Reconnaissance de caractères

Équipe : Dreamers

LI XIANG

BEN SASSI RACHED

1 Introduction

Dans un premier temps nous avions pour but de faire de la reconnaissance et retranscription de texte/mot/caractère. Cependant, on a vite restreins notre sujet de data mining à de la reconnaissance de caractère.

Ce choix à surtout été motivé par nos soucis à réunir un dataset conséquent et 'propre' ainsi par un manque de temps et des ambitions trop grande (pour le temps imposé).

Nous avons donc choisi de scinder notre projet en deux afin de ne pas s'empiéter et de proposé plusieurs solutions au problème. Nous sommes conscients que certains algorithmes utilisés ne sont pas appropriés au problème que nous souhaitons résoudre, cependant nous avons voulu expérimenter un temps sois peu les diverses algorithmes vues en cours.

De plus, le choix de scinder le projet s'explique par le faite que nous avions a disposition un petit dataset de carapates en chinois et que nous souhaitions trouvé et utilisé un plus dataset.

Donc dans un premier temps Xiang, c'est occupé a travailler sur les caractères chinois et Rached, c'est occupé à trouver et nettoyer un plus grand dataset.

Entrée : l'image que l'on a pris dans laquelle il y a un caractère chinois manuscrit.
Sortie : Le vrai caractère

Suite à nos recherche, nous avons trouvé deux datasets :

- Dataset de caractères chinois composé de :
  - un ensemble d'apprentissage est un répertoire contenant 20 caractères, ces caractères sont dans 20 répertoires dont le nom est leur label. Pour chaque caractère, il y a 7 exemples de type différents. Ces 20 caractères sont :
    - 李 想 一 二 三 四 五 六 七 八 九 十 中 国 你 好 大 小 法 德
  - L'ensemble de test est un répertoire contenant des caractères manuscrits de chaque personne.

- Dataset de caractères trouvé sur le site de nist, nous avons regardé toute les archive disponibles et nous avons pris l'archive (~1GB) qui nous semble t-il, aller prendre le moins de temps de traitement. Ce dataset est composé de plusieurs dizaines de milliers d'images pour chaque caractères (a-z A-Z 0-9).
  Le dataset dispose aussi pour chaque caractère d'un ensemble d'entraînement composé de plusieurs dizaines de milliers d'images.

2 Méthodes (pour le dataset de caractères chinois)

a) L'idée principale

Pour décrire une image, on va choisir 3 paramètres :

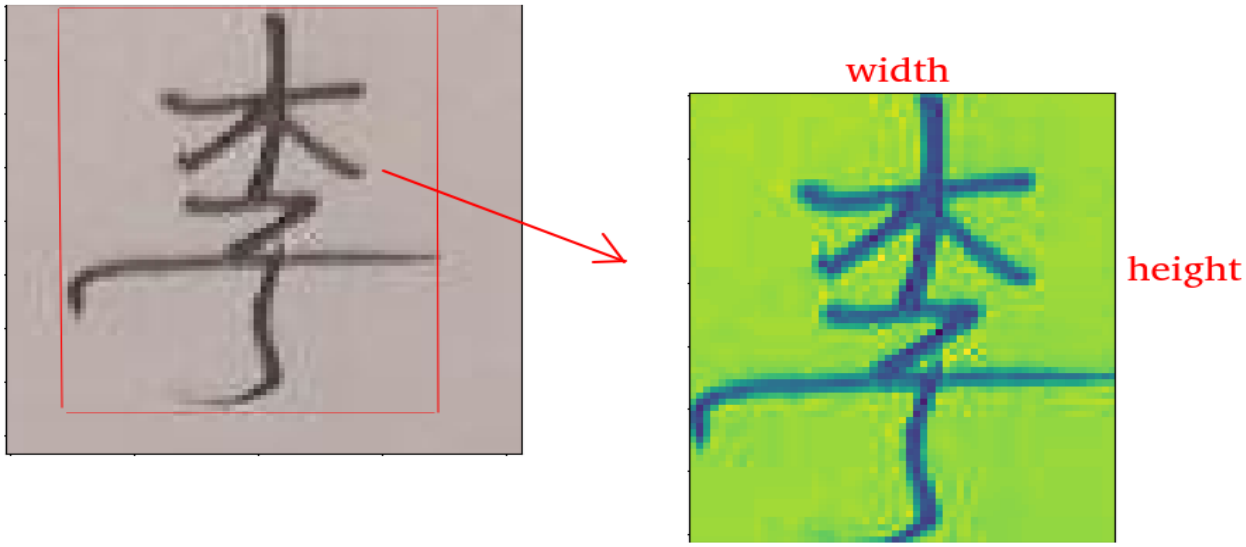
- Le ratio de (hauteur / largeur)
- Des transformations par ligne et par colonne
- La matrice binaire ayant une taille 100*100 représentant le caractère

b) Comment extraire ces trois paramètres

- **Le ratio**

Les images de caractères manuscrits ont une taille inconnue, donc tout d'abord il faut trancher la partie contenant uniquement le caractère.
On parcourt l'image (convertir en une matrice) et trouve deux points : un se situe le plus en haut à gauche, l'autre en bas à droite. Ces deux points construit un rectangle que l'on utilise pour obtenir une petite image qui ne contient que le caractère.
Et puis on enregistre le ratio de type float = hauteur / largeur.

Exemple :

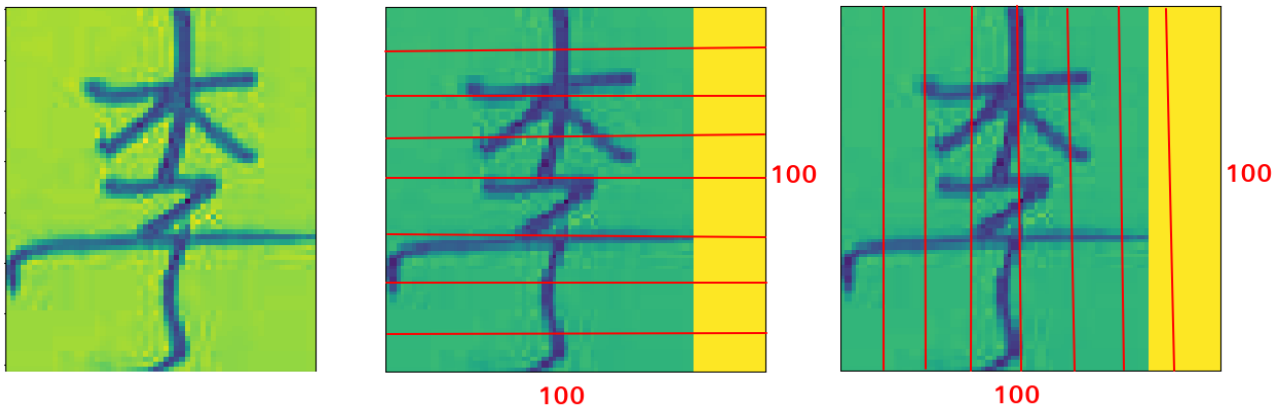


- **Des transformations**

Après l'étape précédente, on a les images de taille différente. Avant de calculer les transformations, on doit remplir ces images de sorte que chacune a une taille 100*100. L'idée est d'élargir le hauteur ou le largeur jusqu'à 100, ensuite ou remplir le reste par blanc (255).

Maintenant toutes les images sont uniformes. On divise le hauteur en 10 intervalles, par exemples, l'intervalle de ligne 1 à 10, 11 à 20 etc. Pour chaque intervalle, on calcule le nombre de transformations total.
Comment calculer la transformation ?
On parcourt une ligne, quand la couleur change de noir à blanc ou de blanc à noir, on augmente le nombre de changement.
A la fin, on obtient une liste de taille 10 dont l'indice 0 vaut le nombre de transformations total de ligne 1 à ligne 10. L'indice 1 vaut ligne 11 à ligne 20 et ainsi de suite.
On applique la même fonction par colonne.

Exemple :



- **La matrice binaire**

Pour chaque image de taille 100*100, on le convertit en une matrice binaire (0 noire, 1 blanc). Pour enregistrer l'information, on transforme la matrice en une chaîne de caractères. Comme on a la taille 100*100, cette chaîne a un longueur de 10000. On convertit cette chaîne de caractères binaires en hexadécimal.
(source : https://www.cnblogs.com/luolizhi/p/5596171.html)

c) Le renforcement d'apprentissage

Pour chaque image dans l'ensemble d'apprentissage, on élargit son hauteur de 1 à 16. Idem pour son largeur.
Donc on enregistre 31 fois pour une image. (1normal+15hauteur+15largeur) Ce renforcement nous permet d'augmenter la précision dans le cas où l'utilisateur ne connais pas exactement le hauteur et le largeur de caractère.

d) Exécution

make load_TrainSet

Après l'apprentissage, on obtient 3 fichiers :

- ***labels.txt*** : tous les vrais labels
- ***hashs.txt*** : des chaînes de caractères hexadécimal représentant une image.
- ***list21s.txt*** : chaque ligne est une liste. L'indice 0 (ratio).

L'indice 1 à 11 (transformations par ligne). Le reste (celles par colonne).
make resultV1 //Les résultats par personne.
make resultV2 //Le rapport par caractère
make resultV3 //Le rapport des 10 caractères simples

e) Résume de l'algorithme

l'étape 1 : Ratio

Comparer le ratio d'image courante et le ratio de tous les images, si la valeur absolue entre les deux est inférieur à **0.2**, passer l'étape suivante :

l'étape 2 : Transformation

Comparer la distance totale des transformations (par ligne et par colonnes), si il est inférieur à **40**, passer l'étape suivante :

l'étape 3 : Hash

Calculer la distance des chaînes de caractères hexadécimal. Avant de comparer deux chaînes, il faut faire une transformation de hexadécimal en binaire Pour chaque indice, 1+1=0 0+0=0 et 1+0=1 0+1=0, et puis on compte le nombre 1.
On enregistre (label) et (ratio,distance_étape2,distance_étape3) dans une dictionnaire.
Si le label existe déjà, on compare la distance d'étape2 et prend celle la plus petite.
Ça veut dire que au sein d'un groupe, on prend une image qui ressemble le plus par rapport à la distance des transformations.

l'étape 4 :

Parmi tous les labels dans la dictionnaire, on fait un tri dans l'ordre croissant selon la distance des transformations.
Ensuite, on compare le premier élément(label 1) à la deuxième(label 2).
Si la distance Hash du label 2 est inférieur à celle du label 1 et (transformation_label_1 – transformation_label_2) < **10**. On prend le label 2.
Sinon on prend la label 1.

Remarque :

Ces paramètres 0.2 , 40 , 10 ainsi que la taille d'image 100*100 sont pris selon de nombre tests que j'ai faits. Bien sûr on peut les optimiser.

3 Résultats

Ces résultats sont très sensibles à la taille de caractère. La valeur de Hash ne sert qu'à mon algorithme. Tous les autres méthodes utilisent le ratio et les transformations comme l'ensemble d'apprentissage.

La liste des méthodes utilisées :

SVM : les Support Vector Machines ⇒ ./SVC.py

- SVC(kernel='linear')
- SVC(kernel='poly')

Ad : Arbres de décision ⇒ ./DecisionTree.py

LR : Régression logistique ⇒ ./LogisticRegression.py

Kn : Les K plus proches voisins ⇒ ./Knearst.py

- KNeighborsClassifier (n_neighbors = 3, 5, 7, 9, 11)

Fa : Forêt aléatoires ⇒ ./RandomForest.py

- RandomForestClassifier (n_estimators = 100, 300, 500, 700, 900)

XL : Algorithme de Xiang ⇒ ./Mon_algo.py

a) Résultats_V1

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
1		XL	SVCli....	SVCpoly	Ad	LR	Kn1	Kn3	Kn5	Kn7	Kn9	Kn11	Fa100	Fa300	Fa500	Fa700	Fa900		average	max
2	chloe	60%	70%	75%	45%	30%	70%	70%	70%	65%	60%	60%	70%	70%	65%	65%	65%		68%	75%
3	etienne	55%	70%	65%	50%	40%	60%	65%	65%	65%	60%	55%	55%	55%	60%	60%	60%		60%	70%
4	chensi	70%	75%	80%	60%	55%	70%	65%	65%	65%	65%	65%	75%	80%	75%	75%	75%		74%	80%
5	marie	70%	65%	70%	50%	40%	65%	65%	70%	55%	50%	50%	60%	55%	55%	65%	60%		66%	70%
6	wangq	80%	85%	95%	65%	70%	85%	80%	85%	85%	80%	80%	75%	80%	80%	80%	80%		85%	95%
7	omar	75%	90%	95%	60%	70%	85%	80%	75%	80%	75%	75%	95%	90%	90%	90%	95%		88%	95%
8	lea	60%	55%	50%	45%	30%	65%	60%	65%	60%	60%	65%	65%	60%	65%	70%	70%		61%	70%
9	xiang2	95%	75%	85%	45%	50%	95%	90%	85%	85%	80%	75%	70%	65%	70%	70%	70%		86%	95%
10	guan	60%	65%	70%	45%	30%	85%	85%	85%	80%	75%	75%	65%	75%	70%	65%	70%		71%	85%
11	quincy	90%	75%	80%	60%	70%	80%	75%	75%	75%	75%	75%	75%	75%	80%	80%	80%		83%	90%
12	yizhe	90%	70%	80%	45%	55%	85%	90%	85%	80%	80%	80%	75%	90%	80%	80%	80%		84%	90%
13	pin	80%	65%	65%	45%	45%	75%	65%	70%	70%	70%	70%	60%	65%	65%	65%	70%		73%	80%
14	xiangli	85%	65%	65%	40%	35%	70%	70%	70%	60%	65%	70%	75%	80%	80%	80%	75%		74%	85%
15	alice	60%	45%	60%	45%	40%	70%	65%	65%	70%	70%	70%	60%	60%	60%	60%	60%		63%	70%
16	clong	75%	95%	95%	65%	55%	90%	85%	85%	85%	85%	80%	85%	90%	85%	85%	85%		86%	95%
17	yuxuan	70%	55%	60%	50%	45%	65%	60%	60%	60%	60%	60%	70%	65%	65%	60%	70%		66%	70%
18	AC	70%	65%	80%	45%	35%	85%	80%	75%	80%	75%	65%	65%	60%	60%	60%	60%		74%	85%
19	FA	65%	30%	45%	30%	10%	55%	65%	45%	45%	45%	45%	35%	35%	35%	35%	30%	ignored	49%	65%
20																				
21	average	73%	70%	75%	51%	47%	76%	74%	74%	72%	70%	69%	70%	71%	71%	71%	72%		74%	76%

Ce résultat est dans le fichier Result_V1.ods.

Chaque colonne est une méthode de classification. Les 4 colonnes en rouge sont les méthodes qui fonctionnent bien. Les pourcentages signifient parmi 20 caractères manuscrits, combien l'algorithme réussie à prédire.

En analysant ce résultat, on peut résumer que :
⇒ Les support Vecteur Machines, Les K plus proches voisins et Forêt aléatoires sont adaptés à mon problème de classification.
⇒ Si on augmente le nombre des voisins, le résultat devient moins précis.
⇒ Si on augmente le nombre d'estimateur, le résultat changent pas beaucoup.

- **Pourquoi l'arbre de décision et la régression logistique ne fonctionnent pas bien ?**

Les caractéristiques sont 21 variables. L'arbre de décision marche avec moins de paramètres. La régression logistique interprète chaque variable de manière probabiliste, donc mauvais résultats avec beaucoup de variables.

- **Comment choisir le noyaux pour SVC**

On teste les noyaux possibles et on prend ce qui nous donne le meilleur résultat.
En fait, SVC marche bien parce que le but de mon problème est de trouver la similarité entre deux caractères. Le noyaux polynomial nous permet configurer plus de paramètres que le noyaux linaire. Si on trouvais les bons paramètres, on aurait un bon résultat.

- **Les K plus proches voisins**

Si au augmente le K, le caractère manuscrit de test peut ressembler à d'autre caractères. Le but est de trouver un caractère imprimants qui le ressemble le plus. C'est pourquoi K = 1 nous donne le meilleur résultat.

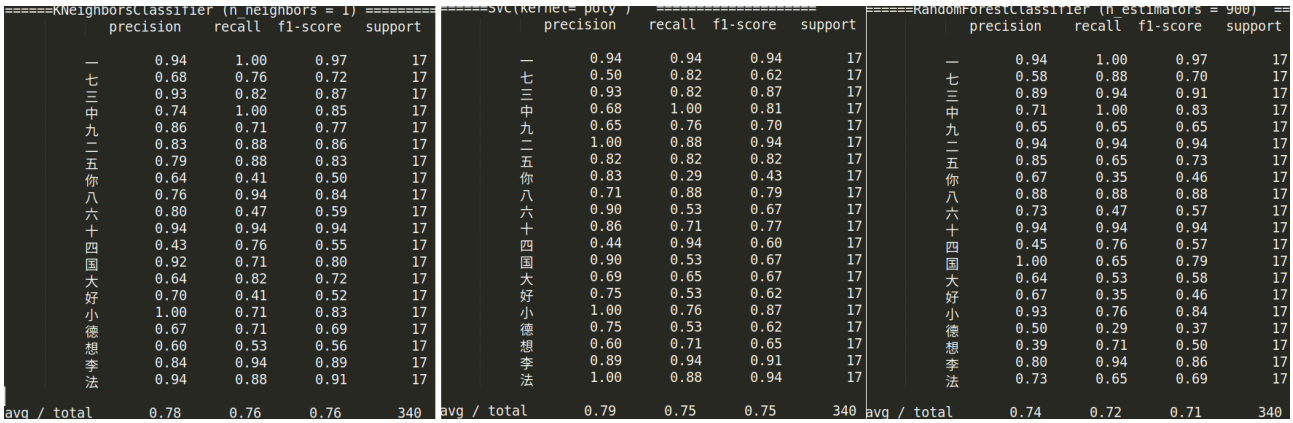
- **Forêt aléatoires**

Normalement, si on augmente le nombre d'estimateur, le résultat devra plus précis. Autrement dit, le pourcentage va augmenter. Mais on voit sur la photo du résultat V1,
pour certaine personnes, le pourcentage baisse. Le principe est on est meilleur ensemble que chacun séparément. À mon avis, il se peut que la vérité soit éclipsée parfois.

b) Résultats_V2

On peut aussi voir le rapport de ces trois méthodes :

make resultv2
less result2.txt



======KNeighborsClassifier (n_neighbors = 1) ========

	precision	recall	f1-score	support
一	0.94	1.00	0.97	17
七	0.68	0.76	0.72	17
三	0.93	0.82	0.87	17
中	0.74	1.00	0.85	17
九	0.86	0.71	0.77	17
二	0.83	0.88	0.86	17
五	0.79	0.88	0.83	17
你	0.64	0.41	0.50	17
八	0.76	0.94	0.84	17
六	0.80	0.47	0.59	17
十	0.94	0.94	0.94	17
四	0.43	0.76	0.55	17
国	0.92	0.71	0.80	17
大	0.64	0.82	0.72	17
好	0.70	0.41	0.52	17
小	1.00	0.71	0.83	17
德	0.67	0.71	0.69	17
想	0.60	0.53	0.56	17
李	0.84	0.94	0.89	17
法	0.94	0.88	0.91	17
avg / total	0.78	0.76	0.76	340

======SVC(kernel= poly) ==========

	precision	recall	f1-score	support
一	0.94	0.94	0.94	17
七	0.50	0.82	0.62	17
三	0.93	0.82	0.87	17
中	0.68	1.00	0.81	17
九	0.65	0.76	0.70	17
二	1.00	0.88	0.94	17
五	0.82	0.82	0.82	17
你	0.83	0.29	0.43	17
八	0.71	0.88	0.79	17
六	0.90	0.53	0.67	17
十	0.86	0.71	0.77	17
四	0.44	0.94	0.60	17
国	0.90	0.53	0.67	17
大	0.69	0.65	0.67	17
好	0.75	0.53	0.62	17
小	1.00	0.76	0.87	17
德	0.75	0.53	0.62	17
想	0.60	0.71	0.65	17
李	0.89	0.94	0.91	17
法	1.00	0.88	0.94	17
avg / total	0.79	0.75	0.75	340

======RandomForestClassifier (n_estimators = 900) ==

	precision	recall	f1-score	support
一	0.94	1.00	0.97	17
七	0.58	0.88	0.70	17
三	0.89	0.94	0.91	17
中	0.71	1.00	0.83	17
九	0.65	0.65	0.65	17
二	0.94	0.94	0.94	17
五	0.85	0.65	0.73	17
你	0.67	0.35	0.46	17
八	0.88	0.88	0.88	17
六	0.73	0.47	0.57	17
十	0.94	0.94	0.94	17
四	0.45	0.76	0.57	17
国	1.00	0.65	0.79	17
大	0.64	0.53	0.58	17
好	0.67	0.35	0.46	17
小	0.93	0.76	0.84	17
德	0.50	0.29	0.37	17
想	0.39	0.71	0.50	17
李	0.80	0.94	0.86	17
法	0.73	0.65	0.69	17
avg / total	0.74	0.72	0.71	340

Si F1-score d'un caractère rapproche 1, c'est plus facile à le reconnaître.
F1-score représente la difficulté pour distinguer ce caractère parmi les 20 caractères.
Il y a quelques caractères qui ressemblent aux autre tel que 六 大, dans ce cas, la taille de caractère et les transformations se rapprochent. On ne peut pas bien prédire c'est quel caractère.

4 Méthode (Dataset pour le caractères latins)

a) L'idée principale

On part d'un grand dataset plutôt bien ordonnée, on effectue quelques traitement dessus afin d'optimiser les temps d'exécution des algorithmes.

b) Traitement des données

- **Labellisation et extraction d'un sous ensemble de donnée**

Dans un premier temps, après avoir téléchargé l'archive contenant le dataset, on parcours le dataset afin de renommer chaque dossier par le caractère représenté par les images contenu. Après avoir fait cela, on a essayé de faire marcher les algorithmes déjà implémenté pour l'autre dataset, cependant ici cela prenait un temps trop long. On a donc décidé de créer un sous ensemble (restreints) du dataset.

On a du restreindre nos données a 160 images par caractères. On est conscients que c'est très peu mais même comme ça on a perdu énormément de temps à chaque exécution (~15 min).

On crée donc deux dossiers :

- un dossier qui prend l' ensemble d'entraînement pour chaque caractères déjà présent dans l'archive téléchargé.
- Un dossier de teste qui est composé d'un certains nombre d'images pour chaque caractères.

- **Transformation des données**

Pour appliquer les algorithmes, on traite les données pendant la création des ensembles d'entraînement et de teste et lors que l'on les charge.

La premier transformation ce fait sur la dimension des images, ainsi pendant la création des ensembles d'entraînement et de teste, on modifie la taille des images (avec l'option ANTIALIAS) pour réduire leurs tailles, ainsi on obtient des images de 40*40.

Lors que l'on charge les données, on extrait les features des images et les labels (les noms de dossiers).

c) Exécution

Nous avons eu le temps d'utiliser deux algorithmes pour ce grand dataset :

Le premier utilise svm :

- ***python3 svm.py*** avec option obligatoire: ***linear*** ou ***poly***

Le second utilise le percetron multicouche :

- ***python3 mlpclassifier.py***

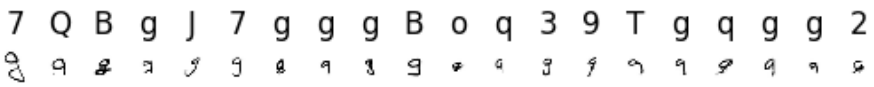
5 Résultats

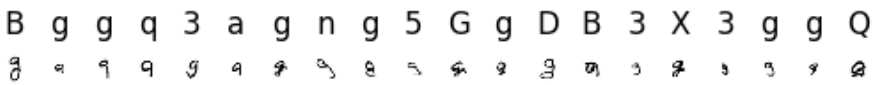
Les résultats ne pas aussi précis que l'on aurait voulu. Cela s'explique principalement par le fait que notre dataset n'est au final pas aussi propre que l'on aurait voulu.

Par exemple, les images ci-dessus sont classés comme des caractères 'g' alors qu'ils coïncident avec des 'a', des 'o', des '9' ou encore des 'q' (on dirai que certaines images sont coupés)

- **SVM (noyau linear)**

ici on vous montre ce que l'on obtient en comme prédiction pour 'q'





On obtient la précision suivante :

```
****************** average_score : 0.583870967742
```

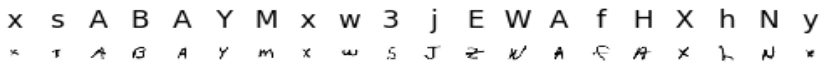
Là encore on pense que c'est notre dataset qui ne permet pas d'avoir de bon résultat.
On pourrai néanmoins mieux configuré svc afin d'obtenir de meilleurs résultats, mais on aura toujours moins que pour le dataset de caractères chinois qui lui est bien plus propre.

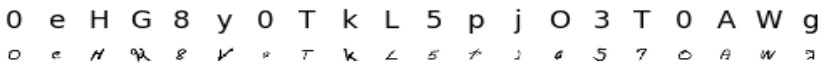
- **SVM (noyau poly)**

```
****************** average_score : 0.600201612903
```

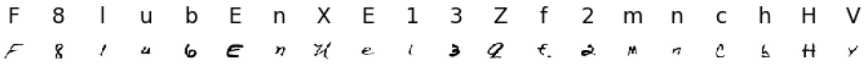
On en conclu la même chose avec l'utilisation de autre noyau.

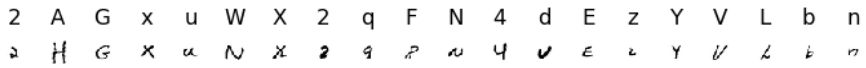
Comparaison noyau poly / linear





kernel linear

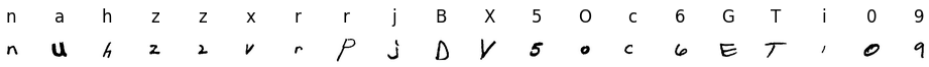


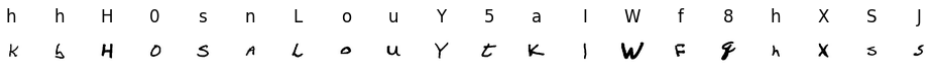


kernel poly

- **Perceptron multicouche (noyau poly)**

Exemple de prédiction obtenu :





Précision obtenu :

```
****************** average_score : 0.486491935484
```

On aurait pu obtenir mieux là aussi, au départ on obtenait une précision très faible (de l'ordre de 1%) mais on faisant quelque réglage on arrive a près de 50 %.
Les réglages que l'on a fait sont assez basique, par exemple, changer le learning rate ou encore l'activation.

6 Discussion

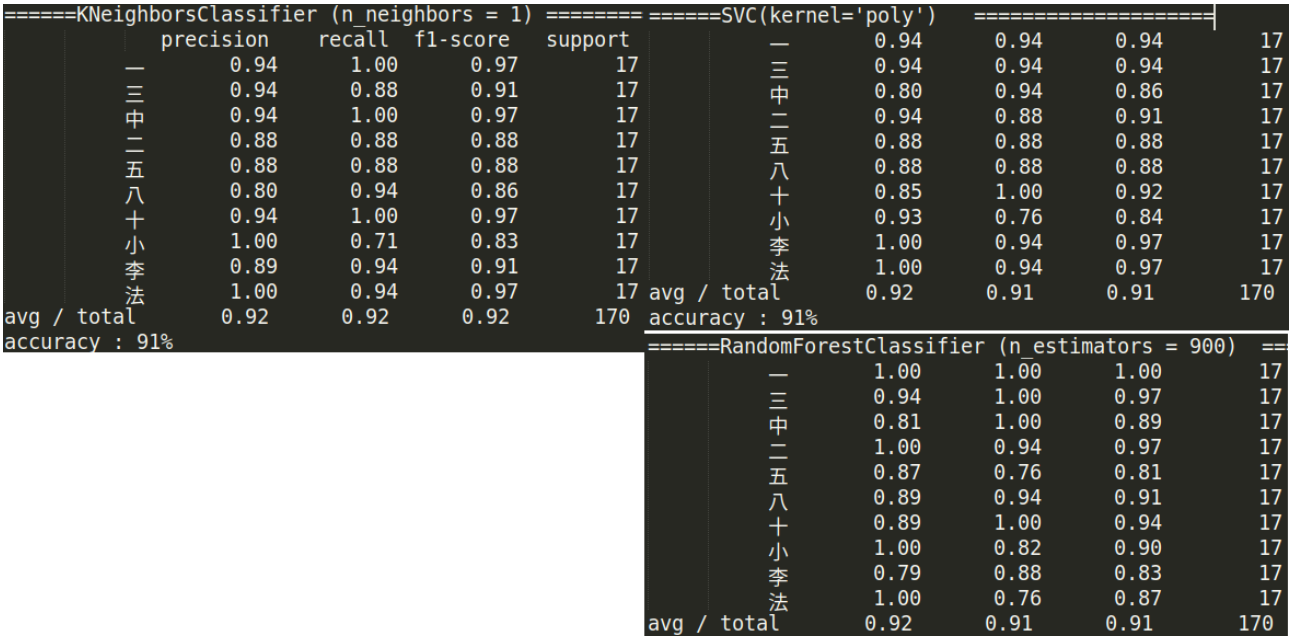
Rappelle : Le problème est de reconnaître des caractères manuscrits à partir des caractères imprimants.

Les facteurs qui pourraient perturber le résultat :

- L'ensemble d'apprentissage : des caractères imprimants. Le nombre total, le nombre d'exemple, la propreté des données, la complexité et la similarité.
- L'algorithme de classification : SVM(poly), K plus proche et Forêt aléatoires.
- Les caractéristiques : la ratio et les transformations.
- Le traitement d'image : surtout les paramètres choisis pour extraire le caractère, donc on va tester des améliorations possibles.

a) Abandonner les caractères compliqués (pour le dataset de caractères chinois)

make resultv3
less result3.txt
On prend 10 caractères qui ont un taux haut en fonction de f1-score.



```
======KNeighborsClassifier (n_neighbors = 1) ========
              precision    recall  f1-score   support

           一       0.94      1.00      0.97        17
           三       0.94      0.88      0.91        17
           中       0.94      1.00      0.97        17
           二       0.88      0.88      0.88        17
           五       0.88      0.88      0.88        17
           八       0.80      0.94      0.86        17
           十       0.94      1.00      0.97        17
           小       1.00      0.71      0.83        17
           李       0.89      0.94      0.91        17
           法       1.00      0.94      0.97        17

avg / total       0.92      0.92      0.92       170

accuracy : 91%
```

```
======SVC(kernel='poly')   ==================
              precision    recall  f1-score   support

           一       0.94      0.94      0.94        17
           三       0.94      0.94      0.94        17
           中       0.80      0.94      0.86        17
           二       0.94      0.88      0.91        17
           五       0.88      0.88      0.88        17
           八       0.88      0.88      0.88        17
           十       0.85      1.00      0.92        17
           小       0.93      0.76      0.84        17
           李       1.00      0.94      0.97        17
           法       1.00      0.94      0.97        17

avg / total       0.92      0.91      0.91       170

accuracy : 91%
```

```
======RandomForestClassifier (n_estimators = 900)  ==
              precision    recall  f1-score   support

           一       1.00      1.00      1.00        17
           三       0.94      1.00      0.97        17
           中       0.81      1.00      0.89        17
           二       1.00      0.94      0.97        17
           五       0.87      0.76      0.81        17
           八       0.89      0.94      0.91        17
           十       0.89      1.00      0.94        17
           小       1.00      0.82      0.90        17
           李       0.79      0.88      0.83        17
           法       1.00      0.76      0.87        17

avg / total       0.92      0.91      0.91       170
```

On voit que on a une précision environ 90%.
En conclusion, si on prenais les caractères qui ont les tailles très différentes et qui ressemblent pas beaucoup aux autres, on aurait un bon résultat de classification.
Mais si on augmente le nombre des caractères, 1000 ou 3000, comment faire ?
Dans ce cas, il faut ajouter d'autre caractéristiques et utilisent deux ou plusieurs algorithmes en même temps.

b) Change les caractéristiques

- Ajouter les transformations à partir de la ligne 5 avec une intervalle de 10.
- Il existe une fonction mesure.find_contours(array,level) dans la librairie « skimage ». Mais on n'ai pas trouvé une façon pour convertir les contours en caractéristiques.
- Les résultats que l'on a obtenus sont très sensibles aux paramètres lors de l'extraction de caractère d'une image, donc c'est mieux de tester plusieurs fois et de choisir les meilleurs paramètres pour l'ensemble de test.

c) Récupérer des caractères manuscrits

Tous les caractères manuscrits que l'on a récupérés sont écrits dans une feuille blanche, donc il n'y a pas une contrainte de taille sur chaque caractère. Si on prenais un carré 100*100 pour chaque caractère manuscrit, ça va augmenter la précision.

De plus, si les caractères sont écrit sur une feuille avec des marquages on aurait sûrement moins de précision.

4.4 L'inverse du problème
Si on inversait l'ensemble de test (les caractères manuscrits) comme l'ensemble d'apprentissage, mon modèle d'algorithme marche encore ou pas ? → LEQUEL ??

7 Conclusion

On a réussi à faire la reconnaissance optique de caractères chinois et latins. Pour le dataset de caractères chinois l' extraction de caractéristiques se base sur la taille de caractère et les transformations par ligne et par colonne.

On obtient des résultats qui sont plutôt bon avec les algorithmes vues en cours.

On a aussi fait un algorithme qui marche bien. L'algorithme de «aHash» (voir 2.2.3) ne fonctionne pas très bien avec des caractères chinois car ils sont trop compliqués. Mais cet algorithme marche avec les 10 chiffres et les 26 lettres en anglais (voir src/).