%d, j_1=%d\n",
                        contatore, colonne_2, righe_1,
                        i_1, j_1);
        }
        for(i = 0; i < colonne_1; i++){
                mtx_1[i]=matrice_1[i_1][i];
                mtx_2[i]=matrice_2[i][j_1];
        }
        if(debug==1){
                stampa(6, NULL, mtx_1, "MTX_1", 0, 0);
                stampa(6, NULL, mtx_2, "MTX_2", 0, 0);
        }
        indici[0]=i_1;
        indici[1]=j_1;
        enciclopedia[contatore].mtx_1 = mtx_1;
        enciclopedia[contatore].mtx_2 = mtx_2;
        enciclopedia[contatore].indici = indici;
        contatore++;
```

```
                if(debug==1){
                        printf("contatore = %d\n", contatore);
                }
                if(debug==1){
                        printf("j_1+1 = %d, colonne_2 = %d, i_1+1 = %d,
                                righe_1 = %d\n", j_1+1, colonne_2,
                                i_1+1, righe_1);
                }
                if(j_1+1<colonne_2){
                        j_1++;
                        if(debug==1){
                                printf("j_1 = %d\n", j_1);
                        }
                }else if(i_1+1<righe_1){
                        i_1++;
                        j_1=0;
                        if(debug==1){
                                printf("i_1 = %d\n", i_1);
                        }
                }else{
                        fine=1;
                }
        }
        contatore=0;
        i=0;
        i_1=0;
        j_1=0;
        fine=0;
        thread=malloc(righe_1*colonne_2*sizeof(pthread_t));
        if(thread==NULL){
                printf("ERRORE - prodotto_matriciale:
                        thread non alloccato.\n");
                exit(0);
        }
        if(debug==1){
                int j = 0;
                for(i=0; i<righe_1*colonne_2; i++){
                        printf("Stampa del contenuto
                                delle strutture.\n");
                        printf("------------\n");
                        for(j=0; j<colonne_1; j++){
```

```
                                printf("enciclopedia[%d].mtx_1[%d]=
                                        %2.2f\n",i, j,
                                        enciclopedia[i].mtx_1[j]);
                        }
                        for(j=0; j<colonne_1; j++){
                                printf("enciclopedia[%d].mtx_2[%d]=
                                        %2.2f\n", i, j,
                                        enciclopedia[i].mtx_2[j]);
                        }
                        for(j=0; j<2; j++){
                                printf("enciclopedia[%d].indici[%d]
                                        =%d\n", i, j,
                                        enciclopedia[i].indici[j]);
                        }
                }
                printf("-------------\n");
        }
        for(i=0; i<righe_1*colonne_2; i++){
                controllo=pthread_create(&thread[i], NULL,
                (void*)thread_function, (void*)&enciclopedia[i]);
                if(controllo!=0){
                        printf("ERRORE - prodotto_matriciale:
                                pthread_create fallito.\n");
                        exit(0);
                }
        }
        controllo=pthread_join(thread[(righe_1*colonne_2)-1], NULL)
        if(controllo!=0){
                printf("ERRORE - prodotto_matriciale:
                        pthread_join fallito.\n");
                exit(0);
        }
        free(mtx_1);
        free(mtx_2);
        free(indici);
        free(enciclopedia);
        return 0;
}
```

4.12 Thread_function

Al thread arriva il cosiddetto "candidato" che non è altro che la struttura "catalogo" contenente: la i-esima riga, la j-esima colonna e gli indici di posizione dove andare poi a salvare il risultato della singola iterazione del prodotto matriciale, nella matrice "data_compression".
I parametri della struttura vengono castati per il loro tipo di origine (dato che la "thread_function" riceve solo di tipo puntatore a void), si procede poi con l'esecuzione del prodotto matriciale (a singoli elementi): si esegue il singolo prodotto, il risultato lo si salva sulla posizione corrente del'array tmp[0]; successivamente si somma il precedente risultato al valore presente nella cella 0 di tmp (tmp[0][0] svolgerá il ruolo di sommatore, senza dover creare una variabile temporanea).
Infine si inserisce il valore nell'apposita cella del "data_compression".

```
void *thread_function(void *candidato){
        int i = 0;
        int *indici;
        double *mtx_1, *mtx_2;
        struct registro *catalogo;
        catalogo=(struct registro*)candidato;
        mtx_1=catalogo->mtx_1;
        mtx_2=catalogo->mtx_2;
        indici=catalogo->indici;
        for(i=0; i<dimensione; i++){
                if(debug==1){
                        printf("mtx_1[%d] = %2.2f, mtx_2[%d] = %2.2f\n",
                        i, mtx_1[i], i, mtx_2[i]);
                }
                mtx_1[i]=mtx_1[i]*mtx_2[i];
                if(debug==1){
                        printf("singolo prodotto: %2.2f\n",
                                mtx_1[i]);
                }
                if(i>0){
                        mtx_1[0]=mtx_1[0]+mtx_1[i];
                        if(debug==1){
                                printf("somma: %2.2f\n",
                                        mtx_1[0]);
                        }
                }
```

```
        }
        i=0;
        data_compression
        data_compression[indici[0]][indici[1]]=mtx_1[0];
        if(debug==1){
                printf("data_compression[%d][%d]=%2.2f\n",
                        indici[0], indici[1],
                        data_compression[indici[0]][indici[1]]);
        }
        return NULL;
}
```

4.13 Inversione_operazione

Per verificare se è stato eseguito nella maniera corretta la riduzione del "data_set" è possibile usare la reversibilitá dell'operazione

$$X_p = (X - M) \cdot C_p$$

e confrontare la matrice ottenuta con la matrice "data_set", e questa funzione costituisce il primo passo.
Essa necessita in ingresso: il numero di righe del "data_set", l'array "media" e la matrice "evec", matrice che ricordiamo essere di tipo "gsl" e rappresenta gli autovettori (ordinati) della matrice di covarianza del "data_set"; con tali dati eseguiamo la formula inversa

$$X = X_p \cdot {C_p}^{-1} + M$$

richiamando in sequenza: la funzione "inversione_matriciale" e passandogli come argomento la matrice "evec", la funzione di "prodotto_matriciale" e passandogli come argomento il "data_compression" e la matrice "verifica" (risultato dell'operazione d'inversione matriciale) ed infine la funzione "addizione_matriciale".
La funzione restituisce la matrice "verifica", presumibilmente uguale al "data_set".

```
double **inversione_operazione(int nrighe, double *media, gsl_matri
        double **verifica;
        verifica=inversione_matriciale(evec);
        if(0 != prodotto_matriciale(nrighe, dimensione, dimensione,
                dimensione, data_compression, verifica)){
                printf("ERRORE - inversione_operazione: funzione
```

```
                        prodotto_matriciale ha riportato un errore,
                        esecuzione arrestata.\n");
                exit(0);
        }
        verifica=addizione_matriciale(nrighe, media);
        return verifica;
}
```

4.14 Inversione_matriciale

La funzione richiede in ingresso la matrice "evec" di tipo "gsl" (matrice degli autovettori) e per la maggior parte del blocco ci si dedica all'alloccamento dinamico di array e matrici: "permutazione" e "segno" (insieme a "evec") saranno utilizzati nella chiamata alla funzione di libreria *gsl_linalg_LU_decomp* per svolgere la decomposizione della matrice di autovettori, mentre "inverse" (insieme a "evec" e "permutazione") saranno utilizzati nella chiamata alla funzione di libreria *gsl_linalg_LU_invert* per realizzare la vera e propria inversione matriciale. In particolare, la matrice "inverse" sará poi ricopiata nella matrice "inversione", quella che costituirá il risultato della funzione.

```
double **inversione_matriciale(gsl_matrix *evec){
        int i, j = 0;
        int *segno;
        double **inversione;
        gsl_matrix *inverse;
        gsl_permutation *permutazione;

        inversione=malloc(dimensione*sizeof(double*));
        if(inversione==NULL){
                printf("ERRORE - inversione_matriciale:
                        inversione non alloccato.\n");
                exit(0);
        }
        for(i=0; i<dimensione; i++){
                inversione[i]=malloc(dimensione*sizeof(double));
                if(inversione[i]==NULL){
                        printf("ERRORE - inversione_matriciale:
                         inversione[%d] non alloccato.\n", i);
                         exit(0);
                }
```

```
        }
        inverse=gsl_matrix_alloc(dimensione, dimensione);
        if(inverse==NULL){
                printf("ERRORE - inversione_matriciale:
                        inverse non alloccato.\n");
                exit(0);
        }
        permutazione=gsl_permutation_calloc(dimensione);
        if(permutazione==NULL){
                printf("ERRORE - inversione_matriciale:
                        permutazione non alloccato.\n");
                exit(0);
        }
        segno=malloc(1*sizeof(int));
        if(segno==NULL){
                printf("ERRORE - inversione_matriciale:
                         segno non alloccato.\n");
                exit(0);
        }
        controllo=gsl_linalg_LU_decomp(evec, permutazione, segno);
        if(controllo!=0){
                printf("ERRORE - inversione_matriciale:
                        gsl_linalg_LU_decomp fallito.\n");
        }
        controllo=gsl_linalg_LU_invert(evec, permutazione, inverse)
        if(controllo!=0){
                printf("ERRORE - inversione_matriciale:
                        gsl_linalg_LU_invert fallito.\n");
        }
        for(i=0; i<dimensione; i++){
                for(j=0; j<dimensione; j++){
                        inversione[i][j]=gsl_matrix_get(inverse, i,
                }
        }
        gsl_matrix_free(inverse);
        gsl_permutation_free(permutazione);
        free(segno);
        return inversione;
}
```

4.15 Addizione_matriciale

La funzione riceve in ingresso l'intero "nrighe" e l'array "media" e si limita ad eseguire una operazione di addizione matriciale fra: la matrice "data_compression" ed una matrice fittizia (su ogni riga i-esima c'è il valore nella cella iesima dell'array "media").
La funzione restituisce la matrice "addizione", alloccata all'inizio del blocco.

```
double **addizione_matriciale(int nrighe, double *media){
        int i, j = 0;
        double **addizione;
        addizione=malloc(nrighe*sizeof(double*));
        if(addizione==NULL){
                printf("ERRORE - addizione_matriciale:
                        addizione non alloccato.\n");
                exit(0);
        }
        for(i=0; i<nrighe; i++){
                addizione[i]=malloc(dimensione*sizeof(double));
                if(addizione[i]==NULL){
                        printf("ERRORE - addizione_matriciale:
                                addizione[%d] non alloccato.\n", i);
                        exit(0);
                }
        }
        for(i=0; i<nrighe; i++){
                for(j=0; j<dimensione; j++){
                        addizione[i][j]=data_compression[i][j]+media[i];
                        if(debug==1){
                                printf("data_compression[%d][%d] = %2.2f,
                                        media[%d] = %2.2f,
                                        addizione[%d][%d] = %2.2f\n",i, j
                                        data_compression[i][j], i, media[
                                        i, j, addizione[i][j]);
                        }
                }
        }
        return addizione;
}
```

4.16 Stampa

Il compito di fornire un feedback all'utente è riservato alla presente funzione, infatti essa restituisce a schermo la stampa delle varie matrici (monodimensionali e bidimensionali) che si vanno a costruire nelle singole fasi del programma. In particolare la struttura di tale funzione consta sostanzialmente di un singolo blocco switch, in cui si identificano, in maniera capillare, il tipo di stampa che è richiesta: matrici monodimensionali di dimensionali, matrici bidimensionali di dimensioni qualsiasi e quelle matrici che costituiscono delle "chiavi di volta" del programma.

```
void stampa(int tipo, double **mt1, double *mt2, char nome[],
                int nr, int nc){
        int i,j = 0;
        switch(tipo){
                case 0:
                        printf("%s: %d \n", nome, nr);
                break;
                case 1:
                        printf("Stampa matrice %s: \n", nome);
                        for(i = 0; i < nr; i++){
                                for(j = 0; j < nc; j++){
                                        if(j == 0){
                                                printf("|");
                                        }
                                        printf("%2.2f\t", mt1[i][j]
                                        if(j == nc-1){
                                                printf("|");
                                        }
                                }
                                printf("\n");
                        }
                break;
                case 2:
                        printf("Stampa valor medio: \n");
                        for(i = 0; i < nr; i++){
                                printf("media[%d]=%2.2f \n", i, mt2
                        }
                break;
                case 3:
                        printf("Stampa autovalori(covarianza): \n")
```

```
                        for(i = 0; i < dimensione; i++){
                                if(i == 0){
                                        printf("|");
                                }
                                printf("%2.4g\t", autovalori[i]);
                                if(i == dimensione-1){
                                        printf("| \n");
                                }
                        }
                break;
                case 5:
                        printf("Errore la verifica non è andata
                                a buon fine\n");
                        printf("La matrice Verifica è diversa dalla
                                matrice di partenza \n");
                break;
                case 6:
                        printf("Stampa Matrice Estesa %s: \n", nome);
                        for(i = 0; i < dimensione; i++){
                                printf("%s[%d]=%2.2f \n", nome, i, mt2[i]
                        }
                break;
                case 7:
                        printf("Stampa Matrice Estesa %s : \n", nome);
                        for(i = 0; i < dimensione; i++){
                                for(j = 0; j < dimensione; j++){
                                        printf("%s[%d][%d]=%2.2f \n",
                                                nome, i, j, mt1[i][j]);
                                }
                        }
                break;
        }
        printf("\n");
}
```

4.17 Main

Il ruolo di "Maestro d'Orchestra" è affidato alla funzione "main", funzione che, oltre a controllare il corretto procedere del programma, allocca le variabili globali ed esegue un debugging sulla riuscita del'operazione di re-

versibilitá di riduzione di dimensione del data_set. Vediamo qual'è l'ordine di esecuzione delle funzioni, giá suggerita dall'ordine di presentazione delle funzioni in questo capitolo:

- lettura: Apertura e lettura del file che contiene il Data Set;
- allocazione: Si alloccano le variabili globali;
- media: Calcolo del valor medio;
- covarianza: Calcolo della matrice di covarianza, matrice che rappresenta la dispersione delle misure;
- eigen: Calcolo degli autovalori e autovettori, che costituiscono gli assi principali;
- ordinamento: Si ordinano gli autovalori (ed autovettori) in maniera decrescente, in modo da stabilire quali assi principali sono riferibili ai dati del Data set e quali siano riferibili al "rumore";
- sottrazione: La matrice "data_set" viene traslata in difetto pari ai suoi valori di "media";
- prodotto_matriciale: Si esegue l'operazione principale dell'algoritmo, il suo risultato sará il Data Compression;
- inversione_operazione: Funzione che svolge il ruolo di "vice-Maestro d'Orchestr"a in quanto gli è affidato il compito di calcolare l'equazione inversa e, quindi, determinare una matrice che sia "identica" alla matrice del Data Set;
- controllo: Il main conclude il suo operato eseguendo la verifica della riuscita dell'operazione di inversione;

Programma concluso.

```
int main(int argc, char **argv){
        int nrighe, i, j = 0;
        double *valor_medio, **matrice_covarianza,
                **sottrazione, **verifica;
        struct gsl pointer_gsl;
        if(debug){
                printf("MODALITA' DEBUG ATTIVA\n");
        }
        nrighe=lettura();
```

```
stampa(0, NULL, NULL, "nrighe", nrighe, 0);
stampa(0, NULL, NULL, "Dimensione del Data_set", dimensione, 0);
stampa(1, data_set, NULL, "Data Set Trasposto",
                nrighe, dimensione);
data_compression=malloc(nrighe*sizeof(double*));
if(data_compression==NULL){
        printf("ERRORE - main: data_compression
                non alloccato.\n");
        exit(0);
}
for(i=0; i<nrighe; i++){
        data_compression[i]=malloc(dimensione*sizeof(double));
        if(data_compression[i]==NULL){
                printf("ERRORE - main: data_compression[%d]
                        non alloccato.\n", i);
                        exit(0);
        }
}
autovalori=malloc(dimensione*sizeof(double));
if(autovalori==NULL){
        printf("ERRORE - main: autovalori non alloccato.\n");
        exit(0);
}
autovettori=malloc(dimensione*sizeof(double*));
if(autovettori==NULL){
        printf("ERRORE - main: autovettori non alloccato.\n");
        exit(0);
}
for(i=0; i<dimensione; i++){
        autovettori[i]=malloc(dimensione*sizeof(double));
        if(autovettori==NULL){
                printf("ERRORE - main: autovettori[%d]
                                non alloccato.\n", i);
                exit(0);
        }
}
valor_medio=media(nrighe);
matrice_covarianza=covarianza(valor_medio, nrighe);
pointer_gsl=eigen(matrice_covarianza);
for(i=0; i<dimensione; i++){
        autovalori[i]=gsl_vector_get(pointer_gsl.vector, i);
```

```
        }
        gsl_vector_free(pointer_gsl.vector);
        ordinamento(pointer_gsl.matrix);
        for(i=0; i<dimensione; i++){
                for(j=0; j<dimensione; j++){
                        autovettori[i][j]=gsl_matrix_get(
                                        pointer_gsl.matrix, i, j);
                        if(autovettori[i]==NULL){
                                printf("ERRORE - main:
                                        gsl_matrix_get fallito.\n")
                                exit(0);
                        }
                }
        }
        sottrazione=sottrazione_matriciale(nrighe, valor_medio);
        if(0 != prodotto_matriciale(nrighe, dimensione, dimensione,
          dimensione, sottrazione, autovettori)){
                printf("ERRORE - main: funzione prodotto_matriciale
                  ha riportato un errore, esecuzione arrestata.\n")
                return 0;
        }
        stampa(1, data_compression, NULL, "Data Compression",
                         nrighe, dimensione);
        free(sottrazione);
        verifica=inversione_operazione(nrighe, valor_medio,
                        pointer_gsl.matrix);
        stampa(1,  verifica, NULL, "Verifica", nrighe, dimensione);
        gsl_matrix_free(pointer_gsl.matrix);
        stampa(1, data_set, NULL, "Dataset", nrighe, dimensione);
        for(i=0; i<nrighe; i++){
                for(j=0; j<dimensione; j++){
                        if((floor(data_set[i][j]*troncamento)/tronca
                                        !=((floor(verifica[i][j]*troncament
                                                /troncamento)){
                                        stampa(5, NULL, NULL, NULL, 0, 0);
                        }
                }
        }
        printf("programma terminato.\n");
        return 0;
}
```

Capitolo 5

Conclusioni

Il programma viene eseguito senza nessun problema. Purtroppo i valori che il programma fornisce in uscita non sono corretti, ma si discostano dai valori originali. Questo errore è dovuto alle approssimazioni che effettua la libreria quando i numeri sono troppo piccoli.