Notebook for Letter Classification

October 27, 2019

Importing Packages needed for the Process

```
[7]: # Import packages needed initially

import numpy as np
import matplotlib.pyplot as plt
import scipy
from scipy import ndimage
import PIL
from persim import plot_diagrams
from ripser import ripser, lower_star_img
import csv
from numpy import genfromtxt
import persim as pm
```

Upload the Letters File

```
[3]: letters = genfromtxt('letters.csv', delimiter=',')
```

The Next Steps are to Perform Filtrations and Convert those Persistent Diagrams to a Meaningful Value: Our first example will be left to right

First we Perform the Filtration

```
[33]: # Left to Right Scanning and Conversion
dgmLR = [None]*26 #Initialize an empty list
for i in range(26):
    letter_one_line=letters[i,:]

    # initialize matrix of size 10x10 with all values 100
    letter=np.full((10, 10), 100)

    # convert one line letter to 10x10 matrix replacing zeros with 100
    for k in range(1,101):
        if letter_one_line[k]==1.0:
            row=int((k-1)/10)
            column=(k-1)%10
            letter[row,column]=k%10
    dgmLR[i] = lower_star_img(letter)
dgmLR
```

[33]:
```
[array([[ 2., inf]]),
 array([[ 3., inf]]),
 array([[ 2., inf]]),
 array([[ 3., inf]]),
 array([[ 3., inf]]),
 array([[ 3., inf]]),
 array([[ 2., inf]]),
 array([[ 3., inf]]),
 array([[ 4.,  5.],
        [ 4., inf]]),
 array([[ 4.,  6.],
        [ 4., inf]]),
 array([[ 3., inf]]),
 array([[ 3., inf]]),
 array([[ 2., inf]]),
 array([[ 2., inf]]),
 array([[ 2., inf]]),
 array([[ 3., inf]]),
 array([[ 6.,  7.],
        [ 2., inf]]),
 array([[ 3., inf]]),
 array([[ 3.,  8.],
        [ 3., inf]]),
 array([[ 2., inf]]),
 array([[ 3., inf]]),
 array([[ 2., inf]]),
 array([[ 2., inf]]),
 array([[ 3.,  4.],
        [ 3., inf]]),
 array([[ 3., inf]]),
 array([[ 3.,  7.],
        [ 2., inf]])]
```

Now we take this list of persistent diagrams and find the pairwise bottleneck distance between letters

[34]:
```python
# Set an empty pairwise distance matrix for future bottleneck distance input
BNDLR = np.zeros((26,26))



# Change infinities to very large numbers
for i in range(26):
    dgmLR[i][np.isinf(dgmLR[i])] = 10000

# Calculate bottleneck distances and input into the pairwise matrix
for i in range(26):
    for j in range(26):
        BNDLR[i,j] = pm.bottleneck(dgmLR[i], dgmLR[j])
```

```
# The very large values should be set to 0 by bottleneck definity (since the␣
 ↪very large distances would be inifinity)
BNDLR[BNDLR>1000]=0
BNDLR
```

```
[34]: array([[0. , 1. , 0. , 1. , 1. , 1. , 0. , 1. , 2. , 2. , 1. , 1. , 0. ,
         0. , 0. , 1. , 0.5, 1. , 2.5, 0. , 1. , 0. , 0. , 1. , 1. , 2. ],
        [1. , 0. , 1. , 0. , 0. , 0. , 1. , 0. , 1. , 1. , 0. , 0. , 1. ,
         1. , 1. , 0. , 1. , 0. , 2.5, 1. , 0. , 1. , 1. , 0.5, 0. , 2. ],
        [0. , 1. , 0. , 1. , 1. , 1. , 0. , 1. , 2. , 2. , 1. , 1. , 0. ,
         0. , 0. , 1. , 0.5, 1. , 2.5, 0. , 1. , 0. , 0. , 1. , 1. , 2. ],
        [1. , 0. , 1. , 0. , 0. , 0. , 1. , 0. , 1. , 1. , 0. , 0. , 1. ,
         1. , 1. , 0. , 1. , 0. , 2.5, 1. , 0. , 1. , 1. , 0.5, 0. , 2. ],
        [1. , 0. , 1. , 0. , 0. , 0. , 1. , 0. , 1. , 1. , 0. , 0. , 1. ,
         1. , 1. , 0. , 1. , 0. , 2.5, 1. , 0. , 1. , 1. , 0.5, 0. , 2. ],
        [1. , 0. , 1. , 0. , 0. , 0. , 1. , 0. , 1. , 1. , 0. , 0. , 1. ,
         1. , 1. , 0. , 1. , 0. , 2.5, 1. , 0. , 1. , 1. , 0.5, 0. , 2. ],
        [0. , 1. , 0. , 1. , 1. , 1. , 0. , 1. , 2. , 2. , 1. , 1. , 0. ,
         0. , 0. , 1. , 0.5, 1. , 2.5, 0. , 1. , 0. , 0. , 1. , 1. , 2. ],
        [1. , 0. , 1. , 0. , 0. , 0. , 1. , 0. , 1. , 1. , 0. , 0. , 1. ,
         1. , 1. , 0. , 1. , 0. , 2.5, 1. , 0. , 1. , 1. , 0.5, 0. , 2. ],
        [2. , 1. , 2. , 1. , 1. , 1. , 2. , 1. , 0. , 1. , 1. , 1. , 2. ,
         2. , 2. , 1. , 2. , 1. , 2.5, 2. , 1. , 2. , 2. , 1. , 1. , 2. ],
        [2. , 1. , 2. , 1. , 1. , 1. , 2. , 1. , 1. , 0. , 1. , 1. , 2. ,
         2. , 2. , 1. , 2. , 1. , 2. , 2. , 1. , 2. , 2. , 1. , 1. , 2. ],
        [1. , 0. , 1. , 0. , 0. , 0. , 1. , 0. , 1. , 1. , 0. , 0. , 1. ,
         1. , 1. , 0. , 1. , 0. , 2.5, 1. , 0. , 1. , 1. , 0.5, 0. , 2. ],
        [1. , 0. , 1. , 0. , 0. , 0. , 1. , 0. , 1. , 1. , 0. , 0. , 1. ,
         1. , 1. , 0. , 1. , 0. , 2.5, 1. , 0. , 1. , 1. , 0.5, 0. , 2. ],
        [0. , 1. , 0. , 1. , 1. , 1. , 0. , 1. , 2. , 2. , 1. , 1. , 0. ,
         0. , 0. , 1. , 0.5, 1. , 2.5, 0. , 1. , 0. , 0. , 1. , 1. , 2. ],
        [0. , 1. , 0. , 1. , 1. , 1. , 0. , 1. , 2. , 2. , 1. , 1. , 0. ,
         0. , 0. , 1. , 0.5, 1. , 2.5, 0. , 1. , 0. , 0. , 1. , 1. , 2. ],
        [0. , 1. , 0. , 1. , 1. , 1. , 0. , 1. , 2. , 2. , 1. , 1. , 0. ,
         0. , 0. , 1. , 0.5, 1. , 2.5, 0. , 1. , 0. , 0. , 1. , 1. , 2. ],
        [1. , 0. , 1. , 0. , 0. , 0. , 1. , 0. , 1. , 1. , 0. , 0. , 1. ,
         1. , 1. , 0. , 1. , 0. , 2.5, 1. , 0. , 1. , 1. , 0.5, 0. , 2. ],
        [0.5, 1. , 0.5, 1. , 1. , 1. , 0.5, 1. , 2. , 2. , 1. , 1. , 0.5,
         0.5, 0.5, 1. , 0. , 1. , 2.5, 0.5, 1. , 0.5, 0.5, 1. , 1. , 2. ],
        [1. , 0. , 1. , 0. , 0. , 0. , 1. , 0. , 1. , 1. , 0. , 0. , 1. ,
         1. , 1. , 0. , 1. , 0. , 2.5, 1. , 0. , 1. , 1. , 0.5, 0. , 2. ],
        [2.5, 2.5, 2.5, 2.5, 2.5, 2.5, 2.5, 2.5, 2.5, 2. , 2.5, 2.5, 2.5,
         2.5, 2.5, 2.5, 2.5, 2.5, 0. , 2.5, 2.5, 2.5, 2.5, 2.5, 2.5, 1. ],
        [0. , 1. , 0. , 1. , 1. , 1. , 0. , 1. , 2. , 2. , 1. , 1. , 0. ,
         0. , 0. , 1. , 0.5, 1. , 2.5, 0. , 1. , 0. , 0. , 1. , 1. , 2. ],
        [1. , 0. , 1. , 0. , 0. , 0. , 1. , 0. , 1. , 1. , 0. , 0. , 1. ,
 1. , 1. , 0. , 1. , 0. , 2.5, 1. , 0. , 1. , 1. , 0.5, 0. , 2. ],
[0. , 1. , 0. , 1. , 1. , 1. , 0. , 1. , 2. , 2. , 1. , 1. , 0. ,
 0. , 0. , 1. , 0.5, 1. , 2.5, 0. , 1. , 0. , 0. , 1. , 1. , 2. ],
[0. , 1. , 0. , 1. , 1. , 1. , 0. , 1. , 2. , 2. , 1. , 1. , 0. ,
 0. , 0. , 1. , 0.5, 1. , 2.5, 0. , 1. , 0. , 0. , 1. , 1. , 2. ],
[1. , 0.5, 1. , 0.5, 0.5, 0.5, 1. , 0.5, 1. , 1. , 0.5, 0.5, 1. ,
 1. , 1. , 0.5, 1. , 0.5, 2.5, 1. , 0.5, 1. , 1. , 0. , 0.5, 2. ],
[1. , 0. , 1. , 0. , 0. , 0. , 1. , 0. , 1. , 1. , 0. , 0. , 1. ,
 1. , 1. , 0. , 1. , 0. , 2.5, 1. , 0. , 1. , 1. , 0.5, 0. , 2. ],
[2. , 2. , 2. , 2. , 2. , 2. , 2. , 2. , 2. , 2. , 2. , 2. , 2. ,
 2. , 2. , 2. , 2. , 2. , 1. , 2. , 2. , 2. , 2. , 2. , 2. , 0. ]])
```

We then run an agglomorative clustering method on the pairwise distance to grab values for the test

[35]:
```
# Now we look to perform clustering on this pairwise distance matrix. For␣
↪simplicity we use sklearn's Agglomerative Clustering
from sklearn.cluster import AgglomerativeClustering

# For now we set 5 clusters, but we can change this and look for better results␣
↪in our model. A lower number here will prevent overfitting.
clusteringLR = AgglomerativeClustering(n_clusters = 5,
                                       affinity = "precomputed",
                                       linkage = "average").fit(BNDLR)

# This will output a vector of length 26 representing a number for each letter␣
↪for this scan.
LR_test = clusteringLR.labels_
LR_test
```

[35]:
```
array([2, 1, 2, 1, 1, 1, 2, 1, 0, 0, 1, 1, 2, 2, 2, 1, 2, 1, 3, 2, 1, 2,
       2, 1, 1, 4], dtype=int64)
```

As you can see, we now have an array of size 26, where each value corresponds to that letter's value. I will now perform this methodology for the other scannings. (Feel free to skip the reading of this code, it is the same as above's code)

[36]:
```
# Right to Left Scanning

dgmRL = [None]*26 #Initialize an empty list
for i in range(26):
    letter_one_line=letters[i,:]

    # initialize matrix of size 10x10 with all values 100
    letter=np.full((10, 10), 100)

    # convert one line letter to 10x10 matrix replacing zeros with 100
    for k in range(1,101):
        if letter_one_line[k]==1.0:
            row=int((k-1)/10)
            column=(k-1)%10
            letter[row,column]=10-k%10
    dgmRL[i] = lower_star_img(letter)

# Set an empty pairwise distance matrix for future bottleneck distance input
BNDRL = np.zeros((26,26))


# Change infinities to very large numbers
for i in range(26):
    dgmRL[i][np.isinf(dgmRL[i])] = 10000

# Calculate bottleneck distances and input into the pairwise matrix
for i in range(26):
    for j in range(26):
        BNDRL[i,j] = pm.bottleneck(dgmRL[i], dgmRL[j])

# The very large values should be set to 0 by bottleneck definity (since the␣
 ↪very large distances would be inifinity)
BNDRL[BNDRL>1000]=0

# Now we look to perform clustering on this pairwise distance matrix. For␣
 ↪simplicity we use sklearn's Agglomerative Clustering
from sklearn.cluster import AgglomerativeClustering

# For now we set 3 clusters, but we can change this and look for better results␣
 ↪in our model. A lower number here will prevent overfitting.
clusteringRL = AgglomerativeClustering(n_clusters = 5,
                                     affinity = "precomputed",
                                     linkage = "average").fit(BNDRL)

# This will output a vector of length 26 representing a number for each letter␣
 ↪for this scan.
RL_test = clusteringRL.labels_

# Angle Scanning

dgmAngle = [None]*26 #Initialize an empty list
for i in range(26):
    letter_one_line=letters[i,:]

    # initialize matrix of size 10x10 with all values 100
    letter=np.full((10, 10), 100)

    # convert one line letter to 10x10 matrix replacing zeros with 100
    for k in range(1,101):
        if letter_one_line[k]==1.0:
            row=int((k-1)/10)
            column=(k-1)%10
            letter[row,column]=max(k%10,int(k-1)%10)
    dgmAngle[i] = lower_star_img(letter)


# Set an empty pairwise distance matrix for future bottleneck distance input
BNDAngle = np.zeros((26,26))


# Change infinities to very large numbers
for i in range(26):
    dgmAngle[i][np.isinf(dgmAngle[i])] = 10000

# Calculate bottleneck distances and input into the pairwise matrix
for i in range(26):
    for j in range(26):
        BNDAngle[i,j] = pm.bottleneck(dgmAngle[i], dgmAngle[j])

# The very large values should be set to 0 by bottleneck definity (since the␣
 ↪very large distances would be inifinity)
BNDAngle[BNDAngle>1000]=0

# Now we look to perform clustering on this pairwise distance matrix. For␣
 ↪simplicity we use sklearn's Agglomerative Clustering
from sklearn.cluster import AgglomerativeClustering

# For now we set 3 clusters, but we can change this and look for better results␣
 ↪in our model. A lower number here will prevent overfitting.
clusteringAngle = AgglomerativeClustering(n_clusters = 5,
                                     affinity = "precomputed",
                                     linkage = "average").fit(BNDAngle)

# This will output a vector of length 26 representing a number for each letter␣
 ↪for this scan.
A_test = clusteringAngle.labels_

# Diagonal Scanning

dgmDiagonal = [None]*26 #Initialize an empty list
for i in range(26):
    letter_one_line=letters[i,:]

    # initialize matrix of size 10x10 with all values 100
    letter=np.full((10, 10), 100)

    # convert one line letter to 10x10 matrix replacing zeros with 100
    for k in range(1,101):
        if letter_one_line[k]==1.0:
            row=int((k-1)/10)
            column=(k-1)%10
            letter[row,column]=(column+row)*k%10
    dgmDiagonal[i] = lower_star_img(letter)

# Set an empty pairwise distance matrix for future bottleneck distance input
BNDDiagonal = np.zeros((26,26))


# Change infinities to very large numbers
for i in range(26):
    dgmDiagonal[i][np.isinf(dgmDiagonal[i])] = 10000

# Calculate bottleneck distances and input into the pairwise matrix
for i in range(26):
    for j in range(26):
        BNDDiagonal[i,j] = pm.bottleneck(dgmDiagonal[i], dgmDiagonal[j])

# The very large values should be set to 0 by bottleneck definity (since the␣
 ↪very large distances would be inifinity)
BNDDiagonal[BNDDiagonal>1000]=0

# Now we look to perform clustering on this pairwise distance matrix. For␣
 ↪simplicity we use sklearn's Agglomerative Clustering
from sklearn.cluster import AgglomerativeClustering

# For now we set 3 clusters, but we can change this and look for better results␣
 ↪in our model. A lower number here will prevent overfitting.
clusteringDiagonal = AgglomerativeClustering(n_clusters = 5,
                                             affinity = "precomputed",
                                             linkage = "average").fit(BNDDiagonal)

# This will output a vector of length 26 representing a number for each letter␣
 ↪for this scan.
D_test = clusteringDiagonal.labels_
```

```
C:\Users\Putts\Anaconda3\lib\site-packages\ripser\ripser.py:342: RuntimeWarning:
invalid value encountered in maximum
  thisD = np.maximum(thisD, tD)
C:\Users\Putts\Anaconda3\lib\site-packages\ripser\ripser.py:342: RuntimeWarning:
invalid value encountered in maximum
  thisD = np.maximum(thisD, tD)
C:\Users\Putts\Anaconda3\lib\site-packages\ripser\ripser.py:342: RuntimeWarning:
invalid value encountered in maximum
  thisD = np.maximum(thisD, tD)
```

Don't worry too much about the runtime warning, it isn't an error in the code. The next step after this is to grab some variables that test the densities of portions of the grid. This helps differentiate R & B, O & Q, and some others. The areas we found the density of should be large enough not to overfit if some noise occurred in the image.

[38]:
```
# A test to differentiate some letters
bottom_test = [None]*26
for i in range(26):
    bottom_test[i]=sum(letters[i][51:101])

right_test = [None]*26
for i in range(26):
    right_test[i]=sum(np.concatenate((letters[i][6:11],
              letters[i][16:21],
              letters[i][26:31],
              letters[i][36:41],
              letters[i][46:51],
              letters[i][56:61],
              letters[i][66:71],
              letters[i][76:81],
              letters[i][86:91],
              letters[i][96:101]
              )))

botright = [None]*26
for i in range(26):
    botright[i] = sum(np.concatenate((
              letters[i][56:61],
              letters[i][66:71],
              letters[i][76:81],
              letters[i][86:91],
              letters[i][96:101]
              )))

top_test = [None]*26
for i in range(26):
    top_test[i] = sum(letters[i][1:51])

density_test = [None]*26
for i in range(26):
    density_test[i] = sum(letters[i][1:101])
```

Now we combine all of our newly found variables into a neat matrix. This will be our new design matrix to input into our prediction process.

[39]:
```
test_array = np.concatenate((LR_test,
                             RL_test,
                             A_test,
                             D_test,
                             bottom_test,
                             right_test,
                             botright,
                             top_test,
                             density_test))
test_array = np.reshape(test_array,(9,26)).T
test_array
```

[39]:
```
array([[ 2.,  0.,  2.,  2., 13., 13.,  6., 14., 27.],
       [ 1.,  0.,  1.,  0., 13., 16.,  7., 20., 33.],
       [ 2.,  2.,  2.,  0., 11.,  7.,  4., 10., 21.],
       [ 1.,  0.,  1.,  0., 15., 16.,  9., 13., 28.],
       [ 1.,  3.,  1.,  1.,  9.,  9.,  3., 14., 23.],
       [ 1.,  2.,  1.,  0.,  4.,  4.,  0., 12., 16.],
       [ 2.,  2.,  2.,  2., 13.,  7.,  5., 10., 23.],
       [ 1.,  4.,  1.,  0.,  6.,  8.,  3., 11., 17.],
       [ 0.,  1.,  0.,  2.,  6.,  2.,  1.,  6., 12.],
       [ 0.,  1.,  0.,  0.,  5.,  7.,  3.,  6., 11.],
       [ 1.,  2.,  1.,  0., 11.,  8.,  4., 11., 22.],
       [ 1.,  4.,  1.,  0.,  8.,  2.,  2.,  4., 12.],
       [ 2.,  0.,  2.,  0.,  9., 12.,  3., 19., 28.],
       [ 2.,  4.,  2.,  2., 10., 10.,  6., 13., 23.],
       [ 2.,  0.,  2.,  0., 13., 12.,  6., 12., 25.],
       [ 1.,  0.,  1.,  0.,  8.,  9.,  1., 14., 22.],
       [ 2.,  0.,  2.,  1., 15., 14.,  8., 13., 28.],
       [ 1.,  1.,  1.,  0., 12., 10.,  4., 13., 25.],
       [ 3.,  2.,  3.,  1., 13., 11.,  8., 10., 23.],
       [ 2.,  0.,  2.,  4.,  4.,  3.,  0., 10., 14.],
       [ 1.,  0.,  1.,  1., 12., 10.,  6.,  8., 20.],
       [ 2.,  0.,  2.,  3.,  9.,  9.,  4., 11., 20.],
       [ 2.,  0.,  2.,  0., 15., 17.,  8., 17., 32.],
       [ 1.,  1.,  1.,  2., 12., 11.,  5., 13., 25.],
       [ 1.,  4.,  1.,  0.,  7.,  6.,  1., 10., 17.],
       [ 4.,  2.,  4.,  0., 11., 11.,  3., 13., 24.]])
```

We decided that we would run it through the most simple algorithm for predicting multiple classes. Just a normal multinomial regression.

[40]:
```python
# Grab our labels
training_labels = [None]*26
for i in range(26):
    training_labels[i] = letters[i][0]


# Fit our model and see if it has 100% accuracy on training data
from sklearn.linear_model import LogisticRegression
LogReg=LogisticRegression()
LogReg.fit(test_array, training_labels)
y_pred = LogReg.predict(test_array)
```

```
y_pred
```

```
C:\Users\Putts\Anaconda3\lib\site-packages\sklearn\linear_model\logistic.py:433:
FutureWarning: Default solver will be changed to 'lbfgs' in 0.22. Specify a
solver to silence this warning.
  FutureWarning)
C:\Users\Putts\Anaconda3\lib\site-packages\sklearn\linear_model\logistic.py:460:
FutureWarning: Default multi_class will be changed to 'auto' in 0.22. Specify
the multi_class option to silence this warning.
  "this warning.", FutureWarning)
```

[40]:
```
array([ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11., 12., 13.,
       14., 15., 16., 17., 18., 19., 20., 21., 22., 23., 24., 25., 26.])
```

As you can see, there is 100% accuracy for the original data. I now look to set up this notebook so that you can input new vectors to test. Please add some random number in the beginning. For example I started with a 1 since it was A. This is just there so the indexing works. You could put a string there if you wanted for reference. This example is a noisy A.

[116]:
```
newLetter=[1,0., 0., 0., 0., 1., 0, 0., 0., 0., 0., 0., 0., 0., 0., 1., 1.,
0., 0., 0., 0., 0., 0., 0., 1., 1., 1., 0., 0., 0., 0., 0., 0., 0.,
1., 1., 1., 1., 0., 0., 0., 0., 0., 0., 1., 0., 1., 1., 0., 0., 0.,
0., 0., 1., 1., 1., 1., 1., 0., 0., 0., 0., 0., 1., 0., 0., 0., 1.,
0., 0., 0., 0., 1., 1., 0., 0., 0., 1., 1., 0., 0., 0., 1., 0., 0.,
    0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]
```

Now we look to perform all filtrations on the new letter, as well as the density tests.

[117]:
```
 #Left to Right Filtration
letter=np.full((10, 10), 100)

    # convert one line letter to 10x10 matrix replacing zeros with 100
for k in range(1,101):
    if newLetter[k]==1.0:
        row=int((k-1)/10)
        column=(k-1)%10
        letter[row,column]=k%10
dgmNLR = lower_star_img(letter)

 # Right to Left Filtration
letter=np.full((10, 10), 100)

    # convert one line letter to 10x10 matrix replacing zeros with 100
for k in range(1,101):
    if newLetter[k]==1.0:
        row=int((k-1)/10)
        column=(k-1)%10
        letter[row,column]=10-k%10
dgmNRL = lower_star_img(letter)
```

```
 # Angle Filtration
letter=np.full((10, 10), 100)

# convert one line letter to 10x10 matrix replacing zeros with 100
for k in range(1,101):
    if newLetter[k]==1.0:
        row=int((k-1)/10)
        column=(k-1)%10
        letter[row,column]=max(k%10,int(k-1)%10)
dgmNA = lower_star_img(letter)

 # Diagonal Filtration
letter=np.full((10, 10), 100)

    # convert one line letter to 10x10 matrix replacing zeros with 100
for k in range(1,101):
    if newLetter[k]==1.0:
        row=int((k-1)/10)
        column=(k-1)%10
        letter[row,column]=(column+row)*k%10
dgmND = lower_star_img(letter)

# A test to differentiate some letters

bottom_test=sum(newLetter[51:101])


right_test=sum(np.concatenate((newLetter[6:11],
              newLetter[16:21],
              newLetter[26:31],
              newLetter[36:41],
              newLetter[46:51],
              newLetter[56:61],
              newLetter[66:71],
              newLetter[76:81],
              newLetter[86:91],
              newLetter[96:101]
              )))

botright = sum(np.concatenate((
              newLetter[56:61],
              newLetter[66:71],
              newLetter[76:81],
              newLetter[86:91],
              newLetter[96:101]
              )))
```

```
top_test = sum(newLetter[1:51])

density_test = sum(newLetter[1:101])
```

```
C:\Users\Putts\Anaconda3\lib\site-packages\ripser\ripser.py:342: RuntimeWarning:
invalid value encountered in maximum
  thisD = np.maximum(thisD, tD)
```

Now we use the filtrations to find the bottleneck distance between the new letter and all of the old ones.

[118]:
```
# Change infinities to very large numbers
dgmNLR[np.isinf(dgmNLR)] = 10000
dgmNRL[np.isinf(dgmNRL)] = 10000
dgmNA[np.isinf(dgmNA)] = 10000
dgmND[np.isinf(dgmND)] = 10000

# Find bottleneck distance between new letter and previous letters

# Left to Right
# Calculate bottleneck distances and input into the pairwise matrix
BNDNLR = [None]*26
for i in range(26):
    BNDNLR[i] = pm.bottleneck(dgmLR[i],dgmNLR)
BNDNLR = np.array(BNDNLR)
BNDNLR[BNDNLR>1000]=0

# Right to Left
# Calculate bottleneck distances and input into the pairwise matrix
BNDNRL = [None]*26
for i in range(26):
    BNDNRL[i] = pm.bottleneck(dgmRL[i],dgmNRL)
BNDNRL = np.array(BNDNRL)
BNDNRL[BNDNRL>1000]=0

# Angle
# Calculate bottleneck distances and input into the pairwise matrix
BNDNA = [None]*26
for i in range(26):
    BNDNA[i] = pm.bottleneck(dgmAngle[i],dgmNA)
BNDNA = np.array(BNDNA)
BNDNA[BNDNA>1000]=0

# Diagonoal
# Calculate bottleneck distances and input into the pairwise matrix
BNDND = [None]*26
for i in range(26):
    BNDND[i] = pm.bottleneck(dgmDiagonal[i],dgmND)
```

```
BNDND = np.array(BNDND)
BNDND[BNDND>1000]=0
```

Convert them into their values using Agglomerative Clustering

[119]:
```
# Left to Right Value
temp=np.vstack((BNDLR,BNDNLR))
temp2=np.append(BNDNLR,0)
NewBNDLR = np.hstack((temp, np.atleast_2d(temp2).T))
LRClust = AgglomerativeClustering(n_clusters = 5,
                                     affinity = "precomputed",
                                     linkage = "average").fit(NewBNDLR)
LRValue=LRClust.labels_[26]

# Right to Left Value
temp=np.vstack((BNDRL,BNDNRL))
temp2=np.append(BNDNRL,0)
NewBNDRL = np.hstack((temp, np.atleast_2d(temp2).T))
RLClust = AgglomerativeClustering(n_clusters = 5,
                                     affinity = "precomputed",
                                     linkage = "average").fit(NewBNDRL)
RLValue=RLClust.labels_[26]


# Angle Value
temp=np.vstack((BNDAngle,BNDNA))
temp2=np.append(BNDNA,0)
NewBNDA = np.hstack((temp, np.atleast_2d(temp2).T))
AngleClust = AgglomerativeClustering(n_clusters = 5,
                                     affinity = "precomputed",
                                     linkage = "average").fit(NewBNDA)
AValue=AngleClust.labels_[26]


# Diagonal Value
temp=np.vstack((BNDDiagonal,BNDND))
temp2=np.append(BNDND,0)
NewBNDD = np.hstack((temp, np.atleast_2d(temp2).T))
DiagonalClust = AgglomerativeClustering(n_clusters = 5,
                                     affinity = "precomputed",
                                     linkage = "average").fit(NewBNDD)
DValue=DiagonalClust.labels_[26]
```

Combine these values into one vector and run our model on that vector

[120]:
```
New_Letter = np.array((LRValue,
                             RLValue,
                             AValue,
                             DValue,
                             bottom_test,
                             right_test,
                             botright,
                             top_test,
                             density_test))
LogReg.predict(New_Letter.reshape(1,-1))[0]
```

[120]:
```
1.0
```

Our classifier isn't the best with noise. It would be a much better classifier if we included the regular persistent homology diagram up to dimension 1. Also, if we include an L_1 regularization term to our multinomial regression then that would also have helped.