Rapport de TP
Reconnaissance de chiffres manuscrits

1. Introduction

L'objectif de ce TP est d'appréhender les différentes étapes d'un système de reconnaissance de formes depuis les prétraitements (localisation/extraction des formes à reconnaître) jusqu'à la reconnaissance (combinaison des résultats de reconnaissance de plusieurs classifieurs).
Il s'agit de développer, sous Matlab, un système complet de reconnaissance de chiffres manuscrits.

2. Données

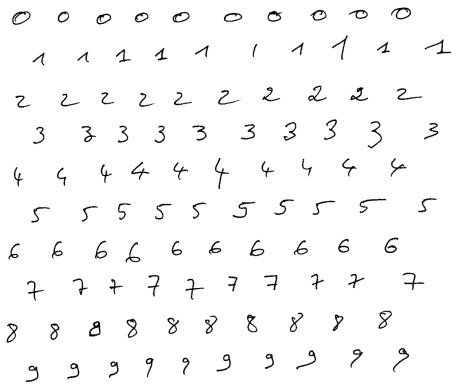

Figure 1 : Image chiffres manuscrits pour test

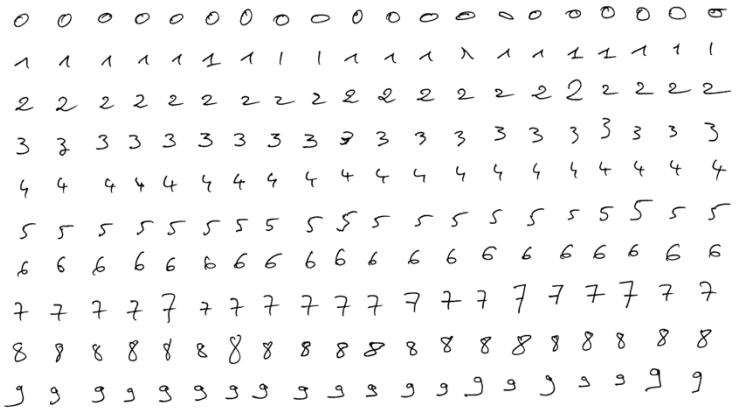

Figure 2 : Image chiffres manuscrits pour entraînement

Nous avons 2 images en formats .tif chacune est composée de chiffres allant de 0 à 10. Les chiffres sont ordonnés de la façon suivante chaque ligne correspond à un chiffre (classé de 0 à 9) et les colonnes correspondent à la répétition du chiffre de la ligne x fois.
L'image de test est composée de 10 lignes (chiffre) et 10 colonnes (répétition), l'image d'entraînement est composée de 10 lignes (chiffre) et 20 colonnes (répétition).

3. Présentation du code

Le script **mainScript.m** permet d'exécuter l'algorithme.

Les autres fichiers sont des fonctions :

Classifieur1 et **classifieur2** sont les fonctions principales qui permettent de lancer les classifieurs 1 et 2 sur l'image de test. Le retour de ces 2 fonctions est un tableau à 3 dimensions. Explication dans le schéma ci-dessous :

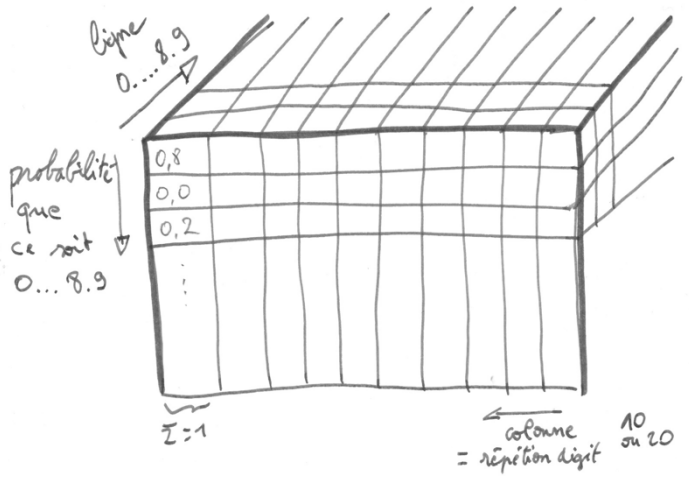

Figure 1: Schéma structure de donnée résultats sortis classifieurs

coordEachDigitsOpt est également une fonction des plus importante pour l'utilisateur, car elle permet d'obtenir les coordonnées de chaque chiffre dans une image. Elle prend en paramètre l'image et le nombre de répétitions de chaque chiffre et retourne un **tableau 10 x 4 x nombres répétition.** Explication dans le schéma ci-dessous :

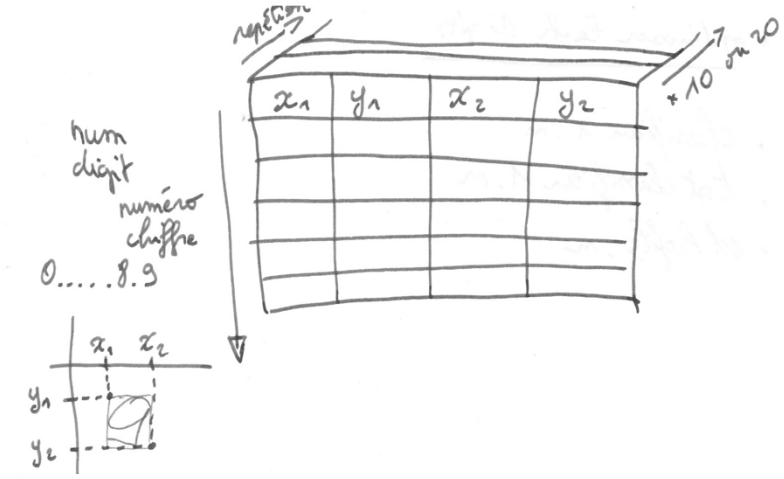

Figure 2: Schéma structure stockage coordonné chiffres

4. Prétraitement : Localisation et extraction des chiffres manuscrits

Tout d'abord il faut détecter dans l'image de test et d'entraînement chaque chiffre présent.

Comme ces chiffres sont bien alignés, nous pouvons commencer par détecter chaque ligne.
Pour ce faire nous utilisons l'histogramme par rapport à l'ordonnée de l'image globale, sur l'histogramme les pics correspondent à une ligne et les zones de zéros correspondent à l'espace entre les lignes. Grâce à ça nous avons obtenons l'image ci-dessous :

C'est les fonctions **histo_horizontale** ainsi que **getCoordVertical** qui vont se charger de cela.

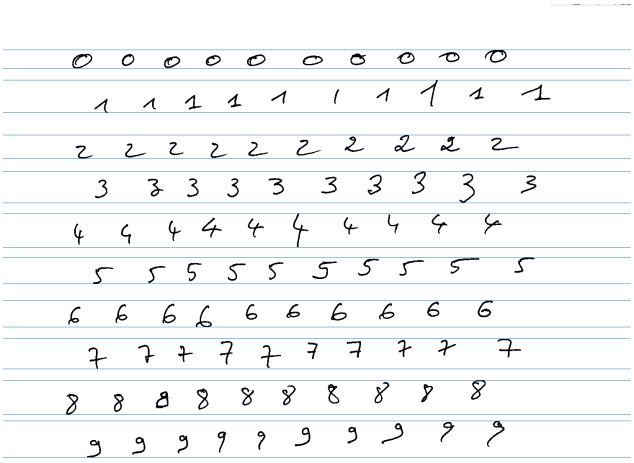

Figure 3: Image Test avec les lignes séparant chaque rangée

Maintenant que nous avons les lignes, nous devons dans chaque ligne séparer chaque chiffre.

Nous pouvons également ici utiliser la méthode précédente avec les histogrammes. Mais cette fois-ci par rapport à l'axe des abscisses de l'image de la ligne obtenue précédemment. En assemblant le résultat de chaque ligne, nous obtenons le résultat de l'image ci-dessous.

C'est la fonction **getCoordHorizontal** qui se charge de cela.

Figure 4: Image d'entraînement après séparation de chaque chiffre grâce à l'histogramme sur l'axe des ordonnées puis sur l'axe des abscisses

Maintenant que nous avons bien séparé chaque chiffre de l'image, nous obtenons une image de chiffre dont la largeur est bien optimisée, mais pas la hauteur. Par exemple sur l'image du dernier chiffre de l'image d'entraînement ci-dessous (le chiffre 9).

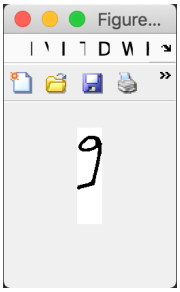

Figure 5: Exemple chiffre individuel, mais non optimisé

La dernière opération qu'il reste à faire est la même que la première. Cette fois encore nous t'utilisons l'histogramme par rapport à l'axe des ordonnées, mais maintenant on le fait individuellement sur chaque chiffre.

C'est la fonction **optimiserEachDigit** qui fait ce travail. La structure de donnée du résultat est expliquée dans la partie 3 du rapport : « 3. Présentation du code ».

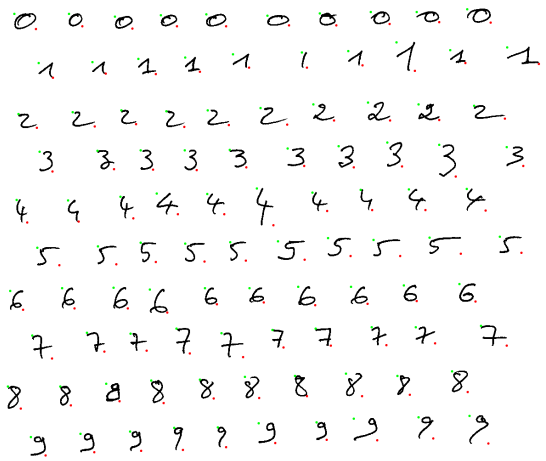

Figure 6: Image avec la séparation des chiffres optimisée. Le point vert représente le premier point en haut à gauche du chiffre et le point rouge le dernier point en bas à droite du chiffre.

5. Classifieurs

Maintenant que nous avons séparé chaque chiffre de l'image de test et de l'image pour l'entraînement, nous pouvons les utiliser dans des classifieurs.

a. Profils et classification par distance euclidienne minimum

L'une des fonctions importantes dans le classifieur est récupérer des données sur chaque chiffre/image. Pour, ce premier classifieur nous allons utiliser la méthode des profils.

a. Profils

Le profil d'un chiffre est obtenu en traçant des x lignes sur l'image. Puis on mesure de chaque côté la taille sur la ligne allant de l'extérieur gauche/droit jusqu'au premier pixel correspondant au chiffre.

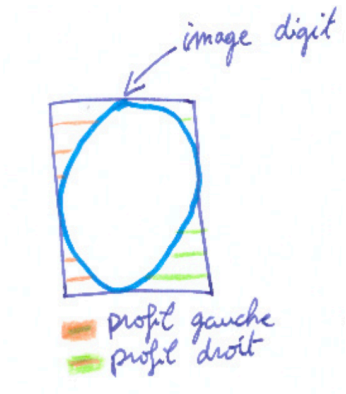

Figure 7: Schéma de l'obtention des profils

Dans le code, le profil est obtenu grâce à la fonction **getProfilImage.** Elle prend en paramètre **l'image du chiffre** que l'on veut traiter ainsi que le **nombre de trait** pour obtenir le profil. La sortie est un vecteur de taille **nombre de trait x 2** qui correspond aux tailles de chaque profil gauche puis les tailles de chaque profil droit.

Pour l'entraînement, la fonction des profils est utilisée sur chaque image. Puis on fait la moyenne des profils obtenus pour chaque classe. Cette moyenne correspond au profil d'entraînement.

C'est la fonction **getProfils** qui permet d'obtenir le profil d'entraînement.

b. Classification par distance euclidienne minimum

Maintenant, nous utilisons les profils d'entraînement de chaque classe pour faire la classification.

C'est la fonction **getProfilsTest** qui fait ce travail, elle fonctionne de la façon suivante : pour chaque image à tester on récupère son profil puis on calcule la probabilité qu'elle appartienne à chaque classe grâce à la classification par distance euclidienne minimum (la formule est ci-dessous) dans lequel on applique le profil d'entraînement ainsi que le profil de l'image à tester.

Formule : Classification par distance euclidienne minimum

Nous avons 10 classes (0...8,9)
p est la probabilité que le chiffre à classer, dont le vecteur de caractéristique est noté x appartienne à la classe C_i, dont le centre est le vecteur noté w_i.

$$p\left(\frac{C_i}{x}\right) = \frac{e^{-dist(x,w_i)}}{\sum_{j=0}^{9} e^{-dist(x,w_j)}}$$

Le résultat de sortie du classifieur est un tableau avec la probabilité d'appartenance à chaque classe pour chaque chiffre de l'image à tester, représentation graphique sur la figure 1.

b. Densités et KPPV

L'une des fonctions importantes dans le classifieur est récupérer des données sur chaque chiffre/image. Pour, le second classifieur nous allons utiliser la méthode des densités.

a. Densités

Les densités d'un chiffre sont obtenues en séparant l'image en x rectangles (sur la hauteur et largeur). Dans chaque rectangle, on va compter le nombre de pixels noir.

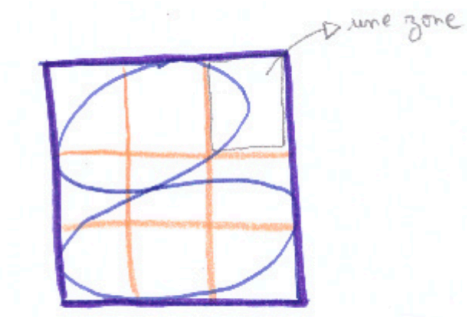

Figure 8: Schéma de l'obtention des densités

Dans le code, les densités sont obtenues grâce à la fonction **getDensityImage.** Elle prend en paramètre **l'image du chiffre** que l'on veut traiter ainsi que le nombre de rectangle sur la largeur/hauteur. La sortie est un vecteur de taille **nombre de trait x 2** qui correspond aux tailles de chaque profil gauche puis les tailles de chaque profil droit. La sortie un vecteur une dimension représentant l'information d'un chiffre/image de la taille nombre de nombre de **rectangles sur largeur x nombres de rectangles sur la hauteur** (nous avons supposé que le nombre de rectangle sur la largeur est égal au nombre de rectangles sur la hauteur).

Pour l'entraînement, la fonction des densités est utilisée sur chaque image/chiffre, ce qui nous donne un tableau de la forme suivante : [nombre zone hauteur * nombre de zones largeur x nombres de chiffres par classe (20) x nombres de digits (10)]. Ce résultat correspond aux densités d'entraînement.

C'est la fonction **getDensities** qui permet d'obtenir les densités d'entraînement.

b. Classification par K-PPV (K Plus Proches Voisins)

Maintenant, nous utilisons les densités de chaque chiffre de la base d'entraînement nous pouvons les utiliser avec K-PPV.

C'est la fonction **getDensitiesTest et compareEachDigitKPP** qui font ce travail. **GetDensitiesTest** fonctionne de la façon suivante : pour chaque image à tester on récupère ses densités puis donne elle donne à **compareEachDigitKPP** les densités de l'image en test et les densités d'entraînement ainsi que le nombre k de voisins à prendre en compte.

Puis, **compareEachDigitKPP** calcule la probabilité qu'elle appartienne à chaque classe grâce à la classification KPPV (la formule est ci-dessous), de la façon suivante : elle compare les distances entre le vecteur densité de notre chiffre à tester avec les vecteurs densités de toutes les images de la base d'entraînement. On trie nos résultats par ordre croissant et on garde les k classes avec les distances les plus faibles. Puis grâce à la formule suivante, on assigne les probabilités d'appartenance à chaque classe au chiffre en test.

Formule :

Le vecteur de probabilité d'appartenance à chaque classe est obtenu de la façon suivante :

$$p\left({}^{C_i}/_{x}\right) = \frac{k_i}{K}$$

Où k_i est le nombre de voisins parmi K appartenant à la classe C_i.

Le résultat de sortie du classifieur est un tableau avec la probabilité d'appartenance à chaque classe pour chaque chiffre de l'image à tester, représentation graphique sur la figure 1.

c. Combinaison de classifieurs

Nous avons obtenu des résultats pour chaque classifieur mais pour améliorer l'efficacité de la classification nous pouvons fusionner les 2 classifieurs ensemble. Nous allons commencer par une **somme** des classifieurs puis un **produit** des classifieurs.

C'est la fonction **combinaisonClassifieurs** qui fait cela, elle prend en entrée les résultats des classifieurs 1 et 2 puis retourne la combinaison faite par la méthode de la somme et la combinaison faite par la méthode du produit.

Voici les formules :

La somme : $p\left(\frac{C_i}{x}\right) = \frac{p_{classifieur1}\left(\frac{C_i}{x}\right) + p_{classifieur2}\left(\frac{C_i}{x}\right)}{\sum_{j=0}^{9}\left(p_{classifieur1}\left(\frac{C_i}{x}\right) + p_{classifieur2}\left(\frac{C_i}{x}\right)\right)}$

Le produit : : $p\left(\frac{C_i}{x}\right) = \frac{p_{classifieur1}\left(\frac{C_i}{x}\right) \cdot p_{classifieur2}\left(\frac{C_i}{x}\right)}{\sum_{j=0}^{9}\left(p_{classifieur1}\left(\frac{C_i}{x}\right) \cdot p_{classifieur2}\left(\frac{C_i}{x}\right)\right)}$

Le dénominateur est là pour normaliser les vecteurs.

6. Analyse et conclusion

a. Classifieur 1

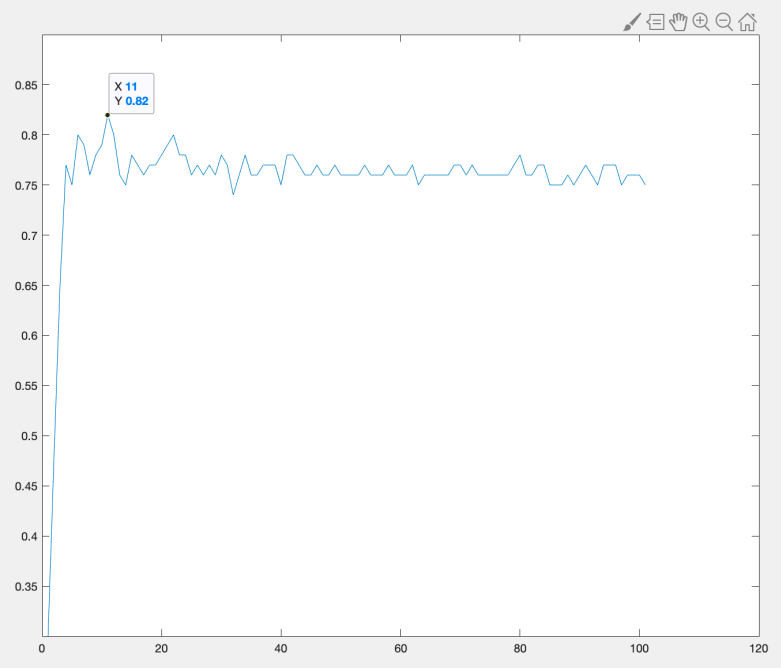

Figure 9: Évolution de la précision du classifieur 1 en fonction du nombre de traits

On fait varier le nombre de traits de 1 à 101 pour obtenir ce graphe.

Grâce au graphe ci-dessus nous pouvons voir que la précision augmente de façon constante de 1 à 4 traits pour obtenir un premier pic pour 4 traits avec une précision de 77 % puis en augmentant le nombre de traits on peut voir que la précision oscille entre 74 % et 82 %.
La meilleure précision est de 82 %, elle est obtenue avec 11 traits.

On peut déduire de ces résultats qu'au bout d'un moment le nombre de traits n'influe presque plus la précision, car il y a une très petite variation de précision.

b. Classifieur 2

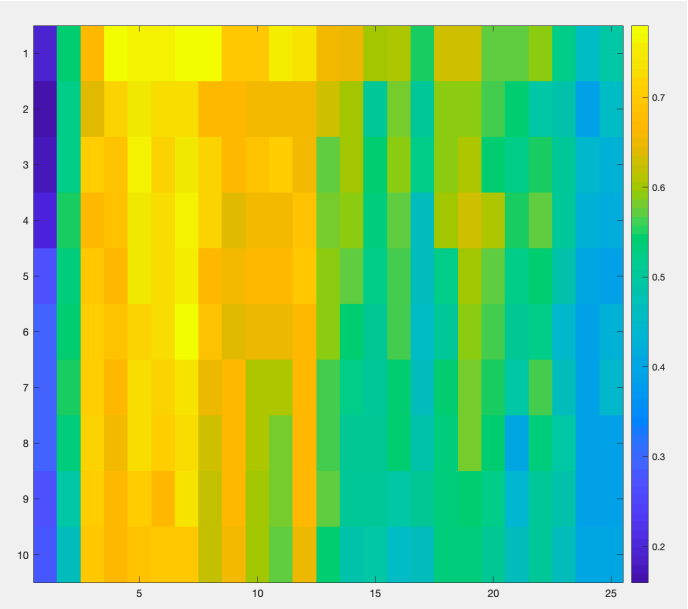

Figure 10: Précision du classieur (0 à 1) en fonction de k (ordonné) et nombre de rectangles sur largeur/hauteur (abscisse)

Pour analyser les résultats du classifieurs 2 on a fait varier le nombre de k voisin de 1 à 10 et le nombre de rectangles sur la largeur/hauteur de 1 à 25 (garder le même nombre de rectangles sur la largeur et le haut permet une analyse plus facile/rapide).

Grâce au graphe dessus nous pouvons en déduire que le meilleur résultat obtenu est de 78 % et ce résultat est obtenu avec les valeurs suivantes :

- k = 6 et 7 rectangles
- k = 1 pour 4, 7, 8 rectangles

On remarque qu'avec 1 rectangle les résultats sont très faibles entre 16 % et 28 %. Si l'on dépasse 13 rectangles, les résultats de la précision deviennent de plus en plus mauvais.
On remarque également que le nombre de k voisin influe beaucoup moins que le nombre de rectangles, toutefois, plus le nombre de k augmente plus les résultats diminuent en précision.

c. Produit

La solution que j'ai trouvée pour calculer la précision est de tester chaque combinaison de k, le nombre de rectangle (classifieur 2) et de trait (classifieur 1) grâce à 3 boucles for, une pour chaque variable. Ce qui me retourne un tableau à 3 dimensions contenant la précision en fonction des 3 variables. Cependant, je n'ai pas trouvé de moyen d'afficher simplement ce tableau 3 d vers 2 d.

Fonction matlab : optimizeAccuracy

Je vais donc décrire les résultats que j'ai obtenus.

Ceci est un résumé des meilleurs résultats obtenus :

0.8300 ➔ k = 2, nombre rectangle = 7, nombre de traits = 4
0.8300 ➔ k = 8, nombre rectangle = 7, nombre de traits = 5
0.8300 ➔ k = 2, nombre rectangle = 7, nombre de traits = 6
0.8300 ➔ k = 8, nombre rectangle = 7, nombre de traits = 6

0.8400 ➔ k = 4, nombre rectangle = 5, nombre de traits = 9
0.8400 ➔ k = 4, nombre rectangle = 7, nombre de traits = 9

0.8400 ➔ k = 4, nombre rectangle = 5, nombre de traits = 10
0.8400 ➔ k = 4, nombre rectangle = 7, nombre de traits = 10

0.8400 ➔ k = 4, nombre rectangle = 5, nombre de traits = 11
0.8400 ➔ k = 4, nombre rectangle = 7, nombre de traits = 11

0.8400 ➔ k = 4, nombre rectangle = 5, nombre de traits = 12
0.8400 ➔ k = 4, nombre rectangle = 7, nombre de traits = 12

0.8400 ➔ k = 4, nombre rectangle = 5, nombre de traits = 13
0.8400 ➔ k = 4, nombre rectangle = 7, nombre de traits = 13

0.8400 ➔ k = 4, nombre rectangle = 5, nombre de traits = 14
0.8400 ➔ k = 4, nombre rectangle = 7, nombre de traits = 14

0.8400 ➔ k = 4, nombre rectangle = 5, nombre de traits = 15
0.8400 ➔ k = 4, nombre rectangle = 7, nombre de traits = 15

Meilleur résultat :
0.8500 ➔ k = 4, nombre rectangle = 7, nombre de traits 21

On peut en déduire, que le nombre de traits influe très peu sur l'optimisation des résultats, mais cependant il influe quand même, car c'est avec 21 traits que nous obtenons le meilleur résultat qui est de **85 %.**

En général pour obtenir les meilleurs résultats (entre 80 % et 85 %) il faut : **nombre rectangle** = 7 et 1 <= **k** <= 10 si **nombre de traits** inférieurs à 9. Puis à partir d'un **nombre de traits** supérieur ou égal à 9, nous avons les meilleurs résultats pour **nombre rectangle** = 7 ou 5 et 1 <= k <= 10

d. Somme

De même que pour la combinaison de produit, la solution que j'ai trouvée pour calculer la précision est de tester chaque combinaison de k, le nombre de rectangle (classifieur 2) et de trait (classifieur 1) grâce à 3 boucles « for » une pour chaque variable. Ce qui me retourne un tableau à 3 dimensions contenant la précision en fonction des 3 variables. Cependant, je n'ai pas trouvé de moyen d'afficher simplement ce tableau 3d vers 2 d. Je vais donc décrire les résultats que j'ai obtenus.

Fonction matlab : **optimizeAccuracy**

Résultats entre 80 % et 84 % :

k = 2, nombre rectangle = 4/5/7, nombre de traits = 5/4
k = 2, nombre rectangle = 4/5/6/7/8/9/10/11, nombre de traits = 6/7/8
k = 2, nombre rectangle = 4/5/7/8/11, nombre de traits = 9
k = 2, nombre rectangle = 4/5/6/7/8/9/10/11, nombre de traits = 10/11/12/13

k = 2, nombre rectangle = 4/5/6/7/8/9/10/11, nombre de traits > 13
2 <= k <= 8, nombre rectangle = 4/5/6/7/8/9/10/11, nombre de traits > 13

Meilleur résultat :
0.8400 =➔k = 4, nombre rectangle = 5, trait = 19/20/21

On peut en déduire, que le nombre de traits influe très peu sur l'optimisation des résultats, mais cependant il influe quand même, car c'est avec 19 traits que nous obtenons le meilleur résultat qui est de **84 %**.

Autrement, si le nombre de traits inférieur ou égal à 13, c'est avec k = 2 et nombre de rectangles compris entre 4 et 11 que nous avons les meilleurs résultats.
Si le nombre de traits supérieur à 13 c'est avec k = 2 et nombre de rectangle compris entre 4 et 11 que nous avons les meilleurs résultats, mais également avec k compris entre 2 et 8 pour un nombre de rectangles compris entre 4 et 11.

c. Conclusion

En conclusion, c'est avec le produit des 2 classifieurs que nous obtenons le meilleur résultat pour **k = 4, nombre rectangle = 7, nombre de traits 21** qui est de 85 %.

Sans combinaison c'est le classifieur 1 qui est le plus efficace.

Globalement, le classifieurs2, somme et produit de classifieurs ont un k optimal qui même s'il n'est pas identique à chaque fois (classifieur2, produit, somme) est généralement compris entre 2 et 8.

Nous avons vu par l'analyse du classifieurs 1 que le nombre de traits influence peu les résultats cependant lorsque l'on fait la combinaison de classifieur le nombre de traits semble devenir plus important.

Les résultats de la combinaison de classifieurs permettent de valider l'idée trouvé sur le classifieur 2 qui est qu'à partir d'un certain nombre de rectangles la précision diminue énormément.

Amélioration possible, il serait bien de tester les classifieurs et combinaisons avec un nombre de k, de rectangle et de trait beaucoup plus important pour en déduire un résultat définitif. Mais cela

demande beaucoup de temps de calcul ou bien l'utilisation de GPU. Nous pouvons également tenter d'améliorer les résultats en trouvant d'autres caractéristiques intéressantes sur les images.

Choisir les paramètres optimaux pour le classifieur 1 et 2 individuellement n'implique pas qu'on obtienne de meilleurs résultats sur la combinaison.

Globalement, les résultats que j'ai obtenus sont ceux attendus. Qui est plus les vecteurs caractéristiques d'entrainement sont bien représentant de l'ensemble des l'images à traiter (ici des chiffres) plus le résultat de la classification sera bon. Cependant je n'espérais pas trouver un nombre de k voisin optimal ou plus efficace pour chaque méthode.