

Apprentissage et Reconnaissance des Formes

[4I802]

Rapport du projet



RÉALISÉ PAR
BIZZOZZÉRO NICOLAS
&
ADOUM ROBERT

Table des matières

Table des matières	2	
1	Préambule : Régression linéaire, régression ridge et LASSO	3
1.1	Calculs préliminaires	3
1.1.1	Régularisation L2	3
1.1.2	Régularisation L1	3
1.2	Description du protocole	3
1.3	Analyse des résultats	4
2	LASSO et Inpainting	5
2.1	Introduction	5
2.1.1	Principe	5
2.1.2	Déroulement & implémentation	5
2.2	Résultats sur des images bruitées	6
2.3	Résultats sur des patchs manquants	7
2.4	De l'importance dans l'ordre de remplissage	8

1 Préambule : Régression linéaire, régression ridge et LASSO

1.1 Calculs préliminaires

Soient f_w la fonction de prédiction et \hat{y} l'ensemble des vrais labels de la base d'apprentissage.

1.1.1 Régularisation L2

On souhaite minimiser la fonction de coût suivante :

$$L_2(w) = \frac{1}{2N} \sum_{i=1}^N (\hat{y}_i - f_w(x_i))^2 + \alpha \|w\|_2^2$$

On va donc l'optimiser par descente de gradient, en utilisant le gradient suivant :

$$\frac{\partial L_2}{\partial w} = \frac{1}{N} \sum_{i=1}^N (\hat{y}_i - f_w(x_i))x_i + 2\alpha w$$

1.1.2 Régularisation L1

On souhaite minimiser la fonction de coût suivante :

$$L_1(w) = \frac{1}{2N} \sum_{i=1}^N (\hat{y}_i - f_w(x_i))^2 + \alpha \|w\|_1$$

On va donc l'optimiser par descente de gradient, en utilisant le gradient suivant :

$$\frac{\partial L_1}{\partial w} = \frac{1}{N} \sum_{i=1}^N (\hat{y}_i - f_w(x_i))x_i + \alpha \cdot sign(w)$$

1.2 Description du protocole

Le classifieur utilisé est, comme décrit dans l'énoncé, une simple **régression linéaire** adaptée à la classification binaire par la méthode du **plug-in**. Nous comparerons différentes fonctions de coût : **MSE**, **régularisation L₂** et **régularisation L₁**, ainsi que plusieurs valeurs du coefficient α . De plus, nous comparons aussi nos résultats avec les implémentations *LinearRegression* et *Lasso* de la bibliothèque **sklearn** et adaptées par la *méthode du plug-in*.

La base de données **USPS** étant déjà séparée en une base d'apprentissage et une base de test, nous pouvons facilement calculer le score obtenu sur cette même base de test. Nous proposons de comparer les résultats obtenus en classifiant **chaque classe contre chaque autre** dans un premier temps, puis dans du **1 contre tous** par la suite. Enfin, nous étudierons l'apparence du vecteur de poids obtenu par chaque méthode de classification en comptant le **nombre de poids nuls** ainsi que la **moyenne de la valeur absolue des poids**.

1.3 Analyse des résultats

Classifieur	Score moyen	Nombre de 0	moyenne de $ w $
Régression linéaire plug-in, MSE	0.5290	0	0.25045
Régression linéaire plug-in, $L_2, \alpha = 0.00$	0.4853	0	0.24862
Régression linéaire plug-in, $L_2, \alpha = 0.25$	0.5179	0	0.00285
Régression linéaire plug-in, $L_2, \alpha = 0.50$	0.5256	0	0.00308
Régression linéaire plug-in, $L_2, \alpha = 0.75$	0.5276	0	0.00265
Régression linéaire plug-in, $L_2, \alpha = 1.00$	0.5095	0	0.00258
Régression linéaire plug-in, $L_1, \alpha = 0.00$	0.5638	0	0.24974
Régression linéaire plug-in, $L_1, \alpha = 0.25$	0.5179	0	0.00387
Régression linéaire plug-in, $L_1, \alpha = 0.50$	0.5016	0	0.00473
Régression linéaire plug-in, $L_1, \alpha = 0.75$	0.5225	0	0.00573
Régression linéaire plug-in, $L_1, \alpha = 1.00$	0.5229	0	0.00642
sklearn.LinearRegression plug-in, $\alpha = 1.00$	0.9732	17	99305970.4
sklearn.Lasso plug-in, $\alpha = 0.10$	0.9535	10832	0.00444
sklearn.Lasso plug-in, $\alpha = 1.00$	0.5607	11520	0.00000

Résultats obtenus pour chaque classifieur sur la base **USPS** en **classe contre classe**

Classifieur	Score moyen	Nombre de 0	moyenne de $ w $
Régression linéaire plug-in, MSE	0.8022	0	0.25628
Régression linéaire plug-in, $L_2, \alpha = 0.00$	0.8089	0	0.24960
Régression linéaire plug-in, $L_2, \alpha = 0.25$	0.7428	0	0.00421
Régression linéaire plug-in, $L_2, \alpha = 0.50$	0.7517	0	0.00317
Régression linéaire plug-in, $L_2, \alpha = 0.75$	0.9000	0	0.00512
Régression linéaire plug-in, $L_2, \alpha = 1.00$	0.8263	0	0.00279
Régression linéaire plug-in, $L_1, \alpha = 0.00$	0.8152	0	0.25186
Régression linéaire plug-in, $L_1, \alpha = 0.25$	0.7354	0	0.00524
Régression linéaire plug-in, $L_1, \alpha = 0.50$	0.6530	0	0.00643
Régression linéaire plug-in, $L_1, \alpha = 0.75$	0.5762	0	0.00547
Régression linéaire plug-in, $L_1, \alpha = 1.00$	0.5982	0	0.00702
sklearn.LinearRegression plug-in, $\alpha = 1.00$	0.9690	0	0.03485
sklearn.Lasso plug-in, $\alpha = 0.10$	0.9204	2452	0.00125
sklearn.Lasso plug-in, $\alpha = 1.00$	0.9000	2560	0.00000

Résultats obtenus pour chaque classifieur sur la base **USPS** en **1 contre tous**

On remarque que nos régressions appliquées à la classification par la méthode **plug-in** sont peu efficaces. Celles de *sklearn* en revanche obtiennent un score correct. De plus, toujours sur les implémentations

tions de *sklearn*, nous voyons que le **Lasso** augmente énormément le nombre de poids nuls et fait que les poids non-nuls se rapprochent beaucoup plus de 0 qu'avec une simple régression linéaire. Nous en déduisons que le **Lasso** permet d'obtenir un vecteur de poids très sparse, utile lorsqu'on ne veut prendre en compte que quelques dimensions.

Après quelques recherches, il s'avère que si notre implémentation du Lasso ne fait que réduire les valeurs des poids sans les rendre nuls, c'est parce que nous utilisons la descente de gradient pour optimiser notre modèle. En effet, pour obtenir un vecteur sparse, la méthode d'optimisation par "coordinate descent" est plus appropriée.

2 LASSO et Inpainting

2.1 Introduction

2.1.1 Principe

Le principe de l'**Inpainting** est qu'une partie d'une image, un **patch**, peut être approximé par une **combinaison linéaire d'autres patchs** de l'image. Cela permet ainsi de pouvoir restituer une partie manquante d'une image, de la débruiter, ou encore de supprimer de plus larges objets (défauts du visage, touristes, ...). On va vouloir prendre en compte qu'un petit nombre de patchs : ceux qui contiennent le plus d'informations en commun avec la partie manquante. On va donc utiliser la fonction de coût **Lasso** afin d'avoir un vecteur de poids le plus sparse possible.

2.1.2 Déroulement & implémentation

Tout d'abord, la **construction du dictionnaire de patchs** est effectuée : On récupère tous les patchs d'une certaine taille dont le signal est pur (qui ne contiennent pas de pixel manquant).

Ensuite, on va boucler tant qu'il existe encore des pixels manquants dans l'image :

- On détermine le prochain patch à approximer
- On fait un apprentissage : Pour chaque emplacement de pixel non-vide du patch cible, on considère que les pixels de chaque patch du dictionnaire au même emplacement peuvent le reconstruire. On constitue donc nos ensembles d'apprentissage de cette façon.
- On prédit les valeurs de chaque pixel manquant du patch.

Ainsi, nous remplissons l'image patch-par-patch et non pixel-par-pixel, pour garder une certaine cohérence spatiale dans notre reconstruction.

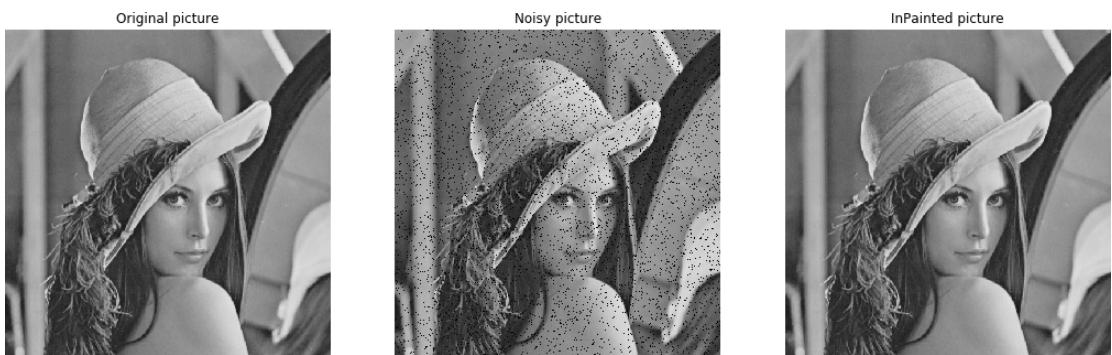
Enfin, notre implémentation peut être paramétrée selon les critères suivants :

- **patch_size** : La taille d'un patch de l'image.
- **step** : Le pas d'itération utilisé pour la construction du dictionnaire.
- **alpha** : L'importance accordée à la sparsité du vecteur de poids.
- **max_iterations** : Le seuil d'itérations maximum.
- **tolerance** : Le critère d'optimisation minimum avant arrêt.

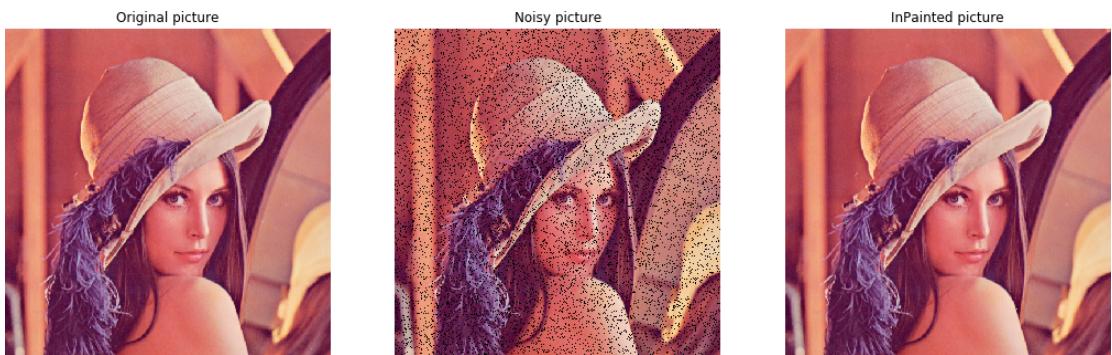
2.2 Résultats sur des images bruitées



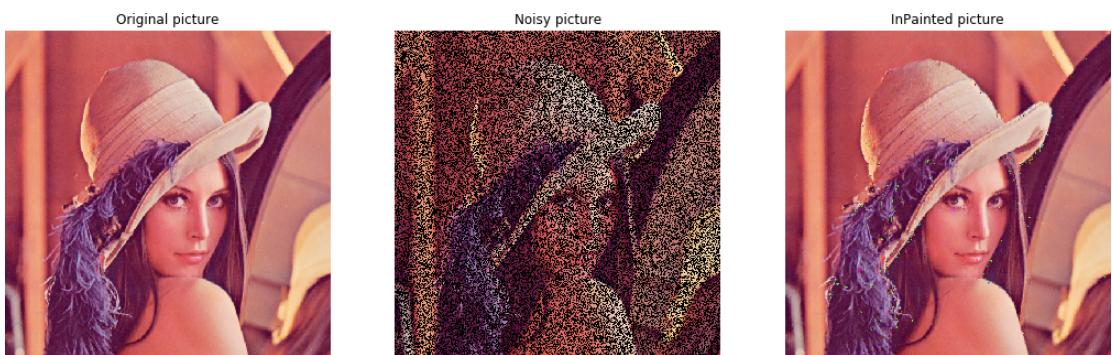
Inpainting de l'image res/pictures/lena_color_512.tif, 0,5% de l'image bruitée.



Inpainting de l'image res/pictures/lena_gray_512.tif, 5% de l'image bruitée.



Inpainting de l'image res/pictures/lena_color_512.tif, 10% de l'image bruitée.



Inpainting de l'image res/pictures/lena_color_512.tif, 50% de l'image bruitée.

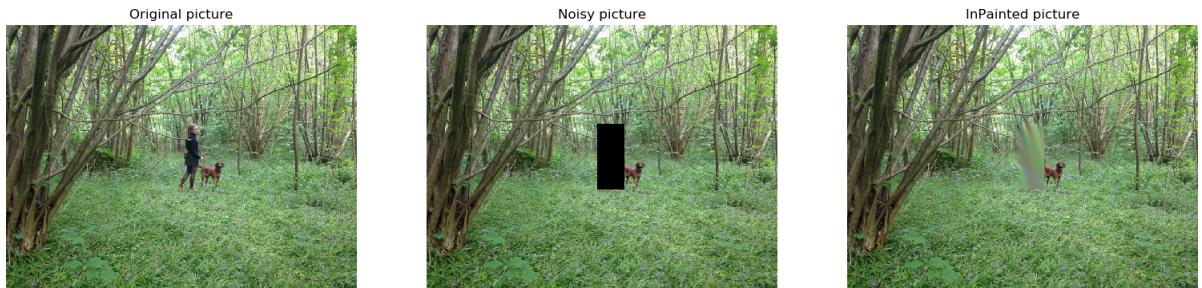
Nous obtenons des résultats visuellement très satisfaisants sur des pixels isolés. Et ce sans trop s'attarder sur les paramètres. Une petite taille de patch est cependant nécessaire pour des images beaucoup bruitées. On remarque aussi la difficulté d'approximer des détails non-redondants de l'image (l'extérieur des plumes, le contour du visage).

2.3 Résultats sur des patches manquants

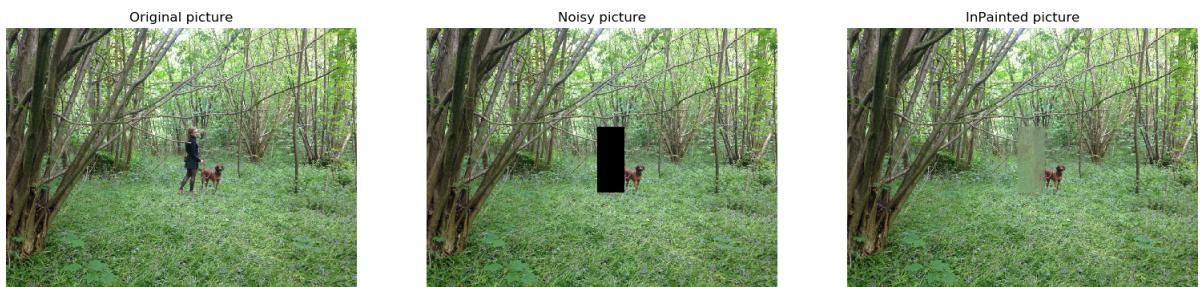
La reconstruction d'une plus grosse partie d'une image devient un peu plus compliquée. En effet, les patchs à reconstituer étant de taille plus importante, nous disposons de moins d'information pour chaque patch. Il va alors falloir s'y reprendre à plusieurs fois pour trouver de bons paramètres. Testons sur l'image res/pictures/outdoor.jpg :



Ici, $\alpha = 0.5$, ce qui est trop gros. On effectue une combinaison linéaire avec trop de patchs, ce qui explique cet effet de flou.



Ici, $patch_size = 5$, ce qui est trop petit. Les patchs ne contiennent pas assez d'information pour être correctement reconstruit. De plus, les pixels qui viennent juste d'être approximé sont trop pris en compte, ce qui donne un effet un peu "baveux".



Ici, $max_iterations = 1000$, ce qui est trop peu. On dépasse ce seuil avant d'avoir trouvé les poids optimaux pour notre reconstruction.



Ici, $\alpha = 0.0001$, $patch_size = 101$, $max_iterations = 100.000$. La reconstruction paraît visuellement correcte, mais elle peut sûrement être encore améliorée.

2.4 De l'importance dans l'ordre de remplissage

L'ordre dans lequel on choisit de remplir les patchs à toute son importance. En effet, nous utilisons un remplissage naïf de gauche à droite et de haut en bas. Ceci entraîne cependant une espèce de "bavure" (comme dans la deuxième figure obtenue). Ce résultat est décrit dans le papier de recherche [3] donné en référence dans l'énoncé : Nous venons de reconstruire une forme particulière (comme un tronc d'arbre), et en remplissant les pixels juste en dessous à l'itération suivante, nous prendrons trop en compte les pixels que nous venons de reconstruire.

Voici une manière efficace pour remplir un trou :

Tout d'abord, assigner une valeur de **priorité** à chaque patch. Plus elle est haute, plus il faudra traiter ce patch en premier. On peut décompenser cette priorité en un *produit de deux valeurs* : un coefficient d'**information**, indiquant à quel point un patch est proche de pixels purs, et un coefficient **isophotique**, caractérisant la force des *isophotes* (la direction et l'intensité des pixels) autour du patch.

Grossièrement, cette heuristique permet de reconstruire en premier lieu les contours du trou, tout en priorisant les gros changements d'intensité autour du trou (souvent liés à une coupe perpendiculaire au trou).