# TP2: Semi-Supervised Learning (SSL)

omar (dot) darwiche-domingues (at) inria.fr
pierre (dot) perrault (at) inria.fr

November 12, 2019

**Abstract**

The report and the code are due in 2 weeks (deadline 23:59 26/11/2019). You will find instructions on how to submit the report on piazza, as well as the policies for scoring and late submissions. All the code related to the TD must be submitted.

## 1 Harmonic Function Solution (HSF)

Semi-supervised learning (SSL) is a field of machine learning that studies learning from both labeled and unlabeled examples. This learning paradigm is extremely useful for solving real-world problems, where data is often abundant but the resources to label them are limited. We will use the following notation:

- $G = (V, E)$ is a weighted graph where $V = \{x_1, \ldots, x_n\}$ is the vertex set and $E$ is the edge set
- Associated with each edge $e_{ij} \in E$ is a weight $w_{ij}$. If there is no edge present between $x_i$ and $x_j$, $w_{ij} = 0$.
- Associated with each node there is a label with value $y_i \in \mathbb{R}$.
- Only a subset $S \subset V$, $|S| = l$ of the nodes' label is revealed to the learner, the remaining $u = n - l$ nodes are placed in the subset $T = V \backslash S$.

We wish to predict the values of the vertices in $T$. In doing so, we would like to exploit the structure of the graph. Since we believe that nodes close in the graph (similar) should share similar labels, we would like to have each node be

surrounded by a majority of nodes with the same label. In particular, if our recovered labels are encoded in the vector $f \in \mathbb{R}^n$ we can express this principle as

$$f_i = \frac{\sum_{i \sim j} f_j w_{ij}}{\sum_{i \sim j} w_{ij}}$$

where we use $f_i = f(x_i)$.

If we want to give a motivation for this belief, we can draw this interpretation in terms of random walks. In particular if the weight $w_{ij}$ expresses the tendency of moving from node $x_i$ to node $x_j$, then we can encode transition probabilities as $P(j|i) = \frac{w_{ij}}{\sum_k w_{ik}}$. If we compute a stationary distribution, it will be a valid solution for our criteria.

The same preference can be expressed with a penalization on all solutions that are not "smooth" w.r.t. the graph weights. This can be expressed as a function using the Laplacian matrix $L$

$$\Omega(\, f) = \sum_{i \sim j} w_{ij} (f_i - f_j)^2 = \, f^T L \, f$$

Initially, we assume that the labels that we receive with the data are always correct, and it is in our best interest to enforce them exactly. In practice, this means that we have first to guarantee that the labeled point will be correctly labeled, and then to promote smoothness on the unlabeled points. As an optimization problem, this can be formulated as

$$\min_{f \in \{\pm 1\}^n} \infty \sum_{i=1}^{l} \left(f(x_i) - y_i\right)^2 + \lambda \sum_{i,j=1}^{n} w_{ij} \left(f(x_i) - f(x_j)\right)^2 .$$

If we relax the integer constraints to be real and fully enforce the label constraints, we end up with

$$\begin{aligned} &\min_{f \in \mathbb{R}^n} \sum_{i,j=1}^{n} w_{ij} \left(f(x_i) - f(x_j)\right)^2 \\ &s.t. \quad y_i = f(x_i) \quad \forall i = 1, \ldots, l \end{aligned}$$

1.1. Complete `hard_hfs` and `two_moons_hfs` . Select uniformly at random 4 labels ($S$), and compute the labels for the unlabeled nodes ($T$) using the hard-HFS formula. Plot the resulting labeling and the accuracy.

1.2. At home, run `two_moons_hfs` using the `data_2moons_large.mat`, a dataset with 1000 samples. Continue to uniformly sample only 4 labels. What can go wrong?

What happens when the labels are noisy, or in other words when some of the samples are mislabeled? In some cases relabeling nodes might be beneficial. For this reason, soft-HFS seeks to strike a balance between smoothness and satisfying the labels in the training data. Define $C$ and $y$ as

$$C_{ii} = \begin{cases} c_l & \text{for labeled examples} \\ c_u & \text{otherwise.} \end{cases} \qquad y_i = \begin{cases} \text{true label} & \text{for labeled examples} \\ 0 & \text{otherwise.} \end{cases}$$

Then soft-HFS's objective function is

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

1.3. Complete `soft_hfs` and test it with `two_moons_hfs`. Now complete `hard_vs_soft_hfs`. Compare the results you obtain with soft-HFS and hard-HFS.

## 2 Face recognition with HFS

We can now start to think of applying HFS to the task of face recognition, or in other words to classify faces as belonging to different persons. Since faces all share common features, it is a effective idea to leverage a large quantity of unlabeled data to improve classification accuracy. As our first exercise, we will begin by completing the code necessary to define the similarity between faces. To extract the faces we will use OpenCV face detection software to localize them, and the same library to apply a series of preprocessing steps to improve their quality. Complete `offline_face_recognition` to classify the faces, and plot the results.

2.1. How did you manage to label more than two classes?

2.2. Which preprocessing steps (e.g. cv.GaussianBlur, cv.equalizeHist) did you apply to the faces before constructing the similarity graph? Which gave the best performance?

2.3. Does HFS reach good performances on this task?

Now try to augment the dataset with more unlabeled data, and test if additional data improves performance. In the archive `extended_dataset.tar.gz`

you will find additional pictures to expand (augment) the dataset you already processed. The file `offline_face_recognition_augmunted` starts as an identical copy of `offline_face_recognition`. Modify it to handle your extended dataset and answer the following questions:

2.4. Did adding more data to the task improve performance? If so, which kind of additional data improves performance?

2.5. If the performance does not improve when including additional data, try to justify why. Which kind of additional data degrades performance instead of improving it?

# 3 Online SSL

Semi-Supervised Learning was introduced as a solution designed for problems where collecting large quantities of supervised training data (usually labels) is not possible. On the other hand, in SSL scenarios it is usually inexpensive to obtain more samples coming from the same process that generated the labeled samples, but without a label attached. A simple approach is to collect as much additional data as possible, and then run our algorithm on all of it, in a batch or off-line manner. But in other cases it is possible to start the learning process as early as possible, and then refine the solution as the quantity of available information increases. An important example of this approach is stream processing. In this setting, a few labeled examples are provided in advance and set the initial bias of the system while unlabeled examples are gathered online and update the bias continuously. In the online setting, learning is viewed as a repeated game against a potentially adversarial nature. At each step $t$ of this game, we observe an example $x_t$ , and then predict its label $f_t$ . The challenge of the game is that after the game started we do not observe any more true label $y_t$. Thus, if we want to adapt to changes in the environment, we have to rely on indirect forms of feedback, such as the structure of data. Another difficulty posed by online learning is computational cost. In particular, when $t$ becomes large, a naive approach to hard-HFS, such as recomputing the whole solution from scratch, has prohibitive computational costs. Especially in streaming settings where near real-time requirements are common, it is imperative to have an approach that has scalable time and memory costs. Because most operations related to hard-HFS scale with the number of nodes, one simple but effective approach to scale this algorithm is subsampling, or in other words to compute an approximate solution on a smaller subset of the data in a way that generalizes well to the whole dataset. Several techniques can give different guarantees for the approximation. As you saw in class, incremental $k$-centers **?** guarantees on the distortion introduced by the approximation allows us to provide theoretical guarantees.

**Algorithm 1** Incremental $k$-centers (simplified)

**Input:** an unlabeled $x_t$, a list of centroids $C_{t-1}$, a list of multiplicities $v_{t-1}$, taboo list $b$ containing the labeled centroids.
**if** $(|C_{t-1}| = k)$ **then**
  $c_1, c_2 \leftarrow$ two closest centroids such that at least one of them is not in $b$.
  // Decide which centroid is $c_{\text{rep}}$, that will represent both $c_1$ and $c_2$, and which centroid is $c_{\text{add}}$, that will represent the new point $x_t$.
  **if** $c_1$ in b **then**
    $c_{\text{rep}} \leftarrow c_1$
    $c_{\text{add}} \leftarrow c_2$
  **else if** $c_2$ in b **then**
    $c_{\text{rep}} \leftarrow c_2$
    $c_{\text{add}} \leftarrow c_1$
  **else if** $v_{t-1}(c_2) \leq v_{t-1}(c_1)$ **then**
    $c_{\text{rep}} \leftarrow c_1$
    $c_{\text{add}} \leftarrow c_2$
  **else**
    $c_{\text{rep}} \leftarrow c_2$
    $c_{\text{add}} \leftarrow c_1$
  **end if**
  $v_t \leftarrow v_{t-1}$
  $v_t(c_{\text{rep}}) \leftarrow v_t(c_{\text{rep}}) + v_t(c_{\text{add}})$
  $c_{\text{add}} \leftarrow x_t$
  $v_t(c_{\text{add}}) = 1$
**else**
  $C_t \leftarrow C_{t-1}$.append$(x_t)$
  $v_t \leftarrow v_{t-1}$.append$(1)$
**end if**

**Algorithm 2** Online HFS with Graph Quantization

**Input:** $t$, a list of centroids $C_t$, a list of multiplicities $v_t$ and labels $y$.
$V \leftarrow \text{diag}(v_t)$
$[\widetilde{W}_q]_{ij} \leftarrow$ weight between centroids $i$ and $j$.
Compute the Laplacian $L$ of the graph represented by $W_q = V\widetilde{W}_q V$
// Infer labels using hard-HFS.
$\widehat{y}_t \leftarrow \text{hardHFS}(L, y)$
// Remark: with the preceding construction of the centroids, $x_t$ is always present in the reduced graph and does not share the centroid with any other node.

Some practical considerations:

- The labeled nodes are fundamentally different from unlabeled ones. Because of this, it is always a good idea to keep them separate, and never

merge them in a centroid. In the implementation this is accomplished with a taboo list $b$ that keeps track of nodes that cannot be merged together.

- In streaming applications, it is not always possible to stop execution to partition the centroids, and it is often preferable to pay a small price at every step to keep execution smooth. In our case, the centroids are updated at every step.
- Whenever a new node arrives, and we have too many centroids, we choose the two closest centroids $c_{\text{add}}$ and $c_{\text{rep}}$. $c_{\text{add}}$ will forget the old centroid and will point to the new sample that just arrived, and $c_{\text{rep}}$ will take care of representing all nodes that belonged to $c_{\text{add}}$.

Use the function `create_user_profile` (in the file `helper_online_ssl.py`) 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` (you have to adjust the subjects name in the file).

3.1. Complete `online_ssl_update_centroids` using the pseudocode **??**.

3.2. Complete `online_ssl_compute_solution` following the pseudocode **??**

3.3. Read the function `preprocess_face` (in `helper_online_ssl.py`) and run `online_face_recognition`. Include some of the resulting frames (not too similar) in the report showing faces correctly being labeled as opposite, and comment on the choices you made during the implementation.

3.4. What happens if an unknown person's face is captured by the camera? Modify your code to be able to disregard faces it cannot recognize, and include some of the resulting frames (not too similar) in the report showing unknown faces correctly being labeled as unknown.