# Modele4

November 15, 2025

[224]:
```
import sys
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(2)
```

[225]:
```
import numpy as np

def extract_features(img):
    """
    Version légère: ~100 features au lieu de 337
    Garde les plus importantes
    """
    features = []
    img = img.astype(np.float32)
    gray = img.mean(axis=2)

    # 1. RGB stats (18)
    for c in range(3):
        channel = img[:, :, c]
        features.append(channel.mean())
        features.append(channel.std())
        features.append(np.percentile(channel, 10))
        features.append(np.percentile(channel, 50))
        features.append(np.percentile(channel, 90))
        features.append(channel.max())

    # 2. Grayscale (6)
    features.append(gray.mean())
    features.append(gray.std())
    features.append(np.percentile(gray, 25))
    features.append(np.percentile(gray, 75))
    features.append(gray.min())
    features.append(gray.max())

    # 3. Edges (3)
    gx = np.abs(gray[:, 1:] - gray[:, :-1]).mean()
    gy = np.abs(gray[1:, :] - gray[:-1, :]).mean()
```

```
    features.extend([gx, gy, np.sqrt(gx**2 + gy**2)])

    # 4. Quadrants (8)
    H, W = gray.shape
    h2, w2 = H//2, W//2
    for quad in [gray[:h2, :w2], gray[:h2, w2:], gray[h2:, :w2], gray[h2:, w2:
↪]]:
        features.append(quad.mean())
        features.append(quad.std())

    # 5. Color ratios (3)
    R, G, B = img[:,:,0], img[:,:,1], img[:,:,2]
    features.append(R.mean() / (G.mean() + 1e-8))
    features.append(R.mean() / (B.mean() + 1e-8))
    features.append(G.mean() / (B.mean() + 1e-8))

    # 6. LBP (64 bins au lieu de 256)
    def lbp(gray):
        H, W = gray.shape
        padded = np.pad(gray, ((1, 1), (1, 1)), mode="edge")
        lbp_img = np.zeros((H, W), dtype=np.uint8)
        offsets = [(-1,-1),(-1,0),(-1,1),(0,-1),(0,1),(1,-1),(1,0),(1,1)]
        for idx, (dy, dx) in enumerate(offsets):
            neigh = padded[1+dy:H+1+dy, 1+dx:W+1+dx]
            bit = (neigh >= gray).astype(np.uint8)
            lbp_img |= (bit << idx)
        return lbp_img

    lbp_img = lbp(gray)
    hist_lbp = np.histogram(lbp_img, bins=64, range=(0, 256))[0]
    features.extend(hist_lbp.tolist())


    return np.array(features, dtype=np.float32)
```

[226]:
```
import numpy as np
import itertools

def polynomial_expansion_degree3(X):
    N, d = X.shape
    features = []

    # Degree 1
    features.append(X)

    # Degree 2
    for i, j in itertools.combinations_with_replacement(range(d), 2):
```

```
        features.append((X[:, i] * X[:, j]).reshape(N, 1))

    # Degree 3
    for i, j, k in itertools.combinations_with_replacement(range(d), 3):
        features.append((X[:, i] * X[:, j] * X[:, k]).reshape(N, 1))

    return np.hstack(features)
```

[227]:
```
class StandardScaler:
    def fit(self, X):
        self.mu = X.mean(axis=0)
        self.sigma = X.std(axis=0) + 1e-8
    def transform(self, X):
        return (X - self.mu) / self.sigma
    def fit_transform(self, X):
        self.fit(X)
        return self.transform(X)
```

[228]:
```
class SoftmaxClassifier:
    def __init__(self, input_dim, num_classes, reg=0.0, seed=None):

        if seed is not None:
            np.random.seed(seed)

        self.W = 0.01 * np.random.randn(input_dim, num_classes).astype(np.
 ↪float32)
        self.reg = reg  # L2
        self.b = np.zeros(num_classes)
        self.seed = seed

    def _softmax(self, scores):
        # scores: (N, K)
        scores = scores - scores.max(axis=1, keepdims=True)
        exp_scores = np.exp(scores)
        return exp_scores / exp_scores.sum(axis=1, keepdims=True)

    def loss(self, X, y, sample_weights=None):
        N = X.shape[0]

        scores = X @ self.W + self.b
        probs = self._softmax(scores)

        correct_logprobs = -np.log(probs[np.arange(N), y] + 1e-12)

        if sample_weights is None:
            sample_weights = np.ones(N)
```

```
        loss = np.sum(sample_weights * correct_logprobs) / N
        loss += 0.5 * self.reg * np.sum(self.W**2)

        return loss, probs

    def grad(self, X, y, probs, sample_weights=None):
        N = X.shape[0]

        if sample_weights is None:
            sample_weights = np.ones(N)

        dscores = probs.copy()
        dscores[np.arange(N), y] -= 1
        dscores *= sample_weights[:, None]
        dscores /= N

        dW = X.T @ dscores + self.reg * self.W
        db = dscores.sum(axis=0)
        return dW, db


    def fit(self, X, y, lr=1e-4, n_steps=1000, sample_weights=None,␣
↪verbose=True):
        losses = []
        for step in range(n_steps):

            loss, probs = self.loss(X, y, sample_weights)
            dW, db = self.grad(X, y, probs, sample_weights)

            self.W -= lr * dW
            self.b -= lr * db

            losses.append(loss)

            if verbose and step % 100 == 0:
                print(f"Step {step}, loss = {loss:.4f}")

        return losses


    def predict_proba(self, X):
        scores = X @ self.W
        probs = self._softmax(scores)
        return probs

    def predict(self, X):
        probs = self.predict_proba(X)
```

```
        return probs.argmax(axis=1)
```

[229]:
```
def accuracy(y_true, y_pred):
    y_true = np.asarray(y_true)
    y_pred = np.asarray(y_pred)
    return np.mean(y_true == y_pred)
```

[230]:
```
def confusion_matrix_np(y_true, y_pred, num_classes=None):
    y_true = np.asarray(y_true).astype(int)
    y_pred = np.asarray(y_pred).astype(int)

    if num_classes is None:
        num_classes = max(y_true.max(), y_pred.max()) + 1

    cm = np.zeros((num_classes, num_classes), dtype=int)
    for t, p in zip(y_true, y_pred):
        cm[t, p] += 1
    return cm
```

[231]:
```
def balanced_accuracy(y_true, y_pred):
    cm = confusion_matrix_np(y_true, y_pred)
    TP = np.diag(cm)
    real_pos = cm.sum(axis=1)
    recall = np.where(real_pos > 0, TP / real_pos, 0.0)
    return recall.mean()
```

[232]:
```
def recall_per_class(cm):
    """
    cm : matrice de confusion (numpy array KxK)
    retourne un vecteur de recall par classe
    """
    TP = np.diag(cm)
    real_pos = cm.sum(axis=1)   # total de vrais échantillons par classe

    # recall par classe (évite division par zéro)
    recall = np.where(real_pos > 0, TP / real_pos, 0.0)
    return recall
```

[233]:
```
import numpy as np

def rbf_kernel(x, X, sigma=1.0):
    np.random.seed(0)
    # ||x - X||^2 = sum( (x_i - X_i)^2 )
    dists = np.sum((X - x)**2, axis=1)
    return np.exp(-dists / (2 * sigma**2))
```

```python
class KernelPerceptron:
    def __init__(self, kernel_fn, n_classes, sigma=1.0):
        self.kernel_fn = kernel_fn
        self.n_classes = n_classes
        self.sigma = sigma


    def fit(self, X, y, max_epochs=10):
        """
        Entraînement multiclasse One-vs-Rest
        """
        N = X.shape[0]
        self.X = X
        self.classes = np.unique(y)
        self.train_y = y
        np.random.seed(0)

        # On crée  pour chaque classe
        self.alpha = np.zeros((self.n_classes, N))

        # Precompute Gram matrix
        print("Computing Gram matrix...")
        self.K = np.zeros((N, N))
        for i in range(N):
            self.K[i] = self.kernel_fn(X[i], X, self.sigma)

        # Entraînement One-vs-Rest
        for c in self.classes:
            print(f"Training classifier for class {c}...")
            # target binaire : +1 si y=c, sinon -1
            y_bin = np.where(y == c, 1, -1)

            count = 0
            i = 0
            n_iter = 0
            n_iter_max = max_epochs * N

            while count < N and n_iter < n_iter_max:
                # prédiction : sign( Σ _j y_j K(x_j, x_i) )
                pred = np.sign(np.sum(self.alpha[c] * y_bin * self.K[i]))

                if pred != y_bin[i]:
                    # mise à jour du perceptron
                    self.alpha[c][i] += 1
                    count = 0
                else:
                    count += 1
```

```
                i = (i + 1) % N
                n_iter += 1

        print("Training done.")

    def compute_output(self, X_test):
        """
        Retourne la classe prédite pour chaque x_test
        """
        np.random.seed(0)
        outputs = []
        for x in X_test:
            # compute kernel K(x, X_train)
            k = self.kernel_fn(x, self.X, self.sigma)

            # score pour chaque classe
            scores = []
            for c in self.classes:
                y_bin = np.where(self.train_y == c, 1, -1)
                score = np.sum(self.alpha[c] * y_bin * k)
                scores.append(score)

            # classe la plus probable
            outputs.append(np.argmax(scores))

        return np.array(outputs)
```

[236]:
```
import pickle
path_to_data = 'ift-3395-6390-kaggle-2-competition-fall-2025/train_data.pkl'

# --- Load training data ---
with open(path_to_data, "rb") as f:
    train_data = pickle.load(f)

X_imgs = train_data["images"]
y = train_data["labels"].reshape(-1)

# --- Feature extraction ---
X = np.array([extract_features(img) for img in X_imgs], dtype=np.float32)
#X = polynomial_expansion_degree3(X)
# --- Normalize ---
scaler = StandardScaler()
X = scaler.fit_transform(X)

# --- Split ---
idx = np.random.permutation(len(X))
```

```
X = X[idx]
y = y[idx]
n_train = int(0.8 * len(X))
X_train, X_test = X[:n_train], X[n_train:]
y_train, y_test = y[:n_train], y[n_train:]

# --- Class weights ---
class_counts = np.bincount(y_train)
class_weights = (1.0 / class_counts)
class_weights /= class_weights.sum()
sample_weights = class_weights[y_train]
#{0.1, 0.5, 1, 2, 5, 10}
# --- Train model ---
num_classes = len(np.unique(y))
model = KernelPerceptron(kernel_fn=rbf_kernel, n_classes=5, sigma=5)
model.fit(X_train, y_train, max_epochs=20)

'''model = SoftmaxClassifier(input_dim=X.shape[1], num_classes=num_classes,␣
 ↪reg=0.05, seed=0)
model.fit(X_train, y_train, lr=0.5, n_steps=5000,␣
 ↪sample_weights=sample_weights)'''

# --- Evaluate ---
y_pred = model.compute_output(X_test)
acc = (y_pred == y_test).mean()
print("Test accuracy =", acc)

# --- Save model ---
pickle.dump((model, scaler), open("model_softmax.pkl", "wb"))

#print("Train accuracy:", acc)
print("Test accuracy:", acc)
cm = confusion_matrix_np(y_test, y_pred)
acc = accuracy(y_test, y_pred)
bal_acc = balanced_accuracy(y_test, y_pred)
rec = recall_per_class(cm)

print("Accuracy :", acc)
print("Balanced acc :", bal_acc)
print("Confusion matrix:\n", cm)
print("Recall par classe :", rec)
print("Recall moyen (macro):", rec.mean())
```

```
Computing Gram matrix…
Training classifier for class 0…
Training classifier for class 1…
Training classifier for class 2…
```

```
Training classifier for class 3…
Training classifier for class 4…
Training done.
Test accuracy = 0.48148148148148145
Test accuracy: 0.48148148148148145
Accuracy : 0.48148148148148145
Balanced acc : 0.3343941083071518
Confusion matrix:
 [[74  6 11  5  3]
 [ 9  4  6  2  2]
 [15  5  8  8  3]
 [11  4 11 16  2]
 [ 2  1  2  4  2]]
Recall par classe : [0.74747475 0.17391304 0.20512821 0.36363636 0.18181818]
Recall moyen (macro): 0.3343941083071518
```

[235]:
```python
import pickle
import numpy as np
import pandas as pd

# --------------------------
# 1. Charger le modèle entraîné
# --------------------------
model, scaler = pickle.load(open("model_softmax.pkl", "rb"))

# --------------------------
# 2. Charger le test_data.pkl
# --------------------------
with open("ift-3395-6390-kaggle-2-competition-fall-2025/test_data.pkl", "rb")␣
 ↪as f:
    test_data = pickle.load(f)

X_test_imgs = test_data["images"]


# Apply to test set
X_test_feats = np.array([extract_features(img) for img in X_test_imgs],␣
 ↪dtype=np.float32)

# --------------------------
# 4. Normaliser avec les stats du train
# --------------------------
X_test_norm = scaler.transform(X_test_feats)

# --------------------------
# 5. Prédire
# --------------------------
```

```python
y_pred = model.compute_output(X_test_norm).astype(int)

# --------------------------
# 6. Générer le CSV Kaggle
# --------------------------
df = pd.DataFrame({
    "ID": np.arange(1, len(y_pred)+1),
    "Label": y_pred
})

df.to_csv("ift3395_YamirPoldoSilvaV5.csv", index=False)

print("Fichier 'submission.csv' généré !")
```

```
Fichier 'submission.csv' généré !
```