Manifold Learning in the context of Partially Labelled Classification

Telmo Cunha[1]

[1] *73487 - Course: Mathematics for Machine Learning (21/22) - Instituto Superior Técnico*

[*] *telmocunha@gmail.com*

Abstract: Under the assumption that the considered data approximately lies within a submanifold of a significantly larger space, we tackle classification problems where a large portion of said data is unlabeled. In particular, we employ an algorithm based on manifold learning to build a classifier supported on a discretized submanifold inferred from the unlabeled data. This is done by approximating a submanifold via a graph formulation and producing a classifier function based on the eigenfunctions of the Laplace Beltrami operator in the discrete setting. A computational implementation is realized in Python and the algorithm is tested using the MNIST (Modified National Institute of Standards and Technology) dataset of handwritten digits. The goal of this work is to replicate the results on the MNIST dataset given in [2] and to understand the theory behind the method by probing mainly [2] and [3].

1. Introduction

In classification settings it is often very costly to obtain a large number of labeled examples in order to employ a form of supervised learning. This is typically the case, for example, in biological and medical research or medical applications where it is required that a specialist labels the data, which turns out be expensive and extremely time consuming. Thus, there is a huge advantage in being able to construct solid classifiers from datasets having a large proportion of unlabeled data. The method we apply, based on [2], explores the assumption that our data satisfies the manifold hypothesis, i.e. that the data approximately lies within a submanifold of an higher dimensional space. Using unlabeled data one can infer a discretized version (graph) of this submanifold and build a classifier supported on the nodes of the graph which is then optimized with recourse to the labeled data. Specifically, we consider an adjacency graph based on kNN (k-Nearest Neighbours) using the entire dataset, to obtain a discretized approximation of an underlying submanifold. Then, we approximate a classifying function supported on the nodes of the graph, with recourse to the labeled examples, which is given within a basis of eigenfunctions of the (graph) Laplace operator.

2. Motivation via a (synthetic) example

Using the synthetic example, provided in [2], we motivate the consideration of the manifold structure for partially supervised learning. Suppose we have an unknown distribution as the one of panel 1 from which we know only the labeled datapoints seen in panel 2, fig. (1).

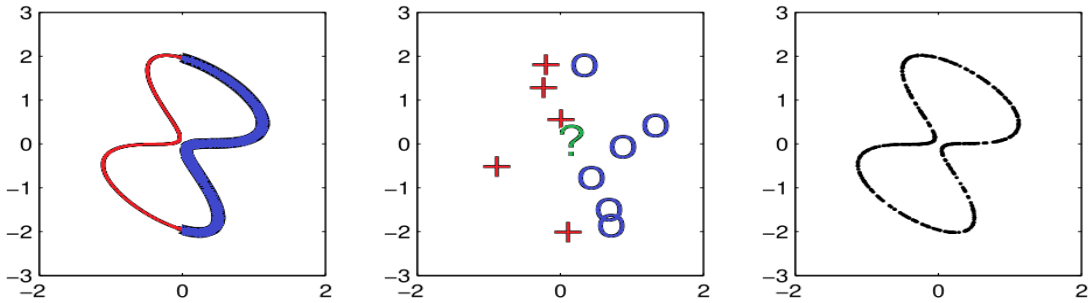

Fig. 1. **Panel 1:** Underlying (unknown) distribution; **Panel 2:** Labeled datapoints where "?" is a point we want to classify. **Panel 3:** (500) Unlabeled datapoints. Source: [2].

In the interest of classifying the unlabeled point, marked "?", under a supervised-learning setting, consider kNN using an Euclidean distance. For $k = 1$ we would classify this points as a "+". For $k = 2$ we would have to pick at random since the two nearest labeled datapoints are "+" and "∘". For $k = 3$ we would label this points as "∘" and for $k = 4$ we would again be choosing at random. It is clear that the Euclidean distance does not seem very informative in order to classify this particular point. Furthermore, if we attempt to find a separating line as a method for classification it is clear we could potentially miss-label a significant portion of the data (by slightly varying the example) and still, the point marked as "?" would always lie very close to this separating line reducing the confidence level over the predicted label.

However, if we introduce unlabeled data (panel 3 of fig. (1)) we see an underlying structure on the feature space, i.e. the data seems to clearly lie on a submanifold of the $\mathbb{R}^2$ plane. From this apparent submanifold, if we want again to classify the unlabeled data point "?", it is now clear that the geodesic distance within the submanifold seems much more relevant for classification than the Euclidean distance, since the points "can not cross" to the other side on the center patch, they are restricted to move along the submanifold.

From this setting the goal is to find a transformation of the data such that the coordinates vary as slowly as possible when traversing the curve. This desired embedding can be very closely approximated by using Laplacian Eigenmaps, we briefly show this in section 3.3 following [3]. For this particular example, the ideal representation can be seen in fig. (2) where the "?" datapoint now lies clearly in between the "+" labeled datapoints on panel 2.

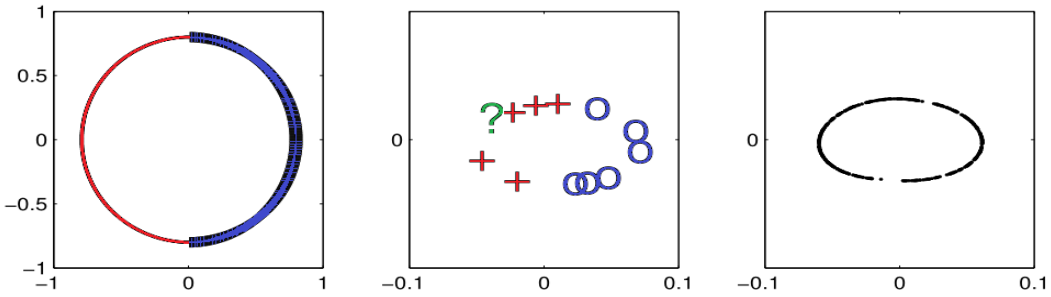

Fig. 2. **Panel 1:** Transformed (unknown) distribution; **Panel 2:** Labeled datapoints and "?" under the transformation. **Panel 3:** Transformed unlabeled datapoints. Source: [2].

The overall procedure is the following, first we recover the submanifold by making use of the entire dataset, labeled and unlabeled datapoints. Since a smooth manifold is a "continuous object" we need to formulate an approximation based on a finite number of datapoints. For this we shall consider a weighted graph formulation. Next, we want to build a classifier on this submanifold by considering the labeled datapoints. A straightforward approach would be to consider the geodesic distances between points, i.e. the closest distances restricted to the submanifold (following edges in the adjacency graph). According to [2] however, this method can potentially be unstable which is why the approach considered by the authors is to instead approximate a classification function by considering functions supported on the embedding provided by the eigenfunctions of the graph Laplacian, which is an operator that can be estimated by using the unlabeled datapoints.

3. Theoretical Foundation

The ideas in the discrete setting are built by analogy with the continuous case. We shall only provide an overview of the latter as motivation for the discrete formulation. For the theory in the continuous case see for example [5].

3.1. Continuous case

Consider a closed Riemannian manifold $\mathcal{M} = (M, g)$ isometrically embedded in $\mathbb{R}^n$, for some n. The Hilbert space of functions on the manifold $L^2(M)$, i.e. the square integrable functions defined on M, has an orthonormal basis provided by the eigenfunctions of a self-adjoint positive semidefinite operator called the Laplace-Beltrami operator Δ, which is a generalization of the usual Laplace operator for Riemannian manifolds. This result is known as the Lichnerowicz–Obata theorem. Thus, for any $f \in L^2(M)$ we can write $f(x) = \sum_{i=0}^{\infty} \alpha_i e_i(x)$ where the e_i are the eigenfunctions of Δ, i.e. $\Delta e_i = \lambda_i e_i$. The discreteness of the spectrum for Δ is a consequence of the

compactness of M and is also shown in the Lichnerowicz–Obata theorem[1].

Besides giving a basis for $L^2(M)$, the Laplace-Beltrami operator also provides a measure of smoothness for functions defined on M from the following definition, where smoothness here should be interpreted in the sense of a function varying slowly.

Definition 1. ***Smoothness Operator $\mathcal{S}$***
Given $f : M \to \mathbb{R}$, $f \in C^2(M)$, the smoothness operator $\mathcal{S}$ is defined as:

$$\mathcal{S}(f) := \int_M \|\nabla f\|^2 d\mu = \int_M f \Delta f \, d\mu = \langle \Delta f, f \rangle_{L^2(M)} \tag{1}$$

The justification for the equalities above follows from Stoke's theorem, see section 3.2 of [3]. Just as a remark, since the result on the eigenfunctions of the Laplace-Beltrami operator comes from the Lichnerowicz–Obata theorem, which assumes a Riemanninan manifold without boundary[2], then the above integration by parts showing no boundary terms makes sense.

Given an eigenfunction e_i of Δ we have $S(e_i) = \langle \Delta e_i, e_i \rangle_{L^2(M)} = \lambda_i \langle e_i, e_i \rangle_{L^2(M)} = \lambda_i$, using the fact that these eigenfunctions have unit norm. Thus, we see that the eigenvalues hold the information regarding the smoothness properties of the eigenfunctions.

Given an arbitrary function $f \in L^2(M)$, its representation in the basis provided by Δ is given by $f = \sum_i \alpha_i e_i$ therefore, using again the fact that the basis is orthonormal, $\mathcal{S}(f)$ is given by:

$$\mathcal{S}(f) = \langle \Delta f, f \rangle_{L^2(M)} = \left\langle \sum_i \alpha_i \Delta e_i, \sum_j \alpha_j e_j \right\rangle = \left\langle \sum_i \alpha_i \lambda_i e_i, \sum_j \alpha_j e_j \right\rangle = \sum_i \lambda_i \alpha_i^2 \tag{2}$$

From an intuitive conception of smoothness it is no surprise that one would consider constant functions to be the "most smooth". By the definition it is clear that any constant function c defined on M satisfies $\mathcal{S}(c) = 0$ and thus corresponds to the zero eigenvalue by comparison with equation (2), since at least one $\alpha_i \neq 0$. In fact, the eigenfunctions of Δ with the lowest eigenvalues correspond to the "smoothest" functions. Therefore, approximating a function by retaining only the first, say p, eigenfunctions is a way to control the smoothness of the approximation. To obtain the best approximation to a general function f on M we consider the optimization problem:

$$\min_{a_1,\ldots,a_p} \int_M (f(x) - \sum_{i=1}^{p} a_i e_i(x))^2 d\mu \tag{3}$$

Which is essentially least squares in the continuous case and approximates f by projecting into the p eigenfunctions of Δ with smallest eigenvalues, where $a_i = \langle e_i, f \rangle$, in order to force the approximating function $\hat{f} = \sum_i a_i e_i$ to have certain smoothness properties.

Since we have a finite number of datapoints we need to transpose this theory into a discrete scenario. We now present a framework for obtaining operators defined on graphs as discrete versions of continuous differential operators, based on [1]. For a more general framework, and mostly for future reference, see also [4].

3.2. Discrete case

We begin by introducing the notion of a weighted graph which will be a way to have a discretized representation of a manifold, for our purposes.

Definition 2. ***Finite weighted graph***
We denote the triple $G = (V, E, w)$ as a finite directed graph G where $V = \{v_1, v_2, ..., v_n\}$, $n \in \mathbb{N}$, is a finite set of vertices, $E \subset V \times V$ is a finite set of edges connecting a subset of the vertices in V and where $w : E \to \mathbb{R}^+$ is a non-negative symmetric weight function defined on the edges of E.

We will consider only undirected graphs since edge directions will not be important when defining the adjacency matrix in order to obtain the graph Laplacian. Thus we have $(u, v) \in E \Leftrightarrow (v, u) \in E$, so an undirected graph is a particular case of a directed graph. The weight function encapsulates the relative importance given to each edge,

[1] Source: Wikipedia's page on the Laplace-Beltrami operator.
[2] Source: Again, Wikipedia's page on the Laplace–Beltrami operator.

we shall consider the following weight function (other weight functions could be considered, see the discussion in [3]):

$$w((u,v)) := \begin{cases} 1, \text{if } (u,v) \in E \\ 0, \text{otherwise} \end{cases} \tag{4}$$

Definition 3. ***Functions defined on vertices***
Let $\mathcal{H}(V)$ denote the Hilbert space of real-valued functions defined on the vertices of a graph $G = (V, E, w)$, i.e. $f \in \mathcal{H}(V)$, $f : V \to \mathbb{R}$.

A function is fully specified by giving its values on the vertices which can be represented as a vector $f = [f(v_1) \dots f(v_n)]^T \in \mathbb{R}^n$. Thus, we consider the inner product on $\mathcal{H}(V)$ to be given by $\langle f, g \rangle = \sum_{v \in V} f(v)g(v)$, where $f, g \in \mathcal{H}(\mathcal{V})$.

Definition 4. ***Functions defined on edges***
Let $\mathcal{H}(E)$ be the Hilbert space of real-valued functions defined on the edges of a graph $G = (V, E, w)$, i.e. $F \in \mathcal{H}(E)$, $F : E \to \mathbb{R}$. The inner product is given by $\langle F, H \rangle = \sum_{(u,v) \in E} F((u,v))H((u,v))$ for $F, H \in \mathcal{H}(E)$.

Definition 5. ***Difference Operator*** *d*
Consider the graph $G = (V, E, w)$ and $f \in \mathcal{H}(V)$, we define the difference operator $d : \mathcal{H}(V) \to \mathcal{H}(E)$ on the edge $(u, v) \in E$ by:

$$(df)(u,v) = \sqrt{w(u,v)}(f(v) - f(u)) \tag{5}$$

The difference operator d encapsulates the notion of a directional derivative along the edges. Given a vertex $u \in V$, edge $e = (u, v) \in E$ and a function $f \in \mathcal{H}(V)$ we define the derivative of f at vertex u along edge e as:

$$\left.\frac{\partial f}{\partial e}\right|_u = \partial_v f(u) = (df)(u,v) \tag{6}$$

Definition 6. ***Adjoint Difference Operator*** *d^**
Given $f \in \mathcal{H}(V)$ and $H \in \mathcal{H}(E)$, the adjoint operator $d^* : \mathcal{H}(E) \to \mathcal{H}(V)$ is a linear operator defined by:

$$\langle df, H \rangle_{\mathcal{H}(E)} = \langle f, d^*H \rangle_{\mathcal{H}(V)} \tag{7}$$

Proposition 1. *Let $H \in \mathcal{H}(E)$ and $u \in V$, we can express $(d^*H)(u)$ as,*

$$(d^*H)(u) = \sum_{u \sim v} \sqrt{w(u,v)}(H((v,u)) - H((u,v))) \tag{8}$$

Proof. From the inner-product on $\mathcal{H}(E)$ and the definition of the difference operator (5) we have:

$$\begin{aligned}
\langle df, H \rangle_{\mathcal{H}(E)} &= \sum_{(u,v) \in E} (df)(u,v)H((u,v)) = \sum_{(u,v) \in E} \sqrt{w(u,v)}(f(v) - f(u))H((u,v)) = \\
&= \sum_{(u,v) \in E} \sqrt{w(u,v)}f(v)H((u,v)) \; - \sum_{(u,v) \in E} \sqrt{w(u,v)}f(u)H((u,v)) = \\
&= \sum_{u \in V} \sum_{v \sim u} \sqrt{w(u,v)}f(u)H((v,u)) \; - \sum_{u \in V} \sum_{v \sim u} \sqrt{w(u,v)}f(u)H((u,v)) = \\
&= \sum_{u \in V} f(u)\sqrt{w(u,v)}\,(H((v,u)) - H((u,v)))
\end{aligned}$$

where on the previous to last step we use the fact that $(u, v) \in E \Leftrightarrow (v, u) \in E$, for undirected graphs, and we turn a sum on edges into a sum on each vertex u of the graph and all corresponding connected vertices $v \sim u$. From definition (6) and the inner product on $\mathcal{H}(V)$ we obtain:

$$\begin{aligned}
\langle df, H \rangle_{\mathcal{H}(E)} &= \sum_{u \in V} f(u)\sqrt{w(u,v)}\,(H((v,u)) - H((u,v))) = \\
= \langle f, d^*H \rangle_{\mathcal{H}(V)} &= \sum_{u \in V} f(u)d^*H(u)
\end{aligned}$$

The conclusion follows. □

Definition 7. ***Divergence Operator***
The divergence operator is defined to be $-d^*$, and measures the net outflow of a function in $\mathcal{H}(E)$ at each vertex of the graph.

Definition 8. ***Gradient Operator***
The weighted gradient operator of a function $f \in \mathcal{H}(V)$ at vertex $u \in V$ is defined by:

$$\nabla_w f(u) = [\partial_{v_i} f(u) : (u, v_i) \in E]^T \tag{9}$$

In analogy to the continuous case, by computing the norm of the gradient vector we obtain a measure of the local variation of the function at a certain vertex, in particular this gives us a measure of the smoothness of the function. We have:

$$\|\nabla_w f(u)\|_2 = \left(\sum_{v \sim u} (\partial_v f(u))^2 \right)^{1/2} \tag{10}$$

We now introduce the Laplace operator. Here we use a slightly different notation compared to the one in [1] (and in particular we consider only the case $p = 2$), in this way we think the relation with the usual Laplace operator is made clearer.

Definition 9. ***Laplace Operator***
Given $f \in \mathcal{H}(V)$ we define the weighted Laplace Operator $\Delta_w : \mathcal{H}(V) \to \mathcal{H}(V)$ by:

$$\Delta_w f(u) = -\frac{1}{2} d^*(\nabla_w f(u)) \tag{11}$$

This is analogous to the continuous case where the Laplacian is the divergence of the gradient, i.e. given $f \in C^2(\mathbb{R}^n)$, we have $\Delta f = \nabla \cdot \nabla f$.

Proposition 2. *Given $f \in \mathcal{H}(V)$ the following holds:*

$$\Delta_w f(u) = \sum_{v \sim u} w(u, v)(f(u) - f(v)) \tag{12}$$

Proof. Using definitions (9), (8) and (5) we have:

$$\Delta_w f(u) = -\frac{1}{2} d^*(\nabla_w f(u)) = -\frac{1}{2} d^*((\partial_{v_1} f(u), ..., \partial_{v_k} f(u))) = -\frac{1}{2} d^*(df((u, v_1)), ..., df((u, v_k))) \tag{13}$$

$$= -\frac{1}{2} d^*(\sqrt{w(u, v_1)}(f(v_1) - f(u)), ..., \sqrt{w(u, v_k)}(f(v_k) - f(u))) \tag{14}$$

This last term can be seen as $-\frac{1}{2} d^* H(u)$, where $H \in \mathcal{H}(E)$, thus using proposition (1), we have:

$$\begin{aligned}
&-\frac{1}{2} d^* \left(\sqrt{w(u, v_1)}(f(v_1) - f(u)), ..., \sqrt{w(u, v_k)}(f(v_k) - f(u)) \right) = \\
&= -\frac{1}{2} \left(\sqrt{w(u, v_1)}^2 (f(v_1) - f(u) - (f(u) - f(v_1))) + ... + \sqrt{w(u, v_k)}^2 (f(v_k) - f(u) - (f(u) - f(v_k))) \right) = \\
&= \sum_{v \sim u} w(u, v) \, (f(u) - f(v))
\end{aligned}$$

□

This Laplacian operator is known as the Graph Laplacian. We now show how to define these operators as matrices, acting on functions defined on the vertices or edges of a graph, by looking at a particular example.

Definition 10. ***Incidence matrix***
Consider the graph $G = (V, E, w)$, let $v_i \in V$ and $e_j \in E$ and let K be a matrix such that the rows correspond to edges and columns to vertices, we have:

$$K_{ij} := \begin{cases} 1, & \text{if edge } i \text{ enters vertex } j \\ -1, & \text{if edge } i \text{ leaves vertex } j \\ 0, & \text{otherwise} \end{cases} \tag{15}$$

Given a function $f \in \mathcal{H}(V)$ we have $\nabla f = Kf$. Consider the following graph G, fig. (3), where $V = \{A, B, C, D, E\}$ and $E = \{e_1 = (A, B), e_2 = (B, C), e_3 = (D, B), e_4 = (C, D), e_5 = (B, E), e_6 = (E, D)\}$.

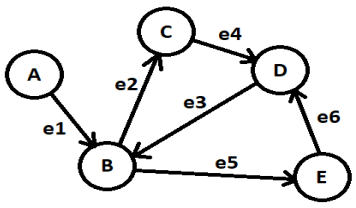

Fig. 3. Directed graph example

The incidence matrix for this example is given by:

$$K = \begin{bmatrix} -1 & 1 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & -1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & -1 \end{bmatrix} \tag{16}$$

Thus, given $f \in \mathcal{H}(V) = (f(A), f(B), f(C), f(D), f(E))$ we can compute:

$$\nabla f = \begin{bmatrix} -1 & 1 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & -1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} f(A) \\ f(B) \\ f(C) \\ f(D) \\ f(E) \end{bmatrix} = \begin{bmatrix} f(B) - f(A) \\ f(C) - f(B) \\ f(B) - f(D) \\ f(D) - f(C) \\ f(E) - f(B) \\ f(D) - f(E) \end{bmatrix} \tag{17}$$

Consider now $\nabla f = g \in \mathcal{H}(E)$ where $g = (g(e_1), g(e_2), g(e_3), g(e_4), g(e_5), g(e_6))$ the divergence of g on each node is given by proposition (1) which can be seen to be equivalent to $K^T g$:

$$-d^*(g) = \begin{bmatrix} -g(e_1) \\ g(e_1) - g(e_2) + g(e_3) - g(e_5) \\ g(e_2) - g(e_4) \\ -g(e_3) + g(e_4) + g(e_6) \\ g(e_5) - g(e_6) \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 & 0 & 0 & 0 \\ 1 & -1 & 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} g(e_1) \\ g(e_2) \\ g(e_3) \\ g(e_4) \\ g(e_5) \\ g(e_6) \end{bmatrix} = K^T g \tag{18}$$

Thus the Laplacian, being the composition of the operators K and K^T, is given by $\Delta f = K^T K f$. In fact, it can also be computed as $L = K^T K = D - W$ considering the weight matrix W is a $V \times V$ matrix called the adjacency matrix and is defined by:

$$W_{ij} = \begin{cases} 1, & \text{if } (i, j) \in E \\ 0, & \text{otherwise} \end{cases} \tag{19}$$

Note that W is symmetric since we are considering w to be a symmetric function. The matrix D is diagonal and is defined by:

$$D_{ij} = \begin{cases} deg(v_i), & \text{if } i = j \\ 0, & \text{otherwise} \end{cases} \tag{20}$$

We have $L_{ij} = K^T_{(i,:)} K_{(:,j)} = \left\langle K_{(:,i)}, K_{(:,j)} \right\rangle$ (the notation, : , represents all rows or columns) and the diagonal entries are just $\left\langle K_{(:,i)}, K_{(:,i)} \right\rangle = deg(v_i)$, where v_i represents node i, we are just counting the number of edges since we always have terms of the form $(-1)(-1)$ and $(1)(1)$ besides the zero case. The off-diagonal terms just measure if there is an edge between node i and node j, if there is an edge in common we have $(-1)(1) = -1$ and otherwise we get 0, this is precisely $-W$ and thus we have $L = D - W$.

The operator L can be seen to be a self-adjoint positive semidefinite operator and, by the finite dimensional spectral theorem, the eigenfunctions of L provide a basis for functions defined on the vertices of the graph G. As shown in equation (10), and by analogy again with the continuous case, we define smoothness for a function on the vertices of a graph.

Definition 11. ***Graph smoothness functional***
Given a graph G and a function f defined on V we define the smoothness of f by:

$$\mathcal{S}_G(f) := \sum_{i\sim j}(df(i,j))^2 = \sum_{i\sim j} w_{ij}(f(j)-f(i))^2 \tag{21}$$

Proposition 3. *We have $S_G(f) = \langle f, Lf\rangle$*

Proof.

$$\begin{aligned}\langle f, Lf\rangle &= \left\langle \sum_j f(j)\hat{j}, \sum_j \sum_{i\sim j} w_{ij}(f(j)-f(i))\hat{j}\right\rangle = \sum_j \sum_{i\sim j} w_{ij}(f(j)-f(i))f(j) = \\ &= \sum_{i\sim j} w_{ij}(f(j)-f(i))f(j) - w_{ij}(f(j)-f(i))f(i) = \sum_{i\sim j} w_{ij}(f(j)-f(i))(f(j)-f(i)) = \\ &= \sum_{i\sim j} w_{ij}(f(j)-f(i))^2\end{aligned}$$

□

Approximating $f : V \to \mathbb{R}$ by the eigenfunctions of the Graph Laplacian L, i.e. $f = \sum_i \alpha_i e_i$ where $Le_i = \lambda_i e_i$, we have:

$$\mathcal{S}_G(f) = \langle Lf, f\rangle_{f\in\mathcal{H}(V)} = \left\langle \sum_i \alpha_i L(e_i), \sum_i \alpha e_i\right\rangle = \sum_i \lambda_i \alpha_i^2 \tag{22}$$

which is again totally analogous to the continuous case, the eigenfunctions with lower eigenvalues represent again "smoother" functions with the zero eigenvalue corresponding to a constant function on the vertices.

Again, we can approximate a function by retaining only the first p eigenvectors by solving the following least-squares problem:

$$\min_{\mathbf{a^T}=(a_1,\dots,a_p)} \sum_{i=1}^{n}\left(f(x_i) - \sum_{j=1}^{p} a_j e_j(x_i)\right)^2 \implies \mathbf{a} = (EE^T)^{-1}E\mathbf{y}$$

where E is a $p\times n$ matrix with $E_{ij} = e_i(x_j)$ and $y^T = (f(x_1),\dots,f(x_n))$.

3.3. Justification for the Embedding

In this section, following [3], we very briefly look as to why the eigenvectors of the graph Laplacian L provide the desired embedding.

Suppose we have a given dataset $\{x_1,\dots,x_n\}$ and we have the corresponding (assumed to be connected) graph $G = (V,E)$. Consider that we want to find a lower dimensional representation of these points $y = (y_1,\dots,y_n)^T$ which preserve the locality of the points, i.e. if $x_i \sim x_j$ then x_i and x_j are close in a certain sense and we want the y_i and y_j to be close in the same sense. We look to minimize the objective function $\sum_{ij}(y_i-y_j)^2 W_{ij}$ under some appropriate constraints. We can write for any $\mathbf{y}$:

$$\frac{1}{2}\sum_{ij}(y_i-y_j)^2 W_{ij} = \mathbf{y^T L y} \tag{23}$$

where $L = D - W$ since:

$$\begin{aligned}\sum_{ij}(y_i-y_j)^2 W_{ij} &= \sum_{ij}(y_i^2+y_j^2-2y_iy_j)W_{ij} = \sum_i y_i^2\sum_j W_{ij} + \sum_j y_j^2\sum_i W_{ij} - 2\sum_{i,j} y_iy_jW_{ij} = \\ &= \sum_i y_i^2 D_{ii} + \sum_j y_j^2 D_{jj} - 2\sum_{i,j} y_iy_jW_{ij} = 2\left(\sum_i y_i^2 D_{ii} - \sum_{i,j} y_iy_jW_{ij}\right) = 2\mathbf{y^T L y}\end{aligned}$$

This shows also that L is positive semidefinite. The optimization problem can be rewritten as:

$$\begin{aligned} \underset{\mathbf{y}}{\text{argmin}} \quad & \mathbf{y^T L y} \\ \text{s.t.} \quad & \mathbf{y^T D y = 1} \end{aligned} \tag{24}$$

with the constraint removing a scaling factor on the embedding and giving higher relevance to nodes with more edges, since D_{ii} is the degree of node i. The vector $\mathbf{y}$ that minimizes the objective function is given by the generalized eigenvalue equation $L\mathbf{y} = \lambda \mathbf{D y}$.

For the general problem where the data lives in $\mathbb{R}^m$ we consider $\mathcal{Y} = [\mathbf{y_1}\, \mathbf{y_2} \ldots \mathbf{y_m}]$, a $k \times m$ matrix where each row represents the embedding of a datapoint $\mathbf{x_i}$ from $i = 1, ..., k$. Again, following [3], the optimization problem is now:

$$\begin{aligned} \underset{\mathbf{y}}{\text{argmin}} \quad & \sum_{i.j} \|y^{(i)} - y^{(j)}\|^2 W_{ij} = tr(\mathcal{Y}^T L \mathcal{Y}) \\ \text{s.t.} \quad & \mathbf{y^T D y = 1} \end{aligned} \tag{25}$$

where the solution is again given by $L\mathbf{y} = \lambda \mathbf{D y}$. This result together with the smoothness properties of the eigenfunctions of the Laplace operator is what justifies their use as providing the desirable embedding. This justification can be seen in much more detail in [3].

4. Algorithm

Suppose we have $x_1, ..., x_n$ datapoints where each $x_i \in \mathbb{R}^d$, for $i = 1, 2, ..., n$. Out of these n datapoints assume that only the first $s < n$ are labeled with the remaining ones unlabeled. The algorithm can be split into four steps:

- **Adjacency matrix:** To build the adjacency matrix we consider k-nearest neighbours with an Euclidean distance. Picking a point x_i, we set $W_{ij} = 1$ if x_j is within k-nearest neighbours of x_i (or x_i is within k-nearest neighbours of x_j), for $i \neq j$ and we set $W_{ij} = 0$ otherwise.

- **Eigenfunctions of the Graph Laplace Operator:** From the previous step we can compute the Graph Laplace operator $L = D - W$, where D is a diagonal matrix satisfying $D_{ii} = \sum_j W_{ij}$, and solve the eigenvalue problem $Le = \lambda e$, where each $e_i \in \mathbb{R}^n$ can be thought of as a function on the vertices of the graph and gives the ith embedding coordinate for all datapoints. We keep only the first p eigenfunctions (in practice we consider p to be about 20% of the number of labeled points s) to obtain the desired embedding.

- **Classifier:** To build the classifier function we solve the following optimization problem:

$$\min_{a_1,\ldots,a_p} \quad \sum_{i=1}^{s} \left(c_i - \sum_{j=1}^{p} a_j e_j(i) \right)^2 \tag{26}$$

  Thus we are essentially finding a function defined on the vertices of the graph that minimizes the error when comparing on the vertices for which we know the labels. The solution of the least squares problem can be obtained by the normal equations and is given by:

$$\mathbf{a} = (E^T E)^{-1} E^T \mathbf{c} \tag{27}$$

  where $\mathbf{c} = (c_1, ..., c_s)^T$ are the labels on the s datapoints and E is an $s \times p$ matrix with $E_{ij} = \mathbf{e_j}(i)$. In practice we will have multiple classes and we consider a one-against-all classifier which means training the classifier on each label. Thus, the vector $\mathbf{c} = (c_1, ..., c_s)^T$ will have $c_i = 1$ whenever point i has the label under consideration and $c_i = -1$ otherwise.

- **Classification:** For any unlabeled point x_i, i.e. with $s < i \leq n$, we classify it according to:

$$c_i = \begin{cases} 1, & \text{if } \sum_{j=1}^{p} a_j \mathbf{e}_j(i) \geq 0 \\ -1, & \text{if } \sum_{j=1}^{p} a_j \mathbf{e}_j(i) < 0 \end{cases} \tag{28}$$

Again, in the situation where we have several classes, we construct a classifier on each label attributing to each point i a value c_i given by $\sum_{j=1}^{p} a_j \mathbf{e}_j(i)$. These values correspond to confidence values for each label, since we are considering positive results as 1 and negative results as -1, the label with highest c_i value will be the chosen label for point x_i.

5. Python Implementation with the MNIST dataset

We now discuss some details of the implementation in Python for the MNIST dataset. The MNIST dataset consists of gray-scale images of handwritten digits in the set $\{0, 1, ..., 9\}$ with $28 \times 28 = 784$ pixels, where each pixel has a value in $\{0, 1, ..., 255\}$, together with a vector of labels corresponding to each digit.

We use the *mnist* library to import the data, each datapoint x_i, $i = 1, 2, ..., 60000$, is a vector of size $x_i = 784$ for which we have a label $c(x_i) \in \{0, 1, ..., 9\}$. As done in [2], a significant amount of pixels in the images will not be relevant for classification, think for example on the pixels on the boundary. Therefore, we first apply PCA (Principal Component Analysis) to the data and reduce our feature space to 100 principal components, so now each $x_i \in \mathbb{R}^{100}$. Since all pixel values vary in the same range we do not standardize the data when performing PCA.

The computation of the adjacency graph, by calculating nearest neighbours using *for* loops, revealed itself to be extremely inefficient. Therefore, we use the *kneighbors_graph* library from *scikit-learn* to compute the k nearest neighbours matrix A with $k = 8$ (as used in [2]). The A matrix will be a *compressed sparse row (csr)* matrix which is not yet the adjacency matrix W. If node i has k neighbours, one of each is node j, but node j has k neighbours which does not include node i then $A_{ij} = 1$ and $A_{ji} = 0$ where we would want both to be 1. Therefore we need to symmetrize the matrix. For this we compute $W = A + A^T$ and then do $W[W >= 1] = 1$, which converts the entries that become 2 to 1, to finally obtain the adjacency matrix W. Computing D from W, by summing W along the rows, we obtain the graph Laplacian by $L = D - W$, as shown before. This is implemented on the *graph_laplacian()* function.

At this point we choose a value of s which corresponds to the number of labeled examples that we want to consider. This just means that we copy a subset of the the entire dataset with size s and keep the labels to later build the classifying function. This is done using the *sample_label_points()* function.

Considering then $p = \lfloor 0.2s \rfloor$, where $\lfloor x \rfloor$ mean the closest integer smaller to x, we compute the p eigenfunctions of L corresponding to the eigenvalues of smallest magnitude, e_j, $j = 1, ..., p$, where $e_j \in \mathbb{R}^{60000}$. Since L is a symmetric semi-definite positive matrix all eigenvalues will be real and non-negative. Here we use the libraries *scipy.sparse* and *scipy.sparse.linalg* to handle matrix methods for large sparse matrices, since L is a 60000×60000 matrix.

We now have to compute a classifier function for each label, i.e. for each digit in $\{0, 1, ..., 9\}$ we need to compute $\mathbf{a} = (E^T E)^{-1} E^T \mathbf{c}$ where E is the $s \times p$ matrix of eigenfunctions of L valued on each labeled node and c is a function on the labeled nodes with value 1 on the digit under consideration and -1 on all other digits (one-against-all classification). The computation of E is done in the *E_matrix()* function and of c in the *c_vector()* function. The coefficients $\mathbf{a}$ are computed using the *classifier_coefs()* function.

We now build the function *confidence_measure()* which computes a matrix $M_{60000 \times 10}$ where M_{ij} has the value $c_i(d) = \sum_{j=1}^{p} a_j^d \mathbf{e}_j(i)$ which corresponds to the confidence level of labeling point i with digit d.

To obtain the labels for each unlabeled point i we just do $\text{label}(i) = \underset{d}{\text{argmax}}\, M_{(i,:)}$ which corresponds to the digit with highest confidence level. This is done using the *computing_labels()* function.

Finally, to compute the error rate we use the function *error_rate()* which just takes the ratio $\frac{FP+FN}{60000-s}$, i.e. the sum of all false positives and false negatives over the number of unlabeled datapoints.

Remark: This is the description for a single run of the method using the entire dataset, for the 60000 datapoints. Since the results are dependent on the sampling of a labeled dataset we should consider several trials and average the results over different choices of the labeled points. In our tests we considered 10 trials when $s \in \{20, 50, 100, 500, 1000\}$ however, for $s \in \{5000, 20000\}$ it takes a really long time to determine the p smallest eigenvalues and corresponding eigenfunctions, so for $s = 5000$ we do only 3 trials and for $s = 20000$ we do not provide results (running over night was not enough to compute a single run).

6. Results on MNIST and discussion

The results obtained can be seen in the following figure, fig. (4).

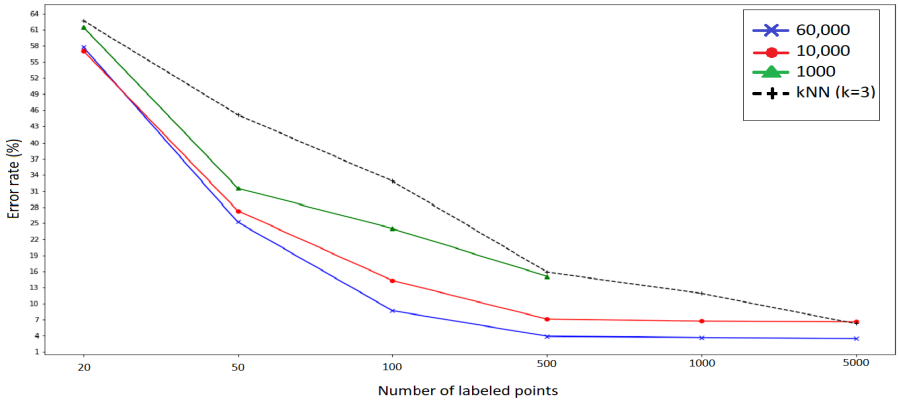

Fig. 4. Error rates (%) on the MNIST dataset

In order to compare our results with the ones obtained in [2], figure (4) shows the error rates obtained as a function of the size of the labeled dataset for $s \in \{20, 50, 100, 500, 1000, 5000\}$, for different sizes of the total points considered $\{60000, 10000, 1000\}$. We further compare our results with kNN, considering $k = 3$, using again different sizes of the labeled dataset. Again, we do not show results for $s = 20000$ (as done in [2]) since the time to compute the eigenvalues is extremely long. The actual error rates can be seen in the following table (1).

# of datapoints	$s = 20$	$s = 50$	$s = 100$	$s = 500$	$s = 1000$	$s = 5000$
60000	55.08%	24.76%	8.92%	4.07%	3.65%	3.48%
10000	57.1%	27.30%	14.32%	7.13%	6.74%	6.62%
1000	61.51%	31.51%	23.97%	15.09%	-	-
kNN (k=3)	62.69%	45.17%	32.82%	15.89%	11.90%	6.26%

Table 1. Error rates

For comparison sake we show in figure (5) the results from [2] on the MNIST dataset.

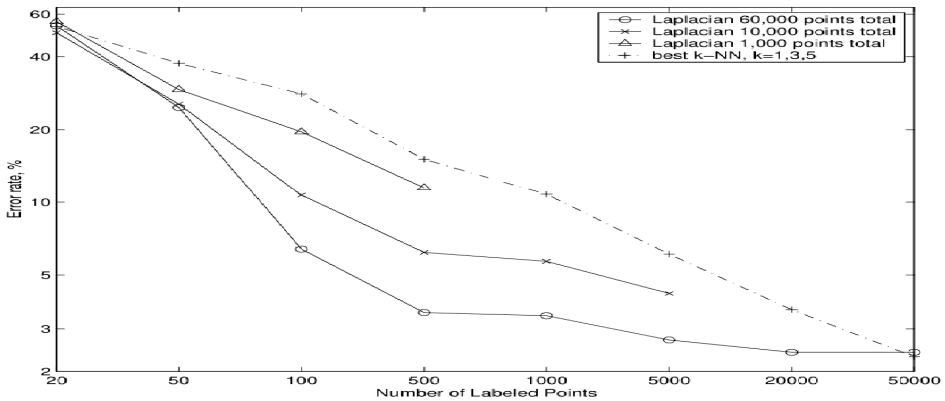

Fig. 5. Error rates (%) on the MNIST dataset from [2]

Overall the results follow a similar trend, having a larger number of unlabeled examples gives lower error rates since it better captures the underlying manifold structure. Similarly, having a higher number of labeled examples allows for a better approximation of the underlying distribution, which leads to a better classifier resulting in lower error rates as well. Note that the plot in figure (5) is not linear so slight differences in performance are clearer than those of figure (4). The error rates we obtained seem slightly worse, i.e. higher, comparing to [2] however we used a smaller number of trials, 10 compared to 20, and yet smaller trials for $s = 5000$ due to computation time. That might somewhat justify the discrepancy. Nonetheless, in all cases we see a significant improvement over kNN with $k = 3$. In fig. (5) we actually see a comparison with the best kNN for $k \in \{1, 3, 5\}$ and we further see a converging of performance when using a set with $s = 50000$ justified by the fact that kNN underlies the whole procedure when constructing the adjacency matrix. Further tests could be done, there was no clear indication as to why the number of eigenfunctions was taken to be 20% of the number of labeled examples, we assume this was tested and that it was the value leading to the best results. Also, other values of k could be tested when constructing the adjacency matrix, we assume again that this was tested and that $k = 8$, as used in [2], is what leads to the best results.

References

1. Sébastien Bougleux, Abderrahim Elmoataz, Olivier Lezoray. Nonlocal discrete regularization on weighted graphs: a framework for image and manifold processing. *IEEE Transactions on Image Processing*, 17(7):1047–1060, 2008.
2. Mikhail Belkin and Partha Niyogi. Using manifold structure for partially labelled classification. *Neural Information Processing Systems (NIPS)*, 15, 2002.
3. Mikhail Belkin and Partha Niyogi. Laplacian eigenmaps for dimensionality reduction and data representation. *Neural Computation*, 15(6):1373–1396, 2003.
4. Ronny Bergmann and Daniel Tenbrinck. A graph framework for manifold-valued data. *SIAM Journal on Imaging Sciences*, 11(1), 2017.
5. Steven Rosenberg. *The Laplacian on a Riemannian Manifold*. Cambridge University Press, 1997.