Due: April 25, 2019 **Homework 4** CPSC 476

Non-linear Dimensionality Reduction

PCA works very well for data sets where the points basically form an ellipsoidal cloud (e.g. points sampled from a multivariate normal distribution). What if our data has a more complicated structure? Take a look at the first principal component we get when our data is scattered (non-uniformly) along a spiral as in Figure 1. PCA in this case doesn't capture particularly useful information about our data. Projecting the points onto this principal component gives us a jumbled mess.

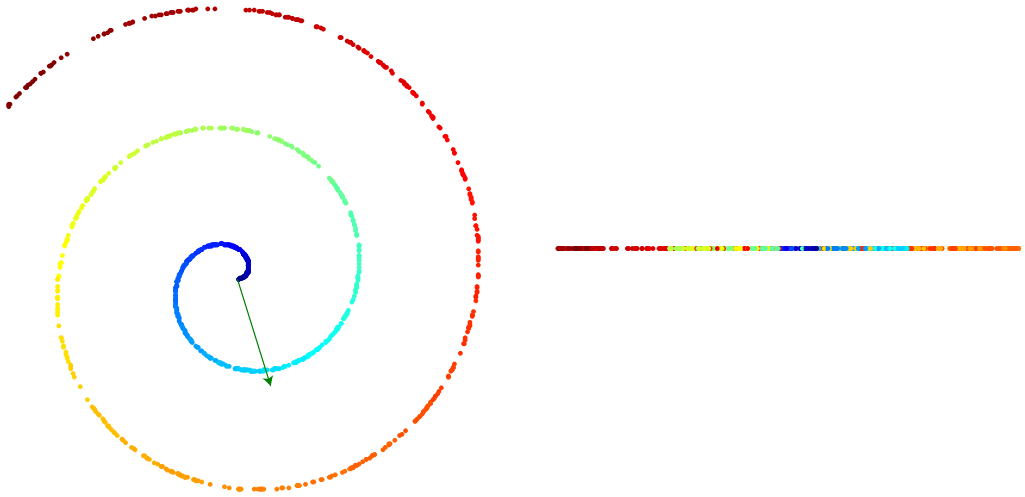

Figure 1: On the left are points distributed on a spiral, colored by their position along the spiral. The green vector is the first principal component of the data. On the right are the points projected onto the first principal component.

What might we want out of a dimensionality reduction algorithm instead? One answer, in the case of a simple curve like this, is that we'd like to get coordinates describing the position of our points along the curve. Such a coordinate function (from points to their position along the curve) is highly non-linear, which makes it clear why techniques like PCA can't help us—the coordinate functions given by PCA are derived from the projection of the observations onto (nice) basis vectors, and projection is a linear operation. More generally, if our data is distributed on a manifold, we might like to *learn the manifold* that our data lies on, and recover coordinate functions respecting the arrangement of our points on this manifold—points far apart from each other on the manifold should be mapped to points that are far apart (in a Euclidean sense) in the space given by our coordinate functions.

Notions of Distance

Note that Euclidean distance in the ambient space doesn't give us a reliable notion of distance along the manifold. The straight-line distance between the start and end points of the spiral in Figure 1 is *much* shorter than the distance between them when we're only allowed to travel along the spiral itself. We need a way of measuring some notion of distance *within* the manifold.

While Euclidean distance and manifold distance don't behave in compatible ways most of the time, they do qualitatively agree in the extremes. Observations that are very far apart in the ambient space will still be very far apart if distance is measured along the manifold. More importantly, assuming the manifold is covered reasonably densely by our observations, observations that are very close in a Euclidean sense are likely to be very close on the manifold. We can leverage this property to build a graph whose nodes correspond to our observations, and whose edges connect observations we think are nearby on the manifold (i.e. are very close in the ambient space). We can then measure a notion of manifold distance between observations by considering paths between them in this graph.

Random Walks

It's tempting to try to use shortest path ("geodesic") distance between nodes in our graph as our distance measure, but it turns out this isn't ideal—it's sensitive to noise and to topological "short circuits" in our graph. Consider the data points in Figure 2. We'd like the "distance" between points A and B to be larger than than the distance between points B and C—due to the bottleneck in the graph, it's "harder" to go from B to A than it is to go from B to C. Geodesic distance only cares about the length of the shortest path between two points, and so is unaffected by the bottleneck. We can construct a distance measure that *is* affected by features like the bottleneck by considering random walks within our graph—two nodes in the graph are "close" if a random walk starting from one node is likely to hit the other in a limited number of steps. For a fixed path length, there are far fewer paths connecting A and B than there are paths connecting B and C, so traveling through the bottleneck is relatively unlikely, and a distance based on random walks more closely reflects our intuition.

Let's formalize this picture. We'll construct a weighted graph with edge weights related to the "similarity" between nodes (observations in our data)—it should be easier to move between similar nodes, and this will be reflected in a high edge weight between highly similar nodes. Let's first define a matrix W containing pairwise similarity values:

$$W_{ij} = \exp\left(-\frac{\|\mathbf{x}_i - \mathbf{x}_j\|^2}{2\sigma^2}\right). \tag{1}$$

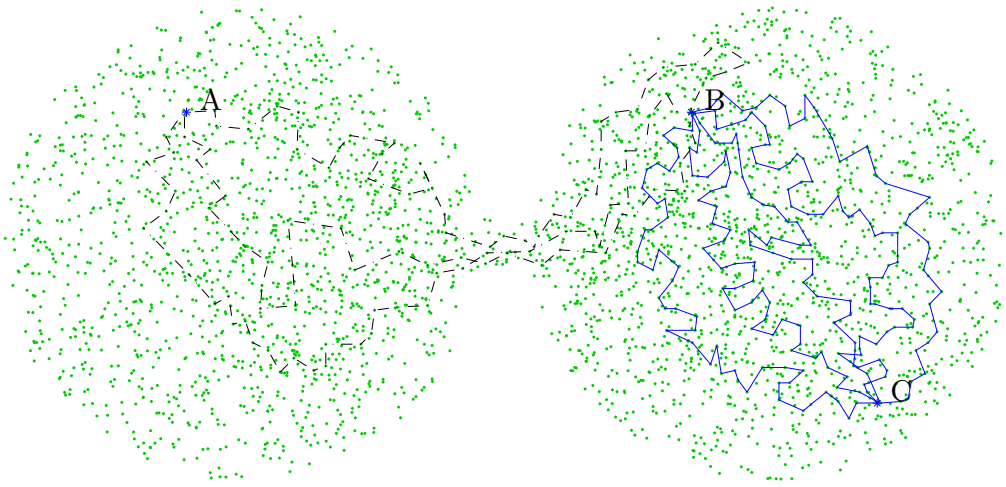

Figure 2: Data distributed among two disks with a "short circuit" connecting them. Using geodesic distance, points A and B aren't much farther apart than B and C. Using diffusion distance, because a random walk starting from B is much more likely to hit C than A in a limited number of steps, B is closer to C than to A.

W_{ij} contains a similarity value between observation i and observation j. Observations that are very close in a Euclidean sense have a similarity close to 1, and as the distance between observations gets larger the similarity goes to 0. We can control how quickly this falloff occurs by varying the bandwidth parameter σ. Setting σ appropriately is a bit of an art—we want to connect observations that are close on the manifold, and only observations that are close on the manifold (consider what might happen if the manifold curves back and almost intersects itself), so σ should be as small as possible while still strongly connecting nearby points.

Note that passing Euclidean distance through a Gaussian function is by no means the only way we could construct a similarity matrix—we could define other "similarity kernels" $k(\mathbf{x}_i, \mathbf{x}_j)$ that provide a similarity value between two observations, and in many applications this choice of similarity kernel is extremely important—but the Gaussian kernel we have used to define W is simple and works well in many cases.

Now that we have a similarity matrix, we'd like to perform a random walk among the nodes in our graph. At a given node i, we'll decide randomly what node j to visit next, but the probability of choosing j should be in proportion to the similarity between i and j relative to the other nodes connected to i. Let's define the probability of walking to j given that we're at i as follows:

$$p(j|i) = \frac{W_{ij}}{\sum_k W_{ik}}.$$

Note that $p(i|i) \neq 0$, so we have a non-zero likelihood of staying right where we are. This

makes the walk a *lazy* random walk. The denominator of $p(j|i)$ is simply the sum along row i of W. Let D be a matrix that has the row sums of W along its diagonal (i.e. $D_{ii} = \sum_k W_{ik}$), and define a matrix M given by

$$M = D^{-1}W.$$

Since multiplying a matrix by a diagonal matrix from the left simply scales rows, M has row sums normalized to 1 and we can now simply write $p(j|i) = M_{ij}$. Row i of M now contains the probability of stepping from node i to every other node in the graph.

Imagine we start our walk with a unit of "probability mass" located at node i. Let e_i be a probability vector containing all zeros except for a 1 in the i-th position. After one step of the walk, our probability mass has "diffused" through the graph, and we have a new probability vector, given by $e_i^\mathsf{T} M$, describing the new distribution of probability mass. This simply selects the i-th row of M (since we're multiplying from the left), which makes sense: Our probability mass has diffused to node j exactly according to the probability of stepping from i to j. We can take another step by multiplying this new probability vector against M from the left once more, giving $e_i^\mathsf{T} M^2$. This results in a new vector, with element j containing the probability of reaching node j from node i in exactly two steps of our walk. In fact, for t steps we can write

$$p(t, j|i) = e_i^\mathsf{T} M^t e_j, \tag{2}$$

to be read as the probability of reaching node j from node i in exactly t steps of the walk.

Eigendecomposition of the Walk Matrix

What happens to our probability mass as we let the number of steps in our walk t go to infinity? Even after many steps in our walk, $e_i^\mathsf{T} M^t$ is still a probability vector (to see this, just multiply it by the all ones vector from the right to compute the row sums, and remember that M has row sums equal to 1). We might expect that once the probability mass has diffused sufficiently throughout the graph, we reach a steady state in which the distribution of mass is unaffected by further steps of the walk.

Indeed, in the case of a lazy random walk matrix like M (strictly, for any walk matrix derived from a connected non-bipartite graph), this is exactly what happens. We can even find this "stationary" distribution: It's simply the diagonal of D (i.e. the row sums of W). Let's let $\mathbf{d}$ be a vector containing the row sums of W (diagonal of D). Then $\mathbf{d}^\mathsf{T} M = \mathbf{d}^\mathsf{T} D^{-1} W = \mathbb{1}^\mathsf{T} W = \mathbf{d}^\mathsf{T}$, where $\mathbb{1}$ is the all ones vector and $\mathbb{1}^\mathsf{T} W = \mathbf{d}^\mathsf{T}$ because W is symmetric (so the column sums are equal to the row sums).

This means $\mathbf{d}$ is a left eigenvector of M with eigenvalue 1. Its corresponding right eigenvector is simply the all ones vector. This is easy to see since the row sums of M are all 1, so $M\mathbb{1} = \mathbb{1}$. In addition to being the stationary distribution, $\mathbf{d}$ tells us something about the density of

our observations. Densely packed observations in a region get strongly connected by our similarity kernel, which is reflected in row sums $W\mathbb{1} = \mathbf{d}$.

Do the other eigenvectors of M tell us something useful about our data? What *are* the other eigenvectors of M? After all, M isn't symmetric, since we normalized its rows so that they summed to 1. This potentially makes finding and working with its eigendecomposition a bit of a pain. Luckily, we can do something a little sneaky to make M easier to analyze. Define

$$M_s = D^{1/2} M D^{-1/2} = D^{1/2} D^{-1} W D^{-1/2} = D^{-1/2} W D^{-1/2}.$$

Now, since $D^{-1/2}$ and W are both symmetric, so is M_s! Therefore M_s has an eigendecomposition with nice real eigenvectors and eigenvalues. Let's express it as

$$M_s = \Omega \Lambda \Omega^\mathsf{T}.$$

What about M? We can write M in terms of M_s as $M = D^{-1/2} M_s D^{1/2}$, which allows us to plug in the eigendecomposition we got for M_s:

$$M = D^{-1/2} \Omega \Lambda \Omega^\mathsf{T} D^{1/2}.$$

Let $\Psi = D^{-1/2}\Omega$, and $\Phi = D^{1/2}\Omega$. Then $M = \Psi \Lambda \Phi^\mathsf{T}$. Ψ and Φ have a very simple relationship:

$$\Psi = D^{-1}\Phi.$$

Furthermore, Ψ and Φ are mutually orthogonal, or *bi-orthonormal*: $\Phi^\mathsf{T}\Psi = \Omega^\mathsf{T} D^{1/2} D^{-1/2} \Omega = I$ (since Ω is an orthogonal matrix). In otherwords, $\Phi^\mathsf{T} = \Psi^{-1}$.

The relationship between M and M_s has a special name—two matrices A and B are termed *similar* if $A = PBP^{-1}$, for some invertible matrix P. Thinking of A and B as linear operators, A does exactly the same thing as B, only in a different basis. P^{-1} converts vectors in A's basis to B's basis, applies B, and then P converts back to A's basis. Since $M = D^{-1/2} M_s D^{1/2}$, M and M_s are similar. In fact, a matrix only has an eigendecomposition if it is similar to some diagonal matrix, since similarity to a diagonal matrix of eigenvalues is by definition what an eigendecomposition expresses! Any matrix that is similar to a diagonal matrix (i.e., has an eigendecomposition) is said to be *diagonalizable*. In the right basis (given by the eigenvectors), diagonalizable matrices simply scale each component of a vector by some amount (given by the eigenvalues).

Since we can write $M = \Psi \Lambda \Psi^{-1} = \Psi \Lambda \Phi^\mathsf{T}$, M is diagonalizable, and $\Psi \Lambda \Phi^\mathsf{T}$ is the eigendecomposition for M. Note that Ψ contains the *right* eigenvectors of M, while Φ contains the *left* eigenvectors—this may seem backwards, but you can multiply it out to verify (recalling

the mutual orthogonality of Φ and Ψ).

Now that we know all the eigenvalues of M are real (since it shares its eigenvalues with M_s, a real symmetric matrix), here's a simple proof showing that the largest eigenvalue of M is 1: Suppose $\lambda > 1$ was an eigenvalue of M, with corresponding right eigenvector $\mathbf{v}$. Since M has rows summing to 1, each element of $M\mathbf{v}$ is just a weighted average (convex combination) of the elements in $\mathbf{v}$. Therefore the maximum element of $M\mathbf{v}$ can't be greater than $\mathbf{v}_{\max}$, the maximum element of $\mathbf{v}$. But if $\lambda > 1$, the maximum element of $M\mathbf{v}$ is $\lambda\mathbf{v}_{\max} > \mathbf{v}_{\max}$, a contradiction.

One final thing worth noting—powers of diagonalizable matrices have a particularly simple form when we consider their eigendecomposition. If we look at M^2, we get $\Psi\Lambda\Phi^\mathsf{T}\Psi\Lambda\Phi^\mathsf{T} = \Psi\Lambda^2\Phi^\mathsf{T}$ (since Ψ and Φ are mutually orthogonal). In general, $M^t = \Psi\Lambda^t\Phi^\mathsf{T}$, so if we have the eigendecomposition for M, we can take t steps of our walk by simply taking the eigenvalues to the t-th power.

Coordinate Functions

Let's finally do what we came for—find coordinate functions on our observations that allow us to embed them in a lower dimensional space where Euclidean distance relates to random walks among points in our graph. M was derived from W, a matrix describing pairwise similarities between all observations in our data set. So M has dimensions $n \times n$, where n is the number of observations in our data. This means the eigenvectors of M are also length n, and we've already seen that the elements of the first left eigenvector indicate how densely our data is sampled in the region around each observation. This behavior is very much like a function—a mapping from observations in our data to quantities of interest. It turns out that using the other eigenvectors of M as coordinate functions provides a very natural notion of distance based on the evolution of random walks in our graph.

Recall that the eigendecomposition of M was given by $M = \Psi\Lambda\Phi^\mathsf{T}$, where Φ contained as its columns the left eigenvectors of M, and Ψ contained the right eigenvectors of M. If the eigenvectors assign coordinates to our points, then we should be able to get the "embedded" coordinates of a point by simply taking the corresponding row of Φ or Ψ (i.e., each column provides one of the coordinates). We'd like to eventually relate these embeddings to steps in our random walk. A step in our walk starting at node i is given by $e_i^\mathsf{T}M = e_i^\mathsf{T}\Psi\Lambda\Phi^\mathsf{T}$—$e_i^\mathsf{T}\Psi$ simply selects the i-th row of Ψ.

If we use the columns of Ψ as coordinate functions, we can calculate the embedded distance of two observations by looking at the difference between two rows of Ψ: $\left\|e_i^\mathsf{T}\Psi - e_j^\mathsf{T}\Psi\right\|^2$. Expanding this computation out and attempting to simplify leaves us with some stray Λ^{-1}'s that foul up the works, so to save us the trouble of a dead end, let's use $\Upsilon = \Psi\Lambda$ as the

coordinate functions instead. This simply scales the eigenvectors in Ψ by their corresponding eigenvalues. Looking at the distance between rows i and j of Υ (embeddings of observations i and j) gives

$$\begin{aligned}\left\|e_i^\mathsf{T}\Upsilon - e_j^\mathsf{T}\Upsilon\right\|^2 &= (e_i^\mathsf{T}\Upsilon - e_j^\mathsf{T}\Upsilon)(\Upsilon^\mathsf{T}e_i - \Upsilon^\mathsf{T}e_j) && \text{(since } e_i^\mathsf{T}\Upsilon - e_j^\mathsf{T}\Upsilon \text{ is a row vector)}\\ &= (e_i - e_j)^\mathsf{T}\Upsilon\Upsilon^\mathsf{T}(e_i - e_j)\end{aligned}$$

Note that $M = \Psi\Lambda\Phi^\mathsf{T} = \Upsilon\Phi^\mathsf{T} \implies M\Psi = \Upsilon$ (once again due to biorthonormality between Ψ and Φ), so we can expand $\Upsilon\Upsilon^\mathsf{T}$ to give $M\Psi\Psi^\mathsf{T}M^\mathsf{T}$. Let's see what this does to the expression above:

$$\begin{aligned}(e_i - e_j)^\mathsf{T}\Upsilon\Upsilon^\mathsf{T}(e_i - e_j) &= (e_i - e_j)^\mathsf{T}M\Psi\Psi^\mathsf{T}M^\mathsf{T}(e_i - e_j)\\ &= (e_i - e_j)^\mathsf{T}MD^{-1/2}\Omega\Omega^\mathsf{T}D^{-1/2}M^\mathsf{T}(e_i - e_j) && \text{(definition of } \Psi)\\ &= (e_i - e_j)^\mathsf{T}MD^{-1}M^\mathsf{T}(e_i - e_j)\\ &= (e_i^\mathsf{T}M - e_j^\mathsf{T}M)D^{-1}(M^\mathsf{T}e_i - M^\mathsf{T}e_j).\end{aligned}$$

These terms look pleasantly familiar—$e_i^\mathsf{T}M$ (and its transpose $M^\mathsf{T}e_i$) is just the distribution of mass after a step in our walk starting at i, and similarly for j. What's more, this whole expression is just calculating the weighted length of the difference vector between these distribution vectors—in other words, a distance. The weighting is by the inverse of the empirical density of points (row sums of W). We can look at this distance after t steps instead of just one by replacing M with M^t (if you feel like it, you can go back and make sure nothing else in our derivation above is affected by doing this). Let's continue the expansion to make this more explicit:

$$\begin{aligned}\left\|e_i^\mathsf{T}\Upsilon - e_j^\mathsf{T}\Upsilon\right\|^2 &= (e_i^\mathsf{T}M^t - e_j^\mathsf{T}M^t)D^{-1}({M^t}^\mathsf{T}e_i - {M^t}^\mathsf{T}e_j)\\ &= \left\|e_i^\mathsf{T}M^t - e_j^\mathsf{T}M^t\right\|_\mathbf{w}^2\\ &= \sum_k (e_i^\mathsf{T}M^te_k - e_j^\mathsf{T}M^te_k)^2 \cdot w(k)\\ &= \sum_k (p(t,k|i) - p(t,k|j))^2 \cdot w(k), \qquad (3)\end{aligned}$$

where $w(k) = 1/d(k) = D_{kk}^{-1}$ is k-th element of a vector of weights $\mathbf{w}$ given by the inverse of the row sums of W (an estimate of the densities around the observations).

Diffusion Distance

The distance given by (3) is called the *diffusion distance* at time t between two observations, and the coordinate functions given as the columns of $\Upsilon = \Psi\Lambda$ are called the *diffusion map*. The first column in Υ is trivial, since it is constant across all the observations in our data

(and so contributes nothing to distance calculations). Interesting structure starts with the second coordinate function (the "first non-trivial" coordinate function).

How can we interpret diffusion distance? When we start our walk at node i and let it run for t steps, we're left with a probability vector describing the resulting distribution of probability mass. We can do the same thing starting from node j. We now have two distributions of probability mass after t steps, one from a walk starting at i, the other from j, and we can define a natural notion of distance between i and j by looking at the sum of the squared differences of these mass distributions at each node in the graph. So we don't overcount differences in dense areas of the graph, we weight the elements of the difference vector by the inverse of the densities around our observations. Starting at observations in the same neighborhood of the graph will result in similar mass distributions after a given number of steps, while walks starting at observations far apart on the graph won't have much overlap in their mass distributions.

Diffusion distance depends on t, the number of steps we take in the random walk. Not much diffusion happens when t is small, and so observations remain far apart in the diffusion map space. As t goes to infinity, the mass distribution resulting from a walk starting anywhere in the graph approaches the stationary distribution, so distances in the diffusion map space go to zero. Different structures of our data emerge at different times for intermediate values of t.

Let's see how diffusion mapping handles the spiral that was so problematic for PCA. Figure 3 shows how the second coordinate function (first non-trivial) of the diffusion map embeds the spiral. Points distant along the spiral remain distant in the embedding, so the points are laid out by the second coordinate function in exactly the order they occur along the spiral. This means it provides a parametrization of the spiral. It's possible to show this happens for any open curve (in arbitrarily high dimensions)! This is a very nice feature of diffusion mapping. Closed curves get mapped to a circle in the space given by the first two non-trivial coordinate functions.

1 Implement diffusion mapping. This should be a MATLAB function with the signature

```
[map vals] = diffmap(xs, sigma, t, m)
```

Once again, the `xs` variable is a data matrix, `sigma` is the bandwidth parameter used to calculate the similarity matrix W, `t` is the time parameter of the diffusion, and `m` is the number of coordinate functions we care about. The `map` output should contain the first

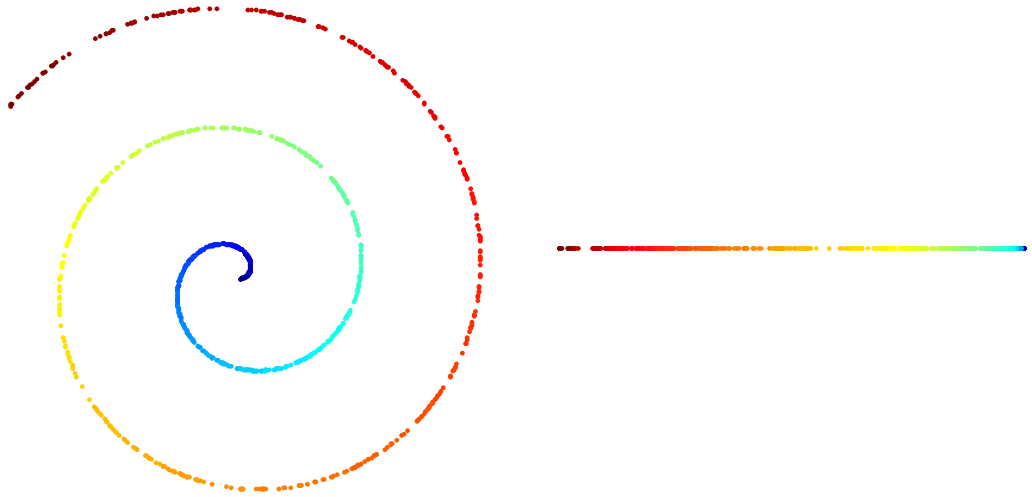

Figure 3: The spiral from Figure 1, this time embedded using the first non-trivial coordinate function given by the diffusion map. Notice how the points are laid out exactly in the order they occur along the spiral, implying the first coordinate of the diffusion map gives a parametrization of the spiral.

`m` non-trivial coordinate functions as its output, and `vals` should be an `m`-length vector containing the eigenvalues corresponding to the first `m` non-trivial coordinate functions. Here is a list of the steps involved:

1. Compute pairwise squared distances between all the columns in `xs`. You can do this with two for-loops, or use the `pdist` function available in both MATLAB and Scipy.

2. Construct the similarities matrix W, as defined in (1).

3. Construct a walk matrix M by normalizing the rows of W so that they sum to 1.

4. Compute the first `m + 1` right eigenvectors of M using the `eigs` function (in Python, use `scipy.sparse.linalg.eigs`).

5. Important: depending on your version of MATLAB or Scipy, `eigs` might not sort the eigenvectors/eigenvalues for you; if that's your case, you'll have to sort the eigenvalues in descending order, and correspondingly rearrange the columns of your eigenvectors.

6. Finally, construct the diffusion map by taking m columns of your eigenvectors starting from the second one, thereby skipping the trivial one—in MATLAB, this means `2:(m + 1)`; in Python, `1:(m + 1)`!—, scaled by their corresponding eigenvalues taken to the `t`-th power.

Test your code on the spiral data set by loading `spiral.mat`. This contains a data matrix in the variable `spirals`, and the position of each point along the spiral in `thetas`. You can use `thetas` to color your embedded points using the `'CData'` option of `scatter` (or, equivalently, the `c` argument of `scatter` in Matplotlib).

2 Now you will apply diffusion maps to the MNIST handwritten digits dataset.

1. Load the data in `digits.mat`. The `X` matrix contains 8×8 images of the digits 0–4 as row vectors, and `y` contains the digit labels corresponding to each data point. Visualize a few of the images (use `reshape`). What would you expect to be a good two-dimensional embedding for this data?

2. Run `diffmap` on the `digits` data. The number of coordinates m should be at least 2. Starting with $t = 1$, experiment with different values for σ. You will visualize the results by creating a scatter plot of the embedded data points using their corresponding digit labels as color. Notice that since this is a 2-D plot you will need to choose two coordinates—the first two non-trivial diffusion coordinates are usually a good first choice.

3. ***Optional***: another cool visualization is drawing the actual digit images at their corresponding positions in the diffusion map. You can do that in MATLAB with `imagesc` by providing coordinates as the first two arguments. In Python, use `AnnotationBbox` from `matplotlib.offsetbox`.

4. After you're happy with the results, try advancing the time and see what happens to the positions of the digits. Comment on the results.

5. Which digits are most similar according to your embedding? Make a new scatter plot using different coordinates (e.g., the 2nd and 3rd ones). Is that still true? Which pair of coordinates do you think is the best one?