Reconnaissance Optique de Caractères via K-Plus-Proches-Voisins

RAGOT-RAILLAT Antoine & LELAURE Arnaud

Projet de Mathématiques et Calcul Scientifique (MaCs)
Université de Bourgogne Europe

Année Universitaire 2025-2026

Résumé

Ce rapport détaille la conception et l'implémentation algorithmique d'un moteur de Reconnaissance Optique de Caractères (OCR) développé intégralement en Python. S'écartant des approches modernes de type « boîte noire » (réseaux de neurones convolutifs), notre étude privilégie une modélisation mathématique explicite et l'utilisation de l'algorithme des k-plus-proche-voisin. Nous passons par une normalisation d'un grand nombre d'images afin d'en extraire des caractéristiques chiffrées pour permettre la classification de **nouvelles** images grace à notre **modèle** utilisant le principe d'apprentissage supervisé.

Table des matières

1 Introduction 4

2 Fonctionnement général 4
- 2.1 Général 4
- 2.2 Caractéristiques des images 4
- 2.3 Classification des données (K-NN) 4
- 2.4 Les différentes étapes 5
- 2.5 Organisation des fichiers du projet 5

3 Normalisation des Images 5
- 3.1 Filtrage et Binarisation 5
- 3.2 Correction de l'Inclinaison par régression linéaire 5
- 3.3 Recadrage et Redimensionnement 6
- 3.4 Seconde Binarisation (Nettoyage) 6
- 3.5 Illustration du processus de Normalisation 6

4 Caractérisation et Extraction des données 7
- 4.1 Le Pavage Spatial (Zoning dynamique) 7
- 4.2 Analyse Topologique par Intersections 7
- 4.3 Rapport d'Aspect (Ratio) 7
- 4.4 Synthèse du Vecteur de Caractéristiques 7

5 Classification par les K-Plus-Proches-Voisins (K-NN) 8
- 5.1 Constitution de la Base de Référence (Mémoire du Modèle) 8
- 5.2 Le Principe du K-NN : Analogie géométrique en 2D 8
- 5.3 La Mesure de Similarité : Distance Euclidienne en Dimension 52 9
- 5.4 Prédiction par Vote Majoritaire 9

6 Autres Algorithmes 9
- 6.1 Standardisation du Jeu de Données : Inversion de Couleur 10
- 6.2 Automatisation de la Capture : Création du Dataset iPad 10
- 6.3 Indexation Automatisée : Fichier de Renommage 10
- 6.4 Optimisation du Temps d'Exécution : Pré-calcul de la Base 10

7 Implémentation et Normes de Développement 11
- 7.1 Environnement et Bibliothèques 11
- 7.2 Conventions et Typage Strict (PEP 8 et PEP 484) 11

8 Exemple d'utilisation des programmes 11
- 8.1 Création d'un Dataset (`creation_ds_Ipad.py`) 11
- 8.2 Calcul de la base de données (`db.py`) 12
- 8.3 Test pour obtenir le taux de réussite (`train.py`) 13
- 8.4 Programme principal de prédiction (`main.py`) 14

9 Résultats, Expérimentation et Analyse 15
- 9.1 Processus d'évaluation (`train.py`) et Apprentissage Actif 15
- 9.2 Gestion et Optimisation des Hyperparamètres 16
  - 9.2.1 Standardisation de l'Espace (28x28 et Lanczos) 16
  - 9.2.2 Le Paradoxe de la Double Binarisation 16
  - 9.2.3 L'Évolution de la Caractérisation : Vers le Zoning 7×7 16
  - 9.2.4 Ajustement Dynamique du Paramètre K 17

9.3 Optimisation du Code et Complexité Algorithmique 17
9.4 Nos Résultats Obtenus et Limites du Modèle 17
9.5 Résultat de notre étude 18
9.5.1 Dataset : MNIST 18
9.5.2 Dataset : Chars74K 18
9.5.3 Dataset : iPad (Personnalisé) 18
9.5.4 Synthèse et Enseignements de l'Étude 18

10 Conclusion du Projet 19
10.1 Perspectives et Réflexions : Un Win Rate de 100 % est-il possible ? 19
10.1.1 L'Erreur Irréductible et l'Ambiguïté Humaine 19
10.1.2 Les Limites Structurelles du modèle K-NN 19
10.2 Mot de la fin 20

1 Introduction

La classification des données non structurées, telles que les images de caractères manuscrits, constitue un problème complexe en calcul scientifique et en intelligence artificielle. Le défi majeur réside dans les petites variations d'un même caractère (par exemple, le chiffre « 8 »), qui peuvent subir des transformations arbitraires (translation, rotation, ...) et des déformations non linéaires induites par le scripteur.

L'objectif de ce projet est de concevoir un système de classification invariant à ces transformations, reposant exclusivement sur des outils mathématiques fondamentaux. Plutôt que de confier l'extraction des motifs à un algorithme d'apprentissage profond, nous avons réalisé un ensemble de programmes en Python permettant la classification de données chiffrées à l'aide d'un algorithme des k-plus-proches-voisins.

Ce rapport s'articule autour des étapes mathématiques et algorithmiques de la chaîne de traitement : la normalisation de nos images, l'extraction des caractéristiques, la classification via l'algorithme des K-Plus-Proches-Voisins (K-NN) et l'étude statistique des performances sur un jeu de données généré en conditions réelles.

2 Fonctionnement général

2.1 Général

Notre approche repose sur un modèle traitant différents chiffres sous forme d'image. Afin d'optimiser les calculs et d'accroître la précision, les images subissent des transformations tels que leurs normalisations et le cropping, fonctions qui permettent de comparer des images qui seront systématiquement identiques en terme de taille et d'orientation. Puis, chaque image est définie par plusieurs caractéristiques numériques (dimensions) qui permettent de placer chacune des images dans un espace, permettant ensuite de réaliser des prédictions via l'algorithme des K-Plus-Proches-Voisins.

2.2 Caractéristiques des images

Pour définir ces fameuses dimensions, nous avons écarté l'analyse pixel par pixel de l'image, car elle est trop fragile face aux variations d'écriture. À la place, notre programme résume chaque chiffre à l'aide de trois grands critères :

— **La distribution spatiale (Zoning)** : l'image est découpée en une grille régulière pour analyser la répartition de l'encre zone par zone, ce qui rend la lecture indépendante de l'épaisseur du trait.

— **La topologie** : le comptage des intersections entre le tracé et les axes centraux permet de capturer la complexité structurelle du chiffre (la présence de boucles ou de lignes continues).

— **La morphologie globale** : le calcul du ratio entre la largeur et la hauteur différencie rapidement les chiffres naturellement étroits (comme le « 1 ») des chiffres plus larges (comme le « 0 » ou le « 8 »).

2.3 Classification des données (K-NN)

L'algorithme des K-Plus-Proches-Voisins (K-NN) constitue le cœur décisionnel du projet. Au lieu de suivre des règles de calcul complexes, il compare simplement chaque nouveau tracé aux milliers d'exemples qu'il possède déjà en réserve. En plaçant le chiffre inconnu dans notre espace mathématique, il identifie ses voisins les plus « proches » (ceux qui lui ressemblent le plus) et laisse la majorité l'emporter : si la plupart des voisins sont des « 4 », le tracé sera identifié comme un « 4 ».

2.4 Les différentes étapes

1. **Prétraitement** : Inversion des couleurs (fond blanc/encre noir), et binarisation des images.
2. **Normalisation** : Orientation de l'image en mettant à la vertical la droite de regression de cette dernière, recadrage (cropping) et redimensionnement en 28 par 28 pixels.
3. **Indexation** : On extrait les caractéristiques de milliers d'images que l'on stocke dans un fichier csv (database).
4. **Reconnaissance** : Pour cette étape on a la capture d'un chiffre, puis sa normalisation et calcul de vecteur et enfin le vote selon K-NN.

2.5 Organisation des fichiers du projet

L'architecture logicielle est modulaire :

— `normalisation.py` : Fonctions de binarisation, régression linéaire et rotation.
— `caracterisation.py` : Calcul du zoning, des intersections et du ratio.
— `db.py` : Gestion des entrées/sorties du fichier CSV (Base de données).
— `knn.py` : Implémentation de la métrique de distance et de l'algorithme de vote.
— `main.py` : Point d'entrée pour la prédiction unitaire.
— `Autres` : Ces derniers ne sont exécutés qu'une seule fois lors du tout premier lancement de l'algorithme. On retrouve `colorInversion.py`, `creation_ds_Ipad.py`, `rename.py` et `train.py`.

3 Normalisation des Images

L'écriture manuscrite présente une forte variabilité. L'objectif de la phase de normalisation est de lisser ces disparités afin d'obtenir des images standardisées, de dimensions fixes (28×28 pixels), pouvant être comparées de manière fiable par notre algorithme.

3.1 Filtrage et Binarisation

La première étape consiste à transformer l'image brute (souvent en couleurs RVB) en une matrice purement binaire. Cela permet de réduire l'information à son essentiel : l'encre et le fond.

Pour chaque pixel, composé de trois canaux (Rouge, Vert, Bleu), nous calculons sa luminance moyenne. Si cette moyenne est inférieure à un seuil défini expérimentalement (ici 128, soit la moitié de l'intensité maximale 255), le pixel est considéré comme de l'encre.

Soit un pixel de coordonnées (x, y) et de valeurs (R, V, B). La valeur binarisée $B_{x,y}$ est calculée ainsi :

$$B_{x,y} = \begin{cases} (0,0,0) \text{ (Noir)} & \text{si } \frac{R+V+B}{3} < 128 \\ (255,255,255) \text{ (Blanc)} & \text{sinon} \end{cases} \quad (1)$$

3.2 Correction de l'Inclinaison par régression linéaire

Le trait d'un caractère est naturellement soumis à l'inclinaison propre au scripteur. Pour redresser le caractère, nous utilisons une méthode basée sur la régression linéaire.

Nous cherchons la droite d'équation $X = aY + b$ qui représente l'axe vertical moyen du tracé (donc en fonction de la répartition de l'encre). Nous isolons les coordonnées (x_i, y_i) des pixels d'encre. Le coefficient directeur a est estimé par la méthode des moindres carrés, en calculant le rapport entre la covariance croisée et la variance sur l'axe Y :

$$a = \frac{\text{Cov}(X,Y)}{\text{Var}(Y)} = \frac{\sum_{i=1}^{N}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{N}(y_i - \bar{y})^2} \quad (2)$$

Une fois cette pente a obtenue, nous déterminons l'angle d'inclinaison θ par rapport à la verticale grâce à la fonction arc-tangente :

$$\theta = \arctan(a) \tag{3}$$

L'image subit alors une rotation d'angle $-\theta$ pour garantir que le chiffre est vertical.

3.3 Recadrage et Redimensionnement

Une fois l'image redressée, il faut la mettre à la bonne taille. L'algorithme cherche d'abord les bords extrêmes du chiffre (les pixels noirs les plus hauts, les plus bas, les plus à gauche et les plus à droite) pour couper tout l'espace blanc inutile autour : c'est le recadrage (*cropping*).

Ensuite, cette image découpée est agrandie ou rétrécie (via l'application d'un filtre de Lanczos) pour rentrer dans une grille de 28×28 pixels. Pour ne pas déformer le chiffre, on conserve strictement ses proportions : son côté le plus grand touchera les bords du carré.

3.4 Seconde Binarisation (Nettoyage)

Lors du redimensionnement, l'ordinateur crée automatiquement des pixels gris sur les bords du tracé pour lisser l'image (phénomène d'anti-aliasing). Notre algorithme ayant besoin de valeurs strictement noires ou blanches, nous repassons une deuxième fois notre filtre de binarisation (avec le même seuil de 128) sur l'image finale. Cela garantit un tracé net et parfaitement binaire avant l'extraction des caractéristiques.

3.5 Illustration du processus de Normalisation

Afin de valider notre modèle de redressement et de standardisation, nous avons développé un outil de visualisation utilisant la bibliothèque *Matplotlib*. La Figure 1 illustre les trois états majeurs de la matrice lors de son passage dans notre algorithme.

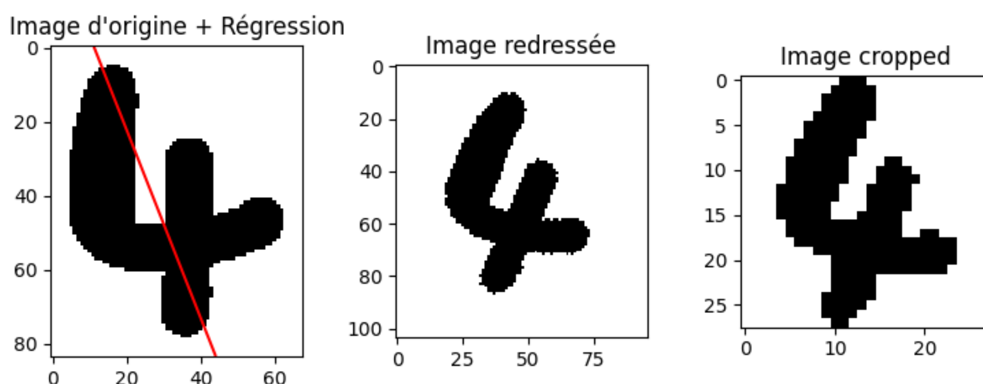


FIGURE 1 – Étapes de normalisation appliquées à un chiffre manuscrit.
À gauche : Matrice binarisée avec superposition de la droite de régression linéaire minimisant la variance sur l'axe horizontal.
Au centre : Matrice redressée après rotation isométrique d'angle $-\theta$.
À droite : Matrice finale après recadrage (*cropping*), redimensionnement proportionnel avec filtre Lanczos, et centrage dans une grille stricte de 28×28 pixels.

Comme le démontre cette figure, l'algorithme parvient à transformer un tracé initialement incliné et mal centré en une caractéristique géométrique standardisée, prête à être découpée par notre fonction de pavage spatial (*Zoning*).

4 Caractérisation et Extraction des données

L'algorithme de classification ne peut pas comparer efficacement deux images de 784 pixels (28×28) en les superposant pixel par pixel. En effet, un léger décalage ou une variation dans l'épaisseur du trait fausserait totalement le calcul d'erreur géométrique. L'objectif de la caractérisation est de « résumer » l'image en une liste de valeurs numériques (un vecteur) qui décrit la forme globale du caractère de manière robuste.

4.1 Le Pavage Spatial (Zoning dynamique)

La méthode d'extraction principale est le pavage spatial, ou *Zoning*. L'image est découpée en une grille régulière de 7×7 blocs. Pour gérer les dimensions d'images qui ne sont pas des multiples de 7, nous utilisons une subdivision dynamique basée sur une répartition linéaire des limites de blocs. Cela garantit qu'aucun pixel n'est oublié lors du découpage.

Pour chaque bloc, nous calculons la densité relative d'encre par rapport à la quantité totale d'encre du chiffre. Soit N_{total} le nombre total de pixels noirs dans l'image entière, et B_k l'ensemble des pixels appartenant au k-ième bloc de la grille. La densité d_k du bloc est :

$$d_k = \frac{\sum_{(x,y)\in B_k} \text{pixel_noir}(x, y)}{N_{total}}$$

L'algorithme génère ainsi 49 valeurs comprises entre 0 et 1, cartographiant la répartition spatiale de la masse d'encre de manière indépendante de l'épaisseur du tracé.

4.2 Analyse Topologique par Intersections

Pour enrichir le vecteur de caractéristiques, nous mesurons la complexité du tracé en comptant les transitions entre le fond et l'encre le long des axes médians (horizontal et vertical).

Afin que ces mesures (qui sont des entiers naturels) ne pèsent pas démesurément face aux densités du zoning dans le calcul de distance, elles sont normalisées par un facteur de 5.0. Ce coefficient permet de ramener le nombre d'intersections dans un ordre de grandeur comparable aux autres caractéristiques (proche de l'intervalle $[0, 1]$). Les valeurs extraites sont donc :

$$I_x = \frac{\text{nb_intersections_horizontales}}{5} \quad \text{et} \quad I_y = \frac{\text{nb_intersections_verticales}}{5}$$

4.3 Rapport d'Aspect (Ratio)

Enfin, nous intégrons une mesure de la morphologie globale du caractère : le ratio largeur/-hauteur. Cette caractéristique permet de distinguer facilement des chiffres aux proportions très différentes, comme le « 1 » (très étroit) et le « 0 » ou le « 8 » (plus larges/carrés).

$$R = \frac{\text{largeur_image}}{\text{hauteur_image}}$$

4.4 Synthèse du Vecteur de Caractéristiques

À l'issue de ces étapes, chaque image est transformée en un vecteur mathématique $X \in \mathbb{R}^D$ de dimension $D = 52$. Ce vecteur concatène les 49 densités de zones, les 2 valeurs d'intersections normalisées et le ratio d'aspect. C'est ce vecteur final qui servira de signature unique pour la classification métrique.

5 Classification par les K-Plus-Proches-Voisins (K-NN)

Une fois l'image caractérisée sous forme de vecteur numérique à 52 dimensions, il faut déterminer à quel chiffre elle correspond. Pour cela, nous utilisons l'algorithme des K-Plus-Proches-Voisins (K-NN). C'est un algorithme d'apprentissage supervisé dit « paresseux » : il n'y a pas de phase d'entraînement mathématique complexe. L'algorithme conserve simplement en mémoire une base de données de référence et procède à des comparaisons géométriques directes.

5.1 Constitution de la Base de Référence (Mémoire du Modèle)

Avant de pouvoir deviner une image inconnue, l'algorithme a besoin d'une mémoire constituée d'exemples connus. Lors de la phase de création de la base de données, notre programme traite des centaines d'images d'entraînement et sauvegarde les résultats dans un fichier texte (au format `.csv`).

Pour chaque image traitée, une ligne est ajoutée dans ce fichier. Cette ligne contient un *tuple* regroupant :

1. Les 52 valeurs mathématiques extraites de l'image (49 zones + 2 intersections + 1 ratio).
2. **L'étiquette réelle de l'image** (le vrai chiffre de 0 à 9). Cette étiquette est déduite automatiquement par notre script grâce au nom du fichier lu (par exemple, le fichier `3_dataset_12.png` indique que l'image est un « 3 »).

C'est la présence de ce vrai chiffre à la fin de chaque vecteur qui rend l'apprentissage « supervisé » : l'algorithme connaît les réponses des images de sa mémoire.

5.2 Le Principe du K-NN : Analogie géométrique en 2D

Pour bien comprendre comment l'algorithme prend sa décision de manière géométrique, prenons l'exemple le plus simple possible : un espace à seulement 2 dimensions (un graphique avec un axe X et un axe Y).

Imaginons que nous avons une mémoire contenant des données connues, séparées en deux catégories : la **Classe A** (les ronds bleus) et la **Classe B** (les carrés rouges). Ces points sont placés sur le graphique selon leurs caractéristiques. Naturellement, les objets similaires ont tendance à se regrouper pour former des « nuages » de points.

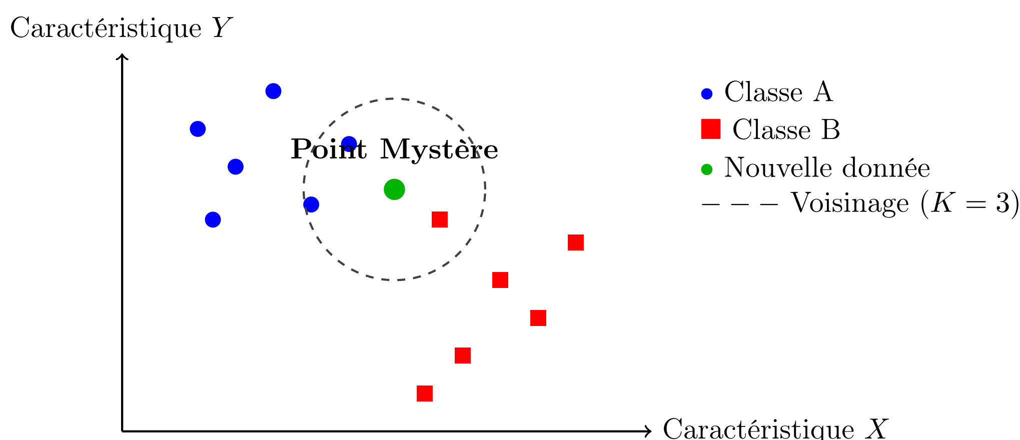


FIGURE 2 – Illustration du fonctionnement du K-NN. Le point mystère cherche ses 3 plus proches voisins pour déterminer sa classe.

Lorsqu'une nouvelle donnée inconnue arrive (le **Point Mystère** vert sur la Figure 2), l'algo-

rithme doit décider s'il s'agit d'un point de la Classe A ou de la Classe B. Le processus suit trois étapes :

1. **Calcul des distances :** L'algorithme calcule la distance géométrique entre le point mystère et absolument tous les autres points de la mémoire.
2. **Sélection du voisinage :** Si nous définissons un paramètre $K = 3$, l'algorithme trace un cercle imaginaire qui s'agrandit jusqu'à capturer exactement les 3 points les plus proches (le cercle en pointillés sur la Figure 2).
3. **Vote majoritaire :** Dans ce cercle, nous comptons 2 ronds bleus et 1 carré rouge. La majorité l'emporte : l'algorithme déduit que le point mystère appartient à la Classe A.

Dans notre véritable projet d'OCR, le principe mathématique reste strictement identique, à la différence près que notre graphique ne possède pas 2 axes, mais 52 axes (les 52 caractéristiques extraites de notre image).

5.3 La Mesure de Similarité : Distance Euclidienne en Dimension 52

Pour évaluer à quel point l'image mystère (notre vecteur X) ressemble à une image de la mémoire (notre vecteur Y), nous mesurons la distance qui les sépare.

Nous utilisons la norme $\mathcal{L}_2$, communément appelée distance Euclidienne :

$$d(X, Y) = \sqrt{\sum_{i=1}^{52} (X_i - Y_i)^2}$$

Dans notre code, ce calcul est optimisé grâce à la fonction `np.linalg.norm(X - Y)` de la bibliothèque *NumPy*. L'algorithme calcule cette distance entre notre image mystère et **absolument toutes les lignes** de notre fichier `csv`, puis il trie ces résultats de la distance la plus petite (le point le plus proche) à la plus grande.

5.4 Prédiction par Vote Majoritaire

Si nous ne prenions que le voisin le plus proche (soit $K = 1$), notre modèle serait vulnérable. En effet, si l'image de la mémoire la plus proche s'avère être une donnée aberrante (une erreur d'écriture), l'algorithme se tromperait bêtement.

Pour rendre le système robuste, nous isolons les K voisins les plus proches (dans notre implémentation finale, nous utilisons généralement $K = 5$ ou $K = 7$). Le processus final se déroule ainsi :

1. L'algorithme récupère les vraies étiquettes (les chiffres de 0 à 9) des K vecteurs les plus proches.
2. Il compte les occurrences de chaque chiffre dans ce voisinage.
3. L'étiquette retenue pour la prédiction finale est le chiffre qui a obtenu le plus grand nombre de votes.

Ce système de vote agit comme un filtre spatial, permettant à notre programme d'ignorer les petites anomalies de la base de données et de prendre une décision statistiquement très fiable.

6 Autres Algorithmes

Le succès d'un modèle d'apprentissage supervisé repose autant sur la qualité des algorithmes de calcul que sur la structure des données. Pour ce projet, nous avons développé une série d'outils utilitaires permettant de transformer des données brutes (images réelles ou bases de données externes) en un format exploitable par notre moteur K-NN.

6.1 Standardisation du Jeu de Données : Inversion de Couleur

Le jeu de données de référence *MNIST*, largement utilisé en recherche, présente initialement des caractères blancs sur un fond noir. Or, dans une situation réelle (encre sur papier), nous travaillons avec des caractères noirs sur fond blanc.

Pour assurer la cohérence entre notre base d'apprentissage et nos tests en conditions réelles, nous avons conçu le script `colorInversion.py`. Ce dernier parcourt l'arborescence du dataset et applique une transformation bit à bit via la bibliothèque `PIL.ImageOps.invert`. Pour un pixel d'intensité p sur 8 bits, la nouvelle valeur p est :

$$p' = 255 - p \tag{4}$$

Cette étape garantit que la binarisation (seuil à 128) fonctionnera de manière identique sur n'importe quelle source d'image.

6.2 Automatisation de la Capture : Création du Dataset iPad

Pour tester la robustesse de notre OCR, nous avons créé notre propre jeu de données en photographiant une grille de chiffres manuscrits. Le script `creation_ds_Ipad.py` utilise la bibliothèque *OpenCV* pour automatiser le découpage de cette grille en images individuelles. Le processus suit quatre étapes clés :

1. **Seuillage :** L'image est binarisée pour isoler les formes.
2. **Détection de contours :** Via la fonction `findContours`, l'algorithme identifie chaque tracé indépendant sur la page.
3. **Tri spatial :** C'est l'étape la plus critique. Pour attribuer le bon label (0, 1, 2...) à chaque image découpée, le script trie les contours par colonnes. En divisant la coordonnée X par la largeur d'une colonne, nous regroupons les chiffres de 0 à 9.
4. **Extraction et Sauvegarde :** Chaque contour valide (suffisamment grand pour ne pas être du bruit) est extrait avec une marge de sécurité, puis sauvegardé dans le dossier correspondant à son chiffre.

Ce script nous a permis de générer rapidement des centaines d'échantillons de test sans intervention manuelle fastidieuse.

6.3 Indexation Automatisée : Fichier de Renommage

Le bon fonctionnement de notre phase d'indexation repose sur une convention de nommage stricte. Le script `rename.py` a été conçu pour normaliser le nom de chaque fichier du dataset sous la forme : `label_dataset_index.png`.

L'utilité est double :

— **Extraction facilitée :** Notre fonction `indexFile` peut extraire le *label* (le chiffre réel) simplement en lisant le premier caractère du nom du fichier.

— **Unicité :** Cela évite les conflits de noms lors de la fusion de plusieurs sources de données (par exemple, mélanger des images de MNIST avec nos propres captures iPad).

6.4 Optimisation du Temps d'Exécution : Pré-calcul de la Base

Enfin, le fichier `db.py` remplit une fonction d'optimisation cruciale. Extraire 52 caractéristiques sur des milliers d'images est une opération lourde en calcul. Plutôt que de recalculer ces vecteurs à chaque nouvelle prédiction, le script parcourt une fois pour toutes le dossier de *train* et exporte les vecteurs dans `database.csv`.

Lors de l'utilisation réelle de l'OCR, l'algorithme ne traite que l'image mystère. Pour la comparaison, il se contente de charger le fichier CSV en mémoire (opération quasi instantanée),

ce qui permet d'obtenir une prédiction en quelques millisecondes, rendant le système utilisable en temps réel.

7 Implémentation et Normes de Développement

La fiabilité d'un calcul scientifique repose grandement sur la robustesse du code qui l'exécute. L'intégralité de ce projet a été développée en langage Python, en respectant les standards de l'industrie logicielle.

7.1 Environnement et Bibliothèques

Afin de ne pas réinventer la roue sur des opérations bas niveau tout en gardant la maîtrise de notre algorithme d'intelligence artificielle, nous nous sommes appuyés sur des bibliothèques reconnues :

— **NumPy** : Cœur mathématique du projet, utilisé pour la vectorisation des calculs et l'optimisation de la distance Euclidienne (norme $\mathcal{L}_2$).
— **Pillow (PIL)** : Manipulation matricielle des images et application du filtre de rééchantillonnage de Lanczos lors du recadrage.
— **OpenCV (cv2)** : Utilisé exclusivement dans nos scripts utilitaires pour la détection de contours et l'extraction automatisée de notre jeu de données iPad.
— **Matplotlib** : Génération des graphiques analytiques et tracé des droites de régression pour la validation visuelle de la normalisation.
— **Bibliothèques natives (`csv`, `os`, `math`)** : Gestion des entrées/sorties et indexation de la base de données.

7.2 Conventions et Typage Strict (PEP 8 et PEP 484)

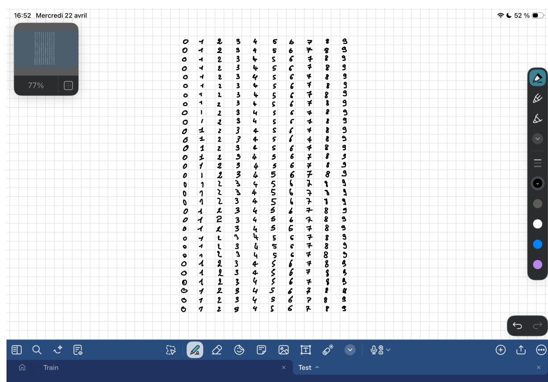
Le code source a été rédigé en stricte conformité avec la **PEP 8**, garantissant une lisibilité et une structure standardisée. Plus important encore pour un projet mathématique, nous avons appliqué la **PEP 484** en typant explicitement toutes nos fonctions avec l'analyseur statique *Pylance* configuré en mode basic. Ainsi, la signature de chaque fonction décrit mathématiquement ses entrées et sorties (par exemple, une fonction retournant un vecteur de caractéristiques est strictement typée `-> tuple[list[float], str]`). Cela a permis d'éliminer les erreurs d'incohérence de types en amont de l'exécution et de garantir l'intégrité des flux de données.

8 Exemple d'utilisation des programmes

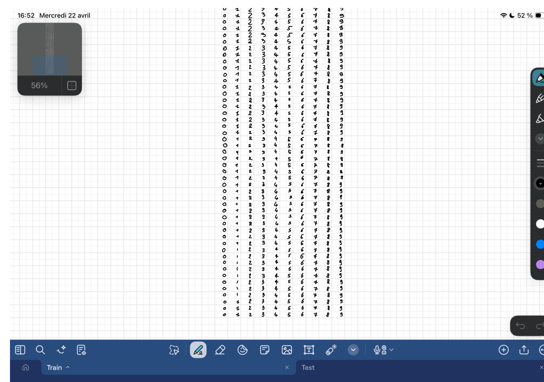
Afin de démontrer l'opérabilité de notre solution, cette section détaille le flux de travail typique d'un utilisateur, de la préparation des données jusqu'à la prédiction finale. Notre architecture modulaire permet d'exécuter chaque étape indépendamment.

8.1 Création d'un Dataset (`creation_ds_Ipad.py`)

Initialement, le modèle a été entraîné sur des bases de données publiques telles que `MNIST (Chiffre manuscrit)` ou encore `Chars74K (Chiffre numérique)`. Mais pour tester le modèle sur de nouvelles écritures que nous sommes aller récolter, nous utilisons notre script d'automatisation. L'utilisateur fournit une image contenant une grille de chiffres manuscrits (par exemple, 10 lignes allant de 0 à 9).



(a) Grille utilisé par la suite pour testé le modèle



(b) Grille pour entraîné le modèle

FIGURE 3 – Capture d'écran de l'application Goodnote ou à été saisie les chiffres pour crée notre propre dataset

Ensuite, le script `creation_ds_Ipad.py` ouvre l'image avec *OpenCV*. Il épaissit légèrement l'encre pour recoller les chiffres dont les traits sont coupés, puis il repère chaque forme. En triant ces formes d'abord ligne par ligne, puis de gauche à droite, l'algorithme découpe chaque chiffre et le range automatiquement dans le bon dossier (de `0` à `9`) tout en le renommant de manière standardisée `n_dataset_i` (n -> le chiffre écrit et i -> le numéro du chiffre)

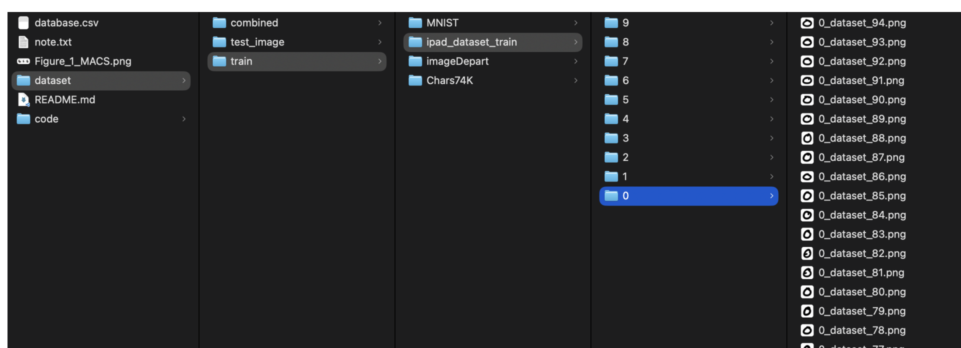


FIGURE 4 – Architecture de dossier et contenu du dossier du dataset crée

Maintenant les autres fichiers du projet vont pouvoir utiliser ces images.

8.2 Calcul de la base de données (`db.py`)

Avant toute prédiction, le modèle doit constituer sa mémoire. L'exécution du script `db.py` (via la fonction `create_db`) parcourt l'intégralité du dossier d'entraînement (*Train*). Pour chaque image, le programme effectue le pipeline complet : binarisation, redressement, recadrage, et extraction du vecteur à 52 dimensions. Afin de ne pas répéter ces calculs coûteux, les vecteurs générés, associés à leur étiquette réelle (déduite du nom du fichier), sont sérialisés et sauvegardés dans le fichier `database.csv`. Ce fichier devient la matrice de référence pour notre K-NN.

Le Listing 1 illustre la logique d'extraction : la fonction `indexFile` est appelée itérativement pour constituer la mémoire du modèle.

Listing 1 – Code permettant de générer la base de données

```
1  # Parcourt le dossier pour lister tous les chemins d'accès des images
2  def indexation(path: str) -> list[str]:
3      files = []
4      ...
```

```
    return files

# Applique le pipeline (normalisation + extraction) sur une seule image
def indexFile(path: str, taille_grille: int = 4) -> tuple[float, ...]:
    image = normalisation(path)
    data = caracterisation(image, taille_grille)
    return data

# Construit la memoire du modele en associant chaque vecteur a son etiquette
def create_db(path: str, taille_grille: int) -> list[tuple[list[float], str]]:
    tab = indexation(path)
    data_base = []
    for fichier in tab:
        data_base.append(
            (list(indexFile(fichier, taille_grille)), fichier.split("/")[-1])
        )
    return data_base


# Sauvegarde la base de donnees dans un fichier CSV (persistance)
def creer_journal(path: str, db: list[tuple[list[float],str]],  taille_grille:
    int) -> None:
    ...

# Permet de charger en mémoire la base de données csv qui a déjà été générée
def charger_journal(path: str) -> list[tuple[list[float], str]]:
    db = []
    ...
    return db
```

8.3 Test pour obtenir le taux de réussite (train.py)

Pour évaluer objectivement les performances, le script **train.py** confronte le modèle à un jeu de données de test (strictement séparé de la base d'entraînement). Le script charge database.csv en mémoire, puis itère sur toutes les images du dossier de test. Pour chaque image, il calcule les distances euclidiennes avec la base entière et extrait le vote majoritaire des K plus proches voisins. Il compare ensuite cette prédiction au vrai label et incrémente un compteur de victoires. À l'issue du processus, le script affiche dans le terminal le *Win Rate* global et liste les erreurs de prédiction pour permettre une analyse de la matrice de confusion.

Listing 2 – Code permettant de d'entrainer la reconnaissance qui contient le calcul du winrate

```
def main() -> None:
    db_csv = "MacsOCR/database.csv"
    dataset = "ipad_dataset"
    win = 0
    k = 5
    taille_grille = 7
    affichage = True

    if os.path.exists(db_csv):
        database = db.charger_journal(db_csv)

    for i in range(10):
        for j in range(30):
            file = f"MacsOCR/dataset/test_image/{dataset}/{i}/{i}_dataset_{j}.
    png"

            mon_image = (list(db.indexFile(file, taille_grille)), file.split("/"
    )[-1])
```

```python

            distances = knn.calcul_distance_total(database, mon_image)
            prediction = knn.find(distances, k, False)

            if prediction is not None:
                if int(prediction) == int(i):
                    win += 1
                else:
                    if affichage:
                        print(f"dataset : {dataset} | image : {file.split('/')[-1]} | prediction : {prediction} | réponse : {i}")

    print("\ndataset :", dataset)
    print("winrate :", float(win / ((i + 1) * (j + 1))))
```

Les premières variables paramètre le modèle :

— `db_csv` : indique le chemin de la base de données au format csv

— `dataset` : indique le choix du dataset à utiliser (attention il faut supprimer le csv si changement de dataset pour que le bon soi généré

— `k` : indique le k à utiliser pour l'algorithme des k-plus-proche-voisin

— `taille_grille` : correspond a la taille de la grille à utilisé pour le zoning durant la normalisation

— `affichage` : booléen qui affiche ou non les information concernant les mauvaises prédictions

Affichage console :

```
dataset : ipad_dataset | image : 7_dataset_10.png | prediction : 9 | réponse : 7
dataset : ipad_dataset | image : 7_dataset_11.png | prediction : 4 | réponse : 7
dataset : ipad_dataset | image : 8_dataset_1.png | prediction : 9 | réponse : 8
dataset : ipad_dataset | image : 9_dataset_13.png | prediction : 3 | réponse : 9
dataset : ipad_dataset | image : 9_dataset_16.png | prediction : 3 | réponse : 9
dataset : ipad_dataset | image : 9_dataset_27.png | prediction : 3 | réponse : 9
dataset : ipad_dataset | image : 9_dataset_28.png | prediction : 0 | réponse : 9

dataset : ipad_dataset
winrate : 0.9

temps d'execution : 1.7630112171173096
```

Figure 5 – Affichage du winrate et des mauvaises prédictions

8.4 Programme principal de prédiction (`main.py`)

C'est le point d'entrée de l'utilisateur final. Dans `main.py`, l'utilisateur renseigne le chemin d'une image isolée (le « chiffre mystère »). Le script :

Listing 3 – main.py qui réalise une simple prédiction sur une image

```python
def showImage(matrix: list[Any], rotatedImage: Image.Image, imageCropped: Image.Image, pente: float, p: float) -> None:
    figure = plt.figure()
    ...
    plt.show()

def NormalisationFichier(imagePath: str) -> None:
    ...
    print("Nombre intersection en x et y :", intersect(croppedImage))
    showImage(binaryImage, rotatedImage, croppedImage, pente, p)

def CreateDb(dataset: str, k: int, db_csv: str, taille_grille: int, recalcul_db: bool) -> list[tuple[list[float], str]]:
    if os.path.exists(db_csv) and not recalcul_db:
```

```python
        database = db.charger_journal(db_csv)
    else:
        os.remove(db_csv) if os.path.exists(db_csv)
        database = db.create_db(dataset, taille_grille)
        db.creer_journal(db_csv, database, taille_grille)

    return database

def main() -> None:
    dataset = "ipad_dataset_train"
    k = 3
    taille_grille = 7
    db_csv = "MacsOCR/database.csv"
    dataset_train = f"MacsOCR/dataset/train/{dataset}/"
    recalcule_db = False

    file = f"MacsOCR/dataset/test_image/perso/73.jpeg"

    database = CreateDb(dataset_train, k, db_csv, taille_grille, recalcule_db)
    mon_image = (list(db.indexFile(file, taille_grille)), file.split("/")[-1])
    distances = knn.calcul_distance_total(database, mon_image)

    print("Prédiction :", knn.find(distances, k, True))
    print("réponse :", file.split("/")[-1][0])
```

Les différents paramètres de la fonction main :
- `dataset` : nom du dataset à utiliser.
- `k` : k à utiliser pour les k-plus-proche-voisin.
- `taille_grille` : taille de la grille pour le zoning.
- `db_csv` : chemin de la base de données csv.
- `dataset_train` : chemin du dataset d'entrainement.
- `recalcule_db` : booléen qui permet de re-générer à chaque lancement.

Les différentes étapes de la fonction main :
1. Charge le fichier `database.csv` ou le génère.
2. **Normalisation :** Passe l'image mystère dans la fonction `db.indexFile(...)`.
3. **Affichage :** Affiche visuellement (via *Matplotlib*) les étapes de binarisation, la droite de régression et le redimensionnement. (étape optionnelle via la fonction NormalisationFichier())
4. **Caractérisation :** Calcule le vecteur caractéristique et interroge l'algorithme K-NN.
5. **Résultat :** Affiche dans la console la prédiction finale et la répartition détaillée des votes du voisinage.

9 Résultats, Expérimentation et Analyse

La validation d'un modèle d'intelligence artificielle repose sur sa capacité à généraliser son apprentissage à des données qu'il n'a jamais rencontrées. Pour évaluer notre moteur K-NN, nous avons conçu un environnement de test rigoureux.

9.1 Processus d'évaluation (`train.py`) et Apprentissage Actif

Puisque l'algorithme K-NN ne possède pas de phase d'optimisation mathématique de poids (contrairement aux réseaux de neurones), l'« entraînement » réside dans la constitution et la qualité de sa base de connaissances (le dataset de *Train*). Le script `train.py` a pour rôle d'automatiser l'évaluation du modèle : il parcourt un dataset de *Test*, soumet chaque image à l'algorithme, et

compare la prédiction avec la véritable étiquette pour calculer le taux de succès global (*Win Rate*).

Afin d'améliorer nos résultats, nous avons mis en place une méthode de **Hard Example Mining** (recherche d'exemples difficiles). Au lieu d'augmenter aveuglément le volume de données, le script `train.py` isole les images où le K-NN échoue. Nous analysons ces échecs et alimentons la base d'apprentissage avec de nouvelles écritures présentant les mêmes caractéristiques (traits fins, boucles ambiguës). Cela permet de renforcer notre modèle.

9.2 Gestion et Optimisation des Hyperparamètres

Le développement de ce moteur OCR a fait l'objet d'un processus itératif. Nos algorithmes ont subi de multiples refontes au gré de nos expérimentations (grace à `train.py`) pour pallier les défauts mathématiques identifiés lors de nos premiers tests. Cette section retrace l'évolution de nos choix.

9.2.1 Standardisation de l'Espace (28x28 et Lanczos)

Historiquement, la reconnaissance de chiffres manuscrits est standardisée autour du format *MNIST* (28×28 pixels). Nous avons adopté cette dimension pour deux raisons : d'une part, elle offre un compromis idéal entre la conservation de la topologie du chiffre et la légèreté des calculs vectoriels (784 pixels) ; d'autre part, elle nous permettait de confronter notre modèle à des bases de données universitaires.

Pour forcer nos images rectangulaires brutes dans ce format carré parfait, nous avons dû utiliser un algorithme de redimensionnement. Nos premiers tests avec un redimensionnement classique (au plus proche voisin) détruisaient les courbes des chiffres. Nous sommes donc passés au **filtre de rééchantillonnage de Lanczos**. Ce filtre mathématique préserve parfaitement la continuité des courbes.

9.2.2 Le Paradoxe de la Double Binarisation

L'utilisation du filtre de Lanczos a cependant engendré un nouvel obstacle : pour lisser les courbes, le filtre génère un effet d'anti-crénelage (*anti-aliasing*) (le crénelage désigne l'effet "escalier" sur les bords d'une forme numérique), créant des pixels gris sur les bordures du tracé. Or, notre algorithme d'extraction (qui compte les pixels noirs) exige des matrices strictement booléennes. C'est ce qui justifie la présence d'une **seconde binarisation** dans notre pipeline final : la première sert à isoler l'encre de la page brute, et la seconde sert à "nettoyer" le flou mathématique créé par le redimensionnement, garantissant un signal pur. De plus, nous avons implémenté l'inversion des couleurs (`colorInversion.py`) (encre noire sur fond blanc pour nos tests, alors que MNIST est en blanc sur noir) pour garantir que l'algorithme calcule toujours les densités sur le bon canal, quelle que soit la source.

9.2.3 L'Évolution de la Caractérisation : Vers le Zoning 7×7

Notre première approche d'extraction de caractéristiques était naïve. Nous divisions l'image en seulement deux moitiés (haut/bas), puis en quatre quadrants (Haut-Gauche, Haut-Droit, Bas-Gauche, Bas-Droit). Les résultats étaient désastreux : mathématiquement, un « 8 » et un « 0 » possédaient quasiment la même masse d'encre dans ces quatre zones.

Nous avons donc évolué vers le pavage spatial (*Zoning*). La littérature recommande souvent des grilles de 8×8. Cependant, appliquer une grille de 8 sur une matrice de 28 génère des blocs de $3,5$ pixels. L'algorithme devait interpoler ou ignorer des pixels aux frontières, causant des pertes de données flottantes. Nous avons donc corrigé ce paramètre pour adopter une **grille de** 7×7. La division devient entière ($28/7 = 4$). L'image est ainsi parfaitement découpée en 49 blocs stricts de 4×4 pixels, sans aucune perte d'information.

9.2.4 Ajustement Dynamique du Paramètre K

Le paramètre K (le nombre de voisins consultés) a subi de fortes variations lors de nos expérimentations, car il dépend intimement de la nature et du volume du jeu de données :

— **Phase 1 (Expérimentations sur MNIST) :** Lors de nos premiers essais, et sans connaissance préalable des standards d'optimisation, nous avons intuitivement testé des valeurs extrêmement élevées, allant jusqu'à $K = 101$. L'hypothèse initiale était qu'un voisinage très large permettrait d'ignorer les écritures aberrantes. Cependant, nos tests ont démontré que ce choix entraînait un lissage excessif : les caractéristiques locales d'un chiffre se retrouvaient totalement noyées dans la masse globale. Après une série de tests empiriques itératifs, nous avons drastiquement réduit ce paramètre. Les expérimentations ont conclu que la valeur $K = 5$ constituait le point d'équilibre parfait, offrant le meilleur taux de réussite en combinant une grande précision géométrique et une robustesse suffisante face au bruit.

— **Phase 2 (Validation sur le dataset iPad) :** Le passage à notre propre jeu de données a permis de confirmer la pertinence du réglage $K = 5$. Ce dataset étant très qualitatif (encre numérique pure) mais plus restreint en volume (environ 105 images par classe), l'utilisation d'un voisinage court est devenue une nécessité mathématique. Un paramètre élevé, comme le $K = 101$ testé initialement, aurait ici pour effet d'englober la quasi-totalité de la base de données, rendant toute distinction entre les chiffres impossible. En stabilisant le modèle sur $K = 5$ (et $K = 3$), nous avons obtenu des prédictions extrêmement fines : l'algorithme parvient à identifier des similitudes précises entre les styles d'écriture sans être noyé par la masse statistique.

9.3 Optimisation du Code et Complexité Algorithmique

L'implémentation naïve d'un K-NN sur une large base de données peut s'avérer extrêmement coûteuse en temps de calcul. Nous avons donc concentré nos efforts sur deux axes d'optimisation :

1. **La mise en cache des caractéristiques :** Comme vu précédemment, le script `db.py` pré-calcule les vecteurs. Ainsi, lors de la prédiction, la complexité temporelle est réduite. L'algorithme ne refait aucun traitement d'image sur la base d'apprentissage ; il parcourt de simples lignes de texte.
2. **La vectorisation des calculs mathématiques :** Le calcul de la distance Euclidienne en dimension 52 nécessite théoriquement une boucle itérative. Pour contourner la lenteur de la boucle `for` native de Python, nous avons utilisé les structures optimisées de la bibliothèque *NumPy*. L'opération `np.linalg.norm(X - Y)` est exécutée en langage C compilé, offrant une rapidité permettant de traiter plusieurs centaines de comparaisons en une fraction de seconde.

9.4 Nos Résultats Obtenus et Limites du Modèle

Sur notre jeu de données propriétaire généré sur iPad (garantissant des conditions de test non biaisées), le modèle atteint un taux de succès **robuste de 89% à 94%**. Il se montre particulièrement résilient face aux variations d'épaisseur de trait (grâce au calcul de densité relative du zoning) et aux translations (grâce au recadrage dynamique).

Cependant, cette phase de test a également mis en avant les limites mathématiques de notre architecture. La plus notable concerne la **sensibilité aux rotations extrêmes**. Si un utilisateur soumet un chiffre incliné à près de 90 degrés, l'algorithme échoue. La cause est algébrique : dans la formule de notre régression linéaire, la variance sur l'axe Y (le dénominateur) tend vers zéro lorsque le trait devient horizontal, générant une pente aberrante. Il s'agit d'une limite structurelle assumée de notre correction d'inclinaison (communément appelée *slant correction*

dans la littérature), qui suppose que le scripteur respecte une orientation initiale raisonnable (inférieure à 45 degrés).

9.5 Résultat de notre étude

Pour valider la robustesse de nos algorithmes de normalisation et de caractérisation, nous avons confronté notre modèle à trois jeux de données de natures très différentes.

9.5.1 Dataset : MNIST

La base de données *MNIST* est le standard académique mondial pour la reconnaissance de chiffres manuscrits.

- **Particularité :** Les images étant originellement en blanc sur fond noir, nous avons utilisé notre script `colorInversion.py` pour les adapter à notre algorithme.
- **Analyse :** Sur ce dataset gigantesque aux styles d'écritures très variés, notre modèle a obtenu un taux de réussite de **94%** (avec un paramètre ajusté à $K = 5$). L'algorithme s'adapte, mais nous avons rencontré une limite majeure avec ce jeu de données : l'algorithme est perturbé par les tracés trop atypiques ou les écritures américaines coupées, cependant ce dataset est très complet. Mais si nous entrainons notre modèle avec ce dataset il à tout de même du mal à reconnaitre nos propre images bien qu'il reconnaisse bien les images dédiées de test de ce dataset.

9.5.2 Dataset : Chars74K

Le dataset *Chars74K* regroupe des caractères écrit à l'ordinateur provenant de nombreuse polices d'écritures.

- **Particularité :** Contrairement à MNIST, les épaisseurs de trait et la netteté varient drastiquement d'une image à l'autre.
- **Analyse :** Notre méthode de *Zoning* (basée sur une densité d'encre relative) a prouvé son efficacité face aux changements d'épaisseur de stylo, atteignant un score de **94%** avec $K = 5$.

9.5.3 Dataset : iPad (Personnalisé)

Pour maîtriser nos tests de bout en bout, nous avons rédigé notre propre jeu de données sur tablette grâce à la contribution de notre entourage qui a accepté de nous aider en écrivant une centaine de chiffres par personne.

- **Particularité :** L'environnement de capture est très propre (fond blanc, encre pure). L'algorithme peut se concentrer uniquement sur la forme et l'inclinaison naturelle de l'écriture.
- **Analyse :** Dans cet environnement contrôlé, notre système s'est montré très performant avec un taux de réussite culminant à **89%** pour un voisinage de $K = 3$. Bien que ce résultat valide notre approche géométrique, le modèle atteint ses limites structurelles lorsque un utilisateur incline son chiffre à près de 90 degrés ce qui rend la régression linéaire inopérante. Ou encore on a obesrvé que la majorité des volontaire ne font pas de barre en dessous du chiffre 1, ce qui a rendu les tests compliqués pour ce chiffre au debut. Depuis le dataset a été complété de ce chiffre écrit de cette manière.

9.5.4 Synthèse et Enseignements de l'Étude

La confrontation de notre modèle algorithmique à ces trois jeux de données distincts nous a permis de tirer plusieurs conclusions fondamentales sur le comportement d'un OCR comme celui-ci.

L'enseignement principal de cette étude est la **forte dépendance du modèle à la cohérence de son environnement d'acquisition**. Comme le prouvent nos résultats sur le dataset iPad, l'algorithme des K-Plus-Proches-Voisins se révèle redoutablement efficace lorsque les images de test partagent le même cadre technique (contraste, type de stylo, absence de bruit de fond) que le jeu de données d'entraînement. Dans un environnement contrôlé, les signatures vectorielles extraites sont mathématiquement stables, ce qui rend les prédictions du K-NN très fiables.

À l'inverse, les difficultés rencontrées sur *MNIST* et *Chars74K* ont mis en lumière le véritable talon d'Achille de l'extraction manuelle de caractéristiques : **la sensibilité au bruit lors du prétraitement**. Sur des images non standardisées, la majorité des erreurs ne provient pas d'une défaillance du calcul de distance, mais d'un échec en amont (un recadrage faussé par un pixel parasite, ou une régression linéaire perturbée par un tracé atypique).

En conclusion, notre étude démontre qu'une approche purement mathématique (sans réseau de neurones) est une solution algorithmique extrêmement viable, légère et transparente pour traiter des documents numérisés dans un cadre strict (comme la lecture automatique de formulaires administratifs ou de chèques). En revanche, pour traiter des écritures « sauvages » soumises à de fortes variations contextuelles, la flexibilité des architectures d'apprentissage profond reste indispensable.

10 Conclusion du Projet

10.1 Perspectives et Réflexions : Un Win Rate de 100 % est-il possible ?

Face à un taux de succès de 92 %, il est légitime de se demander s'il est possible d'atteindre une précision de 100 %. D'un point de vue purement théorique et pratique en *Machine Learning*, **la réponse est non**. Ce plafond de verre s'explique par deux facteurs distincts.

10.1.1 L'Erreur Irréductible et l'Ambiguïté Humaine

Même pour un cerveau humain, le taux d'erreur sur la reconnaissance de caractères manuscrits isolés n'est pas de 0 %. L'écriture humaine est intrinsèquement ambiguë : un scripteur peut tracer un « 7 » de manière si précipitée qu'il devient topologiquement identique à un « 1 ». Un être humain parvient généralement à lever cette ambiguïté grâce au contexte (le mot ou la phrase qui entoure la lettre), une information dont notre algorithme est totalement dépourvu puisqu'il analyse les chiffres un par un. Il existe donc une *erreur de Bayes* (erreur irréductible) qu'aucun modèle mathématique ne peut surmonter.

10.1.2 Les Limites Structurelles du modèle K-NN

Atteindre des scores supérieurs (de l'ordre de 98 % à 99 %) nécessiterait un changement radical de modèle mathématique. Notre approche repose sur la création *manuelle* de caractéristiques géométriques macroscopiques (les 52 dimensions de notre vecteur). Bien que performante, cette méthode atteint ses limites face aux micro-déformations d'un tracé.

Pour franchir le cap des 92 %, il faudrait abandonner cette extraction manuelle au profit d'un **Réseau de Neurones Convolutifs (CNN)**. Les CNN utilisent des filtres matriciels optimisés par descente de gradient pour apprendre par eux-mêmes des micro-caractéristiques (détection de bords, de courbes, de textures) inaccessibles à l'intuition humaine.

Cependant, l'utilisation d'une telle architecture de type « boîte noire » irait à l'encontre de l'essence même de ce projet de calcul scientifique. Le fait d'avoir pu atteindre 92 % de précision sur des données réelles inconnues, en n'utilisant qu'un espace vectoriel de 52 dimensions défini par la seule force de la géométrie et du traitement du signal, constitue une démonstration solide de la validité de notre modélisation mathématique.

10.2 Mot de la fin

La force de notre implémentation réside dans son interprétabilité : chaque étape de transformation (redressement par régression, extraction par pavage spatial, décision par norme euclidienne) s'appuie sur des théorèmes mathématiques dont le comportement est prédictible et maîtrisable. Ce travail de modélisation, couplé aux optimisations algorithmiques via *NumPy*, aboutit à un système à la fois performant, léger en ressources et scientifiquement rigoureux.