Project #1: Building a Neural Network and the Fisher Discriminant

ECE 566: Machine & Deep Learning

Due: November 8, 2021

Daniel Marten
A20394276

I. **Introduction**

The principles of neural networks — their self-training characteristics and ability to be trained to perform classification or regression analysis on a large variety of datasets, given sufficient quantity and quality of training data — make them an incredibly versatile and high-powered tool in modern machine and deep learning. While our course has covered principles and aspects of neural networks up until this point, the purpose of this project is to design and implement our very own neural network in order to perform classification analysis on a set of data. The data we were given is a set of handwritten arabic numerals (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) that we must analyze. However, one major point of comparison comes from comparing the performance of the neural network with that of pre-existing Fisher discriminant methods, as derived from the features of the individual examples — features like shape radius, perimeter, the area of the 'pencilled-in' digit, and so on — and their means and covariances. As a broad conclusion, both methods can be successful in the right context, with enough resources, and the proper training data, though neural networks have the key advantage of flexibility yet a key drawback of being potentially extremely time-intensive and resource-heavy.

The project is coded in Python and uses numpy, matplotlib, keras, and skimage. Further, this project was made possible thanks to the code provided by Professor Jovan Brankov, as well as the notes from class and ample searches for Python documentation.

II. **Summary**

Question 1

In Question 1, this project implements a Fisher discriminant to separate handwritten 0s and 1s based off of training data, using feature data for area and perimeter, then apply that to our testing data for analysis. This paper reports an accuracy of 99.198% on the training data and 99.388% on the testing data for 0s and 1s. While only ~0.2% different on otherwise great accuracy metrics, a hypothesis for this improvement is given in the **Body**.

The project also implements a Fisher discriminant method to separate handwritten 5s and 6s based off of training data, also using area and perimeter feature data, but the results are not as optimistic. For this set of digits, the methods yield an accuracy of 64.379% and 63.397% for training and testing data, which is much less than preferable. This is expanded upon in the **Body** section, though in summary, it is a great example of how Fisher discriminants on the same set of features (or other values for classes) are not equal at separating different pairs of classes.

Question 2

In Question 2, we design, train, and test a neural network to similarly discriminate between both handwritten 0s and 1s as well as between 5s and 6s. For equal epochs and learning rates, discrimination between 0s and 1s reports above a 99.7% accuracy for both training and testing, while discrimination between 5s and 6s reports a 92.4% accuracy for training and 92.7% accuracy for testing. This accuracy could be improved, given sufficient time and resources. Importantly, this is a significant improvement over 5-6 classification using Fisher discriminants, showing the versatility of neural networks in classification.

III. **Body**

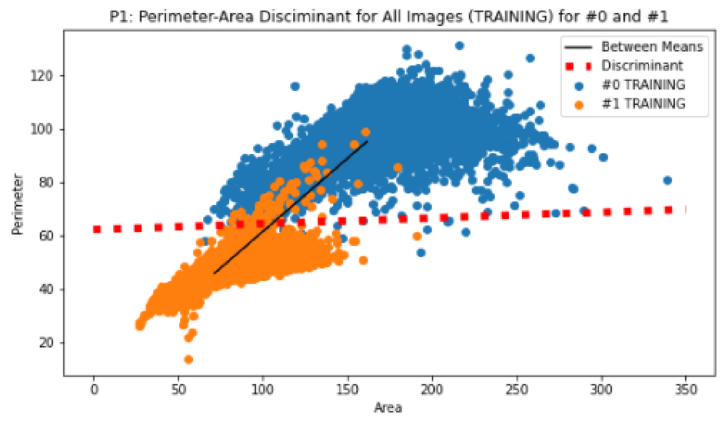
Question 1: 0-1 Classification

To begin Question 1, we classify handwritten 0s and 1s from the keras dataset using extracted feature data — perimeter, area, centroid coordinates in both x- and y-dimensions, eccentricity, and minor axis length. Perimeter and area were first selected to differentiate between 0s and 1s, due to the features with the largest values for each class, as well as both class' values for each being notably different. This summary screen is included as a **Supplemental Figure 1.** Key points from it include that there are 5923 0s and 6742 1s in the training dataset. Additionally, there are 980 0s and 1135 1s in the training dataset, and all images are 28-by-28 pixels.

To find the Fisher's discriminant between the two classes, the means of the training data for each feature for each class and the covariance matrix for each class were found using numpy methods. Assuming equal covariance for each class, these matrices were averaged to produce one matrix. Fisher's linear discriminant was then derived from this value and the differences between the means of each class, producing a 2D-vector. If all of the points were projected onto this 2D vector, it would produce a histogram clearly showing the two classes. However, for our purposes, a boundary between the two classes was assumed to be perpendicular to this vector — with the negative inverse of its slope — and was found iteratively, by starting at different values of y and computing the accuracy of the line as a boundary. Though inefficient, this produces a boundary with an accuracy of 99.198% for the training data. The exact same boundary, when applied to the testing data, produces an accuracy of 99.388%. These graphs and accuracy reporting are seen in **Figure 1**.

a

```
Training Data Accuracy Rate: 99.19804153300691
```



b

```
Testing Data Accuracy Rate: 99.38775510204081
```

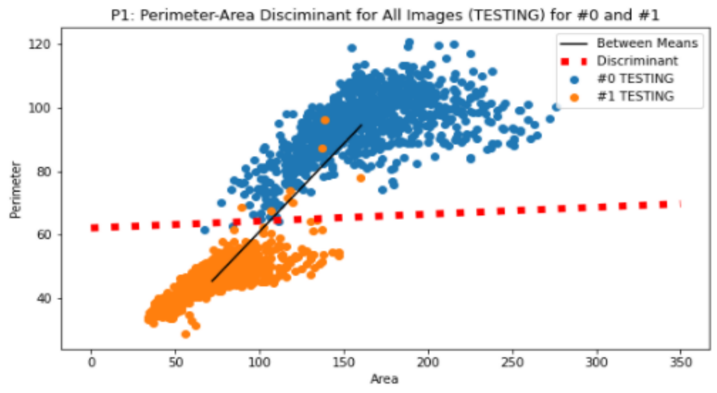


Figure 1: Graphical depictions of the data and discriminant between 0s and 1s for training **(a)** data and testing **(b)** data. Interestingly, the testing data reports a higher accuracy rate than the training data for the same discriminant. This is potentially due to the 'protrusion' of #1 data points with higher than normal perimeter values in **a**. However, the result of a discriminant that performs well for both training and testing data is a strong positive and shows great ability to classify the two handwritten digits.

Question 1: 5-6 Classification

Though there is some ambiguity, I interpreted part 4 of Question 1 as asking us to repeat our classification of 0-1 on 5-6 using the same features, as a way of clarifying the limits of Fisher discriminants and using feature data. Analyses were performed on 5421 images of 5 and 5918 of 6 as training data, and 892 images of 5 and 958 of 6 as testing data. Image dimensions are the same as for 0 and 1. As such, I performed the same methods of analysis, and my program produces an accuracy of 64.379% and 63.397% for training and testing data, respectively, which is over 30% worse than that of 0-1, as seen in **Figure 1**. The result of the training data analysis can be seen in **Figure 2** below. The image is similar for the testing data, with no unique findings. In the interest of space, it is not included, but can be seen and produced from the python code.

```
Training Data Accuracy Rate: 64.37926581811475
```

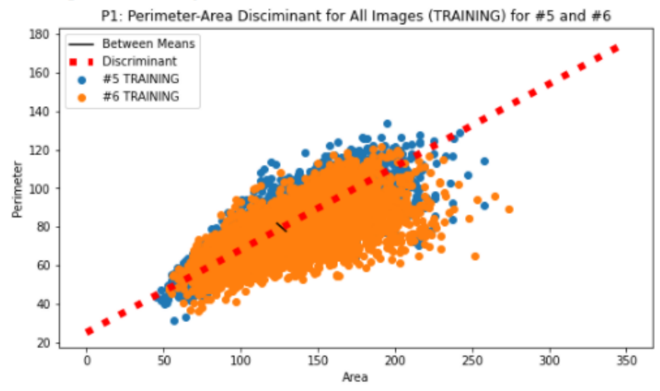


Figure 2: A graphical depiction of the training feature data for handwritten 5s and 6s, as well as the discriminant. Visually, the discriminant does not do a great job of separating the two classes, nor does it have a space to, showing that area and perimeter are not ideal for classifying the two digits.

In regards to the discriminant's accuracy of ~64% for both training and testing data, if there were two different classes with equal circular variances and equal means with a sufficient amount of data points, a decision boundary drawn through that shared mean would likely have an accuracy of (or close to) 50% — with half of one class on one side and half of one class on the other — purely due to symmetry.

Question 1: Main Points

In summary, this question provides a great example of how some features may be incredibly useful in some classification problems, while those same features may not be useful at separating classes in other problems. This puts separating classes using feature data and Fisher discriminants in clear contrast to neural networks, which are significantly more adaptable and versatile, as seen in the following section.

Question 2: 0-1 & Logistic Regression

In contrast to classification using explicitly chosen feature data, the construction of a neural network allows us to develop a classifier for any two classes, given sufficient training data, a good testing dataset for validation, enough time and materials for training, and good computational design. Following the provided codebase, a neural network was designed with sigmoid, propagation, gradient descent, and a prediction function for validation. See the attached code for more information, and refer to the **Recommendations and Improvement** section for some design notes.

The 0-1 classifier was trained on a training dataset of 12,665 images with 784 pixels each, corresponding to a neural network with 784 neurons in its first layer and 784 weights — one for each neuron. A bias term is also included. After initializing the weights of the neurons and the bias term, the total cost and derivative of that total cost with respect to each weight and bias term is found — these are known as the gradients.

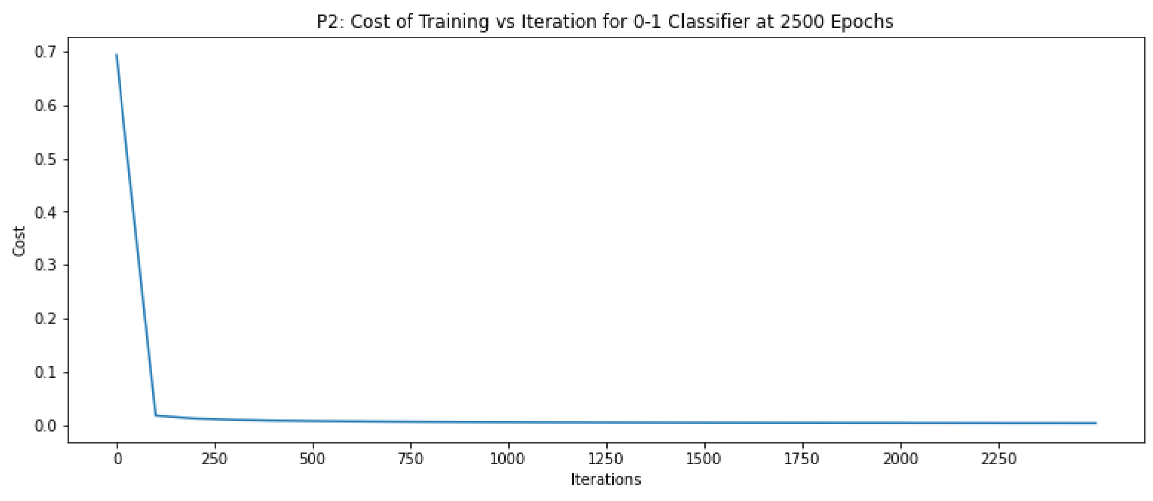
The weights and bias terms are then modified by subtracting the gradients multiplied by some scalar learning rate to give new values for the weights and bias term. A new total cost is calculated — less than the prior total cost, unless an error has occurred — the gradients are figured from this, the weights and bias term are altered again, the process can continue so long as the programmer or user sees fit. It can be stopped when the total cost plateaus, reaches zero, or if the user encounters time or computing constraints. This designed neural network does take considerable time to run, as expanded upon below.

The neural network for classifying handwritten 0s and 1s was trained on the appropriate training data over 2500 epochs (whole cycles through the data) with a learning rate of 0.0003. The training took between 15 and 20 minutes on Google Colabs, and it resulted in a highly effective (if not efficient) classifier for both the training and testing data, with a final cost metric of 0.003671. It reports an accuracy of 99.708% on the training data and 99.716% on testing data. Accuracy (as inversely correlated to cost) improves with time of training, as seen in **Figure 3**. While these are both higher than either accuracy rate for the feature data classifier for 0s and 1s, all figures were above 99% and are considered to be very accurate. Further refer to **Recommendations and Improvement** for a note on these predictions and the low learning rate.

Question 2: 5-6 & Logistic Regression

______The results of the neural network for classifying handwritten 5s and 6s shows significant improvement over the method based on area and perimeter. It was trained identically to the 0-1 classifier — on 2500 epochs with a learning rate of 0.00003 on the 5-6 training data — and also took approximately 15 to 20 minutes. Its results were only modestly less than that of the 0-1 classifier, with a final cost metric of 0.056096 and prediction accuracy rates of 92.380% and 92.703% for training and testing data, respectively. However, compared to its Fisher discriminant feature data classifier, these accuracy rates are above 25% higher for each, which is a very significant improvement.

Though the above training metric of 2500 epochs was designed to make the training design equal to that of the 0-1 classifier, an alternative run for 5000 epochs was trialed for the 5-6 classifier, with a runtime approaching 30 minutes. In this, the final cost metric returned was 0.048375, and the accuracy rates increased slightly to 93.721% for the training data and 93.676% on the testing data. The graph of this is included in **Figure 3**, which helps to illustrate how a doubling in epochs and computation time (up to above 30 minutes) only increased the accuracy by ~1%.



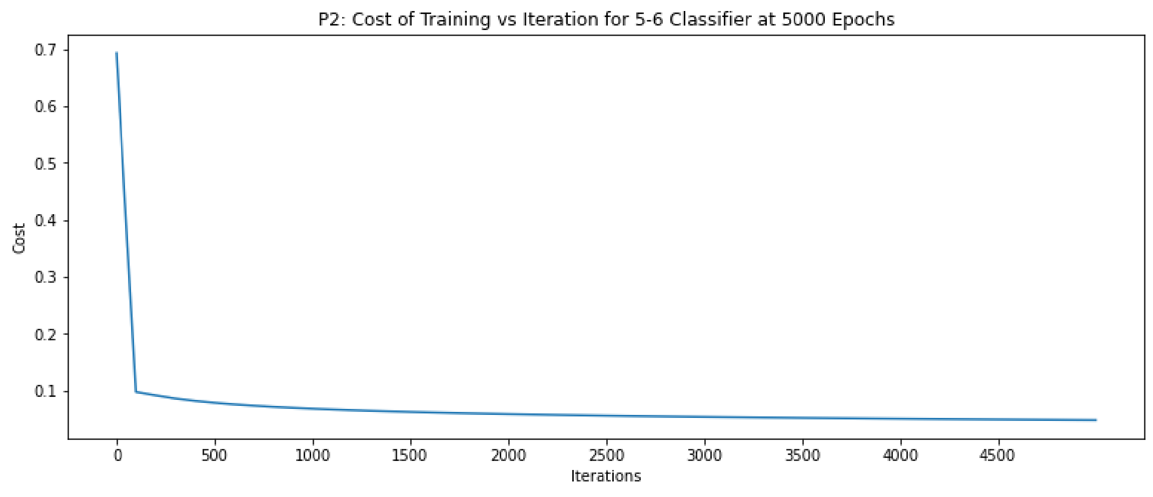


Figure 3: This image shows the sharp plateau of cost as iterations increase, which is conserved for both the 0-1 and 5-6 classifiers. Increasing on the plateau from 2500 to 5000 epochs only resulted in a 1% increase in accuracy for the 5-6 classifier. However, enough computational resources (time and computing power or money) can overcome this plateau and continue iterating.

<u>Question 2: Main Points</u>

It is significant that both 0-1 and 5-6 classifiers reported above 90% accuracy for both training and testing data, which shows the classifiers could potentially be used on future datasets. Further, as mentioned previously, the 5-6 classifier's performance here is greatly improved over the performance of the classifier based on area and perimeter feature data. The fact that the same neural network methods were used in constructing the effective 0-1 and 5-6 classifiers, while the Fisher discriminant-based method only yielded one good classifier, speaks a great deal to the adaptability and flexibility of neural network based methods.

IV. Conclusion

In conclusion, as discussed previously in this report, the use of neural networks methods — utilizing gradient descent and back-propogation — was extremely effective in producing two highly accurate (above 90% accuracy for both training and testing datasets) classifiers for both 0-1 and 5-6 digit pairs. Accuracy of neural networks is a function of resources (time and computing power), though increases in accuracy suffer after a plateau is reached. In contrast to this performance, the linear classifier using Fisher's discriminant methods and area and perimeter feature data was highly effective for a 0-1 classifier (above 99% accuracy for both training and testing datasets) though it was less than 70% accurate for developing a 5-6 classifier for the same feature pair. While both can be highly effective if used correctly, neural network methods have the advantage over Fisher's discriminant methods of being more adaptable and versatile, though having the disadvantage of being more computationally demanding, in terms of both time and resources. The "best" for each situation depends on the context.

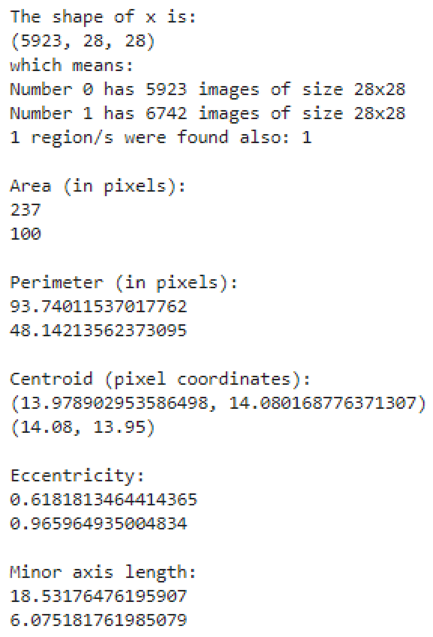
V. Recommendations and Improvement

Preface: This section may not be whole or presented as clearly as the remainder of this report. Rather, it may be a less formal list of future improvements or areas of concern about the current state of the project.

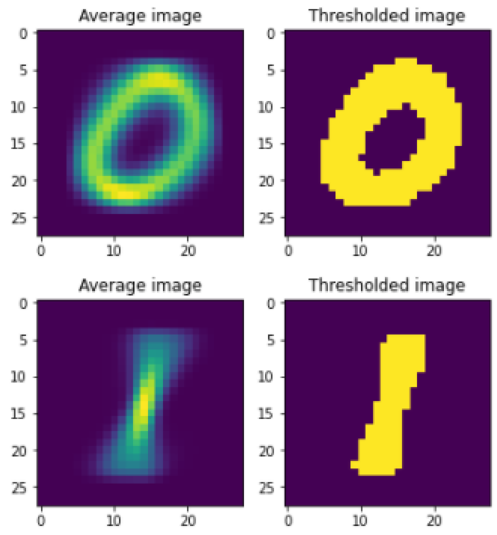
- The current learning rate is abnormally low and results in slow learning. This is due to inefficiencies in my code's propagation section. When my learning rate is not extremely small, these inefficiencies and poor design decisions result in my code taking too large values as either arguments for exp() or the result of exp() with those arguments being too large for Python to fit in 128 bits, resulting in RuntimeWarning values and overflow errors. This results in my code not being able to properly function, and with a loss value stuck at 8.58, if I recall correctly. This is fixed with small learning rates, but the resulting slow rate is unfortunate.
- My prediction equation is overly conservation. While sigmoid functions with gradient back-propogation normally result in values converging to either 0 or 1, with little middle ground, I placed my cut-off for values that would have been predicted to be 1 as 0.95, with everything below that being considered a 0. A more liberal classification algorithm may have only put a cutoff at 0.50, which may have ensured close to a 100% success rate, but I wanted my algorithm to be more selective. However, in this, I see where risk function could be effective. In my decision, I assumed that the function itself should assume that the value is 0 unless it is clearly proven to be 1 — fitting if there was a low cost for incorrectly classifying a 1 as a 0 but a high cost for incorrectly classifying a 0 as a 1. Would some potential uses prefer the default to be 1, unless it is proven to be 0?

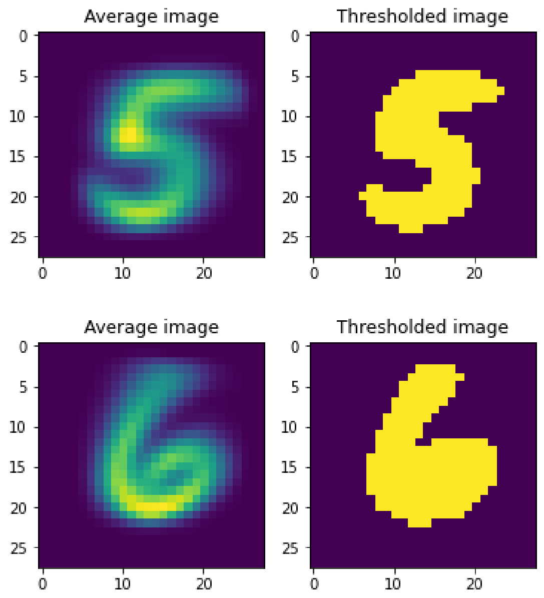
- Future analyses could use PCA-like methods for the sets of feature data to find which “best explain” differences between different data sets by way of dimensional reduction down to two dimensions, some x1 and x2, which could then be used to develop a classifier. Would eccentricity or minor axis length have better classified 5s and 6s than area and perimeter?

VI. Supplemental Figures



```
The shape of x is:
(5923, 28, 28)
which means:
Number 0 has 5923 images of size 28x28
Number 1 has 6742 images of size 28x28
1 region/s were found also: 1

Area (in pixels):
237
100

Perimeter (in pixels):
93.74011537017762
48.14213562373095

Centroid (pixel coordinates):
(13.978902953586498, 14.080168776371307)
(14.08, 13.95)

Eccentricity:
0.6181813464414365
0.965964935004834

Minor axis length:
18.53176476195907
6.075181761985079
```





S.Figure 1: Thresholded images for 0, 1, 5, and 6, as well as feature data statistics for 0 and 1.

Daniel Marten - Project 1 - ECE 566 - Python Notebook & Code

NOTE (and/or warning): the code below was used to make sense of the project and to create the figures, by no means is it meant to be the focus or is it particularly presentable While focus was put into making the report itself presentable and clear, this code is unfortunately anything but Whole sections are repeated for 0-1 and 5-6, instead of being made functions Also, PLEASE run the code from the top as some terminology (x_train) is shared between methods

Resources

Want even more memory and disk space?

Upgrade to Colab Pro+

Python 3 Google Compute Engine backend

Showing resources since 7:59 PM

RAM

Disk

```
# importing
import numpy as np
import matplotlib.pyplot as plt
from keras.datasets import mnist
from skimage import measure

import warnings
warnings.filterwarnings("ignore") # Added this at the end



# PROBLEM 1: SEPARATING CLASSES USING FEATURES

# All Setup:

(x_train, y_train), (x_test, y_test) = mnist.load_data()
number0 = 0
number1 = 1
x0 = x_train[y_train==0,:,:]
x1 = x_train[y_train==1,:,:]
print('The shape of x is:')
print(x0.shape)
print('which means:')
print('Number '+str(number0)+' has '+str(x0.shape[0])+' ir
print('Number '+str(number1)+' has '+str(x1.shape[0])+' ir
# Plotting the average 0 and 1

m = np.mean(x0, axis=0) # IMPORTANT: indexes in python sta

plt.figure()
plt.subplot(1,2,1)
plt.imshow(m)
plt.title('Average image')

mt = 1*(m > 60) # Thresholding
```

```
    # x0 = mt
    plt.subplot(1,2,2)
    plt.imshow(mt)
    plt.title('Thresholded image')

    n = np.mean(x1, axis=0) # IMPORTANT: indexes in python sta

    plt.figure()
    plt.subplot(1,2,1)
    plt.imshow(n)
    plt.title('Average image')

    nt = 1*(n > 60) # Thresholding
    # x1 = nt
    plt.subplot(1,2,2)
    plt.imshow(nt)
    plt.title('Thresholded image')

    # Capturing features using skimage measure

    mt_props = measure.regionprops(mt)
    nt_props = measure.regionprops(nt)
    num_regions_m = len(mt_props)
    num_regions_n = len(nt_props)

    print(str(num_regions_m)+' region/s were found' + ' also:
    print('')

    print('Area (in pixels):')
    area = mt_props[0].area # Remember, index 0 is the first
    area_n = nt_props[0].area # Remember, index 0 is the first
    print(area)
    print(area_n)
    print('')

    print('Perimeter (in pixels):')
    perimeter = mt_props[0].perimeter
    perimeter_n = nt_props[0].perimeter
    print(perimeter)
    print(perimeter_n)
    print('')

    print('Centroid (pixel coordinates):')
    centroid = mt_props[0].centroid
    centroid_n = nt_props[0].centroid
    print(centroid)
    print(str(centroid_n) + '\n')

    print('Eccentricity:')
    eccentricity = mt_props[0].eccentricity
    eccentricity_n = nt_props[0].eccentricity
    print(eccentricity)
```

```
print(eccentricity_n)
print('')

print('Minor axis length:')
minor_axis = mt_props[0].minor_axis_length
minor_axis_n = nt_props[0].minor_axis_length
print(minor_axis)
print(minor_axis_n)
print('')
```

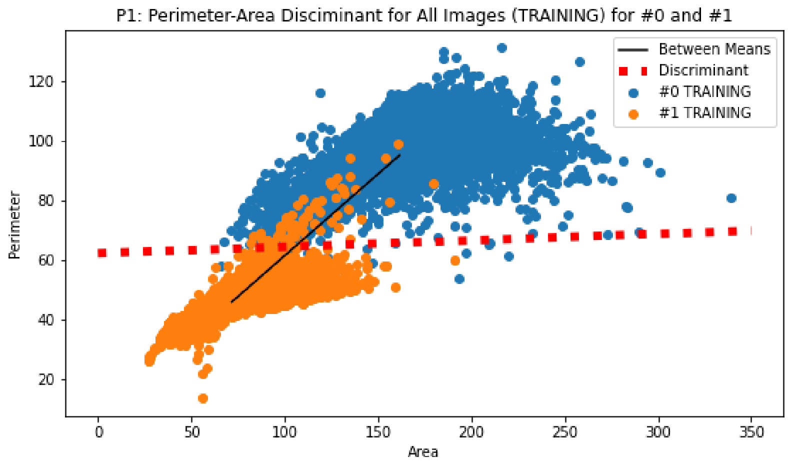
```
The shape of x is:
(5923, 28, 28)
which means:
Number 0 has 5923 images of size 28x28 and in testir
Number 1 has 6742 images of size 28x28 and in testir
1 region/s were found also: 1


237
```

```
# Separating 0 and 1 JUST based on perimeter

t0 = 1*(x0 > 60) # from x0 and x1 as established above
t1 = 1*(x1 > 60)

print('in testing, 0s: ' + str(t0.shape[0]) + ' and 1s: '

# Region properties - from code written previously
area0 = np.zeros(t0.shape[0])
perimeter0 = np.zeros(t0.shape[0])
for i in range(0,t0.shape[0]):
  props = measure.regionprops(t0[i,:,:])
  area0[i] = props[0].area
  perimeter0[i] = props[0].perimeter

area1 = np.zeros(t1.shape[0])
perimeter1 = np.zeros(t1.shape[0])
for i in range(0,t1.shape[0]):
  props = measure.regionprops(t1[i,:,:])
  area1[i] = props[0].area
  perimeter1[i] = props[0].perimeter

a0mean = np.mean(area0)
a1mean = np.mean(area1)
p0mean = np.mean(perimeter0)
p1mean = np.mean(perimeter1)

# print(a0mean)

cov0apnp = np.cov(area0,y=perimeter0)
#print(cov0apnp)

cov0 = cov0apnp
cov1 = np.cov(area1,y=perimeter1)

#covWithin = cov0 + cov1
#covBetween = (a0mean - a1mean)*(p0mean - p1mean)
#print(covBetween)
covAvg = (cov0 + cov1)*0.5
print('covavg: ' + str(covAvg)) # ASSUMPTION THAT COVARIAN

swMinus = np.linalg.inv(covAvg)
```

```
print('swminus: ' + str(swMinus))

v = np.matmul(swMinus,[[a0mean-a1mean],[p0mean-p1mean]]) #
# this and prior work from links:
# https://sthalles.github.io/fisher-linear-discriminant/
# https://www.csd.uwo.ca/~oveksler/Courses/CS434a_541a/Le
# v as the FISHER VECTOR
# equals the inverse of the covariance of each class (equa
# the vector of the differences of the means
# if we projected our points onto this, we could develop a
# that would show a clear discriminant
# however, I used computation methods based off of
# lines perpindicular to this vector

print('v: ' + str(v))

slope = v[1]/v[0]

print('slope: ' + str(slope))

# MAKE THE DISCRIMINANT PERPINDICULAR TO THIS SLOPE
# if we draw the fisher projection starting at the y-axis,
# y =

# yint = 4570
# xint = 4570/45

# py1 = 100
# px1 = (100-4570)/(-45)
# px2 = (4570/45)

# sy = [y for y in range(40,80)]
# y = -45x + 4570
# perp: 1/45
perpSlope = -1/slope

# print('p1 mean: ' + str(p1mean))

total = len(area0)*2
# the total number of points
wrongMat = []
# erroneously named, contains data for the rate of how cor
maxRate = 0
minsy = 0;

# sy as y-intercept of discriminant
# previously did sy in range (20,80) , but narrowed it dow
for sy in range(600,650):
  # print(sy)
  wrongVal = 0
  sy = sy/10 # go by the tenths, ease computational stress
```

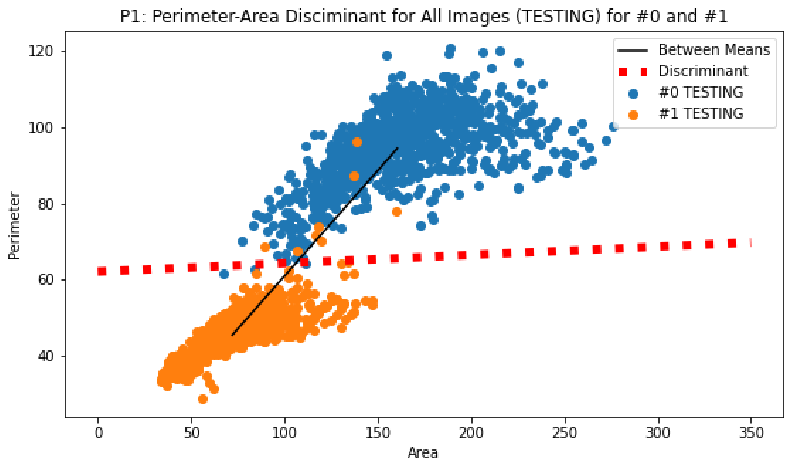
```
    for sx in range(0,len(area0)):
      a0val = area0[sx]*perpSlope + sy # decision boundary 
      a1val = area1[sx]*perpSlope + sy
      b0val = perimeter0[sx] # the actual perimeter values
      b1val = perimeter1[sx]
      if (a0val < b0val):
        wrongVal += 1 # need to count wrong classifications
      if (a1val > b1val):
        wrongVal += 1
        # print(str(aval) + ' , ' + str(bval))
    rate = (1-(1-wrongVal/total)) * 100 # the correctness r
    if rate > maxRate:
      maxRate = rate # finds the maximum rate of correctnes
      minsy = sy
    wrongMat.append(rate)

  # rule is: y=perpSlopex+minsy

  print('Training Data Accuracy Rate: ' + str(maxRate))
  # print(minsy)
  miny0 = minsy
  miny1 = 350*perpSlope + minsy
  # print(miny1)

  minx0 = 0
  minx1 = 350

  # we want to PLOT this rule and discriminant from (0,y0) 
  # this is arbitrary and just based on what would look app


  plt.figure(figsize=(20,5))
  plt.subplot(1,2,1)
  plt.scatter(area0,perimeter0, label='#0 TRAINING')
  plt.scatter(area1,perimeter1, label='#1 TRAINING')
  plt.title('P1: Perimeter-Area Disciminant for All Images 
  plt.plot([a0mean,a1mean],[p0mean,p1mean],color='black',la
  # plt.plot([[px1,px2],[100,0]])
  plt.plot([minx0,minx1],[miny0,miny1],color='red',linewidt
  plt.xlabel('Area')
  plt.ylabel('Perimeter')
  plt.legend()
  plt.show()
  # plt.legend()

  #plt.subplot(1,2,2)
  #plt.scatter(area0[0:100],perimeter0[0:100], label='Numbe
  #plt.scatter(area1[0:100],perimeter1[0:100], label='Numbe
  #plt.plot([minx0,minx1],[miny0,miny1])
  #plt.title('100 images of each class')
  #plt.legend()
```

```
in testing, 0s: 5923 and 1s: 6742
covavg: [[746.05839252 148.65485257]
 [148.65485257  75.78383948]]
swminus: [[ 0.0022004  -0.00431623]
 [-0.00431623  0.02166199]]
v: [[-0.01477226]
 [ 0.67928135]]
slope: [-45.98358637]
Training Data Accuracy Rate: 99.19804153300691
```



```
# How does the SAME discriminant work for the TESTING data
x0test = x_test[y_test==0,:,:]
x1test = x_test[y_test==1,:,:]

t0t = 1*(x0test > 60) # from x0 and x1 as established abov
t1t = 1*(x1test > 60)

print('in testing, 0s: ' + str(t0t.shape[0]) + ' and 1s:


# Region properties - from code written previously
area0t = np.zeros(t0t.shape[0])
perimeter0t = np.zeros(t0t.shape[0])
for i in range(0,t0t.shape[0]):
  props = measure.regionprops(t0t[i,:,:])
  area0t[i] = props[0].area
  perimeter0t[i] = props[0].perimeter

area1t = np.zeros(t1t.shape[0])
perimeter1t = np.zeros(t1t.shape[0])
for i in range(0,t1t.shape[0]):
  props = measure.regionprops(t1t[i,:,:])
  area1t[i] = props[0].area
  perimeter1t[i] = props[0].perimeter
a0tmean = np.mean(area0t)
a1tmean = np.mean(area1t)
p0tmean = np.mean(perimeter0t)
p1tmean = np.mean(perimeter1t)

minsyt = minsy
pslopet = perpSlope

totalt = len(area0t)*2
# the total number of points
wrongMatt = []
wrongVal = 0
# erroneously named, contains data for the rate of how co
maxRatet = 0

# sy as y-intercept of discriminant
# previously did sy in range (20,80) , but narrowed it do
for sx in range(0,len(area0t)):
  a0val = area0t[sx]*pslopet + minsyt # decision boundary
  # print(sx)
  a1val = area1t[sx]*pslopet + minsyt
  b0val = perimeter0t[sx] # the actual perimeter values
  b1val = perimeter1t[sx]
  if (a0val < b0val):
    wrongVal += 1 # need to count wrong classifications f
  if (a1val > b1val):
    wrongVal += 1
    # print(str(aval) + ' , ' + str(bval))
rate = (1-(1-wrongVal/totalt)) * 100 # the correctness ra

print('Testing Data Accuracy Rate: ' + str(rate))

plt.figure(figsize=(20,5))
plt.subplot(1,2,1)
plt.scatter(area0t,perimeter0t, label='#0 TESTING')
plt.scatter(area1t,perimeter1t, label='#1 TESTING')
plt.title('P1: Perimeter-Area Discriminant for All Images 
plt.plot([a0tmean,a1tmean],[p0tmean,p1tmean],color='black
# plt.plot([[px1,px2],[100,0]])
plt.plot([minx0,minx1],[miny0,miny1],color='red',linewidt
plt.xlabel('Area')
plt.ylabel('Perimeter')
plt.legend()
plt.show()
```

```
in testing, 0s: 980 and 1s: 1135
Testing Data Accuracy Rate: 99.38775510204081
```

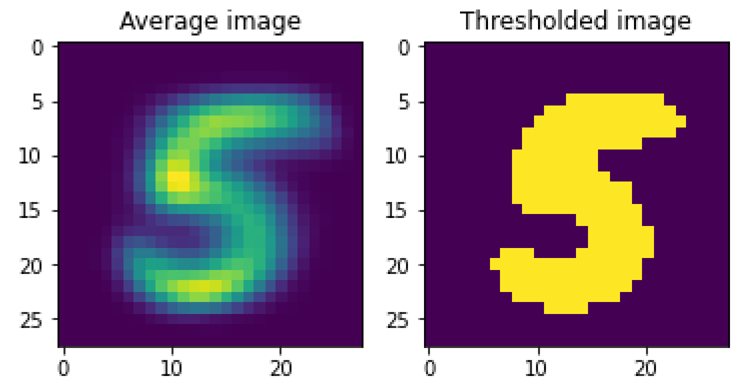


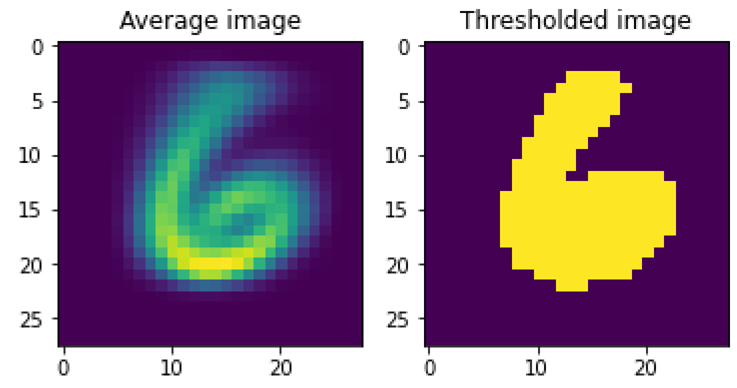
```
# Separating 5 and 6 JUST based on perimeter

# WARNING: reuses the same code and variable names from al
# except when loading, '5' is in '0's place
# and '6' is in '1's place

x5 = x_test[y_test==5,:,:]
x6 = x_test[y_test==6,:,:]

t0 = 1*(x5 > 60) # from x0 and x1 as established above
t1 = 1*(x6 > 60)

print('The shape of x is:')
print(x5.shape)
print('which means:')
print('Number '+str(5)+' has '+str(x5.shape[0])+' images 
print('Number '+str(6)+' has '+str(x6.shape[0])+' images 
# Plotting the average 0 and 1

m = np.mean(x5, axis=0) # IMPORTANT: indexes in python sta

plt.figure()
plt.subplot(1,2,1)
plt.imshow(m)
plt.title('Average image')

mt = 1*(m > 60) # Thresholding
# x0 = mt
plt.subplot(1,2,2)
plt.imshow(mt)
plt.title('Thresholded image')

n = np.mean(x6, axis=0) # IMPORTANT: indexes in python sta
```

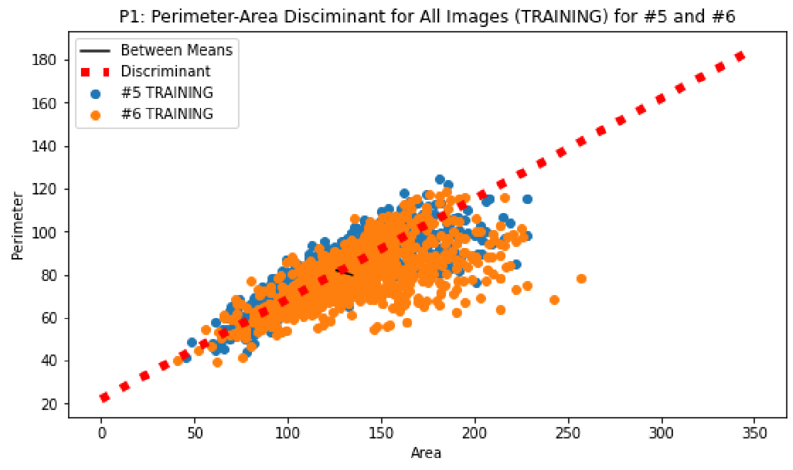
```
plt.figure()
plt.subplot(1,2,1)
plt.imshow(n)
plt.title('Average image')

nt = 1*(n > 60) # Thresholding
# x1 = nt
plt.subplot(1,2,2)
plt.imshow(nt)
plt.title('Thresholded image')

mt_props = measure.regionprops(t0)
nt_props = measure.regionprops(t1)
num_regions_m = len(mt_props)
num_regions_n = len(nt_props)



# Region properties - from code written previously
area0 = np.zeros(t0.shape[0])
perimeter0 = np.zeros(t0.shape[0])
for i in range(0,t0.shape[0]):
  props = measure.regionprops(t0[i,:,:])
  area0[i] = props[0].area
  perimeter0[i] = props[0].perimeter

area1 = np.zeros(t1.shape[0])
perimeter1 = np.zeros(t1.shape[0])
for i in range(0,t1.shape[0]):
  props = measure.regionprops(t1[i,:,:])
  area1[i] = props[0].area
  perimeter1[i] = props[0].perimeter

a0mean = np.mean(area0)
a1mean = np.mean(area1)
p0mean = np.mean(perimeter0)
p1mean = np.mean(perimeter1)

# print(a0mean)

cov0apnp = np.cov(area0,y=perimeter0)
#print(cov0apnp)

cov0 = cov0apnp
cov1 = np.cov(area1,y=perimeter1)

#covWithin = cov0 + cov1
#covBetween = (a0mean - a1mean)*(p0mean - p1mean)
#print(covBetween)
covAvg = (cov0 + cov1)*0.5
print('covavg: ' + str(covAvg)) # ASSUMPTION THAT COVARIAN
```

```
  print('covAvg: ' + str(covAvg)) # ASSUMPTION THAT COVARIAN

  swMinus = np.linalg.inv(covAvg)

  print('swminus: ' + str(swMinus))

  v = np.matmul(swMinus,[[a0mean-a1mean],[p0mean-p1mean]]) #
  # this and prior work from links:
  # https://sthalles.github.io/fisher-linear-discriminant/
  # https://www.csd.uwo.ca/~oveksler/Courses/CS434a_541a/Lec

  print('v: ' + str(v))

  slope = v[1]/v[0]

  print('slope: ' + str(slope))

  # MAKE THE DISCRIMINANT PERPINDICULAR TO THIS SLOPE
  # if we draw the fisher projection starting at the y-axis,
  # y =

  # yint = 4570
  # xint = 4570/45

  # py1 = 100
  # px1 = (100-4570)/(-45)
  # px2 = (4570/45)

  # sy = [y for y in range(40,80)]
  # y = -45x + 4570
  # perp: 1/45
  perpSlope = -1/slope

  # print('p1 mean: ' + str(p1mean))

  total = len(area0)*2
  # the total number of points
  wrongMat = []
  # erroneously named, contains data for the rate of how cor
  maxRate = 0
  minsy = 0;

  # sy as y-intercept of discriminant
  # previously did sy in range (20,80) , but narrowed it dow
  for sy in range(-20,80):
    # print(sy)
    wrongVal = 0
    # sy = sy/10 # go by the tenths, ease computational stre
    for sx in range(0,len(area0)):
      a0val = area0[sx]*perpSlope + sy # decision boundary b
      a1val = area1[sx]*perpSlope + sy
      b0val = perimeter0[sx] # the actual perimeter values
      b1val = perimeter1[sx]
      if (a0val < b0val):
        wrongVal += 1 # need to count wrong classifications
      if (a1val > b1val):
        wrongVal += 1
        # print(str(aval) + ' , ' + str(bval))
    rate = (1-(1-wrongVal/total)) * 100 # the correctness ra
    if rate > maxRate:
      maxRate = rate # finds the maximum rate of correctness
      minsy = sy
    wrongMat.append(rate)

  print('Training Data Accuracy Rate: ' + str(maxRate))
  # print(minsy)
  miny0 = minsy
  miny1 = 350*perpSlope + minsy
  # print(miny1)

  minx0 = 0
  minx1 = 350

  # we want to PLOT this rule and discriminant from (0,y0) t
  # this is arbitrary and just based on what would look appr


  plt.figure(figsize=(20,5))
  plt.subplot(1,2,1)
  plt.scatter(area0,perimeter0, label='#5 TRAINING')
  plt.scatter(area1,perimeter1, label='#6 TRAINING')
  plt.title('P1: Perimeter-Area Disciminant for All Images (
  plt.plot([a0mean,a1mean],[p0mean,p1mean],color='black',lab
  # plt.plot([[px1,px2],[100,0]])
  plt.plot([minx0,minx1],[miny0,miny1],color='red',linewidth
  plt.xlabel('Area')
  plt.ylabel('Perimeter')
  plt.legend()
  plt.show()

  #plt.subplot(1,2,2)
  #plt.scatter(area0[0:100],perimeter0[0:100], label='Number
  #plt.scatter(area1[0:100],perimeter1[0:100], label='Number
  #plt.plot([minx0,minx1],[miny0,miny1])
  #plt.title('100 images of each class')
  #plt.legend()

  # by putting it right throught the middle, you're probably
  # 50% right above and 50% right below
  # nice to see it slightly overperform, but NOT a good clas
  # This is more extensively reported on in the report
```

```
The shape of x is:
(892, 28, 28)
which means:
Number 5 has 892 images of size 28x28
Number 6 has 958 images of size 28x28 and in testing
covavg: [[988.01709197 321.05275398]
 [321.05275398 186.56115565]]
swminus: [[ 0.00229612 -0.00395138]
 [-0.00395138  0.0121601 ]]
v: [[-0.02841511]
 [ 0.06096171]]
slope: [-2.14539789]
Training Data Accuracy Rate: 63.62107623318386
```



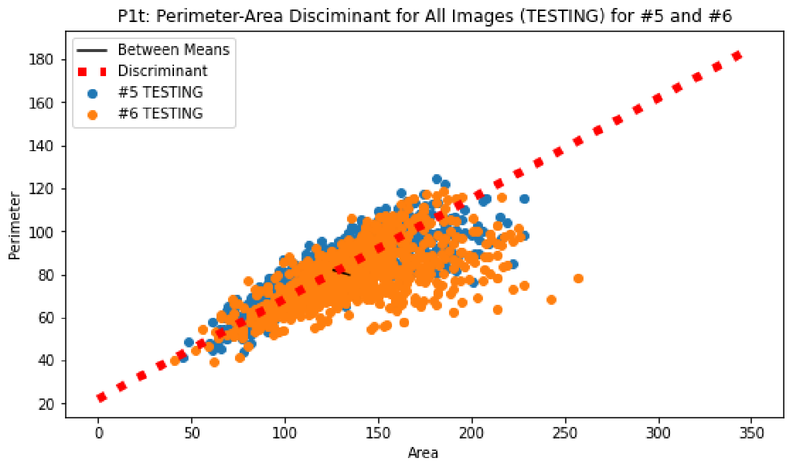




```
# How does the SAME discriminant work for the TESTING data
# for 5 and 6
x0test = x_test[y_test==5,:,:]
x1test = x_test[y_test==6,:,:]

t0t = 1*(x0test > 60) # from x0 and x1 as established abov
t1t = 1*(x1test > 60)

# Region properties - from code written previously
area0t = np.zeros(t0t.shape[0])
perimeter0t = np.zeros(t0t.shape[0])
for i in range(0,t0t.shape[0]):
  props = measure.regionprops(t0t[i,:,:])
  area0t[i] = props[0].area
  perimeter0t[i] = props[0].perimeter

area1t = np.zeros(t1t.shape[0])
perimeter1t = np.zeros(t1t.shape[0])
for i in range(0,t1t.shape[0]):
  props = measure.regionprops(t1t[i,:,:])
  area1t[i] = props[0].area
  perimeter1t[i] = props[0].perimeter

a0tmean = np.mean(area0t)
a1tmean = np.mean(area1t)
p0tmean = np.mean(perimeter0t)
p1tmean = np.mean(perimeter1t)

minsyt = minsy
pslopet = perpSlope

totalt = len(area0t)*2
# the total number of points
wrongMatt = []
wrongVal = 0
# erroneously named, contains data for the rate of how cor
maxRatet = 0

# sy as y-intercept of discriminant
# previously did sy in range (20,80) , but narrowed it dow
for sx in range(0,len(area0t)):
  a0val = area0t[sx]*pslopet + minsyt # decision boundary
  # print(sx)
  a1val = area1t[sx]*pslopet + minsyt
  b0val = perimeter0t[sx] # the actual perimeter values
  b1val = perimeter1t[sx]
  if (a0val < b0val):
    wrongVal += 1 # need to count wrong classifications fo
  if (a1val > b1val):
    wrongVal += 1
    # print(str(aval) + ' , ' + str(bval))
rate = (1-(1-wrongVal/totalt)) * 100 # the correctness rat
```

```
print('rate on testing, not training: ' + str(rate))

plt.figure(figsize=(20,5))
plt.subplot(1,2,1)
plt.scatter(area0t,perimeter0t, label='#5 TESTING')
plt.scatter(area1t,perimeter1t, label='#6 TESTING')
plt.title('P1t: Perimeter-Area Discinimant for All Images
plt.plot([a0tmean,a1tmean],[p0tmean,p1tmean],color='black
# plt.plot([[px1,px2],[100,0]])
plt.plot([minx0,minx1],[miny0,miny1],color='red',linewidth
plt.xlabel('Area')
plt.ylabel('Perimeter')
plt.legend()
```

```
rate on testing, not training: 63.62107623318386
<matplotlib.legend.Legend at 0x7f8bec56fdd0>
```



```
# PROBLEM 2: Developing a Neural Network to do the same

# Everything already imported

# Defining Sigmoid

def sigmoid(z):
    # try:
      # print("we do a sigmoi!")
    s = 1/(1 + np.exp(-z))
    # except:
    #  print("Houston, wir haben eine Problem!")

    return s
```

```
# Initializing our weights

def initialize_weights(dim):

    b = 0
    w = np.zeros((dim,1))

    assert(w.shape == (dim, 1))
    assert(isinstance(b, float) or isinstance(b, int))

    return w, b



# Propogation
# Used to have a VERY extensive amount of print statement

def propagate(w, b, X, Y):
    """
    Implement the cost function and its gradient for the p

    Arguments:
    w -- weights, a numpy array of size (num_px * num_px,
    b -- bias, a scalar
    X -- data of size (number of examples, num_px * num_p
    Y -- true "label" vector of size (1, number of exampl

    Return:
    cost -- negative log-likelihood cost for logistic reg
    dw -- gradient of the loss with respect to w, thus sa
    db -- gradient of the loss with respect to b, thus sa

    """

    m = X.shape[0] # this is the number of examples

    alpha = [] # as the PIECEWISE products of w and x for

    alphaVal = 0 # I forget, do we need to declare variab

    for row in range(m):
       alphaVal = float(np.dot(X[row],w)[0]) # arrays DON
       alpha.append(alphaVal)

    zi = []
    yhat = []
    zi = [aa + b for aa in alpha]

    yhat = [sigmoid(bb) for bb in zi]

    loss = []
    dldz = []

    for i in range(m):
      yhatVal = yhat[i]
      if (yhatVal == 1.00):
        yhatVal = 0.9999999
      elif (yhatVal == 0.00):
        yhatVal = 1-0.9999999

      lossVal = -Y[i] * np.log(yhatVal) - (1-Y[i]) * np.lo

      loss.append(lossVal) # getting value for loss and ap
      zVal = zi[i]
      dldzVal = -Y[i] * (1/(1+np.exp(zVal))) - (1-Y[i]) *

      dldz.append(dldzVal)

    jVal = 1/m * sum(loss) # total loss/cost

    grad = np.zeros((m,6)) # matrix for values going forwa
    gradBack = np.zeros((m,6)) # matrix for gradient going

    # #print('dldz check: ' + str(dldz))
    dldz = np.matrix(dldz)
    db = np.mean(dldz)

    # alpha = np.asarray(alpha)
    alpha = np.matrix(alpha)

    #print('a shape: ' + str(alpha.shape))

    xMat = np.matrix(X)
    at = np.transpose(alpha)
    wt = np.transpose(np.matrix(w))
    xt = np.transpose(np.matrix(xMat))
    dldzt = np.transpose(dldz)

    #print('dw multiplication check: ' + str(xt.shape) +
    dw = np.matmul(xt,dldzt)
    #print('dw check: ' + str(dw.shape))
    cost = jVal
    dw = np.asarray(dw)
    w = np.matrix(w)
    #print('asarray w shape: ' + str(w.shape))


    assert(dw.shape == w.shape)

    assert(db.dtype == float)
    cost = np.squeeze(cost)
    assert(cost.shape == ())
    grads = {"dw": dw,
             "db": db}

    return grads, cost, yhat



# Defining Gradient Descent
# Also used to have some "try" and "except" statements, nc

def gradient_descent(win, bin, Xin, Yin, num_iterations, l
    """
    This function optimizes w and b by running a gradient

    Arguments:
    w -- weights, a numpy array of size (num_px * num_px,
    b -- bias, a scalar
    X -- data of shape (num_px * num_px, number of example
    Y -- true "label" vector of shape (1, number of exampl
    num_iterations -- number of iterations of the optimiza
    learning_rate -- learning rate of the gradient descent

    Returns:
    params -- dictionary containing the weights w and bias
    grads -- dictionary containing the gradients of the we
    costs -- list of all the costs computed during the opt

    Tips:
    You basically need to write down two steps and iterate
        1) Calculate the cost and the gradient for the cur
        2) Update the parameters using gradient descent ru
    """

    costs = []

    for i in range(num_iterations):
        grads, cost, lastYhat = propagate(win, bin, Xin, Y

        dw = grads["dw"]
        db = grads["db"]

        win = win - (learning_rate * dw)

        bin += -1 * learning_rate * db

        if (i % 10 == 0):
          print("Cost after iteration %i: %f" % (i,cost))

        if (i % 100 == 0 or i == num_iterations-1):
            costs.append(cost)
            # print ("Cost after iteration %i: %f" % (i, 

    params = {"w": win,
              "b": bin}

    grads = {"dw": dw,
             "db": db}
    # addition of 'lastYhat' return is NEWQ, so as to make

    return params, grads, costs, lastYhat


# Defining the 'predict' function

def predict(w, b, X):
    '''
    # is this not just the final yHat value we get?
    # OH, using our outputs from above
    # but without Y
    Predict whether the label is 0 or 1 using learned log

    Arguments:
    w -- weights, a numpy array of size (num_px * num_px,
    b -- bias, a scalar
    X -- data of size (num_px * num_px, number of example

    Returns:
    Y_prediction -- a numpy array (vector) containing all
    '''

    m = X.shape[0]
    Y_prediction = np.zeros((1, m))
    w = w.reshape(X.shape[1], 1)

    A = sigmoid(np.dot(w.T, X.T) + b)
    print(A)
    error = 0

    for i in range(A.shape[1]):
      if A[0][i] > 0.95:
        Y_prediction[0][i] = 1
        # WHAT WE WOULD LIKE TO KNOW: is it better to cla
        # different costs?
      elif A[0][i] < 0.1:
        Y_prediction[0][i] = 0
      else:
        # print("Not confident enough! Keep training! val
        Y_prediction[0][i] = 0 # as default
        error+=1

    psuedoErrorRate = error/m
    perPer = psuedoErrorRate*100
    print("Problems: " + str(error))
    print("Problem rate as a percent: " + str(perPer))

    assert(Y_prediction.shape == (1, m))

    return Y_prediction



# Loading, Re-Shaping, and Initial Training

# LOAD DATA
class0 = 0
class1 = 1

(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train = x_train[np.isin(y_train,[class0,class1]),:,:]
y_train = 1*(y_train[np.isin(y_train,[class0,class1])]>cla
x_test = x_test[np.isin(y_test,[class0,class1]),:,:]
y_test = 1*(y_test[np.isin(y_test,[class0,class1])]>class0


# RESHAPE

x_train_flat = x_train.reshape(x_train.shape[0],-1)
print(x_train_flat.shape)
print('Train: '+str(x_train_flat.shape[0])+' images and '

x_test_flat = x_test.reshape(x_test.shape[0],-1)
print(x_test_flat.shape)
print('Test: '+str(x_test_flat.shape[0])+' images and '+s

# STRANDARIZE
x_train_flat = x_train_flat / 255
x_test_flat = x_test_flat / 255

# Initialize parameters with zeros (≈ 1 line of code)
wStart, bStart = initialize_weights(x_train_flat.shape[1]

# Gradient descent (≈ 1 line of code)
# learning_rate = 0.005
# num_iterations = 2000
# parameters, grads, costs = gradient_descent(w, b, x_tra

# w = 0.5*np.ones(784)

# a = propagate(w,0.5,x_train_flat,y_train)
```

```
(12665, 784)
Train: 12665 images and 784 neurons
```

```
(2115, 784)
Test: 2115 images and 784 neurons
```

```
# Loading, Re-Shaping, and Initial Training
# BUT for 5 and 6

# LOAD DATA
class5 = 5
class6 = 6

(x_train56, y_train56), (x_test56, y_test56) = mnist.load_
x_train56 = x_train56[np.isin(y_train56,[class5,class6]),
y_train56 = 1*(y_train56[np.isin(y_train56,[class5,class6]
x_test56 = x_test56[np.isin(y_test56,[class5,class6]),:,:]
y_test56 = 1*(y_test56[np.isin(y_test56,[class5,class6])]


# RESHAPE

x_train_flat56 = x_train56.reshape(x_train56.shape[0],-1)
print(x_train_flat56.shape)
print('Train: '+str(x_train_flat56.shape[0])+' images and

x_test_flat56 = x_test56.reshape(x_test56.shape[0],-1)
print(x_test_flat56.shape)
print('Test: '+str(x_test_flat56.shape[0])+' images and '

# STRANDARIZE
x_train_flat56 = x_train_flat56 / 255
x_test_flat56 = x_test_flat56 / 255

# Initialize parameters with zeros (≈ 1 line of code)
wStart56, bStart56 = initialize_weights(x_train_flat56.sha

# Gradient descent (≈ 1 line of code)
# learning_rate = 0.005
# num_iterations = 2000
# parameters, grads, costs = gradient_descent(w, b, x_tra

# w = 0.5*np.ones(784)

# a = propagate(w,0.5,x_train_flat,y_train)
```

```
(11339, 784)
Train: 11339 images and 784 neurons

(1850, 784)
Test: 1850 images and 784 neurons
```

```
# MOMENT OF TRUTH: TRAINING AND TESTING

# Retrieve parameters w and b from dictionary "parameters'
# These are supposed to be the final w and b after gradier
epoch = 2500
stepSize = 0.00001
stepSizeTest = stepSize*3 # IT WORKS
# this is VERY small, because my code is prone to overflov
# epoch of 5000 takes 30min to reach ~0.0614 cost

# param5,grad5,costs5,finalyhat5 = gradient_descent(wStar
# x5s stored and established, as epoch=5000, stepSize = 0

# setting up param10 as epoch=10,000 and stepSize = 0.000
#

param25,grad25,costs25,finalyhat25 = gradient_descent(wSta
```

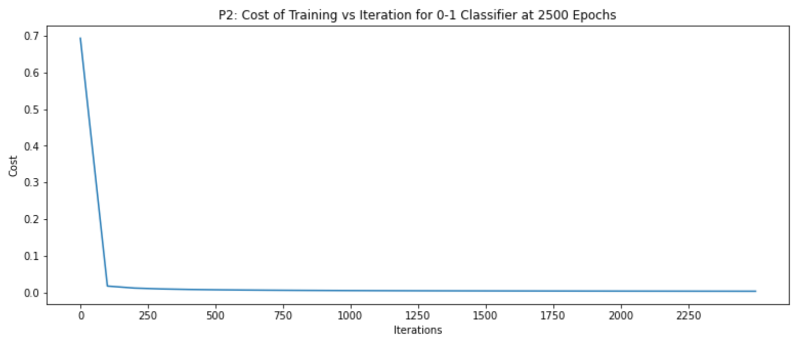
```
Cost after iteration 0: 0.693147
Cost after iteration 100: 0.017764
Cost after iteration 200: 0.012273
Cost after iteration 300: 0.009892
Cost after iteration 400: 0.008521
Cost after iteration 500: 0.007612
Cost after iteration 600: 0.006958
Cost after iteration 700: 0.006459
Cost after iteration 800: 0.006063
Cost after iteration 900: 0.005739
Cost after iteration 1000: 0.005468
Cost after iteration 1100: 0.005236
Cost after iteration 1200: 0.005036
Cost after iteration 1300: 0.004860
Cost after iteration 1400: 0.004704
Cost after iteration 1500: 0.004564
Cost after iteration 1600: 0.004438
Cost after iteration 1700: 0.004323
Cost after iteration 1800: 0.004218
Cost after iteration 1900: 0.004122
Cost after iteration 2000: 0.004033
Cost after iteration 2100: 0.003950
Cost after iteration 2200: 0.003873
Cost after iteration 2300: 0.003801
Cost after iteration 2400: 0.003734
Cost after iteration 2499: 0.003671
```

```
# Same analysis but between 5 and 6

# MOMENT OF TRUTH: TRAINING AND TESTING

# Retrieve parameters w and b from dictionary "parameters'
# These are supposed to be the final w and b after gradie
epoch = 5000
stepSize = 0.00001
stepSizeTest = stepSize*3 # IT WORKS
# this is VERY small, because my code is prone to overflo
# epoch of 5000 takes 30min to reach ~0.0614 cost

# param5,grad5,costs5,finalyhat5 = gradient_descent(wStar
# x5s stored and established, as epoch=5000, stepSize = 0

# setting up param10 as epoch=10,000 and stepSize = 0.000
#

param25561,grad25561,costs25561,finalyhat25561 = gradient_
```

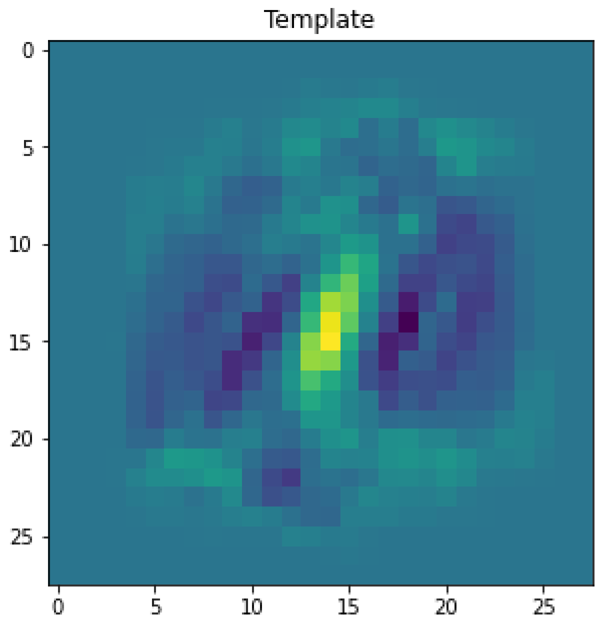
```
    Cost after iteration 3830: 0.051166
    Cost after iteration 3840: 0.051138
    Cost after iteration 3850: 0.051110
    Cost after iteration 3860: 0.051082
    Cost after iteration 3870: 0.051054
    Cost after iteration 3880: 0.051026
    Cost after iteration 3890: 0.050998
    Cost after iteration 3900: 0.050970
    Cost after iteration 3910: 0.050942
    Cost after iteration 3920: 0.050914
    Cost after iteration 3930: 0.050887
    Cost after iteration 3940: 0.050859
    Cost after iteration 3950: 0.050832
    Cost after iteration 3960: 0.050805
    Cost after iteration 3970: 0.050777
    Cost after iteration 3980: 0.050750
    Cost after iteration 3990: 0.050723
    Cost after iteration 4000: 0.050696
    Cost after iteration 4010: 0.050669
    Cost after iteration 4020: 0.050642
    Cost after iteration 4030: 0.050616
    Cost after iteration 4040: 0.050589
    Cost after iteration 4050: 0.050563
    Cost after iteration 4060: 0.050536
    Cost after iteration 4070: 0.050510
    Cost after iteration 4080: 0.050483
    Cost after iteration 4090: 0.050457
    Cost after iteration 4100: 0.050431
    Cost after iteration 4110: 0.050405
    Cost after iteration 4120: 0.050379
    Cost after iteration 4130: 0.050353
    Cost after iteration 4140: 0.050327
    Cost after iteration 4150: 0.050301
    Cost after iteration 4160: 0.050276
    Cost after iteration 4170: 0.050250
    Cost after iteration 4180: 0.050224
    Cost after iteration 4190: 0.050199
    Cost after iteration 4200: 0.050174
    Cost after iteration 4210: 0.050148
    Cost after iteration 4220: 0.050123
    Cost after iteration 4230: 0.050098
    Cost after iteration 4240: 0.050073
    Cost after iteration 4250: 0.050048
    Cost after iteration 4260: 0.050023
    Cost after iteration 4270: 0.049998
    Cost after iteration 4280: 0.049973
    Cost after iteration 4290: 0.049948
    Cost after iteration 4300: 0.049924
    Cost after iteration 4310: 0.049899
    Cost after iteration 4320: 0.049875
    Cost after iteration 4330: 0.049850
    Cost after iteration 4340: 0.049826
    Cost after iteration 4350: 0.049801
    Cost after iteration 4360: 0.049777
    Cost after iteration 4370: 0.049753
    Cost after iteration 4380: 0.049729
    Cost after iteration 4390: 0.049705
    Cost after iteration 4400: 0.049681
    Cost after iteration 4410: 0.049657
```

```
# Continuing after testing,

w = param25["w"]
# print(w)
b = param25["b"]




# Predict test/train set examples (≈ 2 lines of code)
y_prediction_test = predict(w, b, x_test_flat)
y_prediction_train = predict(w, b, x_train_flat)

# Print train/test Errors
print('')
print("train accuracy: {} %".format(100 - np.mean(np.abs(y
print("test accuracy: {} %".format(100 - np.mean(np.abs(y_
print('')

# 99.7% ACCURACY FOR BOTH TRAINING AND TESTING DATA LET'S

plt.figure(figsize=(13,5))
plt.plot(range(0,2600,100),costs25)
plt.title('P2: Cost of Training vs Iteration for 0-1 Class
# P1t: Perimeter-Area Discriminant for All Images (TESTING)
plt.xlabel('Iterations')
plt.ylabel('Cost')
plt.xticks(range(0,2500,250))
plt.show()


plt.figure(figsize=(13,5))
plt.imshow(w.reshape(28,28))
plt.title('Template')
```

```
[[9.98828913e-01 1.51994504e-04 9.99755322e-01 ... 9
  3.52141160e-04 9.99913352e-01]]
Problems: 9
Problem rate as a percent: 0.425531914893617
[[4.53133252e-04 9.99754619e-01 9.99994377e-01 ... 9
  1.36672125e-03 9.98182429e-01]]
Problems: 73
Problem rate as a percent: 0.5763916304776944

train accuracy: 99.70785629688118 %
test accuracy: 99.71631205673759 %
```

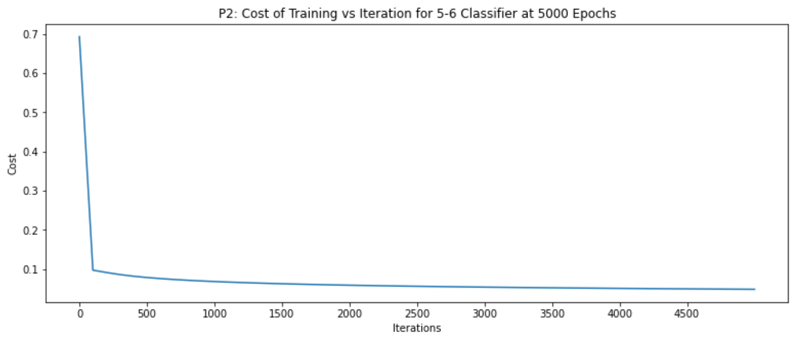


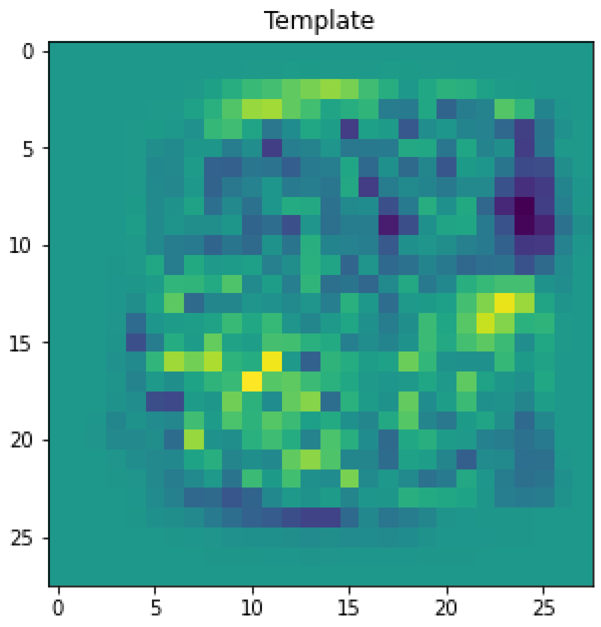
```
Text(0.5, 1.0, 'Template')
```



```
# Continuing after testing,
w561 = param25561["w"]
# print(w)
b561 = param25561["b"]



# Predict test/train set examples (≈ 2 lines of code)
y_prediction_test561 = predict(w561, b561, x_test_flat56)
y_prediction_train561 = predict(w561, b561, x_train_flat5

# Print train/test Errors
print('')
print("train accuracy: {} %".format(100 - np.mean(np.abs(y
print("test accuracy: {} %".format(100 - np.mean(np.abs(y_
print('')

plt.figure(figsize=(13,5))
plt.plot(range(0,5100,100),costs25561)
plt.title('P2: Cost of Training vs Iteration for 5-6 Clas
plt.xlabel('Iterations')
plt.ylabel('Cost')
plt.xticks(range(0,5000,500))


plt.figure(figsize=(13,5))
plt.imshow(w561.reshape(28,28))
plt.title('Template')
```

```
[[9.74274523e-01 9.86385586e-01 9.24443723e-05 ... 9
  5.00419607e-04 9.99960619e-01]]
Problems: 185
Problem rate as a percent: 10.0
[[3.96813680e-05 2.52667424e-02 9.94990552e-01 ... 6
  9.48937404e-07 9.93730648e-01]]
Problems: 1321
Problem rate as a percent: 11.650057324279036

train accuracy: 93.72078666549078 %
test accuracy: 93.67567567567568 %

Text(0.5, 1.0, 'Template')
```





Manage sessions

Change runtime type

✓ 0s completed at 11:22 PM