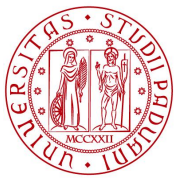


UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Università degli Studi di Padova

DIPARTIMENTO DI MATEMATICA "TULLIO LEVI-CIVITA"

Corso di Laurea Triennale in Matematica

Dalla Sparse Representation al Dictionary Learning: algoritmi K-SVD e SGK

Laureanda:
Francesca Marchese
Matricola 1224392

Relatore:
Prof. Fabio Marcuzzi

Anno Accademico 2023/2024
23 Febbraio 2024

Indice

0.1 Notazioni

In questo paragrafo introduttivo passeremo in veloce rassegna tutte le notazioni che verranno utilizzate in questa tesi:

1. $D \in \mathbb{R}^{n \text{x} K}$ è una matrice *dizionario* le cui K colonne di lunghezza n vengono definite *atomi*;
2. $Y \in \mathbb{R}^{n \text{x} N}$ è una matrice le cui N colonne di lunghezza n vengono definite *segnali-esempio*;
3. $X \in \mathbb{R}^{K \text{x} N}$ è la matrice le cui N colonne di lunghezza K sono le rappresentazioni *sparse* dei segnali-esempio di Y relativamente al dizionario D;
4. t rappresenta l'indice di sparsità che deve avere la soluzione del problema;
5. G è la matrice di Gram;

Capitolo 1

Dictionary Learning

1.1 Introduzione: il problema della *Sparse Representation*

In questo documento verrà presentato uno dei problemi di crescente interesse negli ultimi decenni e sempre più utilizzato nell'applicazione a modelli di uso quotidiano, tra i quali, la video sorveglianza [6] o il riconoscimento facciale [10]: la *rappresentazione sparsa dei segnali*.

Tale problema viene espresso come soluzione di un sistema lineare: [1]

$$y = Dx$$

dove:

- $D \in \mathbb{R}^{nxK}$ è una matrice *dizionario* le cui K colonne vengono definite *atomi*;
- $y \in \mathbb{R}^n$ è il segnale che verrà rappresentato come combinazione lineare di K atomi;
- $x \in \mathbb{R}^K$ sarà l'incognita del nostro sistema lineare.

Viene assunto $n < K$ e, per evitare che il sistema non abbia soluzioni, si assume D matrice a rango massimo, così che y appartenga allo spazio generato dalle colonne di D [4].

Siamo quindi di fronte ad un sistema *sottodeterminato*, cioè con più incognite che equazioni, e, avendo assunto D a rango massimo, avremo infinite soluzioni al nostro problema.[1]

[1] Si noti che se non si fosse scelto D a rango massimo si potrebbe incorrere nel non avere alcuna soluzione al problema.

Per ovviare a questo problema di indeterminatezza del sistema, è necessario apporre dei vincoli alla soluzione e, nel nostro caso, verrà scelta la soluzione *più sparsa.* [2]

Con questi vincoli imposti si delinea il problema di *Sparse Representation*, la cui soluzione viene solitamente espressa in due modi, a seconda della rappresentazione di y: esatta ($y = Dx$) o approssimativa ($y \approx Dx$) [1]:

$$(P_0) : \min_x \|x\|_0 \text{ soggetto a } y = Dx \tag{1.1}$$

$$(P_{0,\varepsilon}) : \min_x \|x\|_0 \text{ soggetto a } \|y - Dx\|_2 \leq \varepsilon \tag{1.2}$$

dove $\|\cdot\|_0$ è detta la norma che conta le entrate del vettore diverse da 0, cioè

$$\|x\|_0 = \#\{i \colon x_i \neq 0\}$$

anche se, in realtà, non è una norma vera e propria poiché non soddisfa una della proprietà che caratterizzano le norme: $\|\cdot\|_0$ come l'abbiamo definita sopra, non soddisfa $\|cx\| = |c|\|x\|$ con $c \in \mathbb{R}$.

In genere si va alla ricerca di una soluzione *approssimativa*, e quindi si andrà a risolvere $(P_{0,\varepsilon})$ utilizzando i cosiddetti *Greedy methods*, come, ad esempio l'Orthogonal-Matching-Pursuit (OMP), il Matching-Pursuit (MP) o il FOCUSS.

La caratteristica di questi metodi è di iniziare da un vettore $x_0 = 0$ e costruire iterativamente un approssimante x_k. Ad ogni iterata l'algoritmo valuta l'errore residuo e, se il suddetto errore è sotto la soglia di tolleranza, l'algoritmo si interrompe e produce il risultato cercato.[4]

1.1.1 L'algoritmo OMP

In questo paragrafo analizziamo brevemente l'algoritmo dell'Orthogonal-Matching-Pursuit e consideriamo il caso particolare di una matrice dizionario D *simmetrica*[1] e *definita positiva* [2]. Questo algoritmo ha come obiettivo quello di calcolare x a partire dal problema (1.2) già visto precedentemente. [11]

L'algoritmo parte, all'iterazione $k = 0$, da una soluzione x_0 nulla e quindi un residuo $r_0 = y$ e da un set di indici, $\Gamma^0 = \emptyset$, che alle iterazioni successive corrisponderà agli indici dei k atomi selezionati di D, ovvero gli atomi corrispondenti alle entrate non nulle del vettore x da recuperare.

[2] Una matrice o un vettore vengono definiti *sparsi* quando la maggior parte delle entrate è pari a 0. Ad esempio la matrice:

$$\begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

contiene 5 elementi diversi da 0 e 11 zeri, quindi è una matrice con *sparsità* al $68,75\%$

[1] $D = D^T$

[2] $\det(D) > 0$

Prendiamo in considerazione la matrice di Gram corrispondente a tutti i coefficienti:

$$G = D^T \cdot D$$

e la matrice di Gram ristretta ai k atomi selezionati:

$$G_{\Gamma^k} = D_{\Gamma^k}^T \cdot D_{\Gamma^k}$$

La matrice G_{Γ^k} è simmetrica e definita positiva [3], quindi può essere scomposta tramite fattorizzazione di Cholesky: esiste un'unica matrice L triangolare inferiore con elementi positivi sulla diagonale principale tale che:

$$G_{\Gamma^k} = L_k \cdot L_k^T$$

A questo punto si può aggiornare il residuo r^k utilizzando il peso

$$s_k = (D_{\Gamma^k}^T \cdot D_{\Gamma^k})^{-1} \cdot D_{\Gamma^k}^T \cdot y = (G_{\Gamma^k})^{-1} \cdot D_{\Gamma^k}^T \cdot y = (L_k \cdot L_k^T)^{-1} \cdot D_{\Gamma^k}^T \cdot y$$

ottenendo così:

$$r_k = y - D_{\Gamma^k} \cdot s_k = y - D_{\Gamma^k} \cdot (L_k * L_k^T)^{-1} \cdot D_{\Gamma^k}^T \cdot y$$

Ad ogni step k viene controllato il valore del residuo in relazione ad una tolleranza assegnata e scelto il prossimo indice di atomo da includere in Γ^{k+1} (vedremo più avanti che i criteri di minimizzazione dell'algoritmo potranno essere di tipo *sparsity-based* oppure di tipo *error-based*). Una volta che l'algoritmo raggiunge la soglia di tolleranza desiderata, esso restituisce il vettore degli atomi $x = s_{\bar{k}}$ dove $\bar{k}$ è l'ultima iterazione che l'algoritmo affronta.

1.2 Un passo in più: il *Dictionary Learning*

Il problema che verrà studiato in questo documento non si fermerà all'obiettivo di trovare il miglior approssimante, il vettore x, ma andrà oltre.

Utilizzeremo nella nostra analisi una matrice $Y = \{y_i\}_{i=1}^N$ che contiene un set di N segnali-esempio e la matrice delle corrispondenti rappresentazioni sparse $X = \{x_i\}_{i=1}^N$.

$$y_i = Dx_i \text{ per } i = 1 \dots N \tag{1.3}$$

A questo punto il nuovo problema, chiamato di *dictionary learning*, si sviluppa in due fasi [7]:

1. un'iterata di *Sparse Recovery*, dove viene aggiornata la matrice X dei coefficienti tramite l'uso di uno dei *Greedy Methods* mentre la matrice dizionario D rimane fissa:

$$X^{(k+1)} = \min_{X \in \mathcal{X}} \|Y - D^k X\|_F^2 \tag{1.4}$$

[3] Per definizione, la matrice di Gram è semidefinita positiva. Inoltre, avendo scelto la matrice D simmetrica e definita positiva, ottengo $G = D^2$ e di conseguenza anche G eredita le proprietà di simmetria e di essere definita positiva.

2.un'iterata di *Dictionary Update*, dove viene aggiornato il dizionario D mentre la matrice dei coefficienti sparsi rimane fissa:

$$D^{(k+1)} = \min_{D \in \mathcal{D}} \|Y - DX^{(k+1)}\|_F^2 \tag{1.5}$$

dove $\|\cdot\|_F$ è la norma di Frobenius e $\mathcal{D}$ e $\mathcal{X}$ sono set ammissibili di, rispettivamente, dizionari e matrici dei coefficienti.

Ci sono vari algoritmi per la risoluzione di questo nuovo problema in due fasi: il METODO DELLE DIREZIONI OTTIMALI (MOD) [5], ad esempio, aggiorna il dizionario partendo dalla minimizzazione dell'errore $E \in \mathbb{R}^{n\text{x}N}$:

$$||E||_F^2 = ||Y - DX||_F^2 \tag{1.6}$$

L'algoritmo cerca il minimo locale di 1.6, analizzando il punto in cui si annulla la sua matrice Jacobiana[4] rispetto a D, ovvero:

$$(Y - DX)X^T = 0 \tag{1.7}$$

$$YX^T - DXX^T = 0 \tag{1.8}$$

$$YX^T = DXX^T \tag{1.9}$$

$$YX^T(XX^T)^{-1} = DXX^T(XX^T)^{-1} \tag{1.10}$$

ottenendo così l'aggiornamento iterativo del dizionario seguendo la regola:

$$D^{(k+1)} = YX^T(XX^T)^{-1} \tag{1.11}$$

Il MOD utilizza come greedy method l'OMP o il FOCUSS, anche se è noto dalla pratica che il risultato migliore viene ottenuto utilizzando il secondo metodo.

Un altro metodo che può essere utilizzato per risolvere questo nuovo tipo di problema è la K-SVD, ovvero la **K-Decomposizione a Valori Singolari**, che utilizza come greedy method l'OMP.

[4]data una funzione $f : \mathbb{R}^n \to \mathbb{R}^m$, la sua MATRICE JACOBIANA è definita come la matrice $\mathcal{J}_f \in \mathbb{R}^{m,n}$ le cui entrate sono le derivate prime parziali della funzione.

Capitolo 2

L'algoritmo della K-SVD

In questo capitolo descriveremo come si sviluppa l'algoritmo iterativo della K-SVD, che ha come obiettivo, partendo da una matrice dei segnali Y, quello di trovare iterativamente un dizionario D e una rappresentazione X corrispondenti a tali segnali secondo l'equazione $Y = DX$. [2] [1]

```
1 function[D,X,err] = ksvd(params, varargin)
```

2.1 I parametri in entrata

In input alla funzione possono essere inseriti vari tipi di parametri iniziali. Elenchiamo di seguito quelli necessari al processo della K-SVD:

1. *params.data* inizializza la matrice dei segnali Y;
2. *params.initdict*$(= D_0)$ inizializza il dizionario da cui partirà il processo iterativo di recupero del dizionario;
3. *params.dictsize*$(= K)$ specifica il numero di atomi che verranno analizzati e, se *params.initdict* non è specificato, questo parametro servirà per inizializzare anche il dizionario di partenza dell'algoritmo;
4. *params.Tdata*/*params.Edata* specifica il tipo di minimizzazione su cui si baserà la K-SVD (questi parametri calcolano l'errore che il metodo compie ad ogni iterata: sono utili per capire la stabilità del metodo in una situazione dove il risultato richiesto non è esatto ma approssimato):

   se $params.Tdata = t$ verrà usato lo *sparsity-based minimization method* dove la funzione obiettivo da minimizzare è data da

$$min_{D,X}|Y - DX|_F \text{ t.c. } |X_i|_0 \leq t$$

[1] Per accedere all'algoritmo completo cliccare qui

se $params.Edata = \epsilon$ verrà usato l'*error-based minimization method* dove la funzione obiettivo da minimizzare è data da

$$min_{D,X}|X|_0 \text{ t.c. } |Y_i - DX|_2 \leq \epsilon$$

se sono entrambi specificati verrà usato $Tdata$;

5. *params.iternum* indica il numero di iterazioni che l'algoritmo deve svolgere (se il parametro non è presente, $params.iternum = 10$ di default);

6. *params.codemode* specifica quale tipo di minimizzazione verrà usata come criterio di stop quando sono presenti sia $params.Edata$ sia $params.Tdata$. Se questo parametro non figura ed entrambi i metodi di minimizzazione sono esplicitati, verrà usato lo *sparsity-based method*.

2.1.1 INPUT/OUTPUT

L'algoritmo ha come input i parametri appena elencati (*params*), a cui si possono aggiungere le stringhe di testo (*varargin*) che danno come output un messaggio per ogni iterazione del metodo. La stringa può contenere una o più delle lettere 'i', 'r', 't' che corrispondono a:

1.'i' numero dell'iterazione;

2.'r' numero degli atomi del dizionario rimpiazzati;

3.'t' valore della funzione obiettivo;

In uscita, il nostro algoritmo ha come output fissi il *trained dictionary* D e la matrice dei coefficienti X. A questi due risultati si può aggiungere anche il vettore *err* che contiene i valori della funzione obiettivo che vengono archiviati ad ogni iterazione e sono molto utili per controllare l'andamento della K-SVD.

2.2 L'algoritmo vero e proprio

Vediamo ora come si sviluppa l'algoritmo della K-SVD.

Come primissima cosa il codice computa tutti i parametri in input, assegnando ad ogni parametro una variabile che verrà poi utilizzata all'interno del loop principale. Prendiamo, ad esempio, l'inizializzazione del dizionario D: se tra i parametri è specificato params.initdict allora il dizionario iniziale sarà:

```
1 D = params.initdict(:,1:dictsize);
```

altrimenti viene inizializzato utilizzando una permutazione random da 1 al numero di indici per i quali la matrice dei segnali, $Y = data$, è maggiore di $1e-6$:

```
data_ids = find(colnorms_squared(data) > 1e-6);
perm = randperm(length(data_ids));
D = data(:,data_ids(perm(1:dictsize)));
```

Passiamo ora al loop principale:

```
foriter = 1:iternum
```

la matrice X viene aggiornata ad ogni iterazione tramite il codice:

```
X = sparsecode(data,D,YtY,G,thresh)
```

che utilizza la tecnica dello Sparse Coding tramite il metodo OMP (Par. 1.1.1) e in cui i parametri in input sono:

1.data = params.data, ovvero la matrice dei segnali Y;

2.D è il dizionario dell'iterazione corrente;

3.YtY = $Y^T \cdot Y$;

4.G = $D^T \cdot D$, ovvero la matrice di Gram;

5.thresh identifica il metodo di minimizzazione utilizzato dall'algoritmo, ovvero thresh = params.Edata oppure thresh = params.Tdata.

Successivamente viene aggiornato anche il dizionario D tramite un processo, anch'esso iterativo, di ottimizzazione degli atomi (le colonne della matrice): si prende in considerazione un atomo di D, d_k, e la riga corrispondente in X, x_T^k. Considero la funzione obiettivo:

$$|Y - DX|_F = |Y - \sum_{j=1}^{K} d_j x_T^j|_F = |Y - (\sum_{j \neq k} d_j x_T^j) - d_k x_T^k|_F \tag{2.1}$$

e definisco:

$$Y - (\sum_{j \neq k} d_j x_T^j) := E_k \tag{2.2}$$

l'errore compiuto quando l'atomo k-esimo viene rimosso;

$$\omega_k := \{i \text{ t.c. } 1 \leq i \leq K, x_T^k(i) \neq 0\} \tag{2.3}$$

l'insieme degli indici per cui gli esempi y_i utilizzano l'atomo d_k;

$$\Omega_k \in \mathbb{R}^{N \text{x} |\omega_k|} \tag{2.4}$$

la matrice con entrate $(\omega_k(i), i)$ pari a 1 e 0 altrove.

A questo punto, usando la matrice Ω_k, restringo la matrice E_k ai soli esempi che utilizzano l'atomo d_k e ottengo $E_k^R = E_k \Omega_k$, a cui posso applicare la

Decomposizione a Valori Singolari (SVD), ottenendo $E_k^R = U\Delta V^T$ [2]. A questo punto la colonna aggiornata $\tilde{d_k}$ del dizionario sarà la prima colonna della matrice U mentre il vettore dei coefficienti aggiornato $\tilde{x_k}$ sarà la prima colonna della matrice V moltiplicata per $\Delta(1,1)$. Si può notare che, usando questo metodo, le colonne di D rimangono dei vettori normalizzati e che il supporto della matrice D, o rimane lo stesso ad ogni iterata, o diminuisce a conseguenza dell'esclusione degli esempi che non utilizzano l'atomo d_k. Vediamo il codice che esegue quanto detto:

```
replaced_atoms = zeros(1,dictsize);
unused_sigs = 1:size(data,2);
p = randperm(dictsize);
fork = 1:dictsize
 [D(:,p(k)),x_k,data_indices,unused_sigs,
    replaced_atoms] = optimize_atom(data,D,p(k),X,
    unused_sigs,replaced_atoms);
 X(p(k),data_indices) = x_k;

end
```

In questo codice vengono inizializzati i vettori *replaced_atoms* e *unused_sigs* che servono, rispettivamente, per segnare quali atomi sono stati rimpiazzati e quali segnali sono stati usati per rimpiazzare gli atomi che non servono più (in modo di non selezionare un segnale per due volte). La funzione *optimize_atom* è interna all'algoritmo della K-SVD e ottimizza la colonna k-esima del dizionario D partendo dal trovare gli indici del vettore *data* che utilizzano l'atomo k-esimo (supponiamo, per semplicità, il caso in cui l'atomo d_k sia effettivamente utilizzato):

```
function[atom,x_k,data_indices,unused_sigs,
   replaced_atoms] = optimize_atom(Y,D,k,X,unused_sigs
   ,replaced_atoms)

[x_k, data_indices] = sprow(x,k);

smallX = X(:,data_indices);
Dk = D(:,k);

atom = collincomb(Y,data_indices,x_k') - D*(smallX*x_k
   ') + Dk*X_k*x_k');
atom = atom/norm(atom);
X_k = rowlincomb(atom,Y,1:size(Y,1),data_indices) - (
   atom'*D)*smallX + (atom'*Dk)*x_k;

end
```

[2] dove U e V sono matrici unitarie e Δ è una matrice diagonale

Infine, il vettore *err* viene aggiornato ad ogni iterata usando la seguente funzione:

```
function err compute_err(D,X,data)

global CODE_SPARSITY codemode

if(codemode == CODE_SPARSITY)
  err=sqrt(sum(reperror2(data,D,X))/numel(data));
else
  err=nnz(X)/size(data,2);
end
```

2.3 Applicazioni

Vediamo in questo capitolo qualche esempio di applicazione dell'algoritmo appena visto:

2.3.1 Tdata

Concentriamoci prima a studiare i casi dove la funzione obiettivo è data da:

$$min_{D,X}|Y - DX|_F \text{ t.c. } |X_i|_0 \leq t$$

Inizializziamo le variabili che ci serviranno: il dizionario che mi servirà per inizializzare la matrice dei segnali Y, $D \in \mathbb{R}^{n \times K}$ lo scelgo in maniera randomica ma tale che abbia le colonne normalizzate all'unità:

```
n=20;
K=50;
D=normcols(randn(n,K));
```

l'indice di sparsità a cui X deve essere sottoposta ad ogni iterazione:

```
t=3;
```

il numero di esempi che verranno analizzati corrisponde a N, che equivale al numero di colonne di X, che anch'essa verrà inizializzata partendo da una permutazione random degli interi da 1 a K che andrà a modificare ad una ad una le colonne di X utilizzando l'indice di sparsità (si noti come N deve essere un numero maggiore di K):

```
N=1500;
X=zeros(K,N);
fori=1:N
  p=randperm(K);
 X(p(1:t),i)=randn(t,1);
end
```

Posso ora inizializzare il segnale Y utilizzando D e X appena introdotti e lo sottopongo ad un rumore, snr, così da avere dei risultati più interessanti da analizzare:

```
Y=D*X;
snr=20;
Y=normcols(Y)+10^(-snr/20)*normcols(randn(n,N));
```

A questo punto introduco i parametri che verranno usati dall'algoritmo e lo posso far girare:

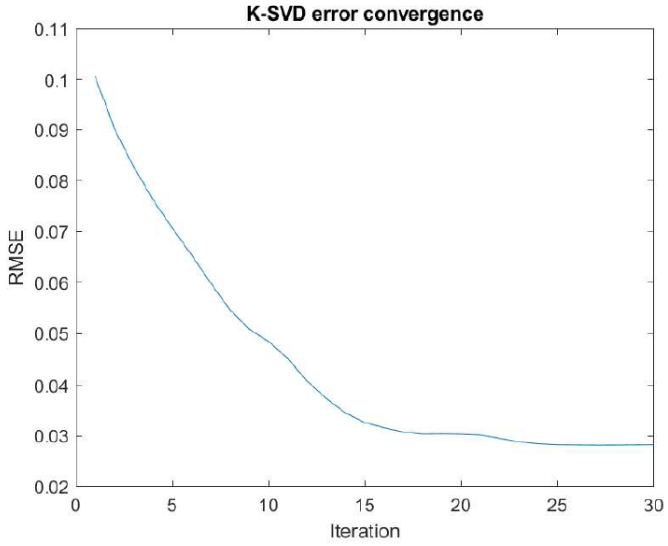
```
params.data=Y;
params.Tdata=t;
params.dictsize=K;
params.iternum=30;

[D_ksvd, X_ksvd, err] = ksvd(params,'tr')
```

Dall'algoritmo ottengo i seguenti risultati:

```
Iteration 1 / 30 complete, RMSE = 0.1007, replaced 0 atoms
Iteration 2 / 30 complete, RMSE = 0.09021, replaced 0 atoms
Iteration 3 / 30 complete, RMSE = 0.08275, replaced 0 atoms
...
Iteration 8 / 30 complete, RMSE = 0.05467, replaced 0 atoms
Iteration 9 / 30 complete, RMSE = 0.05085, replaced 0 atoms
Iteration 10 / 30 complete, RMSE = 0.04845, replaced 1 atoms
Iteration 11 / 30 complete, RMSE = 0.04511, replaced 2 atoms
...
Iteration 18 / 30 complete, RMSE = 0.03039, replaced 1 atoms
...
Iteration 28 / 30 complete, RMSE = 0.0282, replaced 0 atoms
Iteration 29 / 30 complete, RMSE = 0.02824, replaced 0 atoms
Iteration 30 / 30 complete, RMSE = 0.02826, replaced 0 atoms
```

Inoltre, aggiungendo in coda queste righe al codice, posso ottenere ulteriori risultati:

```
figure; plot(err); title('K-SVDerrorconvergence    ');
xlabel('Iteration'); ylabel('RMSE');
[n1,K1]=size(D_ksvd);
printf('  Dictionarysize:%dx%d      ', n, K);
printf('  Numberofexamples:%d     ', N);
printf('  Recovereddictionarysize:%dx%d       ', n1, K1)
    ;

[dist,ratio] = dictdist(D_ksvd,D);
printf('  Ratioofrecoveredatoms:%.2f%%\n      ', ratio
    *100);
```




```
  Dictionary size: 20 x 50
  Recovered dictionary size: 20 x 50
  Number of examples: 15000
  Ratio of recovered atoms: 96.00%
```

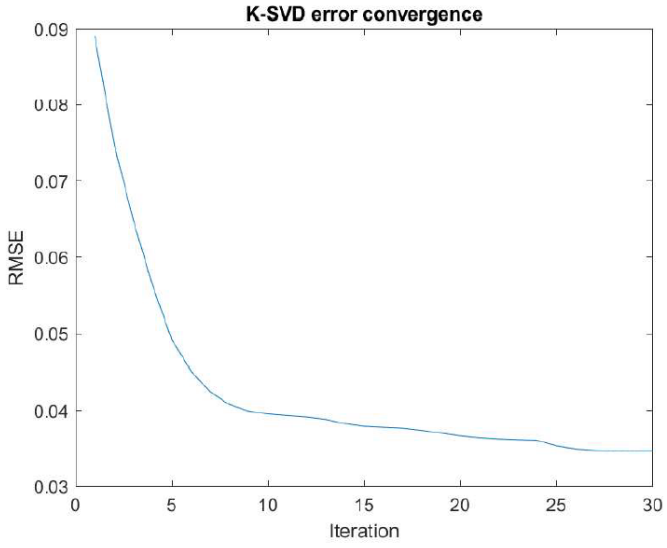
Possiamo notare come l'andamento della radice dell'errore quadratico medio della funzione obiettivo $||Y - DX||_F$ decresce andando a convergere al valore $RMSE = 0.02826$. Da questi risultati capiamo quindi che l'algoritmo della K-SVD è stabile e tende a ritrovare il dizionario di partenza (D che abbiamo usato assieme a X per inizializzare la matrice dei segnali Y) con un successo del 96%, come si evince dall'utilizzo del comando *dictdist*, che misura la distanza tra il dizionario iniziale D e il dizionario prodotto dal nostro algoritmo D_{ksvd}, e la *ratio* tra i due dizionari, ovvero il numero di atomi di D che sono presenti

anche in D_{ksvd} diviso per il numero di colonne di D. Si nota anche che l'algoritmo non sempre rimpiazza atomi del dizionario ad ogni iterazione perché non sempre porta a miglioramenti l'andare a modificare un atomo e, in questo caso, il dizionario ottenuto ha la stessa dimensione del dizionario di partenza, ciò vuol dire che tutti gli esempi y_k utilizzavano l'atomo d_k.

Ripetiamo ora l'esperimento utilizzando gli stessi parametri e cambiando solamente il dizionario iniziale con cui inizializzerò la matrice dei segnali Y, utilizzando un dizionario *overcompleto*[3]:

```
n = 20;
K = 50;
D=odctdict(n,K);
```

In questo caso si notano delle differenze nei risultati:




```
  Dictionary size: 20 x 50
  Recovered dictionary size: 20 x 50
  Number of examples: 15000
  Ratio of recovered atoms: 78.00%
```

Dal grafico si deduce che, in questo caso, il metodo converge più velocemente, di contro la percentuale di atomi del dizionario recuperati è notevolmente più bassa. Questo risultato è dato dal fatto che il comando *odctdict(n,K)* utilizza la Trasformata Discreta del Coseno, andando ad inizializzare una a una le colonne secondo la seguente formula:

[3]Una matrice D si dice OVERCOMPLETA se esiste almeno una colonna di D che è combinazione lineare delle restanti colonne.

```
function D = odctdict(n,K)

D = zeros(n,K);
D(:,1) = 1/sqrt(n);
for k = 2:K
  v = cos((0:n-1)*pi*(k-1)/K)';
  v = v-mean(v);
  D(:,k) = v/norm(v);
end
```

mentre il comando *normcols(randn(n,K))* genera una ad una le entrate della matrice utilizzando la distribuzione normale standard, con la sola proprietà di andare a normalizzare le colonne. Di conseguenza otteniamo più atomi recuperati utilizzando il secondo comando poiché, come visto, il nostro algoritmo opera andando a mantenere la normalizzazione delle colonne del dizionario D e quindi, essendo il dizionario "obiettivo" per costruzione già normalizzato, è più facile da ricostruire. Invece la differente velocità di convergenza è causata dal fatto che il dizionario overcompleto è già in sè una rappresentazione sparsa del segnale, mentre il dizionario random produce una rappresentazione densa del segnale e questo comporta una convergenza più lenta.

Facciamo un ultimo esempio andando a modificare l'indice di sparsità t e tenendo in considerazione un dizionario iniziale con le colonne normalizzate:

```
n = 20;
K = 50;
N = 15000;
t = 1;
snr = 20;
D = normcols(randn(n,K));
X = zeros(K,N);
for i = 1:N
  p = randperm(K);
  X(p(1:t),i) = randn(t,1);
end
Y = D*X;
Y = normcols(Y) + 10^(-snr/20)*normcols(randn(n,N));
params.data = Y;
params.Tdata = t;
params.dictsize = K;
params.iternum = 30;

[D_ksvd,X_ksvd,err] = ksvd(params,'tr');
[dist,ratio] = dictdist(D_ksvd,D);
printf('  Ratioofrecoveredatoms:%.2f%%\n      ', ratio
    *100);
```

Dai risultati si nota che la percentuale degli atomi recuperati è notevolmente più alta e che il metodo ad ogni iterazione sostituisce molti più atomi rispetto a quando si è scelto un indice di sparsità più grande:

```
Iteration 1 / 30 complete, RMSE = 0.104, replaced 11 atoms
Iteration 2 / 30 complete, RMSE = 0.07727, replaced 10 atoms
Iteration 3 / 30 complete, RMSE = 0.06754, replaced 8 atoms
Iteration 4 / 30 complete, RMSE = 0.06356, replaced 7 atoms
Iteration 5 / 30 complete, RMSE = 0.05178, replaced 5 atoms
Iteration 6 / 30 complete, RMSE = 0.04657, replaced 4 atoms
Iteration 7 / 30 complete, RMSE = 0.03475, replaced 2 atoms
Iteration 8 / 30 complete, RMSE = 0.02815, replaced 1 atoms
....
Iteration 27 / 30 complete, RMSE = 0.02175, replaced 0 atoms
Iteration 28 / 30 complete, RMSE = 0.02175, replaced 0 atoms
Iteration 29 / 30 complete, RMSE = 0.02175, replaced 0 atoms
Iteration 30 / 30 complete, RMSE = 0.02175, replaced 0 atoms

Ratio of recovered atoms: 100.00%
```

L'algoritmo raggiunge l'obiettivo dopo 8 iterazioni e trova il dizionario di partenza con una radice quadrata dell'errore quadratico medio pari a 0.02175 e meglio di così non può fare a livello di errore. Questo risultato è dato dal fatto che il metodo OMP utilizzato è più performante di fronte ad un indice di sparsità più basso.

2.3.2 Edata

In questa sezione proveremo degli esperimenti utilizzando la minimizzazione basata sull'errore. La funzione obiettivo che l'algoritmo va a minimizzare è quindi: $min_{D,X}|X|_0$ t.c. $|Y_i - DX|_2 \leq \epsilon$.

Inizializziamo le variabili:

```
n = 20;
K = 50;
```

```
N = 15000;
t = 3;
snr = 20;
D = normcols(randn(n,K));

X = zeros(K,N);
fori = 1:N
  p = randperm(K);
  X(p(1:t),i) = randn(t,1);
end

Y = D*X;

Y = normcols(Y) + 10^(-snr/20)*normcols(randn(n,N));


params.data = Y;
params.Edata = 0.3;
params.dictsize = K;
params.iternum = 30;
```

Facciamo quindi girare l'algoritmo tenendo conto $\epsilon = 0.3$

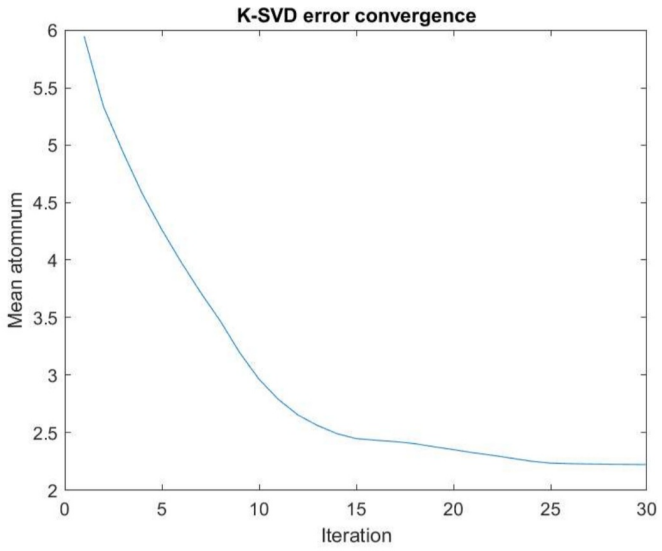
```
[D_ksvd,X_ksvd,err] = ksvd(params,'tr');
```

ed otteniamo ad ogni iterazione i seguenti risultati:

```
Iteration 1 / 30 complete, mean atomnum = 5.945,
   replaced 0 atoms
Iteration 2 / 30 complete, mean atomnum = 5.329,
   replaced 0 atoms
Iteration 3 / 30 complete, mean atomnum = 4.935,
   replaced 0 atoms
Iteration 4 / 30 complete, mean atomnum = 4.572,
   replaced 0 atoms
Iteration 5 / 30 complete, mean atomnum = 4.262,
   replaced 0 atoms
Iteration 6 / 30 complete, mean atomnum = 3.978,
   replaced 0 atoms
Iteration 7 / 30 complete, mean atomnum = 3.716,
   replaced 0 atoms
Iteration 8 / 30 complete, mean atomnum = 3.471,
   replaced 0 atoms
Iteration 9 / 30 complete, mean atomnum = 3.196,
   replaced 1 atoms
Iteration 10 / 30 complete, mean atomnum = 2.964,
   replaced 0 atoms
```

```
Iteration 11 / 30 complete, mean atomnum = 2.788,
    replaced 0 atoms
Iteration 12 / 30 complete, mean atomnum = 2.654,
    replaced 0 atoms
Iteration 13 / 30 complete, mean atomnum = 2.563,
    replaced 0 atoms
Iteration 14 / 30 complete, mean atomnum = 2.493,
    replaced 1 atoms
...
Iteration 30 / 30 complete, mean atomnum = 2.224,
    replaced 0 atoms


Ratio of recovered atoms: 98.00%
```

Notiamo che il valore del *mean atomnum*, ovvero la media$\{|X_i|_0\}$ che equivale a $|X|_0/\text{size}(X, 2)$ decresce in maniera rapida e significativa. Anche in questo caso possiamo dedurre che l'algoritmo della K-SVD risulta stabile e convergente:

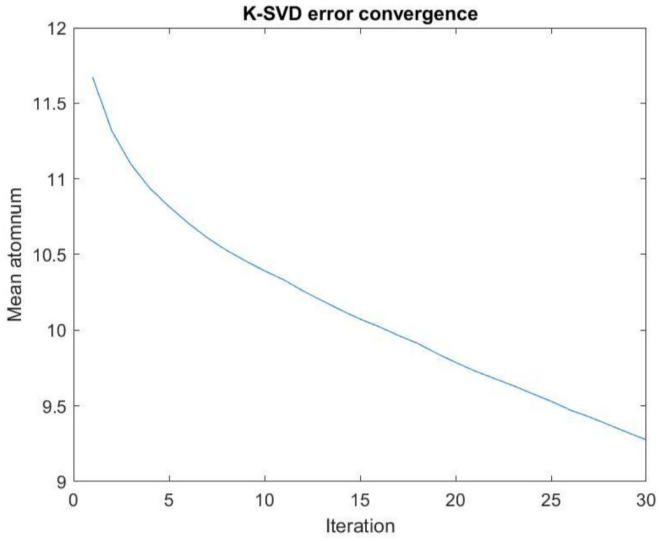


Proviamo ora a diminuire il valore di ϵ e vediamo che il risultato è molto diverso:

```
params.Edata = 0.1;
```

Otteniamo, mantenendo invariato il resto del codice, che il valore di *mean atomnum* rimane molto alto, tende sì a diminuire di iterazione a iterazione, ma non in maniera significativa come nell'esempio precedente:

```
Iteration 1 / 30 complete, mean atomnum = 11.68,
    replaced 0 atoms
Iteration 2 / 30 complete, mean atomnum = 11.32,
    replaced 0 atoms
Iteration 3 / 30 complete, mean atomnum = 11.1,
    replaced 0 atoms
...
Iteration 30 / 30 complete, mean atomnum = 9.275,
    replaced 0 atoms

Ratio of recovered atoms: 8.00%
```



L'algoritmo, quindi, fallisce nel recuperare il dizionario di partenza D con un valore ϵ troppo basso.

Capitolo 3

MOD vs K-SVD: una generalizzazione della K-Means?

Abbiamo visto nel capitolo precedente come l'algoritmo della K-SVD aggiorni in maniera sequenziale K atomi utilizzando la Decomposizione a Valori Singolari, mentre il Metodo delle Direzioni Ottimali aggiorna *tutti* gli atomi in una sola volta tenendo conto del valore minimo della media dell'errore quadratico medio (MMSE) e non in maniera sequenziale. Viene spontaneo chiedersi se questi algoritmi possono essere visti come la generalizzazione di un metodo che aggiorni *solo un atomo* per ogni iterazione?

Si potrebbe pensare alla *K-means*, utilizzata come metodo di Dictionary Training nella VQ (Vector Quantization), un tipo di Sparse Representation estrema, dove solo uno atomo di D viene utilizzato: la matrice dei coefficienti è composta da vettori pari a $x = e_k$, ovvero è un vettore con tutte le entrate nulle tranne la k-esima che è pari a 1 [1]. All'iterazione i, la K-means, nello stadio di Sparse Coding (vedi 1.4), cerca l'indice k tale per cui $y_j = D^i x_j = D^i e_k$ e la matrice Y dei segnali viene partita in K *cluster*, $R_k^i = \{j \text{ t.c. } y_j = D^i e_k\}$, mentre trasforma il problema globale di Dictionary Update, 1.5, in una minimizzazione locale:

$$d_k^{(i+1)} = \min_{d_k} \sum_{j \in R_k^i} ||y_j - d_k||_2^2 = (1/|R_k^i|) \sum_{j \in R_k^i} y_j \tag{3.1}$$

Si vedrà come il MOD è un algoritmo che generalizza la K-means, al contrario della K-SVD[8].

Quando si esegue l'algoritmo della K-SVD, una volta trovato $\tilde{X}$ ristretta ai K atomi, non è garantito il fatto che tutte le entrate della matrice siano 0 oppure 1, perciò viene a mancare la struttura binaria imposta dalla VQ, inoltre la K-SVD richiede che gli atomi del dizionario siano normalizzati, ovvero $||d_k^{(i+1)}||_2 = 1$,

[1] Si noti come le colonne di X siano, per costruzione, ortogonali tra loro.

il che è incompatibile con 3.1. Per questi due motivi si può concludere che la K-SVD non può essere vista come generalizzazione del metodo della K-means.

Dall'altro canto, il MOD non richiede che gli atomi del dizionario abbiano norma unitaria e aggiorna D indipendentemente da X. Considerando che le righe di X sono ortogonali tra loro (per via dei cluster in cui è suddivisa la matrice Y), abbiamo così:

$$XX^T = diag\{|R_1|, ..., |R_K|\} \tag{3.2}$$

dove $|R_k|$ corrisponde al numero dei segnali associati all'atomo d_k e similmente:

$$YX^T = \sum_{j \in R_k^i} y_j \tag{3.3}$$

Mettendo insieme 3.2 e 3.3 e applicandoli alla regola del MOD, 1.11, ottengo esattamente 3.1 e posso concludere che il Metodo delle Direzioni Ottimali è una buona generalizzazione della K-means.

3.1 SGK, una generalizzazione sequenziale della K-means

Gli algoritmi di tipo sequenziale, come la K-means e la K-SVD, sono più accurati e richiedono meno risorse rispetto al MOD. Esiste però un metodo di tipo sequenziale e che sia allo stesso tempo una generalizzazione della K-means? In [8] viene introdotto a questo scopo il metodo della *Generalizzazione Sequenziale della K-Means*, che chiameremo, per brevità, *SGK*. Anche in questo metodo l'aggiornamento del dizionario D avviene indipendentemente dall'aggiornamento di X, non viene richiesto che gli atomi del dizionario abbiano norma unitaria (differentemente dalla K-SVD) e, come per il MOD, l'aggiornamento degli atomi del dizionario può essere semplificato a 3.1. La parte interessante è che l'aggiornamento del dizionario avviene in maniera *sequenziale*. Vediamolo: il problema di ottimizzazione sequenziale si presenta come:

$$d_k^{(i+1)} = \min_{d_k} ||E_k^i - d_k X_{rowk}^i||_F^2 \tag{3.4}$$

dove X_{rowk}^i equivale alla riga di coefficienti di X corrispondente all'atomo d_k, mentre la matrice degli errori è così definita:

$$E_k^i = Y - \sum_{j \neq k} d_j^i X_{rowj}^i$$

Tale problema viene risolto, similmente al MOD, andando a cercare il minimo locale:

$$d_k^{(i+1)} = E_k^i (X_k^i)^T (X_k^i (X_k^i)^T)^{-1} \tag{3.5}$$

Si verifica che, nel caso della VQ,

$$E_k^i (X_k^i)^T = Y (X_k^i)^T - \sum_{j \neq k} d_j^i X_j^i (X_k^i)^T \tag{3.6}$$

applicando poi 3.3 e tenendo presente che le righe di X sono, per costruzione, ortogonali tra loro ($\forall j \neq k \; X_j^T X_k = 0$), ottengo che 3.5 corrisponde a 3.1 e quindi posso concludere che il metodo della SGK è una buona generalizzazione sequenziale della K-means.

3.2 L'algoritmo della SGK

Vediamo ora un breve esempio di implementazione di questo nuovo metodo della SGK.

Per semplificare la nostra analisi, possiamo riprendere l'algoritmo della K-SVD presentato nel capitolo 2 andando a modificarne solo il tratto di codice che va ad aggiornare l'atomo d_k. Infatti la SGK lavora bene con qualsiasi tipo di *sparse coder* e quindi possiamo utilizzare come Greedy Method l'Orthogonal Matching Pursuit, (vedi 1.1.1) che abbiamo visto essere usato anche dalla K-SVD. Anche i parametri in entrata e in uscita saranno gli stessi presentati in 2.1.

```
function[D,X,err] = sgk(params, varargin)
```

Adattiamo il codice che esegue il dictionary update. Si noti che, nel caso della SGK, il vettore X viene aggiornato *solo* dallo sparse coder, in questo caso la OMP, e non dalla fase di dictionary update, come accadeva nella K-SVD:

```
p = randperm(dictsize);
fork = 1:dictsize
 D(:,p(k)) = optimize_atom(data,D,p(k),X);
end
```

Vediamo quindi come adattare la funzione *optimize_ atom* a svolgere quanto detto in 3.5:

```
functionatom = optimize_atom(Y,D,k,X)

d_k = D(:,k);
x_k = X(k,:);

Err = Y - D*X + d_k*x_k;

atom = (Err*x_k')*inv(x_k*x_k');

end
```

3.3 Costi computazionali a confronto

In questa sezione andremo ad esaminare i costi computazionali [2] dei vari algoritmi presentati, ponendo la nostra attenzione alla fase di dictionary update[8]. Riprendiamo il problema introdotto nel Capitolo 2 dove

$$Y = DX$$

con $D \in \mathbb{R}^{n\text{x}K}$, $Y \in \mathbb{R}^{n\text{x}N}$ e $X \in \mathbb{R}^{K\text{x}N}$ e sia t l'indice di sparsità di X. Per computare 2.2 utilizzo $2n(t-1)\omega_k$ FLOP poichè le colonne di X possiedono $t-1$ entrate non nulle che verranno moltiplicate per $d_{j\neq k}$. Applicare poi la SVD a E_k^R costa $2\omega_k n^2 + 11n^3$ FLOP e altri ω_k FLOP per computare $\tilde{x_k}$. Il totale quindi di FLOP utilizzato per aggiornare un atomo in D è di

$$2n(t-1)\omega_k + 2n^2\omega_k + 11n^3 + \omega_k \tag{3.7}$$

Il costo computazionale della fase di aggiornamento del dizionario nell'algoritmo della K-SVD sarà quindi la somma di 3.7 per ogni K atomo, ovvero, dato $\sum_k \omega_k = Nt$:

$$\mathcal{T}_{K-SVD} = \sum_{k=1}^{K}(2nt-1)\omega_k + 2n^2\omega_k + 11n^3 + \omega_k =$$

$$\sum_{k=1}^{K}(\omega_k)(2n(t-1)+2n^2+1) + \sum_{k=1}^{K}11n^3 =$$

$$(Nt)(2nt-2n+2n^2+1) + 11n^3K$$

così da ottenere

$$\mathcal{T}_{K-SVD} = 2nt^2N + 2tn^2N + 11n^3K + tN - 2ntN \text{ FLOP} \tag{3.8}$$

Per quanto riguarda il Metodo delle Direzioni Ottimali l'algoritmo utilizza [8]:

$$\mathcal{T}_{MOD} = 2ntN + 2t^2N + 2nK^2 + \tfrac{K^3}{3} - nK - K^2 \text{ FLOP} \tag{3.9}$$

e per la SGK [8]:

$$\mathcal{T}_{SGK} = 2nt^2N + 2tN - K \text{ FLOP} \tag{3.10}$$

Da un primo confronto tra 3.8, 3.9 e 3.10 si potrebbe dedurre che il MOD abbia il minor costo computazionale in termini di FLOP, poiché è l'unico caso in cui figurano solo termini del terz'ordine.

Facciamo ora un'analisi più dettagliata in termini di K, ovvero tenendo in considerazione il numero di atomi del dizionario (e quindi ponendo l'attenzione alla proprietà di sequenzialità del metodo che abbiamo detto essere più efficace

[2] La funzione che rappresenta il tempo di esecuzione di un algoritmo in termini di FLOP, ovvero di Floating Point Operations [3]

e accurata) e riscriviamo 3.8, 3.9 e 3.10 in termini di $O(K)$ [3]. Consideriamo il caso in cui la matrice D sia una matrice rettangolare, ma senza incorrere in $n \ll K$, ovvero si può scrivere $n = O(K)$, che il numero di segnali-esempio N sia maggiore di K in termini di una costante $a \geq 0$, $N = O(K^{1+a})$ (condizione standard per favorire una migliore rappresentazione dei segnali y_i) e che l'indice di sparsità non sia troppo elevato (anche questa è una condizione standard che viene applicata ad ogni problema di Dictionary Training per far si che gli algoritmi non incorrano in errore, vedi 2.3.1), $t = O(K^b)$ [4].

Applicando quanto sopra riportato, ottengo i seguenti costi computazionali:

$$\mathcal{T}_{K-SVD} \sim O(K^4)$$

$$\mathcal{T}_{MOD} \sim O(K^3)$$

$$\mathcal{T}_{SGK} \sim O(K^{2+2b+a})$$

Si può concludere che, supponendo a, b abbastanza piccoli ($b < \frac{1-a}{2}$), e quindi considerando un problema di Dictionary Training che si potrebbe definire "favorevole", il metodo della SGK risulta quello con costo computazionale minore, poichè è una sintesi tra la sequenzialità della K-SVD e, allo stesso tempo, è una generalizzazione della K-means come il MOD.

3.4 Algoritmi a confronto

Vediamo in questo ultimo capitolo alcuni brevi esperimenti che mettono a confronto i metodi della K-SVD e della SGK.

Andando ad implementare in Matlab un codice come questo:

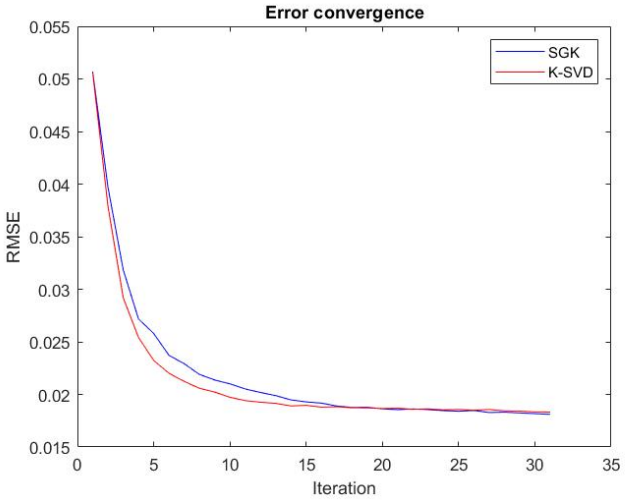
```
function ksvd_sgk

n = 64;
K= 256;
N = 1500;
t = 3;
snr = 20;

D=odctdict(n,K);
X = zeros(K,N);
for i = 1:N
  p = randperm(K);
  X(p(1:t),i) = randn(t,1);
end

Y = D*X;
```

[3] $f(x) = O(g(x))$ per $x \longrightarrow x_0 \iff \lim_{x \to x_0} \frac{f(x)}{g(x)} = \ell \in \mathbb{R}$

[4] dove $b \geq 0$ indica la sparsità del problema rispetto a K

```
17
18 Y = normcols(Y) + 10^(-snr/20)*normcols(randn(n,N));
19
20 params.data = Y;
21 params.dictsize = K;
22 params.iternum = 30;
23 params.Tdata = t;
24
25 [D_sgk,X_sgk,err_sgk] = sgk(params);
26
27 [D_ksvd,X_ksvd,err_ksvd] = ksvd(params);
28
29 figure(1); clf, plot(err_sgk, 'b-'); title('Error
       convergence');
30 xlabel('Iteration'); ylabel('RMSE');
31 hold on, plot(err_ksvd, 'r-'); legend('SGK','K-SVD ');
```

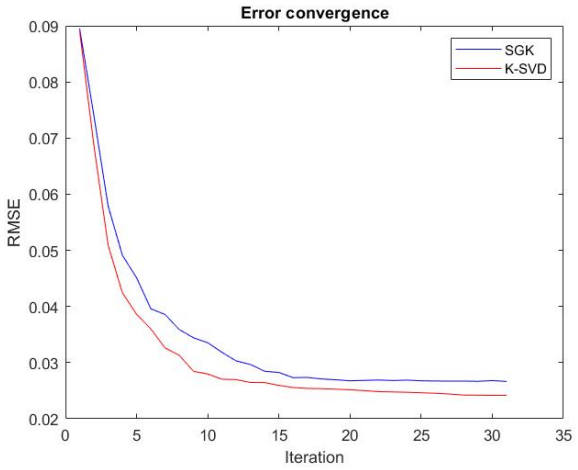
otteniamo i seguenti risultati:



Vediamo ora un altro esperimento tenendo conto delle ipotesi sui dati introdotte in 3.3 e utilizzando per semplicità una costante di proporzionalità pari a 1:

```
1 functionksvd_sgk
2
3 n = 20;%n=O(K)
4 K = 50;
5 a = 0.5;
6 b = 0.2;%b<(1-a)/2;
```

```
N = round(K^(1+a));%N=O(K^(1+a));
t = round(K^b);%t=O(K^b)
snr = 20;

D=odctdict(n,K);
X = zeros(K,N);
fori = 1:N
  p = randperm(K);
  X(p(1:t),i) = randn(t,1);
end

Y = D*X;

Y = normcols(Y) + 10^(-snr/20)*normcols(randn(n,N));

params.data = Y;
params.dictsize = K;
params.iternum = 30;
params.Tdata = t;

[D_sgk,X_sgk,err_sgk] = sgk(params);

[D_ksvd,X_ksvd,err_ksvd] = ksvd(params);

figure(1); clf, plot(err_sgk, 'b-'); title('Error
    convergence');
xlabel('Iteration'); ylabel('RMSE');
hold on, plot(err_ksvd, 'r-'); legend('SGK','K-SVD ');
```



Anche in questo caso si può notare che i due algoritmi mantengono lo stesso andamento in termini di errore quadratico medio su numero di iterazioni. Tenendo presente la scelta di $K = 50$, $a = 0.5$ e $b = 0.2$ e quanto già introdotto in 3.3, si può quindi fare un'analisi più dettagliata sulla differenza tra i due algoritmi:

$$\mathcal{T}_{K-SVD} \sim O(K^4) = O(50^4)$$

$$\mathcal{T}_{SGK} \sim O(K^{2+2b+a}) = O(50^{2+2*0.2+0.5}) = O(50^{\frac{29}{10}})$$

andando a concludere che, a parità di comportamento in termini di iterazioni, l'algoritmo della SGK risulta, come già anticipato, quello con costo computazionale inferiore, e quindi preferibile, in situazioni dove il valore di K è molto alto:

$$\frac{\mathcal{T}_{SGK}}{\mathcal{T}_{K-SVD}} = O(50^{\frac{29}{10}-4}) = O(50^{-\frac{11}{10}}) < 1$$

e quindi si può scrivere $\mathcal{T}_{SGK} < \mathcal{T}_{K-SVD}$

Facciamo ora una riflessione in termini di "FLoating-point OPerations per Second" (FLOPS[5]), e utilizziamo come modello un PC "normale", un PC che utilizziamo tutti i giorni a casa o al lavoro per intenderci, il quale ha come capacità all'incirca 10 GFLOPS, ovvero $\mathcal{C} = 10^{10}$ FLOPS.

$$\frac{\mathcal{T}_{K-SVD}}{\mathcal{C}} \sim \frac{50^4}{10^{10}} \sim 10^{-6}s$$

$$\frac{\mathcal{T}_{SGK}}{\mathcal{C}} \sim \frac{50^{\frac{29}{10}}}{10^{10}} \sim 10^{-7}s$$

Possiamo quindi concludere che, considerando $K = 50$, $a = 0.5$ e $b = 0.2$ e applicando le condizioni di 3.3, l'algoritmo della K-SVD implementato su un computer normalissimo impiega circa $10^{-6}s$ per raggiungere il risultato, mentre l'algoritmo della SGK ne impiega $10^{-7}s$. Questi tempi risultano pressochè irrisori e quindi l'utilizzo di un algoritmo anzichè l'altro non è una scelta determinante in termini di costo computazionale. Se però si vuole analizzare un problema più ampio la situazione cambia: considerando $K = 10^3$, $a = 0.5$ e $b = 0.25$ ottengo:

$$\frac{\mathcal{T}_{K-SVD}}{\mathcal{C}} \sim \frac{10^{12}}{10^{10}} = 10^2 s \sim 1.7'$$

$$\frac{\mathcal{T}_{SGK}}{\mathcal{C}} \sim \frac{10^9}{10^{10}} = 10^{-1}s$$

Si nota subito la differenza: l'algoritmo della K-SVD impiega più di un minuto e mezzo a ottenere il risultato, mente l'algoritmo della SGK impiega 0.1 secondi. Il contrasto è assai rilevante in questo caso (considerando anche il fatto

[5] Attenzione, FLOP $\neq$ FLOPS. Il primo rappresenta il numero di Floating Point OPerations che un algoritmo compie *in totale* per raggiungere il risultato, mentre il secondo rappresenta il numero di Floating Point OPerations *al Secondo* che l'algoritmo compie. Si può quindi scrivere, in termini di unità di misura, che [FLOPS] = $[\frac{\text{FLOP}}{\text{secondi}}]$

che questo genere di algoritmi sono molto spesso inseriti in problemi molto più ampi dove pochi secondi possono fare la differenza tra un codice di successo e un codice che fallisce nel suo intento) e il nuovo algoritmo della SGK non solo è preferibile ma consigliato.

Chiudiamo la nostra dissertazione con un ultimo esempio ispirato da [9], nel quale andiamo a confrontare il tempo effettivo *in secondi* che i due algoritmi impiegano per arrivare ad un risultato. Implementiamo di nuovo un algoritmo di confronto, questa volta andando ad inizializzare i parametri separatamente così da essere sicuri che nei vari esempi venga trattato lo stesso problema $Y = DX$. Consideriamo l'*error-based minimization method* e scriviamo:

$$\epsilon^2 = n(1.15^2)\sigma^2$$

Implementiamo una nuova funzione che ci aiuterà ad analizzare i tempi di risoluzione di uno stesso problema con σ diverse:

```
function tempo(sigma,n, X, K)

epsilon = n*(1.15^2)*sigma^2;

params.data = X;
params.dictsize = K;
params.iternum = 30;
params.Edata = sqrt(epsilon);

tic;
[D_sgk,X_sgk,err_sgk] = sgk(params);
toc;

tic;
[D_ksvd,X_ksvd,err_ksvd] = ksvd(params);
toc;
end
```

Riprendiamo quindi in analisi i dati dell'esperimento precedente e utilizzando la nuova funzione *tempo* andiamo a calcolare lo svolgimento in secondi dei due algoritmi con $\sigma = 5, 10, 15$:

```
n = 20;%n=O(K)
K = 50;
a = 0.5;
b = 0.2;%b<(1-a)/2;
N = round(K^(1+a));%N=O(K^(1+a));
t = round(K^b);%t=O(K^b)
snr = 20;

D=odctdict(n,K);
```

```
X = zeros(K,N);
fori = 1:N
  p = randperm(K);
  X(p(1:t),i) = randn(t,1);
end

Y = D*X;

Y = normcols(Y) + 10^(-snr/20)*normcols(randn(n,N));

tempo(5,n,X,K);
tempo(10,n,X,K);
tempo(15,n,X,K);
```

Riassumiamo nella seguente tabella i risultati ottenuti:

Algoritmo	$\sigma = 5$	$\sigma = 15$	$\sigma = 20$
K-SVD	0.665 s	0.527 s	0.508 s
SGK	0.228 s	0.171 s	0.204 s

Da questi brevi esperimenti pratici abbiamo quindi visto come, sia l'algoritmo della K-SVD, sia l'algoritmo della SGK, siano entrambi efficaci nell'ambito del Dictionary Learning: tutti e due raggiungono l'obiettivo desiderato con una veloce decadenza dell'errore e molto simile nell'andamento. In termini di costo computazionale e conseguente tempo di esecuzione, vediamo però che risulta preferibile l'utilizzo del secondo metodo, con tempi di esecuzione nettamente migliori a parità di performance.

Bibliografia

[1]M. Aharon, M. Elad, and A. Bruckstein. K-svd: An algorithm for designing overcomplete dictionaries for sparse representation. *IEEE Transactions on Signal Processing*, 54(11):4311–4322, 2006.

[2]M. Aharon, M. Elad, and A.M. Bruckstein. *The K-SVD: An Algorithm for Designing of Overcomplete Dictionaries for Sparse Representation*, volume 54. the IEEE Trans. on Signal Processing, 2006.

[3]D. Bhuvana Suganthi, M. Shivaramaiah, A. Punitha, M K Vidhyalakshmi, and S. Thaiyalnayaki. Design of 64-bit floating-point arithmetic and logical complex operation for high-speed processing. In *2023 International Conference on Intelligent and Innovative Technologies in Computing, Electrical and Electronics (IITCEE)*, pages 928–931, 2023.

[4]M. Elad. *Sparse and Redundant Representations: From Theory to Applications in Signal and Image Processing*. Springer Science+Business Media, 2010.

[5]K. Engan, S.O. Aase, and J. Hakon Husoy. Method of optimal directions for frame design. In *1999 IEEE International Conference on Acoustics, Speech, and Signal Processing. Proceedings. ICASSP99 (Cat. No.99CH36258)*, volume 5, pages 2443–2446 vol.5, 1999.

[6]Shih-Chung Hsu, I-Cheng Chang, and Chung-Lin Huang. Object verification in two views using sparse representation. In *2016 23rd International Conference on Pattern Recognition (ICPR)*, pages 504–509, 2016.

[7]Mostafa Sadeghi, Massoud Babaie-Zadeh, and Christian Jutten. Dictionary learning for sparse representation: A novel approach. *IEEE Signal Processing Letters*, 20(12):1195–1198, 2013.

[8]Sujit Kumar Sahoo and Anamitra Makur. Dictionary training for sparse representation as generalization of k-means clustering. *IEEE Signal Processing Letters*, 20(6):587–590, 2013.

[9]Sujit Kumar Sahoo and Anamitra Makur. Replacing k-svd with sgk: Dictionary training for sparse representation of images. In *2015 IEEE Inter-*

national Conference on Digital Signal Processing (DSP), pages 614–617, 2015.

[10]Liu Xia, Luo Wenhui, and Su Yixin. Face recognition algorithm based on improved kernel sparse representation. In *2019 34rd Youth Academic Annual Conference of Chinese Association of Automation (YAC)*, pages 654–659, 2019.

[11]Hufei Zhu, Ganghua Yang, and Wen Chen. Efficient implementations of orthogonal matching pursuit based on inverse cholesky factorization. In *2013 IEEE 78th Vehicular Technology Conference (VTC Fall)*, pages 1–5, 2013.