Sparse Filtrations and Cohomology

October 25, 2017

1 Multi-Scale Projective Coordinates via Persistent Cohomology of Sparse Filtrations

The idea for this notebook is to apply the dimensionality reduction scheme of Jose A. Perea to th MNIST data base and hopefully get some results. The whole idea behind this is that we have $X \subset (\mathbb{M}, \mathbf{d})$, that is, manifold with a given distance $\mathbf{d}$. We will construct a function that sends ourpoints into: $R\mathbb{P}^n \subset R\mathbb{P}^\infty$ (for some n, number of selected points in the set), and then use an extension of PCA in the real projective plane, so we can extract one or two dimensions of the resulting image. We will first test our code with a sample from the tours, to then proceed with the MNIST data set.

In [153]:
```
#Imports Matplotlib and a little bit of setup
from __future__ import print_function
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
#The mnist
from mnist import MNIST

# import jtplot submodule from jupyterthemes
from jupyterthemes import jtplot

# currently installed theme will be used to
# set plot style if no arguments provided
jtplot.style(theme='default')
#jtplot.style()

#A little bit of matplotlib magic so that the images can be showed on the IPython no
%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

#Small function that helps displays images
def imshow_noax(img, normalize=True):
    """ Tiny helper to show images as uint8 and remove axis labels """
    if normalize:
```

```
        img_max, img_min = np.max(img), np.min(img)
        img = 255.0 * (img - img_min) / (img_max - img_min)
    plt.imshow(img.astype('uint8'))
    plt.gca().axis('off')
```

1.1 Torus Data

We work with random sample of 3000 point from the two dimensional torus surface. This will now corresponds to our X

```
In [ ]: #Sample the points from the torus
        n = 3000
        c = 2 #radius
        a = 1 #tube radius

        theta = 2*np.pi*np.random.rand(n)
        phi = 2*np.pi*np.random.rand(n)

        x = (c + a*np.cos(theta)) * np.cos(phi)
        y = (c + a*np.cos(theta)) * np.sin(phi)
        z = a * np.sin(theta)

        X = np.column_stack((x,y,z))

        #Plots the Torus
        fig = plt.figure()
        ax = Axes3D(fig)
        sca = ax.scatter(X[:,0], X[:,1], X[:,2], color = 'red', cmap=plt.cm.jet)
        ax.set_title(str(n) + ' Points of the Torus')
        ax.set_xlabel('x')
        ax.set_ylabel('y')
        ax.set_zlabel('z')


        plt.show()
```

1.2 Greedy Permutations

We will conduct the greedy selection of points from the given sample. By definition we have:

Definition: If we let $\underline{k} = \{0, \dots k\}$ for $k\mathbb{N}$, and if we have $X \subset (\mathbb{M}, d)$ be a finite subset with $n+1$ elements. A **Greedy Permutation** on X will be a bijection $\sigma_g : \underline{n} \longrightarrow X$, which satisifies:

$$\sigma_g(s+1) = argmax_{x \in X}\, \mathbf{d}\big(x, \sigma_g(\underline{s})\big)$$

In practice, since we will not be using the whole set, we will select the greedy permutation by means of **maxmin** sampling. Here is the code for thisprocedure:

```
In [26]: #Fot the different available distances
         import scipy.spatial.distance as distance
```

```
import numpy as np
import math

def maxmin_sampling(X,
                    n,
                    used_indexes = None,
                    metric = 'cosine',
                    low_ram = False,
                    print_progress = False):
    '''
    Description
    ----------------------------
    Selects n points by via maxmin smapling. This corresponds
    to a greedy permutation
    ----------------------------
    Parameters
    ----------------------------
    X : numpy.array(N,D)
        The sample
    n : integer > 0
        The number of points to select from the sample.
    used_indexes : numpy.array(W)
        The indices of a  maxmin sample. This parameter will be
        used to compute larger samples from already computed ones
    metric : string
        The metric to be used when calculating the distance
        between records.
    low_ram : Boolean
        For low ram ambients. If true, the algorithm never calculates
        the entire distance matrix
    print_progress : Boolean
        Prints progress of sampling
    ----------------------------
    Return : (set, indexes)
    ----------------------------
    A numpy array of shape (n,D), with a set that corresponds to a
    greedy permutation of size n and the array of their
    indices inside the set X
    '''
    #Gets the shape of the sample
    N, D = X.shape

    n = min(n, N)

    #Print percentage
    per = 20
    flag = False
```

```
if(n <= 0):
    raise ValueError('''The number of points to be
                    returned (n) must be larger than zero''')

if(used_indexes is None):
    #Selects a random element that will act like a seed
    seed = np.random.randint(N)
    used_indexes = np.array([seed])
    response = np.array([X[seed,:]])
else:
    response = np.array(X[used_indexes,:])


#checks ram mode to calculate the entire matrix
if(not(low_ram)):
    complete_distance_matrix = distance.cdist(X,
                                              X,
                                              metric = metric)

#Costructs the greedy permutation
while(response.shape[0] < n and response.shape[0] < N):
    #if in low_ram mode
    #Calculates the distance from the remaining set, to the current selection
    if(low_ram):
        distance_matrix = distance.cdist(X,
                                         response,
                                         metric = metric)
    else:
        distance_matrix = X[:, used_indexes]
        if(len(used_index == 1)):
            distance_matrix = distance_matrix.reshape(N,1)

    #Gets all the minimum distances
    min_distances = np.amin(distance_matrix, axis = 1)
    #Eliminates the new indices
    min_distances[used_indexes] = -1
    #Gets the new index
    new_index = np.argmax(min_distances)

    #Adds the new element
    response = np.append(response,[X[new_index,:]], axis = 0)
    used_indexes = np.append(used_indexes, new_index)

    if(print_progress):
        prog = int(np.round(response.shape[0]*100/n))
        if(math.fmod(prog,per) == 0):
            if(not(flag)):
                print(str(prog) + '% Completed')
```

```
                    flag = True
                else:
                    flag = False
            if(print_progress):
                print('Finished!')
            return(response, used_indexes)
```

Hence, we use the previous code to select a subset of 101 points following this procedure.

```
In [154]: n = 100
          X_sample, indices = maxmin_sampling(X,
                                              n+1,
                                              metric = 'euclidean',
                                              low_ram = True,
                                              print_progress = False)
```

1.3 Sparse Filtrations

The next item in our procedure is to select an appropiate set of radiuos for the given subsample. Recall that now we have a set of subsamples:

$$X_s = \{x_0, x_1, \ldots, x_s\} \subset X_n \subset X$$

for $s \in [n]$ and from our construction, we have each X_s in form of a greedy permutation. We will now calculate the insertion radius for each x_s (denoted λ_s). This is defined as:

$$\lambda_s = \begin{cases} \infty & \text{if } s = 0 \\ \mathbf{d}(x_s, X_{s-1}) & \text{if } s > 0 \end{cases}$$

Since X_s is ordered as a gready permutation, it follows that we have a descending order of insertion radii:

$$\infty = \lambda_0 > \lambda_1 \geq \lambda_2 \cdots \geq \lambda_n$$

Now, let us define the function that calculates the **insertion radii** for a given set:

```
In [27]: #Fot the different available distances
         import scipy.spatial.distance as distance
         import numpy as np

         def get_insertion_radii(X_s, metric = 'euclidean'):
             '''
             Description
             ----------------------------
             Calculates the insertion radius for all elements
             in the given set, following its order
             ----------------------------
             Parameters
```

```
    ----------------------------
    X_s : numpy.array(n,D)
        The sample of elements
    metric : string
        The metric to be used when calculating the
        distance between records.
    ----------------------------
    Return
    ----------------------------
    A numpy array of dimension n with the corresponding radii
    '''

    #Starts the radii vector with infinity
    radii = np.array([np.inf])

    for i in range(1,X_s.shape[0]):
        #Calculates the distances of the progressed set to the current record
        distance_matrix = distance.cdist(X_s[0:i,:],
                                         [X_s[i,:]],
                                         metric = metric)

        #Gets the minimum (as the distance of a point to a set is defined)
        radius = np.min(distance_matrix)
        #Assignes value
        radii = np.append(radii, radius)

    return radii
```

We calculate the insertion radii for our sample X_{100}

```
In [155]: lambdas = get_insertion_radii(X_sample,
                                        metric = 'euclidean' )
```

Now we proceed with the calculation of the sparse filtration raddii. For this, we select some $0 < \varepsilon < 1$, and for $\alpha \geq 0$ we have that:

$$r_s(\alpha) = \begin{cases} \alpha & \text{if} \quad \alpha < \lambda_s(1+\varepsilon)/\varepsilon \\ \lambda_s(1+\varepsilon)/\varepsilon & \text{if} \quad \lambda_s(1+\varepsilon)/\varepsilon \leq \alpha \leq \lambda_s(1+\varepsilon)^2/\varepsilon \\ 0 & \text{if} \quad \alpha > \lambda_s(1+\varepsilon)^2/\varepsilon \end{cases}$$

This numbers will correspond to the radii when constructing simplices as we progress through the filtration. Its corresponding function is defined below

```
In [28]: def get_radius(alpha, lam_s, eps = 0.5):
             '''
             Description
             ----------------------------
```

```
    Calculates r_s(alpha) for the given parameters
    ------------------------------
    Parameters
    ------------------------------
    alpha : float
        alpha parameter (alpha >= 0)
    lam_s : float
        lambda_s parameter (lam_s >= 0)
    eps : float
        Epsilon parameter (0 < eps < 1)
    ------------------------------
    Return
    ------------------------------
    A float with the corresponding value for r_s(\alpha)
    '''
    #Cases
    #Break 1
    break_1 = lam_s*(1+eps)/eps
    #break 2
    break_2 = lam_s*(1+eps)**2/eps

    if(alpha < break_1):
        return alpha
    elif(break_1 <= alpha and alpha <= break_2):
        return break_1
    else:
        return 0
```

Although there are several softwares for this type of TDA, most focus only in persistent homology. Since the dimensionality reduction formula is based on the fact that:

$$H^n(B,G) \cong [B, K(G,n)]$$

Where $[A,B]$ is the group of functions from A to B under homotopic equivalence and $K(G,n)$ is the topological space that has G as its n-th homotopy group and zero elsewhere. So particulary, for our intended dimensionality reduction we have that:

$$H^1\left(X, \frac{\mathbb{Z}}{2}\right) \cong \left[X, K\left(\frac{\mathbb{Z}}{2}, 1\right)\right] = [X, R\mathbb{P}^\infty]$$

Since we will not only need the persistent bar codes for the the constructed filtration, but the explicit cocycles for the one dimension cohomology, we turn to **Dionysus2** (*http://www.mrzv.org/software/dionysus2/*). Although this software constructs the rips filtration of a given data set, we are interested in the sparse filtration (something not implemented by this software yet), but we can input the birth-times of the edges manually as a distance matrix into this software and get the bar codes and cocycle via rips complex aproximation.

Such birth times can be calculated efficiently using an algorith developed by Nicholas J. Cavanna, Mahmoodreza Jahanseir and Donald R. Sheehy in their paper: *A Geometric Perspective on*

Sparse Filtrations. The following code corresponds to its implementation.

```
In [29]: #Fot the different available distances
         import scipy.spatial.distance as distance
         import numpy as np

         def get_birth_time(x_i,
                            lam_i,
                            x_j,
                            lam_j,
                            eps = 0.5 ,
                            metric = 'cosine'):
             '''
             Description
             ----------------------------
             Calculates the birth-time (alpha) for the
             edge between the given pair of elements,
             with their corresponding insertion radii.
             ----------------------------
             Parameters
             ----------------------------
             x_i : np.array(1,D)
                 first element of the data set
             lam_i : float
                 insertion radius for the first element (lam_i >= 0)
             x_j : np.array(1,D)
                 second element of the data set
             lam_j : float
                 insertion radius for the second element (lam_j >= 0)
             eps : float
                 Epsilon parameter (0 < eps < 1)
             metric : string
                 The metric to be used when calculating the
                 distance between records.
             ----------------------------
             Return
             ----------------------------
             The corresponding birth time for the edge between x_i and x_j
             '''
             #insertion radii should be in decreasing order
             if(lam_i > lam_j):
                 #switches values
                 lam_i, lam_j = lam_j, lam_i
                 x_i, x_j = x_j, x_i

             #cases
             val_1 = 2*lam_i*(1+eps)/eps
             val_2 = (lam_i+lam_j)*(1+eps)/eps
```

```
    #gets the distance
    dist = distance.pdist(
        np.append(x_i,x_j,axis = 0),
        metric = metric)[0]

    if(dist <= val_1):
        return dist/2
    if(dist <= val_2):
        return dist - val_1/2

    return np.inf
```

1.4 Persistent Cogomology, Cocycles and Dionysus 2

With the previuos filtration, we now turn to Dinoysus2 for computing the barcodes of the resulting persistent homology and cocycle for the desired projection. Recall that we are working over $\mathbb{Z}/2$

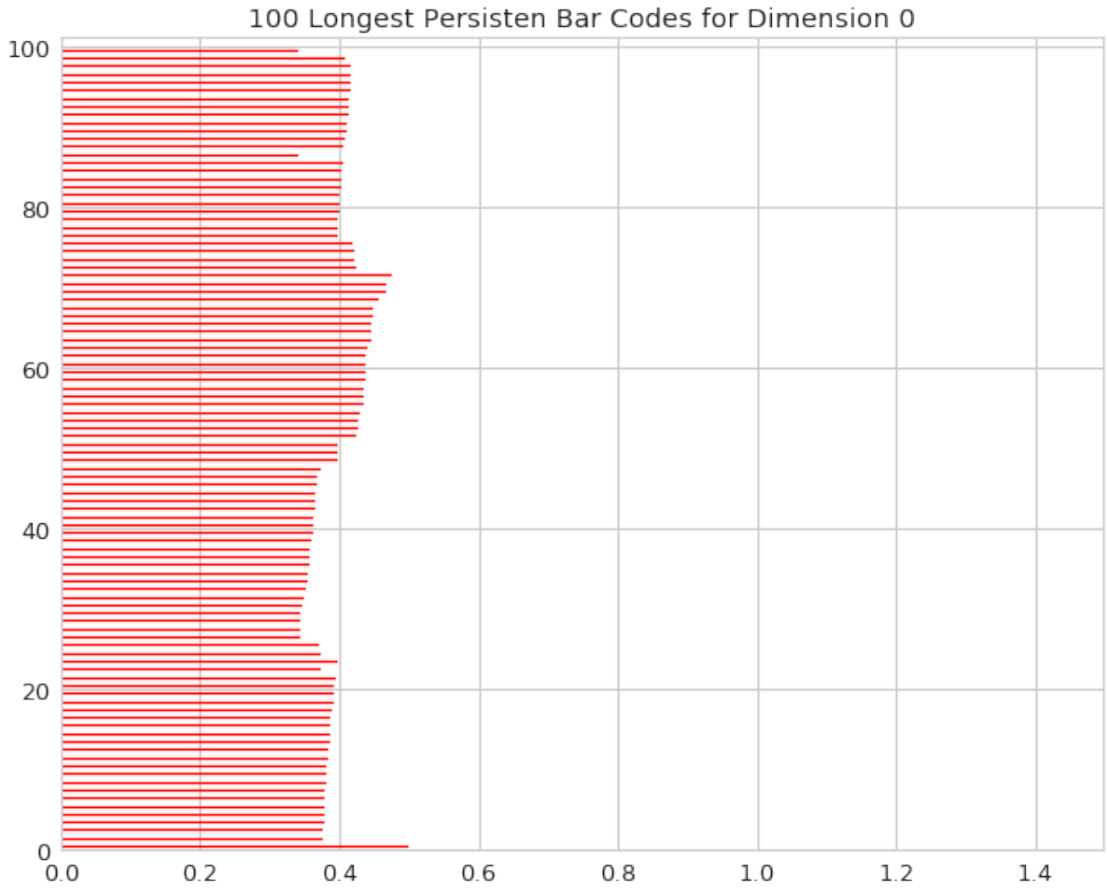
```
In [156]: #We first construct the distance matrix between points
          #The distance here corresponds to the birth-time of the segment between points

          #Small epsilon
          eps = 0.7
          distance_matrix = np.zeros((n+1,n+1))

          for i in range(n+1):
              for j in range(i+1,n+1):
                  distance_matrix[i,j] = get_birth_time(
                                                  [X_sample[i,:]],
                                                  lambdas[i],
                                                  [X_sample[j,:]],
                                                  lambdas[j],
                                                  eps = eps,
                                                  metric = 'euclidean')
                  distance_matrix[j,i] = distance_matrix[i,j]


          #Converts the redundant square form into condensed form
          distance_matrix = distance.squareform(distance_matrix)

In [157]: #The dionysus library
          import dionysus as dio

          #For easier plotting
          from ploting import plot_persistence_diagram


          #Proceeds with the persistent cohomology computation
```

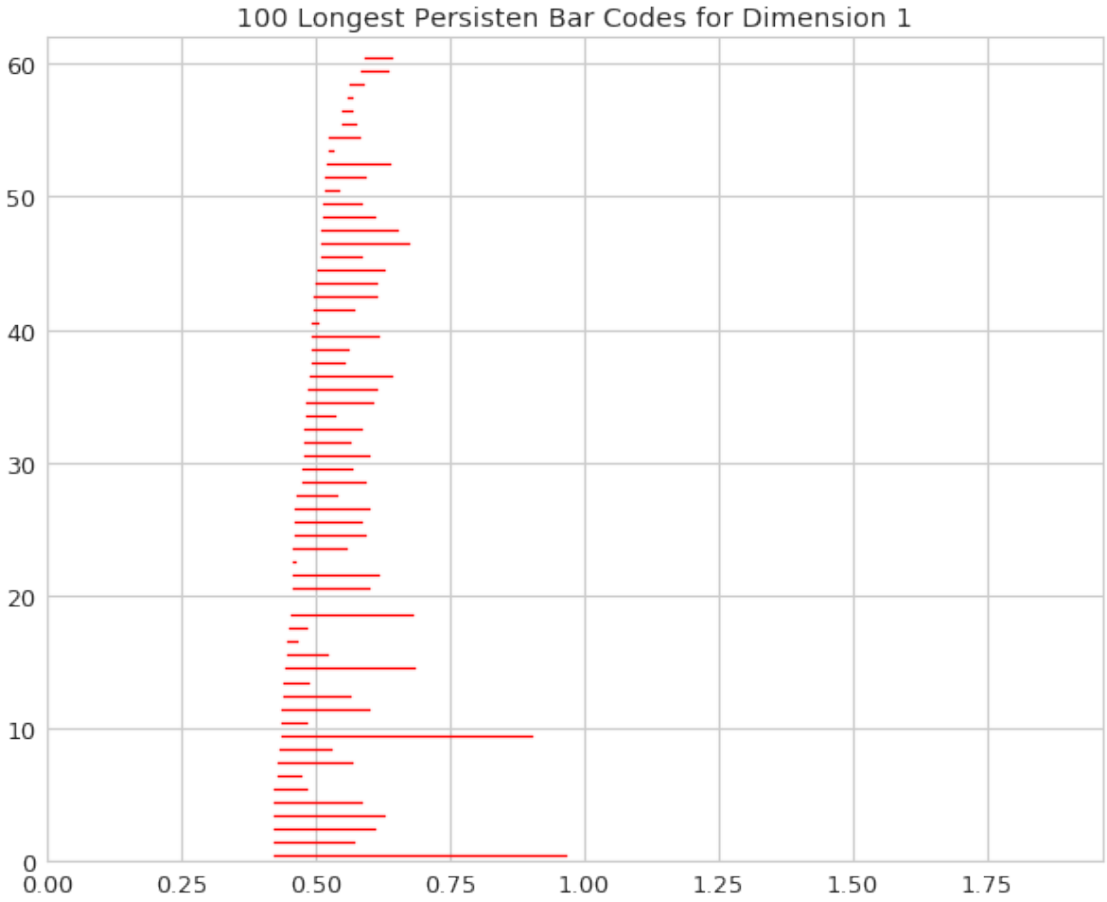
```
prime = 2
dim = 3
max_rad = np.nanmax(
    distance_matrix[np.isfinite(distance_matrix)])/2 + 0.001
f = dio.fill_rips(distance_matrix, dim, max_rad)
p = dio.cohomology_persistence(f, prime, True)
dgms = dio.init_diagrams(p, f)
```

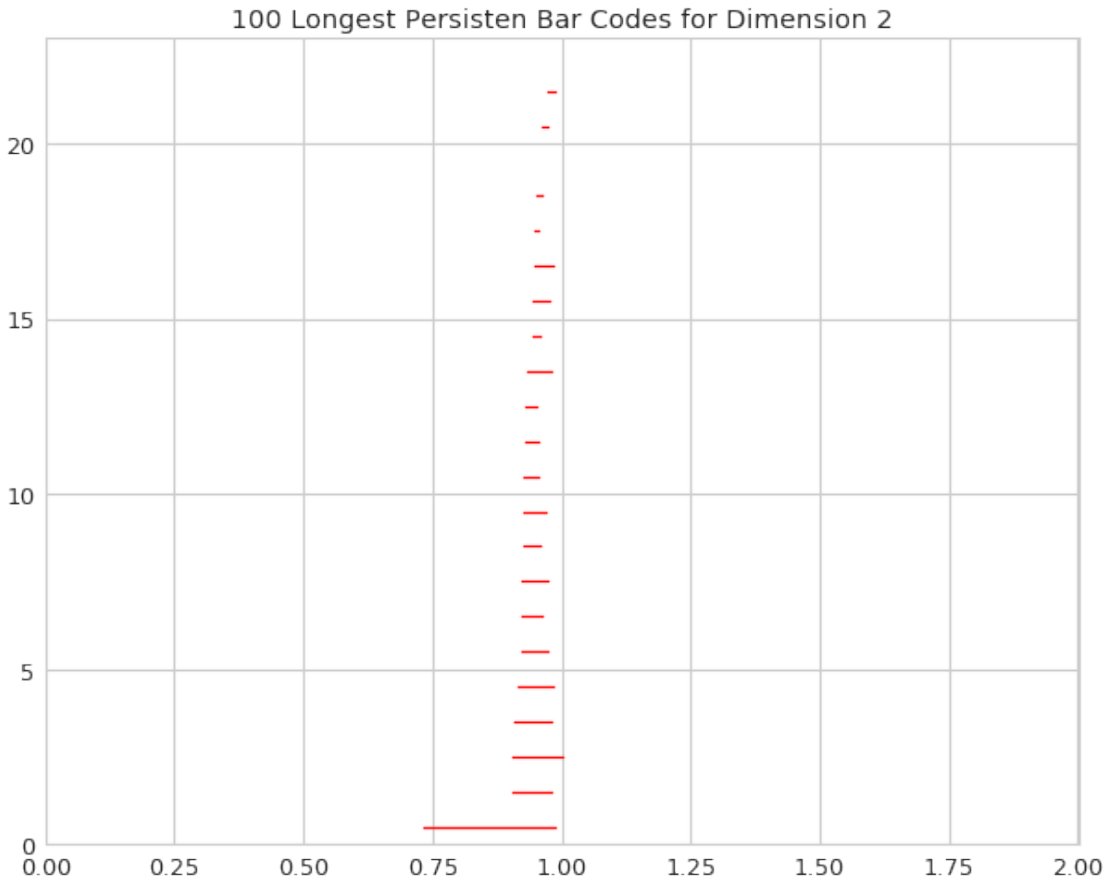
1.4.1 Persistent Cohomology Bar Codes

We now visualize the barcodes of the selected subset

```
In [158]: from ploting import plot_persistence_diagram
          plot_persistence_diagram(dgms, 0, number_of_lines = 100)
          plot_persistence_diagram(dgms, 1, number_of_lines = 100)
          plot_persistence_diagram(dgms, 2, number_of_lines = 100)
```




100 Longest Persisten Bar Codes for Dimension 1
60
50
40
30
20
10
0
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75



1.4.2 Extracting the Cocycles

From the previuos barcdes, for $H^1\left(X, \frac{\mathbb{Z}}{2}\right)$ we see that there are are two dominant (longer) barcodes, these correspond to two non-zero cohomology classes in dimension 1. We will use even further and identify the cocycle representatives μ_1 and μ_2 for these calsses.

```
In [12]: #Extract the longest cocycles
         #Convert diagram into list
         points = list(dgms[1])
         #Sorts from shortest into longest
         points.sort(key = lambda pt: pt.death - pt.birth)
         #Gets the longest points
         p_1, p_2 = points[-1], points[-2]
         #Extracts the cocycles
         mu_1, mu_2 = p.cocycle(p_1.data), p.cocycle(p_2.data)

         #We select alpha as the minimum between the births of mu_1 and mu_2
         if (p_1.birth < p_2.birth):
             mu = mu_1
```

```
        alpha = p_1.death
    else:
        mu = mu_2
        alpha = p_2.death
```

1.4.3 Defines the functions τ_{rt} and $\phi_s^\alpha(b)$

We proceed with the definition of τ_{rt} and $\phi_s^\alpha(b)$. Recall the latter's definirion:

$$\phi_s^\alpha(b) = \frac{|r_s(\alpha) - \mathbf{d}(b, x_s)|_+^2}{\sum_{t\in[n]} |r_t(\alpha) - \mathbf{d}(b, x_t)|_+^2}$$

In [91]:
```
import numpy as np
import scipy.spatial.distance as distance

# In reality we will only use tau_Oi but we will define it anyways
def get_tau(mu, fil, n):
    '''
    Description
    ----------------------------
    Gets the tau function defined in the article
    multiscale projective coordinates via persistent
    cogomology of sparse filtrations by Jose. A Perea.
    ----------------------------
    Parameters
    ----------------------------
    mu : dionysus._dionysus.CoChain
        The dionysus cocycle from the 1 simplexes to Z/2
    fil : dionysus._dionysus.Filtration
        The dionysus filtration with simplex information
    n : integer
        Number of balls in the cover (n+1 balls)
    ----------------------------
    Return
    ----------------------------
    A two dimensioanl numpy array with the values of tau_rs
    '''
    # Initialices the numpy array
    tau = np.zeros((n+1, n+1))
    # iterates over the surviving incides
    for i in mu:
        r = fil[i.index][0]
        t = fil[i.index][1]
        tau[r,t] = 1
        tau[t,r] = 1

    return tau
```

```
def phi(X_n, b, alpha, lambdas = None, metric = 'euclidean', eps = 0.5):
    '''
    Description
    ----------------------------
    Emulates the phi function defined in the article
    multiscale projective coordinates via persistent cogomology
    of sparse filtrations by Jose. A Perea. Returns an array for
    the values of phi(alpha,s)(b) for all s.
    ----------------------------
    Parameters
    ----------------------------
    X_n : numpy.array(n,D)
        The sample of elements
    b : np.array(1,D)
        element of the data set
    alpha : float
        The alpha value corresponding to the radius
    lamdas : numpy.array(n)
        Array with the corresponding insertion raddi.
        If none, will be calculated.
    metric : string
        The metric to be used when calculating
        the distance between records.
    eps : float
        Epsilon parameter (0 < eps < 1)
    ----------------------------
    Return
    ----------------------------
    The value for phi(alpha,s,b)
    '''

    #in case lambdas is none, calculates them
    if(lambdas is None):
        lambdas = get_insertion_radii(X_n, metric = metric )

    #calculates the radiuses
    r = np.apply_along_axis(lambda lam_s:
                            get_radius(alpha,
                                       lam_s[0],
                                       eps = eps), 0,
                            lambdas.reshape(1,len(lambdas)))
    distances = distance.cdist(b, X_n)[0,:]

    #calculates the formula
    #denominator
    values = (r - distances)
    # absolute positive value
```

```
    values = values*(values > 0)
    # Squares it
    values = values**2


    numerator = values
    denominator = sum(values)

    # fraccion
    return numerator / denominator
```

1.4.4 Declares the resulting dimensionality reduction map: f_τ^α

```
In [92]: import numpy as np
         import scipy.spatial.distance as distance


         #Defines the resulting function constructed from the cocycle
         def multiscale_projective_coordinates(X_n,
                                               tau,
                                               b,
                                               j,
                                               alpha,
                                               lambdas = None,
                                               metric = 'euclidean',
                                               eps = 0.5):
             '''
             Description
             ----------------------------
             Calculates the coordinates in the corresponding
             projective plane for the given element
             ----------------------------
             Parameters
             ----------------------------
             X_n : numpy.array(n,D)
                 The sample of elements
             tau : np.array(n+1,n+1)
                 The corresponding values of the cocycle for points [i,j]
             b : np.array(1,D)
                 element of the data set
             j : int
                 The index of the ball the element is contained in
             alpha : float
                 The alpha value corresponding to the radius
             lamdas : numpy.array(n)
                 Array with the corresponding insertion raddi.
                 If none, will be calculated.
```

```
        metric : string
            The metric to be used when calculating
            the distance between records.
        eps : float
            Epsilon parameter (0 < eps < 1)
        ----------------------------
        Return
        ----------------------------
        The value for f(alpha, tau, b)
        '''

        n, D = X_n.shape
        #adjusts n
        n = n -1

        #in case lambdas is none, calculates them
        if(lambdas is None):
            lambdas = get_insertion_radii(X_n, metric = metric )

        phi_res = phi(X_n,b,alpha,lambdas = lambdas,metric = metric,eps = eps)
        result = np.array([((-1)**(tau[:,j])*np.sqrt(phi_res))])

        return result


    def multiscale_projective_coordinates_batch(X,
                                                X_n,
                                                mu,
                                                fil,
                                                alpha,
                                                tau = None,
                                                metric = 'euclidean',
                                                eps = 0.5,
                                                print_progress = False):
        '''
        Description
        ----------------------------
        Calculates the coordinates in the corresponding
        projective plane for the complete set X
        ----------------------------
        Parameters
        ----------------------------
        X : numpy.array(N,D)
            The complete set of elements
        X_n : numpy.array(n,D)
            The sample of elements
        mu : dionysus._dionysus.CoChain
            The dionysus cocycle from the 1 simplexes to Z/2
```

```
fil : dionysus._dionysus.Filtration
    The dionysus filtration with simplex information
alpha : float
    The alpha value corresponding to the radius
tau : np.array(n+1,n+1)
    The corresponding values of the cocycle for
    points [i,j]. If it is None, it will be  calculated
    using mu and the corresponding filtration
metric : string
    The metric to be used when calculating the
    distance between records.
eps : float
    Epsilon parameter (0 < eps < 1)
print_progress : boolean
    Print progress of coordinate calculation
-----------------------------
Return
-----------------------------
The value for f(alpha, tau, x) for all x in X
'''

#Print percentage
per = 20
flag = False

#gets dimensions
N, D = X.shape
n, D = X_n.shape
#adjusts n
n = n - 1

#Calculates the lambda values
lambdas = get_insertion_radii(X_n, metric = metric )
#First gets the radiuses
r = np.apply_along_axis(lambda lam_s:
                        get_radius(alpha,
                                   lam_s[0],
                                   eps = eps),
                        0,
                        lambdas.reshape(1,len(lambdas)))

#gets tau
if(tau is None):
    print('Tau is None, Calculating Tau')
    tau = get_tau(mu, fil, n)

result = np.zeros((N,n+1))
for i in range(N):
```

```
            #extracts b
            b = np.array([X[i,:]])
            #Calculates distances to the sample
            distances_vec = distance.cdist(b,
                                           X_n,
                                           metric = metric)[0,:]
            #Gets the indices of the open balls containing b
            indices = np.where(r > distances_vec)[0]
            if(len(indices) == 0):
                raise ValueError('''The element should belong to
                                 at least one ball, since it's a cover of X''')

            j = indices[0]

            #calculates the ccordinates
            new_coord = multiscale_projective_coordinates(X_n,
                                                          tau,
                                                          b,
                                                          j,
                                                          alpha,
                                                          lambdas,
                                                          metric,eps)

            #saves the coordinates
            result[i,:] = new_coord[0,:]

            if(print_progress):
                prog = int(np.round(i*100/N))
                if(math.fmod(prog,per) == 0):
                    if(not(flag)):
                        print(str(prog) + '% Completed')
                        flag = True
                else:
                    flag = False

        return result
```

If we've done this correctly, we should get a projection of the torus onto $\mathbb{RP}^{100}$. We procced with this final step.

```
In [15]: X_new = multiscale_projective_coordinates_batch(X = X,
                                                        X_n = X_sample,
                                                        mu = mu,
                                                        fil = f,
                                                        alpha = alpha,
                                                        metric = 'euclidean',
                                                        eps = 0.5,
                                                        print_progress = True)
```

```
0% Completed
20% Completed
40% Completed
60% Completed
80% Completed
100% Completed
```

1.5 MNIST data

The following cell imports the data. The data base consists of 60,000 images for training and 10,000 for testing. Each image corresponds to a list of 784 of values between 0 and 255, representing the gray value of the pixel (where 0 equals black and 255 equals white). In turn, each image is of size 28 by 28 pixels in black and white, and can be visualized in $\mathbb{R}^{784}$

```
In [146]: #Imports the data and converts it to numpy arrays
          mndata = MNIST('../python-mnist/data')
          X, y = mndata.load_training()
          X_test, y_test = mndata.load_testing()

          #converts to numpy array and changes coordinates to be between 0 and 1
          #X = np.round(np.array(X)/255, decimals = 3)
          X = np.array(X)
          #X_test = np.round(np.array(X_test)/255, decimals = 3)
          X_test = np.array(X_test)

          y = np.array(y)
          y_test = np.array(y_test)

          #We will use the test set for our computations
          X = X_test
          y = y_test

          #in case we need a subsample
          #subsample_idices = np.random.choice(X.shape[0], 1000)
          #X = X[subsample_idices,:]
          #y = y[subsample_idices]

          #final size
          N, D = X.shape
```

1.5.1 Some Examples

The next cell shows some examples from the data base of digits

```
In [10]: n_row = 3
         n_col = 5
         for i in range(n_row*n_col):
```

```
        ran = np.random.randint(N)
        plt.subplot(n_row, n_col, i+1)
        imshow_noax(X[ran].reshape(28,28), normalize=False)
        plt.title(str(y[ran]))
    plt.show()
```

1.5.2 Extracting the smallest sample necesary

We now proceed with the computation of the smallest sample needed to obtain coordinates for the whole sample. We start with $n = 300$ and increas by 50 elements at a time. This means the smalles max min sample such that after calculating persistent cohomology and the corresponding α we obtain a cover for all of X

```
In [ ]: #The dionysus library
        import dionysus as dio

        n = 100
        step = 25
        ended = False
        X_sample, indices = None, None
        metric = 'euclidean'
```

```
#Persistence parameters
eps = 0.5
prime = 2
dim = 2
N, D = X.shape


while(not ended):
    #gets sample
    X_sample, indices = maxmin_sampling(X,
                                        n+1,
                                        used_indexes = indices,
                                        metric = metric,
                                        low_ram = True,
                                        print_progress = True)
    print(n)
    #persistence
    print('Sample extracted')
    lambdas = get_insertion_radii(X_sample, metric = metric )
    print('Lambdas calculated')
    distance_matrix = np.zeros((n+1,n+1))

    for i in range(n+1):
        for j in range(i+1,n+1):
            distance_matrix[i,j] = get_birth_time([X_sample[i,:]],
                                                  lambdas[i],
                                                  [X_sample[j,:]],
                                                  lambdas[j],
                                                  eps = eps,
                                                  metric = metric)
            distance_matrix[j,i] = distance_matrix[i,j]


    #Converts the redundant square form into condensed form
    distance_matrix = distance.squareform(distance_matrix)
    print('Distance Computed')
    #Proceeds with the persistent cohomology computation
    max_rad = np.nanmax(distance_matrix[np.isfinite(distance_matrix)]) + 0.001
    f = dio.fill_rips(distance_matrix, dim, max_rad)
    p = dio.cohomology_persistence(f, prime, True)
    dgms = dio.init_diagrams(p, f)

    print('Persistence Computed')
    #Extract the longest cocycle
    #Convert diagram into list
    if(len(dgms[1]) == 0):
        print('Nothing in Cohomology Dimension 1')
```

```
    print('----------------------------------')
    n = min(n + step, N-1)
    continue

points = list(dgms[1])
#Sorts from shortest into longest
points.sort(key = lambda pt: pt.death - pt.birth)
#Gets the longest bars
p_1 = points[-1]
#Extracts the cocycles
mu_1  = p.cocycle(p_1.data)

#Assignes the variables
mu = mu_1
alpha = p_1.death
print('min lambda: ' + str(min(lambdas)) + ' max lambda: ' + str(max(lambdas)))
print('alpha: ' + str(alpha))
print('length of cocycle: '  + str(p_1.death - p_1.birth))
if(alpha == np.inf):
    print('Infinite alpha')
    print('----------------------------------')
    n = min(n + step, N-1)
    continue

#Gets the radiuses
r = np.apply_along_axis(lambda lam_s:
                        get_radius(
                            alpha,
                            lam_s[0],
                            eps = eps),
                        0,
                        lambdas.reshape(1,len(lambdas)))

#Checs if we have a cover
temp_dist = distance.cdist(X, X_sample, metric = metric)
#Counts number of points outside cover
missing = N - sum(np.apply_along_axis(sum, 1, (r - temp_dist) > 0 ) > 0)

print('Number of outside points: ' + str(missing))
print('----------------------------------')

ended = (missing == 0)
if(not ended):
    n = min(n + step, N-1)
```

1.5.3 Exctracting the minimum α to obtain a cover

Given a subset $X = \{x_0, \ldots, x_n\}$ with $X \subset \mathbb{X}$, we will calculate the corresponing *tau* and α for this sample. We will then find the minimum $\hat{\alpha} \geq \alpha$ such that:

$$\mathbb{X} \subset \bigcup_{j=0}^{n} B_{r_j(\alpha)}(x_j)$$

```
In [133]: #The dionysus library
          import dionysus as dio
          import sys
          import scipy.spatial.distance as distance


          def extract_minimum_cover(X, n ,eps = 0.5, metric = 'euclidean'):

              #Persistence parameters
              eps = 0.5
              prime = 2
              dim = 2
              N, D = X.shape

              #Computes the diagonal distance of X
              max_vec = np.apply_along_axis(max, 0, X)
              min_vac = np.apply_along_axis(min, 0, X)

              max_alpha = distance.pdist(
                  np.append(np.array([max_vec]),np.array([min_vac]),
                            axis = 0),
                  metric = metric)[0]


              #gets sample
              X_sample, indices = maxmin_sampling(X,
                                                  n+1,
                                                  used_indexes = None,
                                                  metric = metric,
                                                  low_ram = True,
                                                  print_progress = True)

              #persistence
              print('Sample extracted')
              lambdas = get_insertion_radii(X_sample, metric = metric )
              print('Lambdas calculated')
              distance_matrix = np.zeros((n+1,n+1))

              for i in range(n+1):
                  for j in range(i+1,n+1):
```

```
        distance_matrix[i,j] = get_birth_time([X_sample[i,:]],
                                              lambdas[i],
                                              [X_sample[j,:]],
                                              lambdas[j],
                                              eps = eps,
                                              metric = metric)
        distance_matrix[j,i] = distance_matrix[i,j]


#Converts the redundant square form into condensed form
distance_matrix = distance.squareform(distance_matrix)
print('Distance Computed')
#Proceeds with the persistent cohomology computation
max_rad = np.nanmax(distance_matrix[np.isfinite(distance_matrix)]) + 0.001
f = dio.fill_rips(distance_matrix, dim, max_rad)
p = dio.cohomology_persistence(f, prime, True)
dgms = dio.init_diagrams(p, f)

print('Persistence Computed')
#Extract the longest cocycle
#Convert diagram into list
if(len(dgms[1]) == 0):
    print('Nothing in Cohomology Dimension 1')
    print('-----------------------------------')
    sys.exit("Could not complete excecution")

points = list(dgms[1])
#Sorts from shortest into longest
points.sort(key = lambda pt: pt.death - pt.birth)
#Gets the longest bars
p_1 = points[-1]
#Extracts the cocycles
mu_1  = p.cocycle(p_1.data)

#Assignes the variables
mu = mu_1
min_alpha = p_1.death
print('min lambda: ' + str(min(lambdas)) + ' max lambda: ' + str(max(lambdas)))
print('alpha: ' + str(min_alpha))
print('length of cocycle: '  + str(p_1.death - p_1.birth))
if(min_alpha == np.inf):
    print('Infinite alpha')
    print('-----------------------------------')
    sys.exit("Could not complete excecution")


#starts binary search for the smallest alpha
alpha_i = min_alpha
```

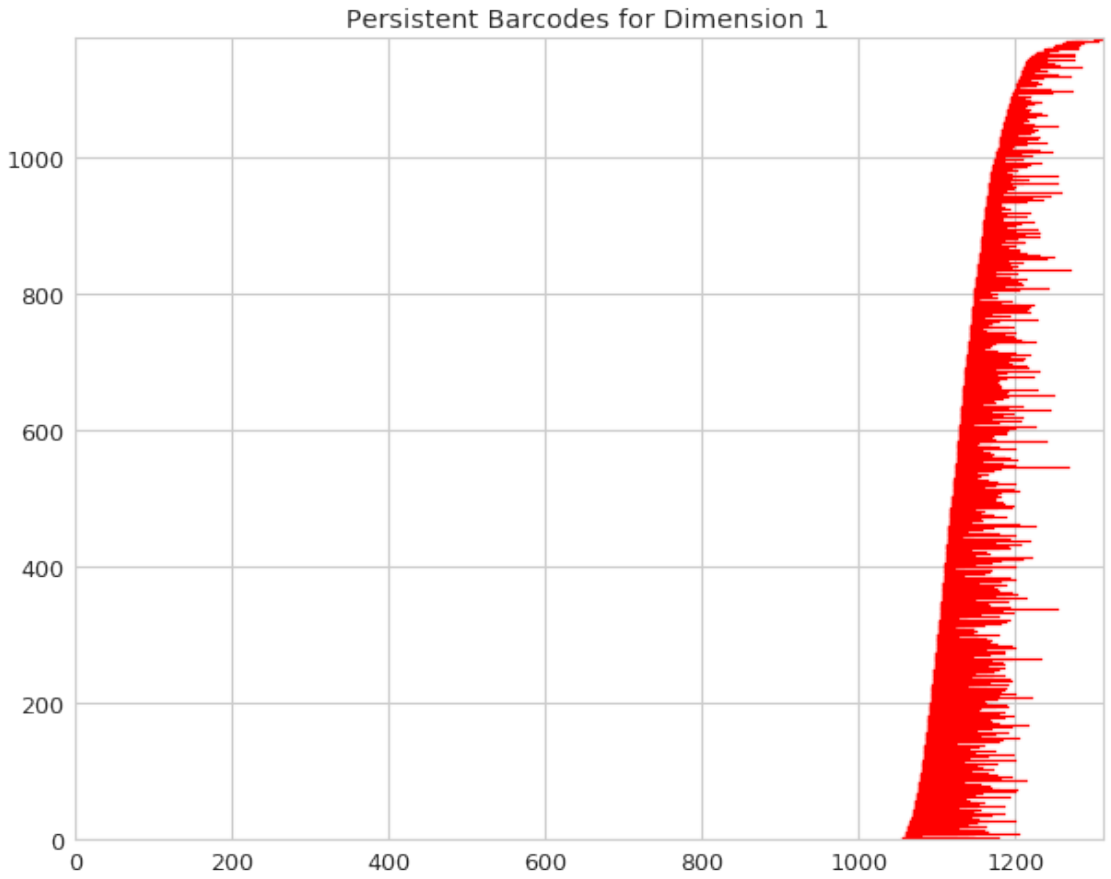
```
alpha_j = max_alpha
alpha_temp = alpha_i
alpha = np.inf
ended = False
diff = 0.1
print('Satarted Looking for Minimum Alpha')
while(not ended):

    #Gets the radiuses
    r = np.apply_along_axis(lambda lam_s:
                            get_radius(alpha_temp,
                                       lam_s[0],
                                       eps = eps),
                            0,
                            lambdas.reshape(1,len(lambdas)))

    #Checks if we have a cover
    temp_dist = distance.cdist(X, X_sample, metric = metric)
    #Counts number of points outside cover
    missing = N - sum(np.apply_along_axis(sum, 1, (r - temp_dist) > 0 ) > 0)
    #print(missing)
    if(missing == 0):
        if(alpha_temp == min_alpha):
            alpha = alpha_temp
            print('Finished')
            ended = True
        elif((alpha_j  - alpha_i) < diff):
            alpha = alpha_temp
            print('Finished')
            ended = True
        else:
            #print('Too High, Keep Looking')
            #print('alpha low: ' + str(alpha_i) + ' alpha high: ' + str(alpha_j))
            alpha_j = alpha_temp
    elif(missing > 0):
        #print('Too low, Keep Looking')
        #print('alpha low: ' + str(alpha_i) + ' alpha high: ' + str(alpha_j))
        alpha_i = alpha_temp
    else:
        sys.exit("Should not enter here")

    #Gets the current alpha
    alpha_temp = alpha_i + (alpha_j - alpha_i)/2

print('----------------------')
print('Initial Alpha: ' + str(min_alpha))
print('Max Alpha: ' + str(max_alpha))
print('Final Alpha ' +  str(alpha_temp))
```

```
        return(X_sample, dgms, f, p, mu, alpha)
```

Extract the minimum α for a compelte cover using 500 points

```
In [ ]: n = 500
        X_sample, dgms, f, p, mu, alpha = extract_minimum_cover(X = X,
                                                                n = n,
                                                                eps = 0.5,
                                                                metric = 'euclidean')
```

```
In [53]: from ploting import plot_persistence_diagram
         plot_persistence_diagram(dgms,
                                  1,
                                  number_of_lines = np.inf,
                                  plot_title = 'Persistent Barcodes for Dimension 1')
```



Notice how, unlike the plot from the torus, there are no predominant classes (long bar codes). So what do we do? Instead of working with the longest one, we can use all of them. Recall that a cocycle is simply a function from edges to $\mathbb{Z}/2$, this means that for each pair of points either we

have a zero or a one. We can, therefore, add all the cocycles togther (modulo 2) and obtain a new cocycle. We will add all the corresponding tau matrices modulo two and use the result as our tau.

```
In [ ]: tau = np.zeros((n+1, n+1))

        j = 0
        #Iterates over all the bars
        for bar in dgms[1]:
            #Extracts the cocycle
            cocycle = p.cocycle(bar.data)
            tau_temp = get_tau(cocycle, f, n)
            #sums modulo 2
            tau = np.fmod(tau + tau_temp, 2)
            j = j +1
            #print(j)
```

1.5.4 Computes the New Coordinates

```
In [ ]: X_new = multiscale_projective_coordinates_batch(X = X,
                                                        X_n = X_sample,
                                                        mu = mu,
                                                        fil = f,
                                                        alpha = alpha,
                                                        tau = tau,
                                                        metric = 'euclidean',
                                                        eps = eps,
                                                        print_progress = True)
```

1.5.5 Saves the New Coordinates

```
In [95]: head_list = list(['proy_' + str(i) for i in range(n+1)])
         head = ', '.join(head_list)
         np.savetxt("multi_scale_coordinates_500_all.csv",
                    X_new,
                    delimiter=",",
                    header = head)
```

1.6 Getting Even Better Results

We will apply the convolutional networks scheme to the images to try and improve the cohomology class. We will slide a window of 6×6 pixels over the images with two pixel overlap. Thus obtaining 49 images of 6×6 for each original mage (we padd the image with zeros to obtain a 30 by 30 matrix)

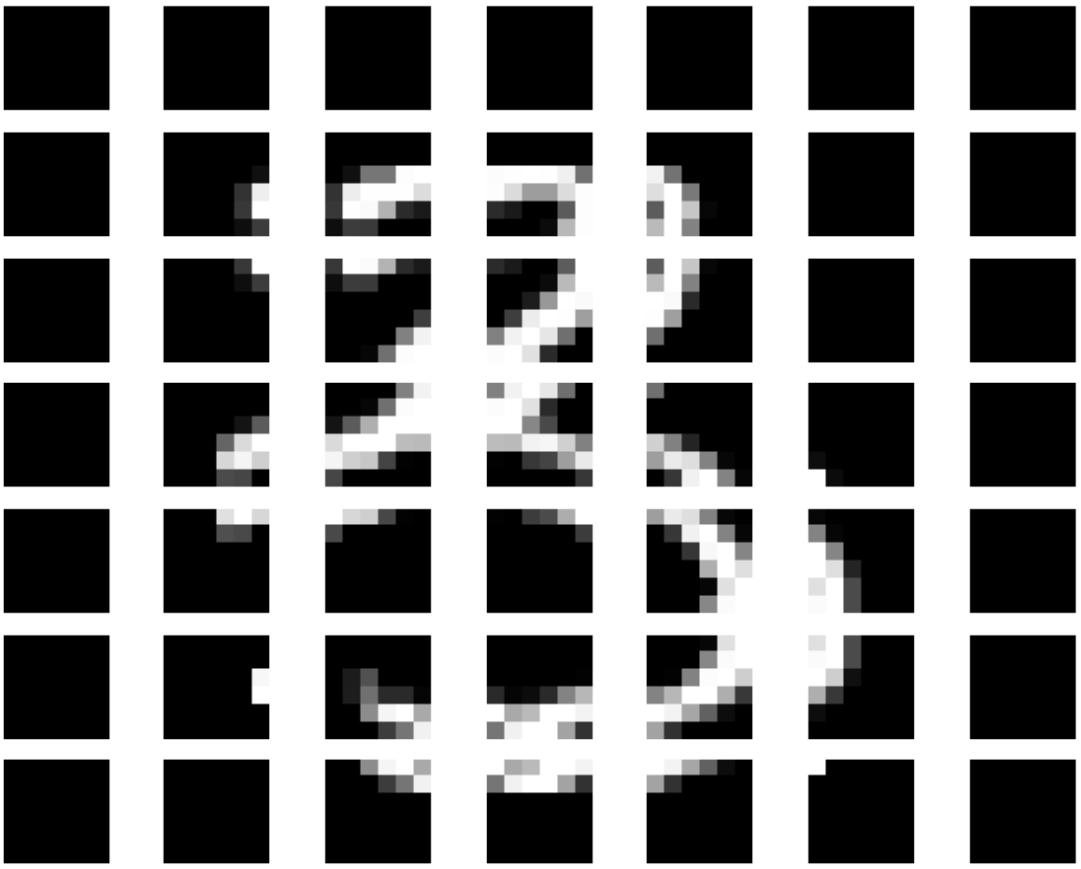
```
In [130]: def extract_patches( row, dim = 6, slide = 2, original_size = 28):
              image = row.reshape(original_size,original_size)
              #Padds the image
              image = np.insert(image, 0, 0, axis=1)
              image = np.insert(image, 29, 0, axis=1)
```

```
    image = np.insert(image, 0, 0, axis=0)
    image = np.insert(image, 29, 0, axis=0)
    num_patches = int((original_size + 2 - dim)/(dim - slide)) +1
    array_images = np.zeros((num_patches*num_patches, dim*dim ))
    step  = dim - slide
    pos = 0
    for i in range(num_patches):
        for j in range(num_patches):
            array_images[pos,:] =  (image[i*step:((i*step) + dim), j*step:((j*step)
            pos = pos +1
    plt.show()
    return array_images
```

1.6.1 Example

Here how the patches look for a digit

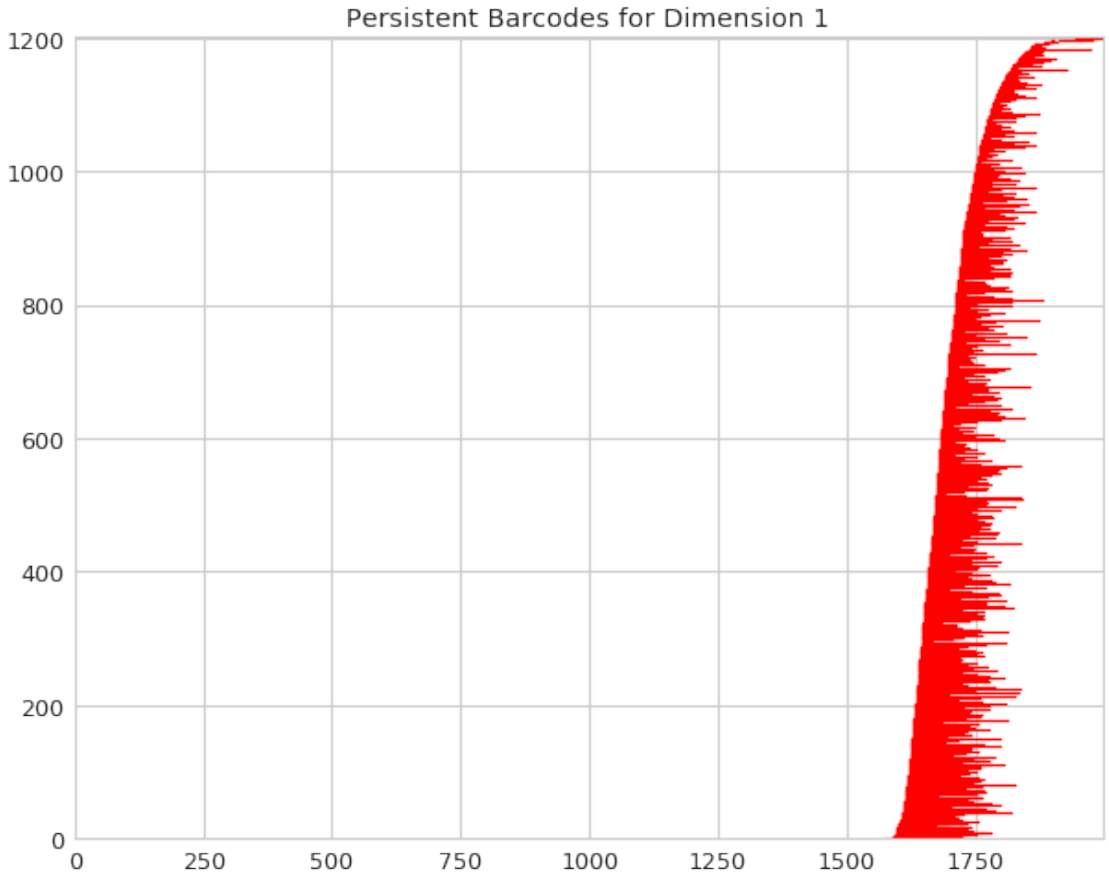
```
In [132]: patches = extract_patches(X[5000]).reshape(-1)
          n_row = 7
          n_col = 7
          for i in range(n_row*n_col):
              image = patches[6*6*i:(6*6*(i+1))]
              plt.subplot(n_row, n_col, i+1)
              imshow_noax(image.reshape(6,6), normalize=False)
          plt.show()
```



We now apply this scheme to all of the MNIST test set and repeat the cohomology computation. Notice that the new representation (with the parches concatenated together) will be of $10,000 \times 1,764$

```
In [ ]: X_patches = np.zeros((10000,1764))
        for i in range(X_patches.shape[0]):
            X_patches[i,:] = extract_patches(X[i]).reshape(-1)

        print('Patches Extracted')

        n = 500
        X_sample, dgms, f, p, mu, alpha = extract_minimum_cover(X = X_patches,
                                                                n = n,
                                                                eps = 0.5,
                                                                metric = 'euclidean')

        print('Finished Persistence Computation')

In [154]: plot_persistence_diagram(dgms,
                                   1,
```

```
                                    plot_title = 'Persistent Barcodes for Dimension 1')
```



Well.... not much of an improvement =(