

SSL

September 6, 2026

1 Semi-Supervised Learning (SSL)

SSL studies how to learn from both labeled and unlabeled data, which can be useful when data is abundant but the resources to label them are limited.

In this exercise, you will:

- Given a simulated dataset with both labeled and unlabeled data, build a similarity graph and use the Harmonic Function Solution (HSF) to predict the labels of the unlabeled data;
- Use HSF for face recognition, given a fixed dataset;
- Implement an online version of HSF to label images as they appear in real time.

1.1 1. Harmonic Function Solution

Let $G = (V, E)$ be a weighted undirected graph where $V = \{x_1, \dots, x_n\}$ is the vertex set and E is the edge set. Each edge $e_{ij} \in E$ has a weight w_{ij} and, if there is no edge between x_i and x_j , then $w_{ij} = 0$.

Let $|V| = n$ be the total number of nodes. Only a subset of the nodes $S \subset V$ with cardinality $|S| = l$ is labeled, and the remaining $u = n - l$ nodes are placed in the subset $T = V \setminus S$.

Our goal is to predict the labels of the vertices in T using the structure of the graph. Since we believe that nodes close in the graph should have similar labels, we would like to have each node surrounded by a majority of nodes with the same label. In order to do so, we impose that the labeling vector $f \in \mathbb{R}^n$ must be an **harmonic function** on the graph, that is:

$$f_i = \frac{\sum_j w_{ij} f_j}{\sum_j w_{ij}}, \forall i \in T$$

One interpretation for this constraint is that w_{ij} represents the tendency of moving from node x_i to node x_j , the stationary distribution of the transition matrix $P(j|i) = \frac{w_{ij}}{\sum_k w_{ik}}$ is a valid solution to our problem.

1.1.1 Hard HFS

It can be shown that f is harmonic if and only if $(Lf)_T = 0$, where $(Lf)_T$ is the vector containing the values of Lf for the nodes in the set T , and L is the graph Laplacian.

Hence, the harmonic function solution to the SSL problem is the solution to the following optimization problem:

$$\min_{f \in \mathbb{R}^n} f^T L f \quad \text{s.t.} \quad y_i = f(x_i) \quad \forall x_i \in S$$

where y_i are the labels available for the vertices $x_i \in S$. This gives us:

$$f_T = L_{TT}^{-1}(W_{TS}f_S) = -L_{TT}^{-1}(L_{TS}f_S)$$

1.1.2 Soft HFS

If the labels are noisy, we might need to replace the “hard” constraint of the optimization problem above by a “soft” constraint. Let C be a diagonal matrix such that $C_{ii} = c_l$ for labeled examples and $C_{ii} = c_u$ otherwise. Also, define $y_i = 0$ for unlabeled examples, that is, for $x_i \in T$.

The soft HFS objective function is

$$\min_{f \in \mathbb{R}^n} (f - y)^T C (f - y) + f^T L f$$

whose solution is

$$f^* = (C^{-1}L + I)^{-1}y$$

1.1.3 Implementation

Implement hard and soft HFS in the function `compute_hfs`. Complete the function `two_moons_hfs` to test your implementation using the datasets `data_2moons_hfs.mat` and `data_2moons_hfs_large.mat`.

- Tips:
 - Don’t forget to choose well the parameters to build the graph and its Laplacian.
 - You can use the functions `build_laplacian_regularized` and `build_similarity_graph`. The function `mask_labels` is used to chose how many labels are revealed.
 - Be careful: the labels are revealed randomly, and each random realization can have different results! Check how the `seed` parameter works.
 - Introduce noisy labels to compare hard and soft HFS.

```
[1]: import numpy as np
import matplotlib.pyplot as plt
import scipy.spatial.distance as sd
from scipy.io import loadmat
import os
from helper import build_similarity_graph, label_noise
from helper import build_laplacian, build_laplacian_regularized
from helper import plot_classification
from helper import mask_labels
```

```
[2]: """
Define parameters for HFS
"""
```

```

params = {}

# regularization parameter (gamma)
params['laplacian_regularization'] = 10**-3

# the sigma value for the exponential (similarity) function, already squared
params['var'] = 1.0

# Threshold eps for epsilon graphs
params['eps'] = 0

# Number of neighbours k for k-nn. If zero, use epsilon-graph
params['k'] = 10

# String selecting which version of the laplacian matrix to construct.
# 'unn': unnormalized, 'sym': symmetric normalization, 'rw': random-walk
↳normalization
params['laplacian_normalization'] = 'unn'

# Coefficients for C matrix for soft HFS
params['c_l'] = 1.0
params['c_u'] = 0.01

```

```

[3]: def compute_hfs(L, Y, soft=False, **params):
    """
    Function to perform HFS (hard or soft!).

    Parameters
    -----
    L : array
        Graph Laplacian, (n x n) matrix (regularized or not)
    Y : array
        (n, ) array with node labels [0, 1, ... , num_classes] (0 means
    ↳unlabeled)
    soft : bool
        If True, compute soft HFS. Otherwise, compute hard HFS.

    Returns
    -----
        Labels, class assignments for each of the n nodes
    """

    num_samples = L.shape[0]
    Cl = np.unique(Y)
    num_classes = len(Cl) - 1

    """

```

```

Build the vectors:
y = (n x num_classes) target vector
l_idx = shape (l,) vector with indices of labeled nodes
u_idx = shape (u,) vector with indices of unlabeled nodes
"""

# Labeled and unlabeled indices
l_idx = np.where(Y != 0)[0]
u_idx = np.where(Y == 0)[0]

# Creating the target vector y
y = np.zeros((num_samples, num_classes))
for ci, c in enumerate(C1[1:]):
    y[Y == c, ci] = 1.0

if not soft:
    """
    Compute hard HFS.

    f_l = solution for labeled data.
    f_u = solution for unlabeled data
    f    = solution for all data
    """

    # Splitting the Laplacian into blocks (labeled vs. unlabeled indices)
    L_ll = L[l_idx][:, l_idx]
    L_lu = L[l_idx][:, u_idx]
    L_ul = L[u_idx][:, l_idx]
    L_uu = L[u_idx][:, u_idx]

    # y_l: values (or one-hot vectors) for labeled nodes
    y_l = y[l_idx, :]
    f_l = y_l

    # y_u: unknown, we need to compute it
    # Solve  $L_{uu} * f_u + L_{ul} * f_l = 0 \Rightarrow f_u = -\text{inv}(L_{uu}) * L_{ul} * f_l$ 

    # Solve for each dimension of y (i.e., for each class)
    f = np.zeros_like(y)
    f[l_idx, :] = f_l # impose values for labeled nodes
    if len(u_idx) > 0: # only if there are actually unlabeled nodes
        # Inversion per class
        L_uu_inv = np.linalg.inv(L_uu)
        f_u = - L_uu_inv @ L_ul @ f_l
        f[u_idx, :] = f_u
    else:

```

```

    """
    Compute soft HFS.
    f = harmonic function solution
    C = (n x n) diagonal matrix with c_l for labeled samples and c_u
    → otherwise
    """
    # We want  $f = (C^{-1} L + I)^{-1} y$ 
    # Constructing the C matrix (n x n)
    c_l = params['c_l']
    c_u = params['c_u']
    C = np.zeros(num_samples)
    C[l_idx] = c_l
    C[u_idx] = c_u
    C = np.diag(C)

    # Compute  $M = C^{-1} L + I$ 
    invC = np.diag(1.0 / np.diag(C)) # inverse of diagonal
    M = invC @ L + np.eye(num_samples)

    # Solve class by class:  $f = M^{-1} y$ 
    f = np.zeros((num_samples, num_classes))
    for ci in range(num_classes):
        # y_ci is the (n,) vector for class ci
        y_ci = y[:, ci]
        #  $f_{ci} = M^{-1} y_{ci}$ 
        f_ci = np.linalg.solve(M, y_ci)
        f[:, ci] = f_ci

    """
    Return the label assignments from the HFS solution, and the function f
    labels: (n x 1) class assignments [1,2,...,num_classes]
    f : harmonic function solution
    """
    # Deduce the predicted label: argmax over the class dimension (or by sign if
    → binary)
    labels_idx = np.argmax(f, axis=1)
    labels = Cl[1:][labels_idx] # Avoid Cl[0] (undefined label)

    return labels, f

```

```

[4]: def two_moons_hfs(l=4, l_noisy=1, soft=False, dataset='data_2moons_hfs.mat',
    → plot=True, seed=None, per_class=False, print_all=True, **params):
    """
    HFS for two_moons data.

    Parameters
    -----

```

```

l : int
    Number of labeled (unmasked) nodes provided to the HFS algorithm.
l_noisy : int
    Number of *noisy* labels to introduce.
soft : bool
    If true, use soft HFS, otherwise use hard HFS
dataset : {'data_2moons_hfs.mat' or 'data_2moons_hfs_large.mat'}
    Which dataset to use.
plot : bool
    If True, show plots
seed : int
    If not None, set global numpy seed before choosing labels to reveal.
"""
if seed is not None:
    np.random.seed(seed)

# Load the data. At home, try to use the larger dataset.
in_data = loadmat(os.path.join('data', dataset))
X = in_data['X']
Y = np.array(in_data['Y'].squeeze(), dtype=np.uint32)

# infer number of labels from samples
num_samples = np.size(Y, 0)
unique_classes = np.unique(Y)
num_classes = len(unique_classes)

# mask labels
Y_masked = mask_labels(Y, l, per_class=per_class)
# assert len(np.unique(Y_masked)) > 2, "only one class in training data!"
# introduce noise
noise_indices = np.where(Y_masked == 0)[0]
np.random.shuffle(noise_indices)
noise_indices = noise_indices[:l_noisy]
Y_masked[noise_indices] = np.random.choice(unique_classes, l_noisy)

"""
compute hfs solution using either soft_hfs or hard_hfs
"""

# Build graph Laplacian using the parameters:
# params['laplacian_regularization'], params['var'], params['eps'],
# params['k'] and params['laplacian_normalization'].

# W = build_similarity_graph(X, var=params['var'], k=params['k'],
→eps=params['eps'])
L = build_laplacian_regularized(X,
                                ↵
→laplacian_regularization=params['laplacian_regularization'],

```

```

        ↪laplacian_normalization=params['laplacian_normalization'],
        var=params['var'], k=params['k'],
        ↪eps=params['eps']

    )

    labels, f = compute_hfs(L, Y_masked, soft, **params)

    # Visualize results
    if plot:
        plot_classification(X, Y, Y_masked, noise_indices, labels,
        ↪params['var'], params['eps'], params['k'])
        accuracy = np.mean(labels == np.squeeze(Y))
        if print_all:
            print(f"Soft={soft}, Accuracy={accuracy}")
        return X, Y, labels, accuracy

```

1.1.4 Question 1.1 - Report the accuracy you obtained for data_2moons_hfs.mat dataset using hard HFS, when $l=10$ and $l_noisy=0$.

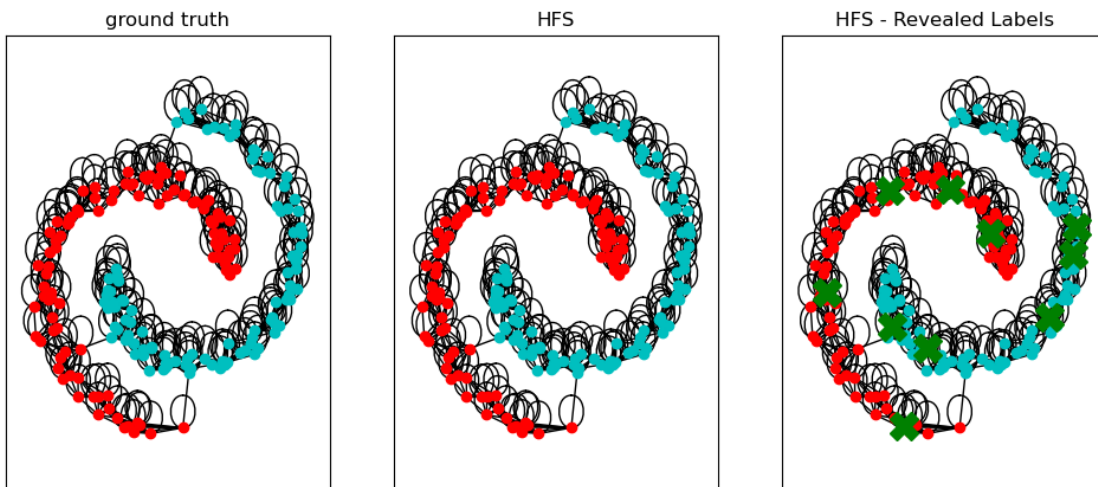
```

[5]: # Threshold eps for epsilon graphs
params['eps'] = 0

# Number of neighbours k for k-nn. If zero, use epsilon-graph
params['k'] = 8

seed = 42
X, Y, hard_labels, hard_accuracy = two_moons_hfs(l=10, l_noisy=0, soft=False,
↪dataset='data_2moons_hfs.mat',
        plot=True, seed=seed, **params)

```



Soft=False, Accuracy=1.0

In this question, we applied Hard HFS with a k-NN graph ($k = 8$) and a Laplacian L_{unn} regularized by $\gamma = 10^{-3}$.

With this configuration, we obtain an **accuracy of 1**, indicating a perfect classification of the unlabeled points.

1.1.5 Question 1.2 - Using data_2moons_hfs_large.mat, run two_moons_hfs several times with $l=4$. What can go wrong?

- Tips:
 - When running two_moons_hfs several times, don't forget to set seed=None

```
[6]: hard_accuracy_mean = []
for ii in range(20):
    X, Y, hard_labels, hard_accuracy = two_moons_hfs(l=4, l_noisy=0, soft=False,
                                                    ↵
    ↪dataset='data_2moons_hfs_large.mat',
                                                    plot=False, seed=None,↵
    ↪**params)
    hard_accuracy_mean.append(hard_accuracy)
print(f"Hard HFS accuracy mean: {np.mean(hard_accuracy_mean)}")
```

```
Soft=False, Accuracy=1.0
Soft=False, Accuracy=1.0
Soft=False, Accuracy=1.0
Soft=False, Accuracy=1.0
Soft=False, Accuracy=1.0
Soft=False, Accuracy=1.0
Soft=False, Accuracy=1.0
Soft=False, Accuracy=1.0
Soft=False, Accuracy=1.0
Soft=False, Accuracy=1.0
Soft=False, Accuracy=1.0
Soft=False, Accuracy=1.0
Soft=False, Accuracy=0.5
Soft=False, Accuracy=1.0
Soft=False, Accuracy=1.0
Soft=False, Accuracy=1.0
Soft=False, Accuracy=1.0
Soft=False, Accuracy=1.0
Soft=False, Accuracy=0.5
Soft=False, Accuracy=1.0
Hard HFS accuracy mean: 0.95
```

If we leave the function as it is, a problem may arise: sometimes, the $l = 4$ labeled points all belong

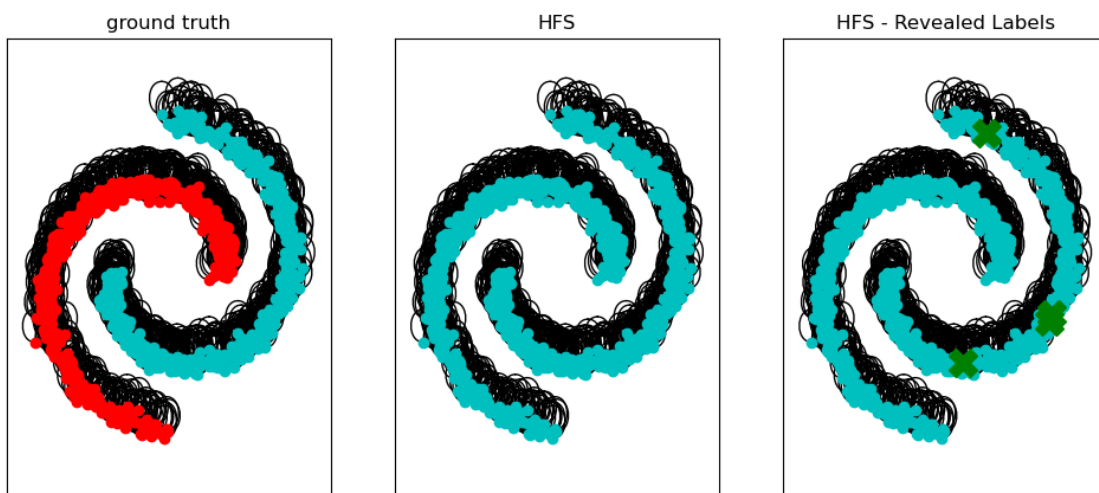
to the same class.

In this case, the semi-supervised algorithm only considers a single class and therefore predicts only that class, which explains the accuracy of 0.5.

Here is an example:

```
[7]: params['k'] = 30

X, Y, hard_labels, hard_accuracy = two_moons_hfs(l=4, l_noisy=0, soft=False,
↪dataset='data_2moons_hfs_large.mat',
plot=True, seed=29,
↪**params)
```



Soft=False, Accuracy=0.5

To avoid this issue, we can modify the `per_class` parameter in `mask_labels(Y, l, per_class=per_class)`.

By setting it to `True`, we ensure that l points are selected per class, thus solving the problem.

```
[8]: hard_accuracy_mean = []
for ii in range(20):
    X, Y, hard_labels, hard_accuracy = two_moons_hfs(l=4, l_noisy=0, soft=False,
↪dataset='data_2moons_hfs_large.mat',
plot=False, seed=None,
↪per_class=True, print_all=False, **params)
    hard_accuracy_mean.append(hard_accuracy)
print(f"Hard HFS accuracy mean: {np.mean(hard_accuracy_mean)}")
```

Hard HFS accuracy mean: 1.0

1.1.6 Question 1.3 - Using data_2moons_hfs.mat, l=10 and l_noisy=5, compare hard HFS to soft HFS. Report the accuracy and comment the results.

```
[21]: # Comparing
params['k'] = 10
params['c_l'] = 0.5
params['c_u'] = 0.1

seed = 72
dataset = 'data_2moons_hfs.mat' # Try also 'data_2moons_hfs_large.mat'
nb_runs = 100

# Storing results
hard_accuracies = []
soft_accuracies = []

for i in range(nb_runs):
    plot = False
    print_all = False

    if i == 0:
        plot = True
        print_all = True
        print("Displaying only the first run")

    # Hard HFS
    _, _, _, hard_accuracy = two_moons_hfs(l=10, l_noisy=5, soft=False,
    ↪dataset=dataset,
                                plot=plot, seed=None,
    ↪print_all=print_all, **params)
    hard_accuracies.append(hard_accuracy)

    # Soft HFS
    _, _, _, soft_accuracy = two_moons_hfs(l=10, l_noisy=5, soft=True,
    ↪dataset=dataset,
                                plot=plot, seed=None,
    ↪print_all=print_all, **params)
    soft_accuracies.append(soft_accuracy)

hard_accuracies = np.array(hard_accuracies)
soft_accuracies = np.array(soft_accuracies)

# Comparing results
print(f"\nResults after {nb_runs} runs:")
print(f"Hard HFS - Mean Accuracy: {hard_accuracies.mean():.4f}, Standard
    ↪Deviation: {hard_accuracies.std():.4f}")
```

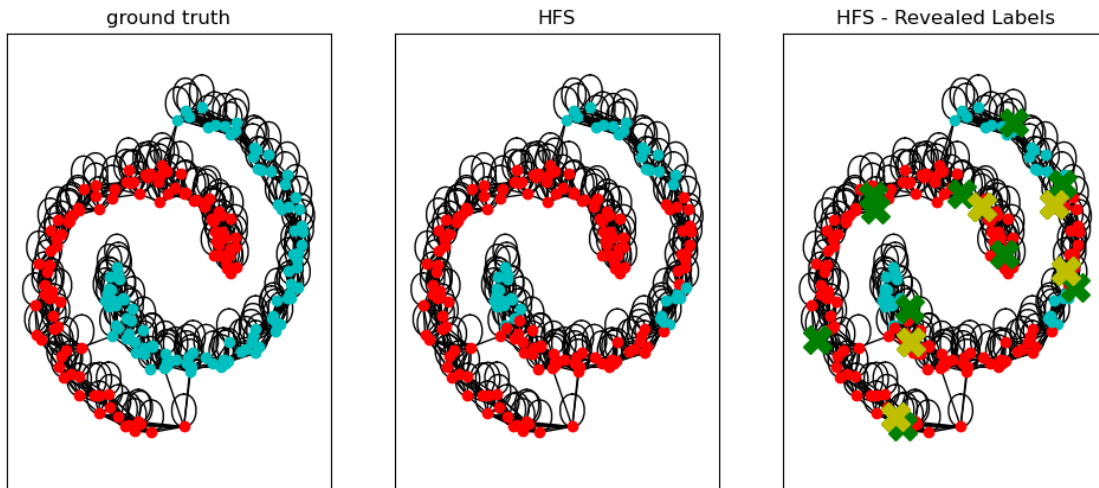
```

print(f"Soft HFS - Mean Accuracy: {soft_accuracies.mean():.4f}, Standard_
↳Deviation: {soft_accuracies.std():.4f}")

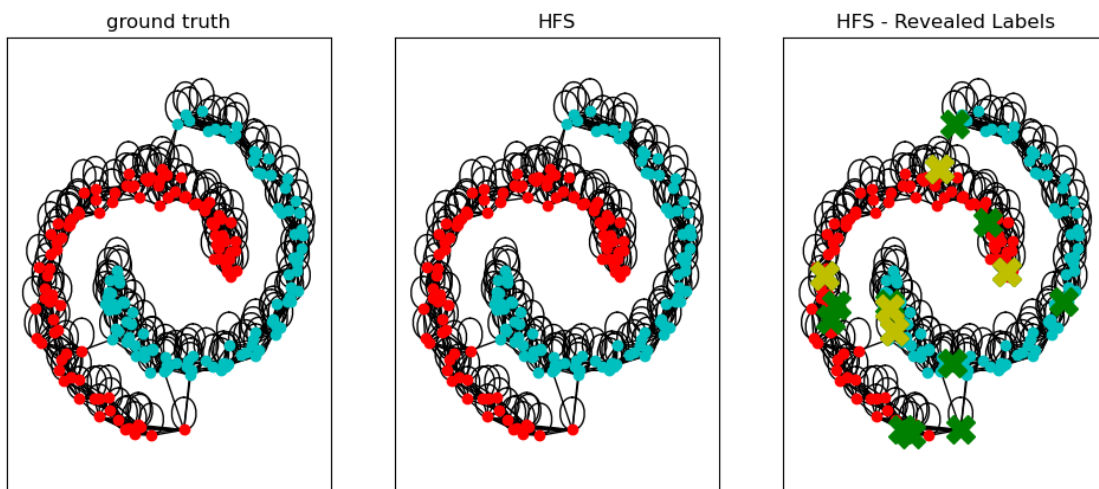
plt.boxplot([hard_accuracies, soft_accuracies], labels=["Hard HFS", "Soft HFS"])
plt.ylabel("Accuracy")
plt.title("Comparison of Accuracy between Hard and Soft HFS")
plt.show()

```

Displaying only the first run



Soft=False, Accuracy=0.725

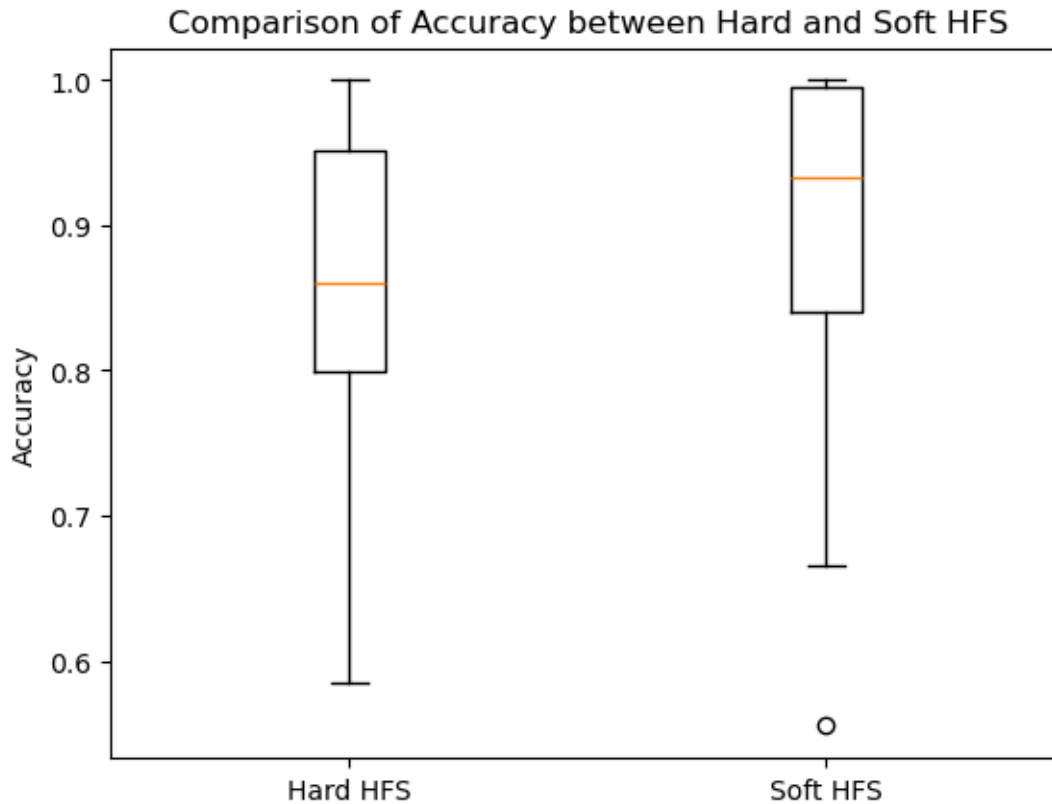


Soft=True, Accuracy=1.0

Results after 100 runs:

Hard HFS - Mean Accuracy: 0.8591, Standard Deviation: 0.1052

Soft HFS - Mean Accuracy: 0.9137, Standard Deviation: 0.0887



With $l=10$ and $l_{\text{noisy}}=5$, 5 labels are randomly replaced, introducing noise.

Hard HFS fixes them, potentially propagating errors, while Soft HFS adjusts them based on the graph structure and the chosen hyperparameters.

After a **grid search optimization**, I found that $k = 10$, $c_l = 0.5$, and $c_u = 0.1$ yield the best results.

With these values, Soft HFS achieves **91.3% accuracy**, compared to **85.9% for Hard HFS**, demonstrating its robustness to noisy labels despite a higher sensitivity to data variations.

1.2 2. Face recognition with HFS

Now, we apply HFS to the task of face recognition, that is, our goal is to classify faces as belonging to different people. Since faces all share common features, it can be a good idea to leverage a large quantity of unlabeled data to improve classification accuracy. In this part of the exercise, you will:

- Extract faces from the images using OpenCV for face detection, and use the same library to apply preprocessing steps;
- Run HFS for classification.

1.2.1 Implementation

Choose the hyperparameters and run HFS for face recognition, using both the small and large dataset. You can try to change the preprocessing steps (e.g. `equalizeHist`, `GaussianBlur`) applied to the images.

Important: make sure your HFS code is able to handle more than two classes!

1.2.2 Question 2.1 - How did you manage to label more than two classes?

To handle more than two classes, I used the `argmax` function along the class dimension to assign each point the class with the highest f value.

The matrix f has a shape of (n, C) , where n is the number of points and C is the number of classes. Each row represents a point, and each column represents a class, with scores indicating the point's affinity to each class.

The line `labels_idx = np.argmax(f, axis=1)` extracts the index of the class with the highest score for each point.

For example, if we have 5 points and 3 classes, a possible matrix f could be:

$$f = \begin{bmatrix} 0.1 & 0.7 & 0.2 \\ 0.3 & 0.4 & 0.3 \\ 0.8 & 0.1 & 0.1 \\ 0.2 & 0.3 & 0.5 \\ 0.4 & 0.5 & 0.1 \end{bmatrix}$$

Applying `argmax(f, axis=1)`, we obtain the predicted class indices:

$$\text{labels_idx} = [1, 1, 0, 2, 1]$$

1.2.3 Question 2.2 - Report the best accuracy you obtained for both (small and augmented) datasets.

- Tips:
 - The small dataset (10 images per person) is loaded with `load_image_data`.
 - Use `load_image_data_augmented` for the augmented dataset (50 images per person).

To preprocess the images, I applied several transformations to enhance data quality before classification:

- `image = cv2.bilateralFilter(image, 5, 30, 30)`: A **bilateral filter** is applied to reduce noise while preserving edges.
- `clahe = cv2.createCLAHE(clipLimit=50.0, tileGridSize=(8,8))` followed by `image = clahe.apply(image)`: A **CLAHE filter** is used to better distinguish facial features.
- `im -= im.mean()` then `im /= (im.std() + 1e-8)`: The image is centered and normalized.

The obtained results are summarized in the following table:

Dataset	Max Accuracy	Laplacian	k-NN
Small (10 images/person)	85%	L_{rw}	5

Dataset	Max Accuracy	Laplacian	k-NN
Large (50 images/person)	73.4%	L_{sym}	8

1.2.4 Question 2.3 - If the accuracy changes when using the augmented dataset, explain why. Does using additional data always increase the performance?

The accuracy decreases with the larger dataset because the semi-supervised algorithm still only has **4 labeled images per person**.

With 10 images per person, this represents **40%** of the labeled data, whereas with 50 images per person, it drops to only **8%**, making label propagation more challenging.

Adding **labeled data** is always beneficial, as it provides more reference points to guide the algorithm.

On the other hand, adding only **unlabeled data** can help if they are well-distributed and representative, but not always, as they can also **degrade performance**.

For example, if they are of “poor quality,” they increase classification uncertainty.

```
[10]: import matplotlib.pyplot as plt
from imageio import imread
import numpy as np
import cv2
import os

from load_images import load_image_data, plot_image_data
from load_images import load_image_data_augmented, plot_image_data_augmented
```

```
[11]: """
Define parameters for face recognition with HFS
"""
params_face_rec = {}
params_face_rec['laplacian_regularization'] = 1.0
params_face_rec['var'] = 10000.0
params_face_rec['eps'] = None
params_face_rec['k'] = None
params_face_rec['laplacian_normalization'] = 'unn'
params_face_rec['c_l'] = 0.95
params_face_rec['c_u'] = 0.05
```

```
[12]: # Function to preprocess the images
# You may try to change it and check the impact on the classification accuracy
def preprocess_image(image):
    """
    Parameters
    -----
    image : array
        (width, height) array representing a grayscale image
```

Returns

(96, 96) preprocessed image

"""

output_frame_size = 96 # do not change the output frame size!

image = cv2.bilateralFilter(image, 5, 30, 30)

image = cv2.equalizeHist(image)

clahe = cv2.createCLAHE(clipLimit=50.0, tileGridSize=(8,8))

image = clahe.apply(image)

image = cv2.GaussianBlur(image, (3, 3), 0)

im = cv2.resize(image, (output_frame_size, output_frame_size)).astype(float)

im -= im.mean()

im /= im.max()

im /= (im.std()+1e-8)

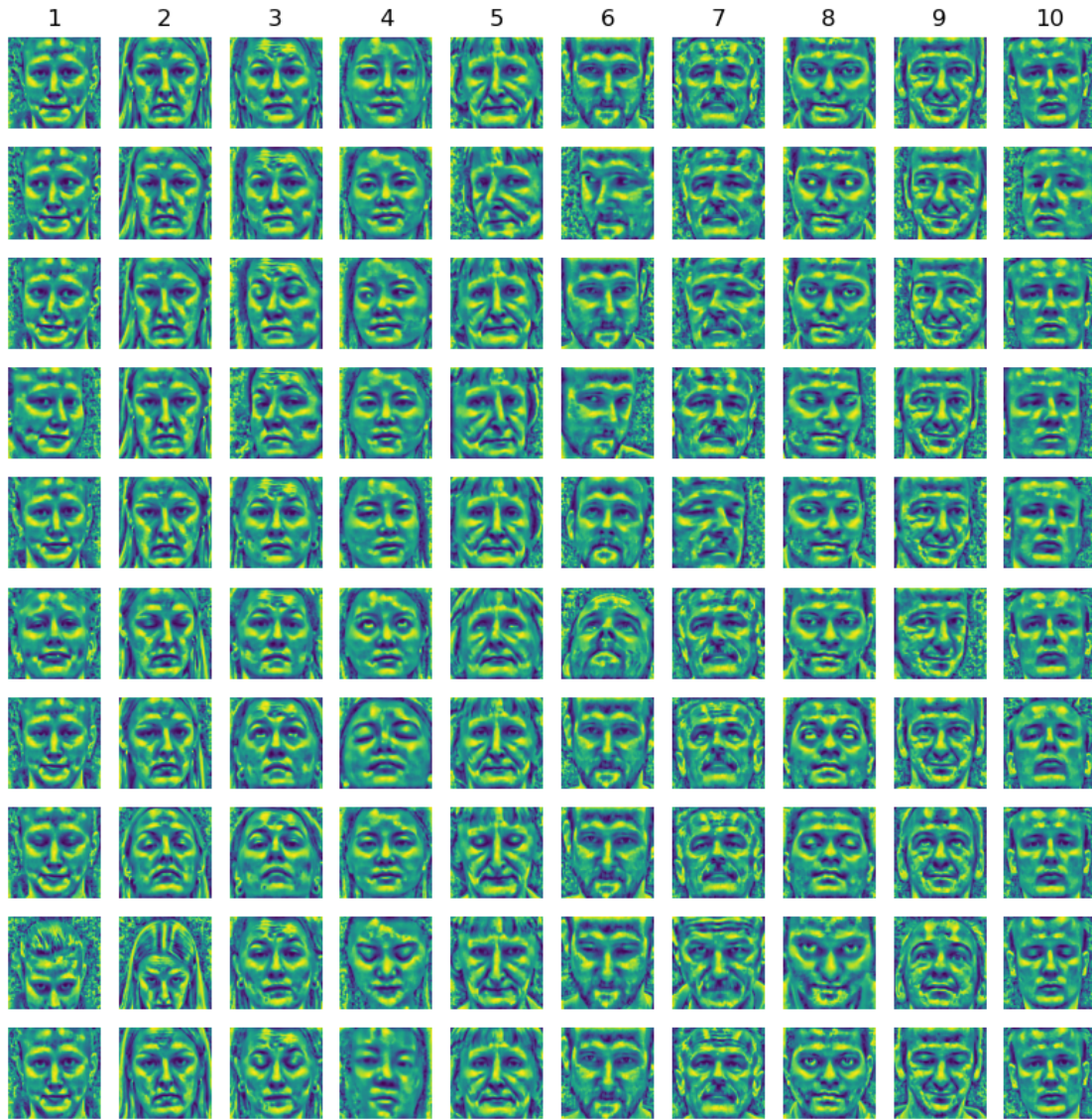
image = im

return image

```
[13]: # 10 images per person
np.random.seed(456) # set seed, since labels are masked randomly
images, labels, masked_labels = load_image_data(preprocess_image)

# # 50 images per person
images_a, labels_a, masked_labels_a = load_image_data_augmented(preprocess_image)
# plot_image_data_augmented(images_a)

# Uncomment below if you want to visualize the images
plot_image_data(images)
print(images.shape)
print(masked_labels.reshape(-1, 10))
```



(100, 9216)

```
[[ 1.  2.  0.  0.  0.  6.  7.  8.  0.  0.]
 [ 1.  2.  0.  4.  0.  0.  7.  0.  0.  0.]
 [ 1.  0.  3.  0.  0.  0.  0.  0.  9.  0.]
 [ 1.  0.  3.  0.  0.  6.  0.  0.  9. 10.]
 [ 0.  0.  0.  4.  5.  6.  0.  8.  0.  0.]
 [ 0.  0.  0.  4.  5.  0.  0.  0.  0.  0.]
 [ 0.  2.  3.  0.  0.  0.  0.  0.  9. 10.]
 [ 0.  0.  0.  0.  5.  6.  7.  0.  0. 10.]
 [ 0.  0.  0.  0.  5.  0.  0.  8.  9. 10.]
 [ 0.  2.  3.  4.  0.  0.  7.  8.  0.  0.]]
```

```
[14]: params_face_rec['var'] = 10000.0
params_face_rec['eps'] = None
params_face_rec['k'] = 5

params_face_rec['laplacian_regularization'] = 0.001
params_face_rec['laplacian_normalization'] = 'rw'

params_face_rec['c_l'] = 0.9
params_face_rec['c_u'] = 0.1

# graph Laplacian
L = build_laplacian_regularized(images,
                                params_face_rec['laplacian_regularization'],
                                params_face_rec['var'],
                                params_face_rec['eps'],
                                params_face_rec['k'],
                                params_face_rec['laplacian_normalization'])

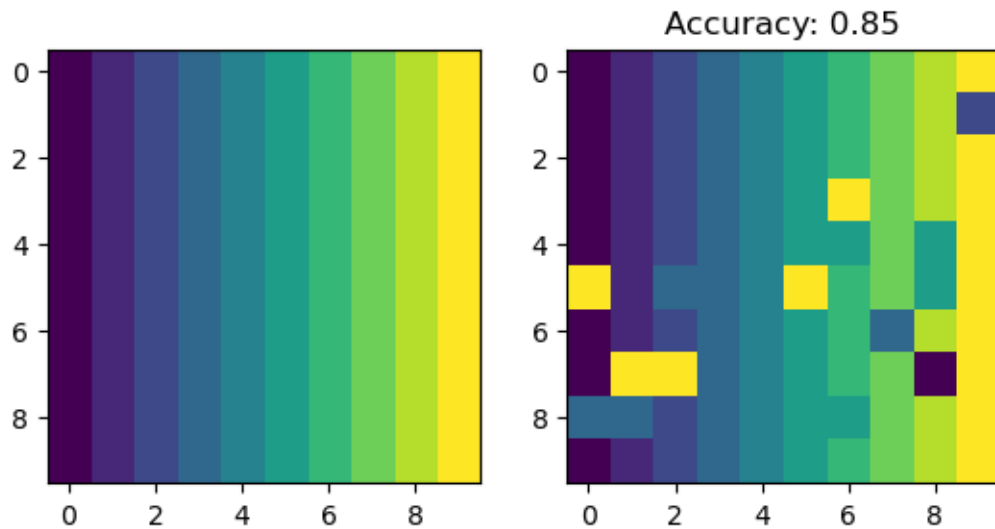
# Run HFS
predicted_labels, f = compute_hfs(L, masked_labels, soft=True, **params_face_rec)
accuracy = np.equal(predicted_labels, labels).mean()
print("Accuracy = ", accuracy)

print(masked_labels)
print(predicted_labels)
print(labels)
# Visualize predicted vs true labels
plt.subplot(121)
plt.imshow(labels.reshape((-1, 10)))
plt.subplot(122)
plt.imshow(predicted_labels.reshape((-1, 10)))
plt.title("Accuracy: {}".format(accuracy))
plt.show()
```

Accuracy = 0.85

```
[ 1.  2.  0.  0.  0.  6.  7.  8.  0.  0.  1.  2.  0.  4.  0.  0.  7.  0.
  0.  0.  1.  0.  3.  0.  0.  0.  0.  0.  9.  0.  1.  0.  3.  0.  0.  6.
  0.  0.  9. 10.  0.  0.  0.  4.  5.  6.  0.  8.  0.  0.  0.  0.  0.  4.
  5.  0.  0.  0.  0.  0.  0.  2.  3.  0.  0.  0.  0.  0.  9. 10.  0.  0.
  0.  0.  5.  6.  7.  0.  0. 10.  0.  0.  0.  0.  5.  0.  0.  8.  9. 10.
  0.  2.  3.  4.  0.  0.  7.  8.  0.  0.]
[ 1.  2.  3.  4.  5.  6.  7.  8.  9. 10.  1.  2.  3.  4.  5.  6.  7.  8.
  9.  3.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10.  1.  2.  3.  4.  5.  6.
 10.  8.  9. 10.  1.  2.  3.  4.  5.  6.  6.  8.  6. 10. 10.  2.  4.  4.
  5. 10.  7.  8.  6. 10.  1.  2.  3.  4.  5.  6.  7.  4.  9. 10.  1. 10.
 10.  4.  5.  6.  7.  8.  1. 10.  4.  4.  3.  4.  5.  6.  6.  8.  9. 10.
  1.  2.  3.  4.  5.  6.  7.  8.  9. 10.]
```

```
[ 1  2  3  4  5  6  7  8  9 10  1  2  3  4  5  6  7  8  9 10  1  2  3  4
  5  6  7  8  9 10  1  2  3  4  5  6  7  8  9 10  1  2  3  4  5  6  7  8
  9 10  1  2  3  4  5  6  7  8  9 10  1  2  3  4  5  6  7  8  9 10  1  2
  3  4  5  6  7  8  9 10  1  2  3  4  5  6  7  8  9 10  1  2  3  4  5  6
  7  8  9 10]
```



Pour les données augmentées :

```
[15]: params_face_rec['var'] = 10000.0
      params_face_rec['eps'] = None
      params_face_rec['k'] = 8

      params_face_rec['laplacian_regularization'] = 0.001
      params_face_rec['laplacian_normalization'] = 'sym'

      params_face_rec['c_l'] = 0.9
      params_face_rec['c_u'] = 0.1

      # graph Laplacian
      L_a = build_laplacian_regularized(images_a,
                                         params_face_rec['laplacian_regularization'],
                                         params_face_rec['var'],
                                         params_face_rec['eps'],
                                         params_face_rec['k'],
                                         params_face_rec['laplacian_normalization'])

      # Run HFS
```

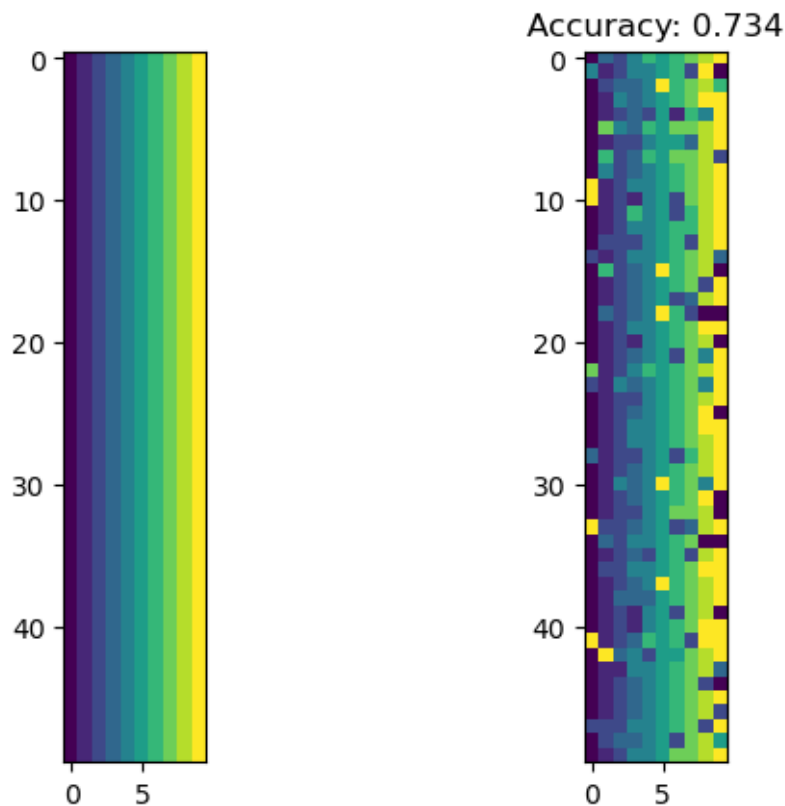
```

predicted_labels_a, f = compute_hfs(L_a, masked_labels_a, soft=True,
↪**params_face_rec)
accuracy_a = np.equal(predicted_labels_a, labels_a).mean()
print("Accuracy = ", accuracy_a)

# print(masked_labels_a)
# print(predicted_labels_a)
# print(labels_a)
# Visualize predicted vs true labels
plt.subplot(121)
plt.imshow(labels_a.reshape((-1, 10)))
plt.subplot(122)
plt.imshow(predicted_labels_a.reshape((-1, 10)))
plt.title("Accuracy: {}".format(accuracy_a))
plt.show()

```

Accuracy = 0.734



1.3 3. Online SSL

Now, instead of having all the data available at once, images will be received online: at each time t , a new image x_t is observed and the algorithm has to output a label y_t .

Use the function `create_user_profile` to capture a training set of labeled data (of your face and someone else). The faces will be preprocessed and saved in the folder `data/faces`. They will be loaded by `online_face_recognition`.

1.3.1 Implementation

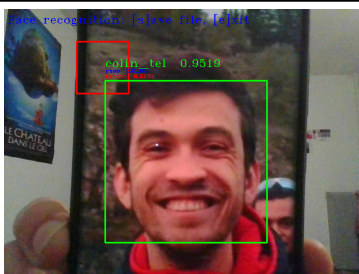
Choose the hyperparameters and complete the functions `online_ssl_update_centroids` and `online_ssl_compute_solution`.

Modify your code to be able to disregard faces it cannot recognize.

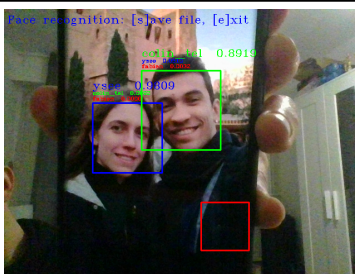
- Tips:
 - You can use the functions `build_similarity_graph` and `build_laplacian`.

1.3.2 Question 3.1 - Attach to this notebook some of the resulting frames of online face recognition.

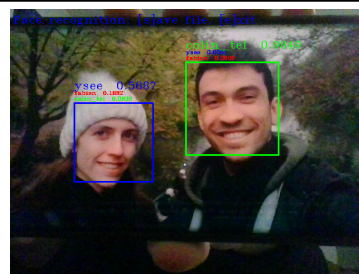
- Tips:
 - You can save the resulting frame and add it to the notebook in a markdown cell as `![title](picture.png)`



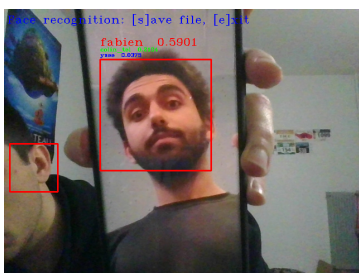
First result



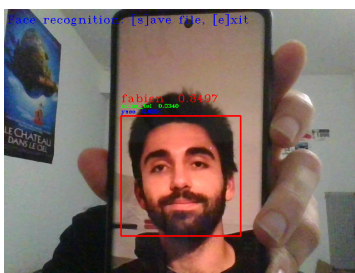
Result with Ysée and me



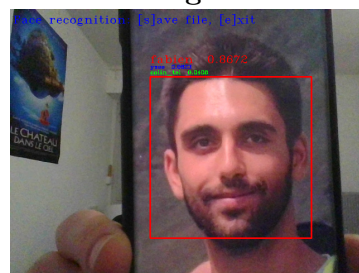
Result with Ysée and me wearing a hat



Fabien result tilted



Fabien result



Fabien result

Analysis of the First Results For these experiments, I used the following parameters:

- $k = 10$ for the k-NN graph,
- $\sigma^2 = 10000$ for the Gaussian kernel variance (value given in the problem statement),
- **Laplacian regularization:** $\gamma = 0.01$,
- **Normalized Laplacian:** L_{rw} ,
- **Recognition threshold** set to **0.3**.

The choice of k and **variance** strongly influences face recognition and can drastically alter the results.

I recorded three different training sets: **colin_tel**, **fabien**, and **ysée**, and after several adjustments, I achieved very good results when testing on other photos.

And, **importantly**: these test images were not part of the initially labeled data for the semi-supervised algorithm.

Here are some observations:

- In the two “Ysée and me” result photos, the algorithm detects Ysée less accurately when she is wearing a hat.
- For Fabien, the algorithm struggles with the first photo where he is **tilted**, whereas the score is much higher for the other images.

These results show that **posture and accessories** (such as a hat) can also affect classification.

1.3.3 Question 3.2 - What strategy did you use to label a face as unknown? Attach to this notebook an example of a unknown face being correctly labeled as unknown.

- Tips
 - If you identify a face as unknown, you can return `[("unknown", score)]` from the function `online_ssl_compute_solution`.

To identify a face as **unknown**, I use two criteria:

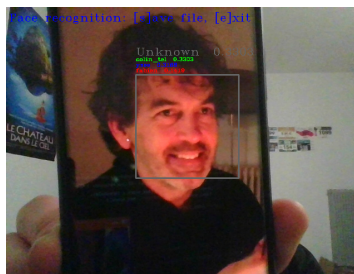
- If the **predicted label** is 0, it means the algorithm did not assign a known class.
- If the **maximum classification score** is below a **threshold set at 0.5**, the model considers that the image does not match any of the known classes.

This approach ensures that only faces with a high similarity to the recorded centroids are classified, thereby reducing recognition errors.

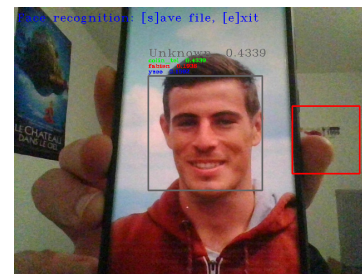
If a face is detected as unknown, the function returns `[("Unknown", score)]`.



My uncle



My dad



My friend Quentin

Here are three results where the algorithm displays “Unknown”.

It can only recognize **Fabien**, **Ysée**, or **me**, so it assigns relatively low scores to other faces.

However, we can notice that my dad and my friend Quentin receive slightly higher scores, which makes sense since they resemble me the most.

```
[16]: import matplotlib.pyplot as plt
import numpy as np
import cv2 as cv
import os
import sys
```

```

from scipy.spatial import distance
import scipy.io as sio

from helper_online_ssl import create_user_profile, online_face_recognition

```

```

[17]: """
      Define parameters for face recognition with HFS
      """
      params_online_ssl = {}
      params_online_ssl['laplacian_regularization'] = 0.01
      params_online_ssl['var'] = 10000.0
      params_online_ssl['eps'] = None
      params_online_ssl['k'] = 10
      params_online_ssl['laplacian_normalization'] = 'rw'

      params_online_ssl['threshold'] = 0.3

[18]: class IncrementalKCenters:
      def __init__(self, labeled_faces, labels, label_names, max_num_centroids=50):
          # Number of labels
          self.n_labels = max(labels)

          # Dimension of the input image
          self.image_dimension = labeled_faces.shape[1]

          # Check input validity
          assert (set(labels) == set(
              range(1, 1 + self.n_labels))), "Initially provided faces should be
          →labeled in [1, max]"
          assert (len(labeled_faces) == len(labels)), "Initial faces and initial
          →labels must have the same size"

          # Number of labeled faces
          self.n_labeled_faces = len(labeled_faces)

          # Model parameter: maximum number of stored centroids
          self.max_num_centroids = max_num_centroids

          # Model centroids (initial labeled faces). Shape = (number_of_centroids,
          →dimension)
          self.centroids = labeled_faces

          # Centroids labels
          self.Y = labels

          # Label names (= user names)
          self.label_names = label_names

```

```

# Variables initialized in online_ssl_update_centroids()
self.centroids_distances = None
self.taboo = None
self.V = None
self.init = True

# Index of  $x_t$  (initialized later)
self.last_face = None

def initialize(self):
    """
    Initialization after the first time the maximum number of centroids is
    ↪reached.
    """
    # Compute the centroids distances
    self.centroids_distances = distance.cdist(self.centroids, self.centroids)

    # Set labeled nodes and self-loops as infinitely distant to avoid
    ↪merging labeled centroids
    np.fill_diagonal(self.centroids_distances, +np.Inf)
    self.centroids_distances[0:self.n_labeled_faces, 0:self.n_labeled_faces]
    ↪= +np.Inf

    # Put labeled nodes in the taboo list
    self.taboo = np.array(range(self.centroids.shape[0])) < self.
    ↪n_labeled_faces

    # Initialize multiplicity
    self.V = np.ones(self.centroids.shape[0])

def online_ssl_update_centroids(self, face):
    """
    Update centroids, multiplicity vector  $V$ , and labels  $Y$ .

    Note: In  $Y$ , set the label to 0 for unlabeled faces.

    Parameters
    -----
    face : array
        New sample

    Returns
    -----
    List with scores for each possible label:
        [(label_1, score_1), (label_2, score_2), ...]

```

```

"""

    assert (self.image_dimension == len(face)), "New image is not of the_
↳correct size"

    n_current = self.centroids.shape[0]

    # Case 1: Maximum number of centroids has been reached
    if n_current >= self.max_num_centroids + 1:
        if self.init:
            # Initialization after the first time the maximum number of_
↳centroids is reached
            self.initialize()
            self.init = False
        """
        Find c_rep and c_add following Algorithm 1.

        - c_1, c_2 = two closest centroids (minimum distance) such that at_
↳least one is not in self.taboo.
        - c_rep = centroid in {c_1, c_2} that is in self.taboo. If neither_
↳is in self.taboo, c_rep is the one
            with the largest multiplicity.
        - c_add = centroid in {c_1, c_2} that is not c_rep.
        """

        i, j = np.unravel_index(np.argmin(self.centroids_distances), self.
↳centroids_distances.shape)
        c_1, c_2 = i, j

        # Choose c_rep, c_add
        if self.taboo[c_1] and not self.taboo[c_2]:
            c_rep, c_add = c_1, c_2
        elif self.taboo[c_2] and not self.taboo[c_1]:
            c_rep, c_add = c_2, c_1
        else:
            # Neither c_1 nor c_2 are taboo
            # => choose c_rep as the one with the largest multiplicity
            if self.V[c_2] <= self.V[c_1]:
                c_rep, c_add = c_1, c_2
            else:
                c_rep, c_add = c_2, c_1

        """
        Update data structures: self.centroids and self.V
        """

        # MERGING: update centroid c_rep (weighted average)

```

```

old_rep = self.centroids[c_rep].copy()
old_add = self.centroids[c_add].copy()
wrep = self.V[c_rep]
wadd = self.V[c_add]
new_rep = (old_rep * wrep + old_add * wadd) / (wrep + wadd)
self.centroids[c_rep] = new_rep
self.V[c_rep] += wadd # Sum multiplicities

# Replace c_add with the new face x_t
self.centroids[c_add] = face
self.V[c_add] = 1
self.Y[c_add] = 0 # Unlabeled
self.last_face = c_add

"""
Update the matrix containing distances.
"""

dist_row = distance.cdist(np.array([self.centroids[c_add]]), self.
↪centroids)[0]
dist_row[c_add] = +np.inf
self.centroids_distances[c_add, :] = dist_row
self.centroids_distances[:, c_add] = dist_row

# Case 2: Create a new centroid with the face
# Remark: The multiplicities vector self.V is initialized in self.
↪initialize()
else:
    self.centroids = np.vstack([self.centroids, face])
    self.Y = np.append(self.Y, 0) # Label 0 => unlabeled
    self.last_face = len(self.centroids) - 1

def online_ssl_compute_solution(self):
    """
Returns a prediction corresponding to self.last_face.
"""

    n_centroids = self.centroids.shape[0]

    # Multiplicity matrix
    if self.init:
        V = np.eye(n_centroids)
        self.last_face = n_centroids - 1
    else:
        V = np.diag(self.V)

    # Build quantized graph and its regularized Laplacian

    W_tilde = build_similarity_graph(

```

```

        self.centroids,
        var=params_online_ssl['var'],
        eps=params_online_ssl['eps'],
        k=params_online_ssl['k']
    )

    W_tilde[W_tilde < 1e-3] = 0.0

    W_q = V @ W_tilde @ V

    L = build_laplacian(
        W_q,
        laplacian_normalization=params_online_ssl.
→get('laplacian_normalization', 'unn')
    )

    gamma = params_online_ssl.get('laplacian_regularization', 0.0)
    Q = L + gamma * np.eye(n_centroids)

    # Compute the hard HFS solution f.
    labels_pred, f = compute_hfs(Q, self.Y, soft=False, **params_online_ssl)

    # Return the score for each possible label
    num_classes = len(np.unique(self.Y)) - 1
    label_scores = []
    for ii in range(num_classes):
        label_name = self.label_names[ii]
        score = f[self.last_face, ii]
        label_scores.append((label_name, score))

    # Handle unknown faces
    # For example, if predicted label == 0 => unknown
    # OR if max of f[self.last_face,:] is < a certain threshold => unknown
    predicted_label = labels_pred[self.last_face] # e.g., 0 => unlabeled
    max_score = np.max(f[self.last_face])
    threshold = params_online_ssl['threshold'] # Threshold for considering
→unknown
    if (predicted_label == 0) or (max_score < threshold):
        label_scores.append(("Unknown", max_score))

    return label_scores

```

```

[19]: # create_user_profile('fabien')          # choose your names here :)

online_face_recognition(['colin_tel', 'ysee', 'fabien'], IncrementalKCenters,
→n_pictures=30)

```

Profile found with 30 images.

Profile found with 30 images.
Profile found with 30 images.

1.3.4 Question 4.1 - You can now classify faces as either known (i.e. belong to a class) or unknown (i.e. belong to no class). How would you modify the algorithm to automatically learn to classify new classes? That is, how can you modify the algorithm so that faces that appear unfrequently are labeled as unknown, but once a specific face has been seen enough times it gets assigned an automatic label (e.g. AutoBob) and from that point it is treated as a new class.

- Tips:
 - Think back to the idea of incremental clustering as unsupervised classification

A simple way to approach this problem is to monitor frequently appearing unknown faces. I see three main ideas to implement this:

1. Replacing an existing label

As soon as we observe a certain number of consecutive images marked as *unknown* but sufficiently similar, we recycle an old, rarely used label to represent this new recurring face. In other words, instead of creating a new label, we **remove an unused label and reassign it to the new face**.

2. Adding a new label

The same detection principle (based on multiple consecutive and similar *unknown* faces), but this time we **increment** the list of existing classes: for example, we create “AutoBob” without sacrificing any previously assigned labels.

3. Incremental clustering approach

The most comprehensive idea: we store all *unknown* faces in small *centroids*, similar to a sub-algorithm of *k-centers*.

Once one of these clusters exceeds a certain multiplicity threshold (e.g., 5 associated images), it is automatically assigned a new label and treated as its own class.

This is a form of **incremental clustering “within” the global incrementing scheme**: the sub-clustering groups recurring unknowns, and if a group becomes large enough, it becomes a real label in the global algorithm.

The first two ideas seem easier to implement, while the last one is more complex but also more relevant.

1.3.5 Question 4.2 - In class we considered different kinds of metric for (semi) supervised learning. Looking at the face classification task, try to quantify how the offline-online, exact-quantized, and inductive-transductive axes influence each other. In particular given l labeled faces, $u = N - l$ unlabeled faces, and m extra/test faces design an experimental study to quantify these trade-offs, both in terms of transductive and inductive error, as well as online/batch error. Examples of combinations that can be used to study these axes are:

- Supervised vs Semi-supervised
 - A comparison between a supervised learner (of your choice) trained on the l labeled faces and a semi-supervised learner (of your choice) trained on the $l + u$ labeled and unlabeled

faces

- A comparison between a supervised learner (of your choice) trained on the N labeled and unlabeled (you reveal everything here) faces and a semi-supervised learner (of your choice) trained on the $l + u$ labeled and unlabeled faces.
- Inductive vs Transductive
 - A comparison between a supervised learner (of your choice) trained on the l labeled faces and a semi-supervised learner (of your choice) trained on the $l + u$ labeled and unlabeled faces evaluated on the N revealed points and then on the m unrevealed points.
- Supervised vs Semi-supervised and Inductive vs Transductive
 - A comparison between a supervised learner (of your choice) trained on the N labeled and unlabeled (you reveal everything here) faces and a semi-supervised learner (of your choice) trained on the $l + u$ labeled and unlabeled faces, evaluated on the N revealed points and then on the m unrevealed points.
- Online vs Batch
 - A comparison between a supervised learner (of your choice) trained on the N labeled faces and an online supervised learner (of your choice) trained revealing the N labels one at a time
- Exact vs Quantized
 - A comparison between a semi-supervised learner (of your choice) trained on the N labeled and unlabeled (you reveal everything here) faces using a certain memory budget, and the same learner with a constrained memory budget.
- Exact vs Quantized and Online vs Batch and Inductive vs Transductive
 - A comparison between a semi-supervised online learner (of your choice) trained with and without quantization, evaluated both on the N faces revealed during training, and m faces unrevealed. You can further compare the online performance of the learner against the performance of an “hindsight” learner that saw the labels all at once