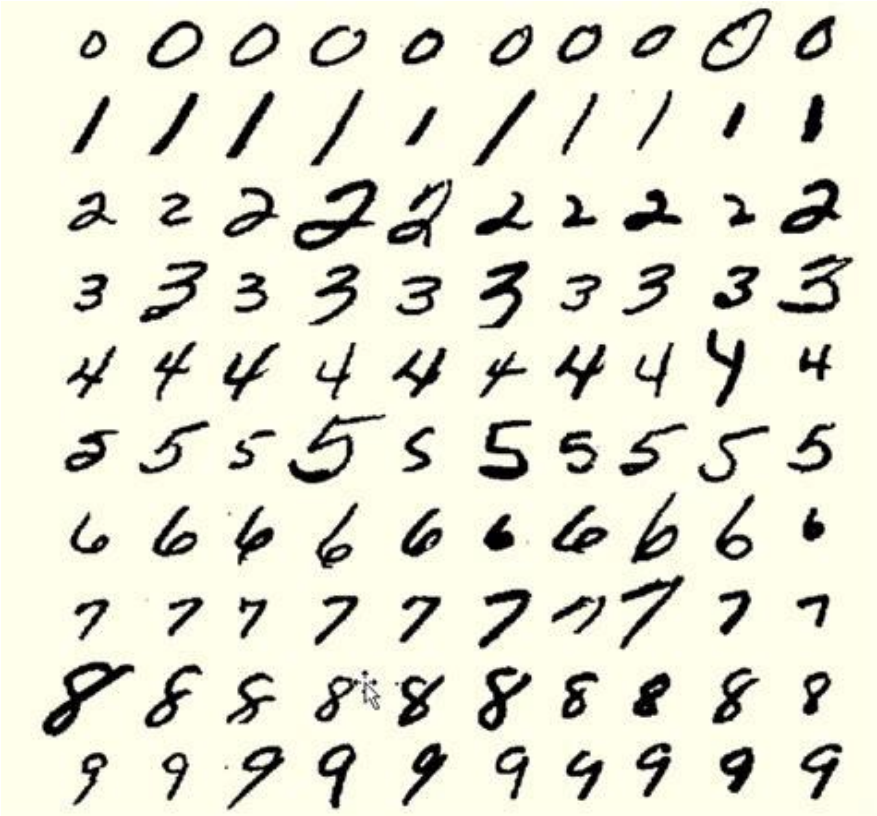
Mathilde DUARTE DOS SANTOS
Lucie MICHELET
Chloé PEDONI
Elsa PETILLON

IPSA
ÉCOLE D'INGÉNIEURS
DE L'AIR, DE L'ESPACE ET DE LA MOBILITÉ DURABLE

Bureau d'étude Ma313 : Variations autour de la reconnaissance de chiffres manuscrits.



M. COUFFIGNAL – Mercredi 15 décembre 2021

Table des matières

/!\ ATTENTION : Nos appareils n'étant pas assez puissants nous avons pris les 6000 premières valeurs des bases de donées *train_data* et *test_data* afin d'effectuer notre étude.

Introduction

Le **machine learning**, en français apprentissage automatique, est au cœur de la science des données, et sert aujourd'hui, à résoudre de nombreux problèmes dans différents secteurs scientifiques. Cet **outil informatique est devenu incontournable** et ses « applications sont nombreuses et variées, pouvant aller des moteurs de recherche et de la reconnaissance de caractères à l'analyse des réseaux sociaux, etc ».[1]

Dans le cadre de ce bureau d'étude de mathématiques, nous allons utiliser cet outil informatique afin d'**implémenter une méthode reconnaissant des chiffres manuscrits** qui se trouvent dans une base de données. Pour cela, on utilisera des **outils mathématiques** tels que la décomposition de Cholesky ou encore l'analyse procustéenne pour **déterminer le meilleur taux de reconnaissance** possible. Le tout sera codé sous Python.

Dans une première partie, nous essayerons de répondre au problème en utilisant la **décomposition de Cholesky**. Ensuite nous aborderons le problème cette fois-ci avec une **approche procustéenne**. Enfin, le but de la troisième et dernière partie est de trouver un **algorithme dit de prédiction**, qui mélangerait les deux méthodes vues précédamment, afin de trouver **une approche plus précise**.

[1] Extrait du livre de Chloe-Agathe Azencott, *Introduction au Machine Learning*

Partie 1 : Inverser le non-inversible.

Dans cette première partie, nous verrons une **méthode de reconnaissance de chiffre manuscrits avec la factorisation de Cholesky**.

On définit les deux ensembles E_1 et E_2 tels que $E_1 = \{u_i \in \mathbb{R}^n |_i \epsilon [1; p]\}$ et $E_2 = \{v_i \in \mathbb{R}^n |_i \epsilon [1; q]\}$ avec p et q des entiers non nuls.

On cherche une fontcion affine de la forme $\boldsymbol{f(x) = w^T x + b}$ de $\mathbb{R}^n$ dans $\mathbb{R}$ avec $w \in \mathbb{R}^n \ et \ b \in \mathbb{R}$. On pose $f(u_i) = 1 \ \forall i \in [1; p]$ et $f(v_i) = -1 \ \forall i \in [1; q]$

Cette fonction devra minimiser la quantité $\sum_{i=1}^{p} (\boldsymbol{f(u_i) - 1})^2 + \sum_{j=1}^{q} \big(\boldsymbol{f(v_j) + 1}\big)^2$ qui quantifie l'erreur de prédiction de f sur les données d'entraînement.

On peut écrire $\sum_{i=1}^{p} (\boldsymbol{f(u_i) - 1})^2 + \sum_{j=1}^{q} \big(\boldsymbol{f(v_j) + 1}\big)^2 = \left\| \begin{pmatrix} \boldsymbol{f(u_1) - 1} \\ \vdots \\ \boldsymbol{f(u_p) - 1} \\ \boldsymbol{f(v_1) + 1} \\ \vdots \\ \boldsymbol{f(v_q) + 1} \end{pmatrix} \right\|_2^2$

$= \left\| \begin{pmatrix} \boldsymbol{w^T u_1 + b - 1} \\ \vdots \\ \boldsymbol{w^T u_p + b - 1} \\ \boldsymbol{w^T v_1 + b + 1} \\ \vdots \\ \boldsymbol{w^T v_q + b + 1} \end{pmatrix} \right\|_2^2 = \left\| \begin{pmatrix} \boldsymbol{u_1^T} & \boldsymbol{1} \\ \vdots & \vdots \\ \boldsymbol{u_p^T} & \boldsymbol{1} \\ \boldsymbol{v_1^T} & \boldsymbol{1} \\ \vdots & \vdots \\ \boldsymbol{v_q^T} & \boldsymbol{1} \end{pmatrix} \begin{pmatrix} \boldsymbol{w} \\ \boldsymbol{b} \end{pmatrix} - \begin{pmatrix} \boldsymbol{1} \\ \vdots \\ \boldsymbol{1} \\ \boldsymbol{-1} \\ \vdots \\ \boldsymbol{-1} \end{pmatrix} \right\|_2^2 = \boxed{\|\boldsymbol{Ax - y}\|_2^2}$

On peut donc identifier A = $\begin{pmatrix} \boldsymbol{u_1^T} & \boldsymbol{1} \\ \vdots & \vdots \\ \boldsymbol{u_p^T} & \boldsymbol{1} \\ \boldsymbol{v_1^T} & \boldsymbol{1} \\ \vdots & \vdots \\ \boldsymbol{v_q^T} & \boldsymbol{1} \end{pmatrix}$, **x** = $\begin{pmatrix} \boldsymbol{w} \\ \boldsymbol{b} \end{pmatrix}$ **et y** = $\begin{pmatrix} \boldsymbol{1} \\ \vdots \\ \boldsymbol{1} \\ \boldsymbol{-1} \\ \vdots \\ \boldsymbol{-1} \end{pmatrix}$

On note $\phi(x) = \|Ax - y\|_2^2$, les points critiques se trouvent en minimisant ϕ. On dérive ϕ par rapport à x avec les conditions du premier ordre. On obtient $2A^T Ax - 2A^T y = 0$. On retrouve alors l'équation normale telle que $\boxed{A^T Ax = A^T y.}$

Sous Python maintenant, on utilise **l'algorithme d'itération QR** afin de déterminer les valeurs propres de la matrice $A^T A$. On peut ainsi approximer le rang de la matrice A en utilisant le cas de l'apprentissage de reconnaissance du chiffre 0. On obtient alors un rang **r = 656**. Cette

matrice ne peut d'ailleurs pas être une matrice inversible car certaines de ces valeurs propres sont nulles.

On réitère ce calcul d'une autre façon, en calculant, cette fois-ci, **les valeurs singulières** de la matrice A^TA. On obtient un rang **r = 785**.

On remarque que le rang obtenu avec la méthode SVD (calcul des valeurs singulières) est plus important que le rang obtenu avec la méthode QR. On peut donc conclure que **la méthode SVD est plus précise** dans les résultats obtenus. En effet, la **bibliothèque Numpy approxime** plus le calcul des valeurs propres de la matrice A, ce qui explique une erreur relative plus grande.

Nous avons également vu en cours qu'il est possible de résoudre ce problème à l'aide du pseudo-inverse de A tel que :

$$x^* \coloneqq A^+ b$$

Lors de cette étude, nous allons plutôt essayer de modifier la matrice A^TA enfin de la rendre inversible. On pose alors :

$$A_\varepsilon \coloneqq A^T A + \varepsilon I_{785}$$

où ε est un réel strictement positif et I_{785} est la matrice identité de taille 785 x 785.

De même que précédamment, en utilisant de nouveau le cas de **l'apprentissage de la reconnaissance du chiffre 0**, on calcule les **valeurs propres approchées de la matrice A_ε** en fonction de ε. A l'aide du calcul Python, on justifie que cette matrice **A^TA est définie positive**, c'est-à-dire que cette dernière **soit inversible**. On créé alors une fonction qui détermine pour chaque valeur d'epsilon ε, si la matrice A^TA est définie positive.

```python
def defsympos(A,k):
    e=1
    condition = True
    while condition == True and e<=k :
        M = A.T@A + e*np.eye(785)
        vpM = iterationQR(M)
        for i in range(785):
            if vpM[i] > 0 and (M.T == M).all() :
                condition = True
            else :
                condition = False
                break
        e +=1
    return condition
```

Fonction Python qui définit si la matrice A^TA est définie positive pour n'importe quelle valeur d'epsilon

Le code nous renvoie donc ***True*** pour n'importe quelle valeur de ε et pour chaque matrice A de chaque chiffre. **Elles sont donc bien définies positives**.

A l'aide de la décomposition Cholesky, sous Python, nous définissons une fonction resChol qui prend en entrée le **nombre de détections, un nombre entier appartenant à $[\![0, 9]\!]$**, qui correspond au chiffre à détecter et **epsilon ε**, un nombre réel. Cette solution :

- Calculera la solution :

$$sol_\varepsilon \coloneqq A^{-1}{}_\varepsilon A^T y$$

  en utilisant la décomposition de Cholesky de A_ε.

- Renverra le taux de réussite de reconnaissance du chiffre que l'on souhaite détecter défini par :

$$T_r := \frac{N_{vp} + N_{vn}}{6000}$$

ainsi que la matrice de confusion M_{conf} associée sur les 6 000 premières données test du fichier test_data. *(Nous avons pris uniquement les 6 000 premières données du fichier du à un problème lié à un manque de mémoire de nos ordinateurs.)*

$$M_{conf} := \begin{pmatrix} N_{vp} & N_{fn} \\ N_{fp} & N_{vn} \end{pmatrix}$$

où N_{vp} est le nombre de **vrais positifs**, N_{fn} le nombre de **faux négatifs**, N_{fp} le nombre de **faux positifs** et N_{vn} le nombre de **vrais négatifs**.

Lorsqu'on lance ainsi le code pour $\varepsilon = 1$, on a donc pour chaque chiffre à détecter entre 0 et 9, la matrice de confusion ainsi que son taux de réussite.

Matrice de confusion pour 0:
[[467 101]
[46 5385]]
Taux de réussite : 0.9754959159859977
Matrice de confusion pour 1:
[[639 47]
[94 5219]]
Taux de réussite : 0.9764960826804467
Matrice de confusion pour 2:
[[341 284]
[43 5331]]
Taux de réussite : 0.9454909151525254
Matrice de confusion pour 3:
[[333 262]
[61 5343]]
Taux de réussite : 0.9461576929488248
Matrice de confusion pour 4:
[[401 198]
[80 5320]]
Taux de réussite : 0.9536589431571929
Matrice de confusion pour 5:
[[205 343]
[41 5410]]
Taux de réussite : 0.9359893315552592
Matrice de confusion pour 6:
[[398 164]
[70 5367]]
Taux de réussite : 0.9609934989164861
Matrice de confusion pour 7:
[[386 222]
[60 5331]]
Taux de réussite : 0.9529921653608935
Matrice de confusion pour 8:
[[231 356]
[55 5357]]
Taux de réussite : 0.9314885814302384

Matrice de confusion pour 9:
[[346 275]
[109 5269]]
Taux de réussite : 0.9359893315552592

Prenons par exemple le chiffre 5. Pour la détection du chiffre 5, on obtient la matrice de confusion :

$$M_{conf(5)} = \begin{pmatrix} 205 & 343 \\ 41 & 5410 \end{pmatrix}$$

Le programme Python a donc détecté, parmi les 6000 premières valeurs, 205 vrais chiffres 5, 343 faux négatifs de 5, c'est-à-dire des chiffres 5 qui semblaient être des 5 mais finalement ne le sont pas, 41 faux positifs, des chiffres qui ne sont pas des 5 mais que l'ordinateur à reconnu comme des 5, et enfin 5410 vrais négatifs, soient des chiffres dont on est sur que ce ne sont pas des chiffres 5.
On obtient alors un taux de réussite de 94% pour la détection du chiffre 5. On peut déjà conclure dans un premier temps, pour $\varepsilon = 1$, **le taux de réussite de reconnaissance est plutôt réussi**. Cela s'explique par le fait que la décomposition de Cholesky résout le système.

Faisons maintenant **varier $\boldsymbol{\varepsilon}$ sur l'intervalle $[\mathbf{10^{-10}}, \mathbf{10^{9}}]$** afin de pouvoir déterminer le meilleur taux de réussite en fonction de ε.

Grâce à notre fonction *optimus_e* nous avons pu calculer le meilleur taux de réussite et son espilon associé, en découpant l'intervalle [10^{-10} ; 10^{9}] en 9 sous-intervalles pour être plus précis. Cette fonction sauvegarde le graphique du taux de réussite en fonction de espilon pour chaque sous-intervalle en PNG. Elle sauvegarde également les meilleures valeurs de epsilon, du taux et leur matrice de confusion associé dans un fichier texte nommé « Solutions » :

Solutions - Bloc-notes

Fichier Modifier Format Affichage Aide

```
==================================
Etude du chiffre 1
==================================
Intervale numéro 0
Taux de réussite max : 0.9764960826804467
espilon max : 1e-10
Matrice de confusion associée : [[ 639   47]
 [  94 5219]]
Intervale numéro 1
Taux de réussite max : 0.9764960826804467
espilon max : 1e-08
Matrice de confusion associée : [[ 639   47]
 [  94 5219]]
Intervale numéro 2
Taux de réussite max : 0.9764960826804467
espilon max : 1e-06
Matrice de confusion associée : [[ 639   47]
 [  94 5219]]
Intervale numéro 3
Taux de réussite max : 0.9764960826804467
espilon max : 0.0001
Matrice de confusion associée : [[ 639   47]
```

Screenshot du fichier Solutions

Enfin, *optimus_e* enregistre **les solutions des taux maximum parmis ces 9 sous-intervalles**, qu'il renvoie sous forme d'un array « *solution i.npy* » où i est le nombre de détections.
Pour le chiffre 0, nous avons trouvé un epsilon de 333 265.
Tous ces fichiers vous sont donnés avec les codes Python associés.

Chiffre de détection	Taux de reussite	Epsilon	Matrice de confusion
0	0.9868311385230871	333 265	[[533 35] [44 5387]]
1	0.9841640273378897	164 102 040	[[615 71] [24 5289]]
2	0.9518253042173695	656 530	[[374 251] [38 5336]]
3	0.9521586931155193	21 387 755	[[358 237] [50 5354]]
4	0.96216036006001	636 326	[[445 154] [73 5327]]
5	0.9438239706617769	50 408	[[246 302] [35 5416]]
6	0.9693282213702283	41 775 510	[[429 133] [51 5386]]
7	0.9648274712452075	62 163 265	[[422 186] [25 5366]]
8	0.9316552758793132	0.01	[[232 355] [55 5357]]
9	0.936156026004334	3	[[347 274] [109 5269]]

Théorisons à présent **pourquoi le remplacement de la matrice A^TA par $A^TA + \varepsilon I_n$ fonctionne**. Nous avons vu précédamment que la solution de :

$$\min_{x \in \mathbb{R}^{n+1}} \|Ax - y\|_2^2$$

revient à rechercher les points critiques de la fonction Φ définie par :

$$\phi(x) := \|Ax - y\|_2^2 = x^T A^T Ax - 2x^T A^T y$$

C'est-à-dire que **les points critiques de Φ vérifie l'équation normale** :

$$A^T Ax = A^T y$$

En remplaçant $A^T Ax \; par \; A^T A + \varepsilon I_n$, nous obtenons :

$$(A^T A + \varepsilon I_n)x = A^T y$$

$$\Leftrightarrow A^T Ax + \varepsilon I_n = A^T y$$

$$\Leftrightarrow A^T Ax \; - \; A^T y + \varepsilon I_n x = 0$$

Or, comme vu précédamment, on sait que les points critiques de la fonction Φ définie par $\Phi(x) = \|Ax - y\|_2^2$ vérifie l'équation normale :

$$\Leftrightarrow A^T Ax - A^T y + \varepsilon I_n x = 0$$

Cela revient donc à rechercher les points critiques de la fonction :

$$\Phi(x) = \|Ax - y\|_2^2 + \varepsilon \|x\|_2^2$$

On remarque également que epsilon à une influence sur le taux de réussite de la fonction. En effet, à l'aide de la fonction optimus on remarque que **plus epsilon est grand plus le taux de réussite est bon**.

Pour terminer cette analyse, nous avons créé la fonction globale de reconnaissance en fonction d'une image vectorisée de la base de donnée de test et la matrice SOL composée des vecteurs solutions de chaque chiffres pour un epsilon optimisé. On définit SOL par la matrice par bloc suivant :

$$SOL = \begin{pmatrix} \omega_0 & \dots & \omega_9 \\ b_0 & \dots & b_9 \end{pmatrix}$$

Cette fonction permet de calculer $f(x) = \omega_i x + b_i$ pour chaque chiffre $i \in [0,9]$, elle renvoie **une liste comprenant les valeurs de f** pour chaque chiffre ainsi que **le résultat correspondant à l'indice du maximum** de la liste.

Cette méthode permet de reconnaître un chiffre manuscrit de *test_data* en utilisant les coefficients ω_i, b_i calculé à l'aide de la base de données *train_data*.

On définit enfin une fonction permettant de calculer le taux de réussite de la fonction globale de Cholesky sur la base de données de test.

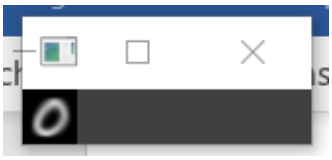
En prenant les 6000 premières images du *test_data* **notre fonction Cholesky à un taux de réussite de 81%**.

Partie 2 : L'approche procustéenne

Dans cette partie, nous allons aborder le problème de la reconnaissance des chiffres manuscrits avec une **analyse procustéenne**. **L'idée est de comparer l'image** « vectorisée » d'un chiffre manuscrit inconnu avec les images « moyennes » des chiffres de 0 à 9 sur les données d'entraînement.

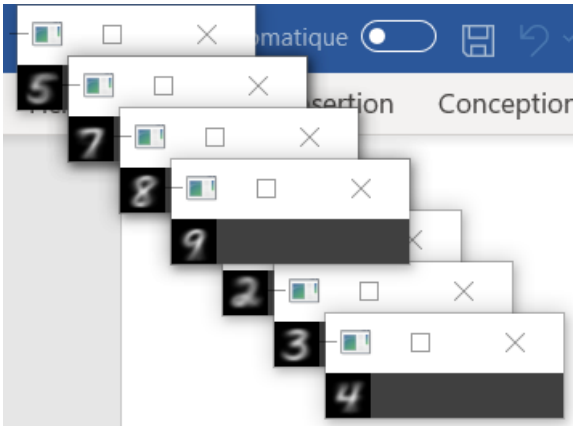
Nous allons tout d'abord nous attarder sur la méthode pour calculer un chiffre moyen (ici 0) afin de généraliser la méthode par la suite à tous les chiffres.

Pour cela, sous Python, en utilisant la commande Numpy *np.mean()*, on crée un vecteur u de taille (1,784) que l'on appelle zeromoyen et qui représente la moyenne de tous les chiffres 0 présent dans la base de données MNIST. On convertit ensuite ce vecteur u en image de 28 pixels par 28 pixels, on obtient ainsi l'affichage de l'image du 0 moyen.



Affichage du zeromoyen

En généralisant ensuite le problème aux autres chiffres manuscrits présents dans la base de données MNIST, on définit alors une fonction chiffremoy(train_data) permettant de créer une matrice (10 x 784) où chaque lignes correspondent à l'image moyenne véctorisée d'un chiffre de 0 à 9. De la même manière que pour l'affichage du 0 moyen, le code Python va parcourir tous les chiffres de 0 à 9, puis sélectionner chaque ligne de la matrice en les convertissant en une matrice 28 par 28, puis les converti en « uint8 ». On obtient alors l'affichage suivant :



Affichage en image uint8 des chiffres moyens de 0 à 9

A l'image de Procuste, l'analyse procustéenne consiste à « faire prendre des données une forme désirée ». Elle est utilisée pour déformer un objet afin de le rendre autant que possible semblable à une référence. Ici, par exmple, l'analyse procustéenne est utilsée pour reconnaître les chiffres manuscrits et les identifier le plus possible à des formes de chiffres connus.

De ce fait, il existe alors une erreur de transformation et un taux de reconnaissance de l'algorithme en prendre en compte.

On considère alors A, B $\epsilon\, M_{mn}(R)$ deux matrices. L'analyse procustéenne linéaire consiste alors à recherche Φ une **transformation affine** qui s'écrit matriciellement :

$$\Phi(A) = \lambda XA + tu$$

Où les inconnues sont :

- $\lambda\, \epsilon$ R le rapport d'homothétie,
- X $\epsilon\, O_m$, une transformation orthogonale,
- t $\epsilon\, R^m$ est un vecteur de tranlation (vecteur colonne) et u le vecteur ligne de R^n

La fonction est telle que Φ minimise la quantité suivante :

$$\min_{\Phi \text{ transformation affine}} \|B - \Phi(A)\|_F^2$$

On rappelle également que la solution de ce problème est donnée par :

$$\boxed{X = V_G U_G^T} \qquad \boxed{t = b_G - \lambda X a_G} \quad \text{et} \quad \boxed{\lambda = \frac{trace\,(\Sigma_G)}{\|A_G\|_F^2}}$$

où U_G et V_G sont données par une décomposition en valeurs singulières (SVD) de ${A_G B^T}_G = U_G \sum_G {V^T}_G$ avec :

$$A_G = A - a_G u \qquad \text{et} \quad B_G = B - b_G u$$

et :

$$b_G := \frac{1}{n}\sum_{j=1}^{n} b_j \quad \text{et} \quad a_G := \frac{1}{n}\sum_{j=1}^{n} a_j$$

avec les a_i et b_i les vecteurs colonnes de A et B respectivement.

Nous écrivons alors sous Python une fonction Procuste(A,B) qui prend en entrée deux matrices de même taille et qui renvoie en sortie le tuple :

$$\left(\lambda, X, t, \|B - \Phi(A)\|_F^2\right)$$

Cette quantité : ${\|B - \Phi(A)\|_F}^2$ s'appelle l'erreur de transformation.

Cette fonction Procuste(A,B) est ensuite implémentée pour calculer une autre fonction appelée comparaison(x,CM) qui renvoie en sortie un tuple : **veccomp**, **resultat** où :

- **veccomp** est un array de taille 10 ayant pour composante j l'erreur de transformation de l'analayse procustéenne pour chaque chiffre,
- **resultat** est l'entier correspondant à l'indice de la composante minimale du vecteur **veccomp**. Cette valeur est le résultat de la prédiction sur le chiffre manuscrit correspondant à x par l'analyse procustéenne.

Cette fonction va alors chiffre par chiffre manuscrit essayer de reconnaître au maximum le chiffre manuscrit donné et va l'identifier à un chiffre « numérique connu ».

Ce qui nous amène enfin à créer une dernière fonction appellée TauxReconnaissanceProcuste permettant de calculer le taux de réussite de l'analyse procustéenne pour répondre au problème posé.

On obtient alors un taux de réussite de reconnaissance d'environ 79%.

Ce taux de réussite est inférieur à celui trouvé avec la méthode de Cholesky car l'analyse procustéenne compare des images avec des moyennes d'autres images déjà existentes, plus la base de données d'entrainement est grande plus nos chiffres moyens seront précis . Néanmoins, par soucis de capacité de mémoire de nos ordinateurs, il nous était impossible de travailler avec les 60 000 données de la base de données MNIST d'entrainement. Nous avons donc pris uniquement les 6000 premières valeurs afin de pouvoir travailler avec. La valeur moyenne du taux de reconnaissance pour chaque chiffre aurait donc été plus précise avec plus de valeurs et obtenir par la suite un meilleur taux de reconnaissance. On peut raisonnablement estimer que le taux de reconnaissance de Procuste avec un entrainement sur les 60 000 valeurs serait de 85%.

Partie 3 : Le mélange

Dans cette partie, nous allons nous servir des deux algorithmes dévelopés dans les parties précédentes afin de réaliser un **algorithme de prédiction** que l'on espère plus précis.

En reprenant les notations précédentes, on rédige une fonction Python de reconnaissance des chiffres manuscrits qui pour une donnée test sous la forme de :

```
x = train_data[i,1:].reshape((784,1))
```

renvoie le vecteur résultat de taille 10 défini de la façon suivante :

```
v = coeff*(1/comparaison(test_data[i,1:],CM)[0])
rep = fglobale(test_data[i,1:].reshape((784,1)),SOL).reshape(np.shape(v))
resultat = v+rep
```

où coeff est un réel strictement positif. La prédiction sera comme dans les parties précédentes l'indice de la composante maximale du vecteur resultat précédent. Le réel coeff est alors un réel flotant que l'on cherche à déterminer pour avoir un taux de réussite maximale. Cette fonction Reconnaissance utilise les méthodes de Cholesky et l'analyse de Procuste afin de reconnaître un chiffre manuscrit de la base de données test.

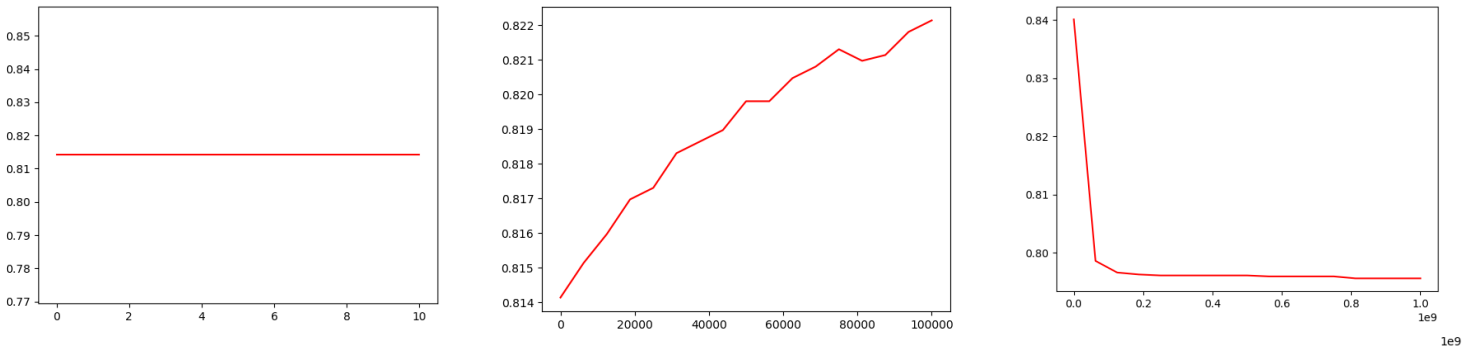
Cette fonction nous permet ensuite de définir une nouvelle fonction TauxGlobal qui prend en entrée comme valeur du coeff que l'on cherche à maximiser. Cette fonction retourne à la fin le nombre de chiffres manuscrits que la fonction Python a réussi à déterminer diviser par le nombre total de chiffre ce qui donne un nouveau taux de réussite.

Par la suite, en prenant un nombre coefficient quelconque, coeff = 240, on obtient un taux de reconnaissance des chiffres manuscrits de 83%. On peut déjà noter que pour un coeff aux alentours de 240, 250, le taux de reconnaissance est déjà assez élevé. En revanche, ici, nous avons pris un nombre coeff quelconque.

Cherchons alors maintenant à déterminer le meilleur coeff pour obtenir le meilleur taux de reconnaissance.

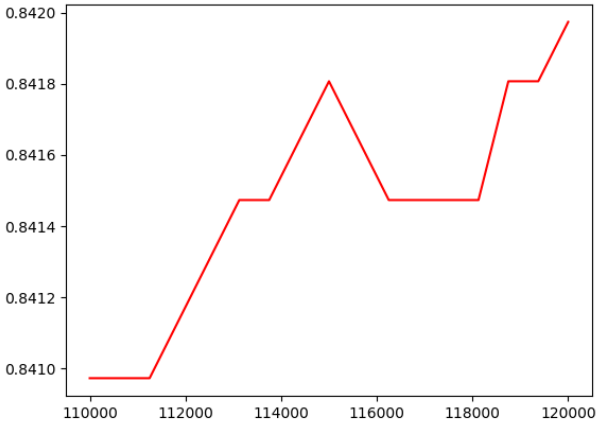
On définit alors un code Python, qui va parcourir toutes les valeurs de coeff possible sur un intervalle de $[10^{-9}, 10^{9}]$ scincé en plusieurs intervals. Il va ensuite calculer, pour chaque coefficient, le taux global de réussite. Ces valeurs vont s'ajouter alors dans une liste appelée Tauxlist. Enfin, on définit le coefficient maximum qui définit le meilleur taux de reconnaissance parmi cette liste. Empiriquement nous allons choisir un intervalle de plus en plus petit se rapprochant d'un taux de réussite maximum.

On commence par scindé l'intervalle en 3.



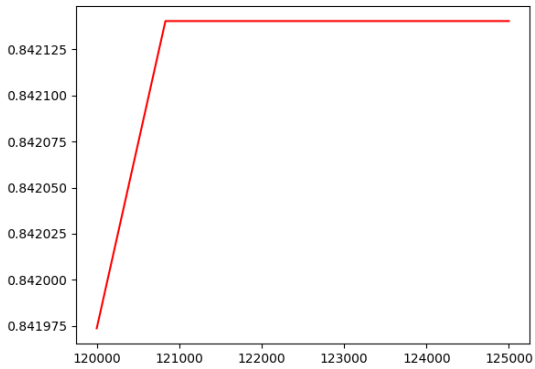
Taux de réussite de reconnaissance en fonction du coefficient de 3 intervalles respectivement de $[10^{-9}, 10]$, de $[10,10^5]$ et de $[10^5, 10^9]$

Nous avons ensuite précisé les intervalles jusqu'à obtenir le coeff le plus précis possible.



Taux de réussite de reconnaissance en fonction du coefficient de $[1.1 * 10^5, 1.2 * 10^5]$

On obtient enfin, avec un intervalle de $[1.2 * 10^5, 1.25 * 10^5]$, le graphique suivant :



On obtient alors un taux de reconnaissance maximum de 84% avec une valeur maximale de coeff de 120833,3.

Conclusion

En conclusion, au travers de ce BE, nous avons créé trois méthodes différentes pour répondre au problème. Dans la première partie, grâce à la décomposition de Cholesky, on obtient un taux de réussite de 81%. A la deuxième partie, de part l'analyse de Procuste, on obtient un taux de réussite de 79%. Cet écart de taux s'explique par le fait que la méthode de Procuste se réalise par comparaison sur des valeurs moyennes. Et plus il y a de valeurs dans la base de données, plus la valeur moyenne de comparaison est précise. Tandis que la décomposition de Cholesky résout le système demandé, ce qui fournit des valeurs plus précises.

Enfin, à la dernière partie, nous avons crée un algorithme de prédiction qui combine les deux méthodes vues aux deux premières parties, afin de rendre ce dernier beaucoup plus précis. Après recherche du coefficient maximal, on obtient un taux de réussite de 84% avec un coeff de 120 833,3.

De plus, on remarque également qu'uniquement les 6 000 premières valeurs de la base de données test_data ont été exploitées par nos programmes comme expliqué ci-dessus. Cela a donc influencé sur les résultats de taux de réussite obtenus tout au long du BE. Si nous avions pris les 60 000 valeurs de la base de données d'entraînement (test_data), les pourcentages de taux de réussite auraient été beaucoup plus elevés.

Bonus : Le machine learning pour les nuls : l'apprentissage avec le module sckit-learn

Il existe un module implémenté sur python permettant de faire du machine learning simplement : Sckit-learn. Après avoir travaillé avec celui-ci pour notre PIG nous avons décidé de l'essayer pour la reconnaissance de chiffres manuscrits. Pour cela il faut tout d'abord choisir le bon estimateur de la bibliothèque sklearn. Pour ce projet, il faut choisir un algoritme de classification, il en existe plusieurs tel que KNeighboor Classifier, Linear SVC ou encore SVM. Ici, nous allons choisir la méthode de classification Support Vector Machine permettant de classifier des données vectorielles. Plus précisement dans la methode SVM nous allons utiliser le module SVC (Support Vector Classifier).

Pour cela nous avons importer la base de données digits_data déjà présente dans le module sklearn permettant d'avoir une base de données de chiffres manuscrits dans le bon format. Il faut également initier notre modèle de classification avec le bon estimateur en écrivant model = svm.SVC(gamma), gamma étant le degré du noyau de la méthode. Ensuite à l'aide de la méthode model.fit(futures, target), il est possible d'entrainer notre modele sur les les chiffres manuscrits en présisants les futures (ici image d'un chiffre) et la target (le label de chaque chiffre de la base de donnée).

Il est possible d'évaluer notre model sur les données d'entrainement avec la méthode model.score(futures, target). Nous obtenons le taux de réussite suivant : 0.9988864142538976. On remarque que ce taux de réussite sur les données d'entrainement est bien meilleur que celui de Cholesky.

On a également la matrice de confusion suivante :

```
[[87  0  0  0  1  0  0  0  0  0]
 [ 0 88  1  0  0  0  0  0  1  1]
 [ 0  0 85  1  0  0  0  0  0  0]
 [ 0  0  0 79  0  3  0  4  5  0]
 [ 0  0  0  0 88  0  0  0  0  4]
 [ 0  0  0  0  0 88  1  0  0  2]
 [ 0  1  0  0  0  0 90  0  0  0]
 [ 0  0  0  0  0  1  0 88  0  0]
 [ 0  0  0  0  0  0  0  0 88  0]
 [ 0  0  0  1  0  1  0  0  0 90]]
```

Avec pour chaque lignes les prédictions pour chaque chiffres. Sur la diagonale il s'agit des vrais positifs.