

TP 2 — Chiffres manuscrits : compresser, regrouper, reconnaître

ACP, k -means, k plus proches voisins et modèle gaussien régularisé

Christophe Ambroise

2026-10-19

Objectifs

Sur des images de chiffres manuscrits, utiliser l'ACP pour compresser et visualiser, k -means pour regrouper et résumer par des prototypes, puis reconnaître les chiffres par k -PPV et par un modèle gaussien. Fil rouge : **la dimension**. Avec 64 pixels et environ 125 images par chiffre, les covariances estimées sont singulières ; une pénalité de type ridge les rend utilisables.

Données et consignes

`data/digits.csv` — jeu UCI *Optical Recognition of Handwritten Digits* (Alpaydin et Kaynak, 1998), dans la version réduite distribuée avec scikit-learn : 1 797 images de 8×8 pixels, chaque pixel étant un entier de 0 (blanc) à 16 (noir). Les colonnes `px00` à `px63` lisent l'image ligne par ligne ; la colonne `chiffre` (0 à 9) est l'étiquette.

- Travailler en Python **ou** en R ; les squelettes sont donnés dans les deux langages.
- Rendre un compte rendu (Quarto ou notebook) qui contient le code, les figures et une réponse **rédigée** à chaque question.
- Fixer les graines (`random_state=0`, `set.seed(0)`).

```
from pathlib import Path

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

chiffres = pd.read_csv(Path("..") / "data" / "digits.csv")
y = chiffres["chiffre"].to_numpy()
X = chiffres.drop(columns="chiffre").to_numpy(dtype=float)  # 64 pixels

def affiche_chiffre(x, ax, titre=""):
    ax.imshow(x.reshape(8, 8), cmap="gray_r", vmin=0, vmax=16)
    ax.set_title(titre); ax.axis("off")
```

```
chiffres <- read.csv("../data/digits.csv")
y <- factor(chiffres$chiffre)
X <- as.matrix(chiffres[, setdiff(names(chiffres), "chiffre")])  # 64 pixels
```

```
# Affiche une image de 64 pixels, lue ligne par ligne, dans le bon sens.
affiche_chiffre <- function(x, titre = "") {
  m <- matrix(x, 8, 8, byrow = TRUE)
  image(t(m)[, 8:1], col = gray(seq(1, 0, length.out = 17)), axes = FALSE,
        main = titre, asp = 1)
}
```

Partie 1 — Exploration

1. Afficher les vingt premières images avec leur étiquette, puis l'**image moyenne** de chaque chiffre.
2. Certains pixels ont une variance nulle sur tout le jeu. Lesquels, et où sont-ils dans l'image ? Pourquoi est-ce prévisible ?
3. Faut-il standardiser les pixels avant une ACP ou un k -PPV ? Donner deux arguments, dont l'un tiré de la question 2.

Partie 2 — ACP : compresser des images

Les étiquettes restent hors du calcul. ACP **centrée**, non réduite, $S = \frac{1}{n} X_c^T X_c$.

```
def acp(A):
    centre = A.mean(axis=0)
    Ac = A - centre
    lam, U = np.linalg.eigh(Ac.T @ Ac / len(Ac))
    ordre = lam.argsort()[::-1]
    return lam[ordre], U[:, ordre], centre
```

```
lam, U, centre = acp(X)
```

```
def reconstruit(x, q):
    """Reconstruction de rang q d'une image x."""
    Uq = U[:, :q]
    return centre + (x - centre) @ Uq @ Uq.T
```

```
acp <- function(A) {
  centre <- colMeans(A)
  Ac <- sweep(A, 2, centre)
  e <- eigen(crossprod(Ac) / nrow(Ac), symmetric = TRUE)
  list(lam = pmax(e$values, 0), U = e$vectors, centre = centre)
}
a <- acp(X)

reconstruit <- function(x, q) {
  Uq <- a$U[, 1:q, drop = FALSE]
  a$centre + drop(((x - a$centre) %*% Uq) %*% t(Uq))
}
```

1. Tracer ρ_q en fonction de q . Combien de composantes faut-il pour conserver 90 % de l'inertie ? Comparer au cas d'Iris (CM 6).
2. Afficher les quatre premiers vecteurs propres $u_1, \dots, u_4$ **comme des images** 8×8 . Que « regarde » chacun d'eux ?

3. Pour une image, afficher ses reconstructions de rang $q \in \{2, 5, 10, 20, 40\}$. À partir de quel q le chiffre est-il lisible ?
4. Calculer l'erreur quadratique moyenne de reconstruction $\frac{1}{n} \|X_c - \widehat{X}_q\|_F^2$ pour $q = 10$ et $q = 20$, et la comparer à $\sum_{j>q} \lambda_j$. Quel résultat du CM 6 vérifie-t-on ?
5. Tracer le plan principal, puis le colorier *a posteriori* par chiffre. Quels chiffres sont bien séparés ? Lesquels se confondent ? Avec ρ_2 sous les yeux, est-ce surprenant ?

Partie 3 — k -means : groupes et prototypes

1. Appliquer k -means avec $K = 10$ (`n_init=50` ou `nstart = 50`). Afficher les dix centres comme des images.
2. Croiser groupes et chiffres et calculer la pureté (définie au TP 1). Quels chiffres k -means sépare-t-il mal ? Est-ce cohérent avec la question 2.5 ?
3. Un centre de groupe est un **prototype** : une image moyenne qui résume ses membres. Combien de nombres faut-il stocker pour décrire le jeu par ses 10 prototypes et l'affectation de chaque image ? Par une ACP de rang q ? Comparer les deux façons de compresser.

Partie 4 — Reconnaître : k -PPV et prototypes

On partage le jeu une fois pour toutes : 70 % d'apprentissage, 30 % de test, stratifié par chiffre. Le test ne sert **qu'à la fin**, pour l'évaluation finale ; tout choix (de k , de K , de q , de λ) se fait par validation croisée à 5 plis **sur l'apprentissage**.

```
from sklearn.cluster import KMeans
from sklearn.model_selection import StratifiedKFold, train_test_split
from sklearn.neighbors import KNeighborsClassifier

X_app, X_test, y_app, y_test = train_test_split(
    X, y, test_size=0.3, stratify=y, random_state=0)
plis = StratifiedKFold(n_splits=5, shuffle=True, random_state=0)

def erreur_vc(ajuste_predit, A=X_app, b=y_app):
    """`ajuste_predit(A, b, V)` apprend sur (A, b) et prédit les lignes de V."""
    erreurs = []
    for i_app, i_val in plis.split(A, b):
        pred = ajuste_predit(A[i_app], b[i_app], A[i_val])
        erreurs.append(np.mean(pred != b[i_val]))
    return np.mean(erreurs)

def knn_ap(k):
    return lambda A, b, V: KNeighborsClassifier(k).fit(A, b).predict(V)
```

```
library(class)

set.seed(0) # partage stratifié : 30 % de chaque chiffre en test
test <- unlist(lapply(levels(y), function(cl) {
  idx <- which(y == cl)
  sample(idx, round(0.3 * length(idx)))
}))
X_app <- X[-test, ]; y_app <- y[-test]
```

```

X_test <- X[test, ]; y_test <- y[test]

plis_stratifies <- function(y, K = 5, graine = 0) {
  set.seed(graine)
  pli <- integer(length(y))
  for (cls in levels(y)) {
    idx <- sample(which(y == cls))
    pli[idx] <- rep_len(seq_len(K), length(idx))
  }
  pli
}
pli <- plis_stratifies(y_app)

erreur_vc <- function(ajuste_predit, A = X_app, b = y_app) {
  mean(sapply(1:5, function(k) {
    pred <- ajuste_predit(A[pli != k, , drop = FALSE], b[pli != k],
                          A[pli == k, , drop = FALSE])
    mean(pred != b[pli == k])
  })))
}
knn_ap <- function(k) function(A, b, V) knn(A, V, b, k = k)

```

1. Choisir $k \in \{1, 3, 5, 7, 9\}$ par validation croisée, puis donner l'erreur de test. Afficher quelques images de test mal classées avec la prédiction : les erreurs sont-elles « excusables » ?
2. **Classer par prototypes.** Apprendre K prototypes par k -means sur l'apprentissage, étiqueter chacun par le chiffre majoritaire de son groupe, puis attribuer à chaque image de test l'étiquette de son prototype le plus proche. Écrire la fonction `prototypes_ap(K)`, puis tracer l'erreur de test pour $K \in \{10, 20, 50, 100, 200\}$.
3. Que devient cette méthode quand K atteint la taille de l'apprentissage ? Situer la courbe de la question 2 par rapport à l'erreur des 1-PPV, et discuter le compromis entre mémoire et erreur.
4. Estimer par validation croisée l'erreur des 3-PPV appliqués aux q premières composantes, $q \in \{2, 5, 10, 20, 30, 40, 64\}$, l'ACP étant apprise dans chaque pli. Combien de composantes suffisent ? Relier au q de la question 2.1.

Partie 5 — Modèle gaussien régularisé

Modèle du CM 2 : dans la classe k , $x \sim \mathcal{N}(\mu_k, \Sigma_k)$, et la règle de Bayes affecte x à la classe qui maximise

$$\delta_k(x) = \log \pi_k - \frac{1}{2} \log \det \Sigma_k - \frac{1}{2} (x - \mu_k)^\top \Sigma_k^{-1} (x - \mu_k).$$

```

def ajuste_gaussien(A, b, lam_reg, commune=False):
    classes = np.unique(b)
    mus, sigmas, pis = [], [], []
    for c in classes:
        Ac = A[b == c]
        mus.append(Ac.mean(axis=0))
        sigmas.append(np.cov(Ac.T, bias=True))      # diviseur n_k
        pis.append(len(Ac) / len(A))
    # À compléter : si `commune`, remplacer chaque Sigma_k par la covariance
    # intra-classe commune ; puis ajouter lam_reg * I à chaque matrice.
    return classes, mus, sigmas, pis

```

```
def predict_gaussien(modele, V):
    classes, mus, sigmas, pis = modele
    scores = []
    for mu, s, pi in zip(mus, sigmas, pis):
        _, logdet = np.linalg.slogdet(s)
        # À compléter : distance de Mahalanobis de chaque ligne de V à mu.
        ...
    return classes[np.argmax(scores, axis=0)]
```

```
ajuste_gaussien <- function(A, b, lam_reg, commune = FALSE) {
    classes <- levels(b)
    mus <- lapply(classes, function(cl) colMeans(A[b == cl, , drop = FALSE]))
    sigmas <- lapply(classes, function(cl) {
        Ac <- A[b == cl, , drop = FALSE]
        crossprod(sweep(Ac, 2, colMeans(Ac))) / nrow(Ac) # diviseur n_k
    })
    pis <- as.numeric(table(b)) / length(b)
    # À compléter : si `commune`, covariance intra-classe commune ;
    # puis ajouter lam_reg * diag(ncol(A)) à chaque matrice.
    list(classes = classes, mus = mus, sigmas = sigmas, pis = pis)
}

predict_gaussien <- function(modele, V) {
    scores <- sapply(seq_along(modele$classes), function(k) {
        s <- modele$sigmas[[k]]
        logdet <- as.numeric(determinant(s, logarithm = TRUE)$modulus)
        # À compléter : distance de Mahalanobis de chaque ligne de V à mu_k (solve).
    })
    factor(modele$classes[max.col(scores, ties.method = "first")],
           levels = modele$classes)
}
```

1. Compléter les deux fonctions.
2. Calculer le rang de chaque $\widehat{\Sigma}_k$ sur l'apprentissage, et le nombre de pixels **constants dans la classe k** . Comparer les deux, et expliquer le lien avec la question 1.2. Constaté ce qui se passe avec $\lambda = 0$. Pourquoi MASS::qda ou QuadraticDiscriminantAnalysis sans réglage échouent-ils ou protestent-ils ?
3. Remplacer $\widehat{\Sigma}_k$ par $\widehat{\Sigma}_k + \lambda I_p$, $\lambda > 0$. Montrer que cette matrice est inversible, et que ses valeurs propres sont celles de $\widehat{\Sigma}_k$ décalées de λ . Rapprocher de la régression ridge (CM 5), où $X^T X$ devient $X^T X + \lambda I$.
4. Choisir λ sur la grille $10^{-3}, 10^{-2.5}, \dots, 10^3$ par validation croisée, pour le modèle à covariances propres (QDA) et pour le modèle à covariance commune (LDA). Tracer les deux courbes d'erreur en fonction de $\log_{10} \lambda$.
5. Pour λ grand, montrer qu'à des termes indépendants de k et des termes en $O(\lambda^{-2})$ près, $\delta_k(x) \approx \log \pi_k - \frac{1}{2\lambda} (\text{tr } \widehat{\Sigma}_k + \|x - \mu_k\|^2)$. Pour le modèle à covariance commune et des π_k égaux, reconnaître la règle du **plus proche centre de classe**. Estimer son erreur par validation croisée et la situer sur les courbes. Que se passe-t-il quand $\lambda \rightarrow +\infty$ si les π_k ne sont pas égaux ?
6. Donner l'erreur de test des deux modèles à leur λ^* . Compter les paramètres de chaque modèle. Pourquoi la QDA, qui en estime bien davantage, peut-elle battre la LDA ici, alors qu'on craint d'ordinaire le contraire ?

Partie 6 — Synthèse

Réunir dans un tableau l'erreur de test de chaque méthode à son réglage choisi par validation croisée : k -PPV, k -PPV après ACP, prototypes k -means, LDA et QDA régularisées. Ajouter pour chacune ce qu'il faut stocker pour prédire (nombre de valeurs réelles). Quelle méthode recommanderiez-vous pour un appareil à mémoire limitée ? Pour la meilleure exactitude ?