Illinois Institute of Technology

Machine and Deep Learning - ECE 566

Project 1

Ishan Loganathan Reddy

CWID : A20472779

Date: 8th November 2021
Professor : Dr. Brankov, Jovan

Fisher Discriminant

Introduction:

Fisher's linear discriminant is a classification method that projects high-dimensional data onto a line and performs classification in this one-dimensional space. The projection maximizes the distance between the means of the two classes while minimizing the variance within each class. Fisher Discriminant is a supervised learning algorithm, which is also known as the linear dimensionality reduction method. FLD is used in binary classification, we can find an optimal threshold t and classify the data accordingly. In this problem set, we have set the threshold value t to be -60. Here the numpy library imported from python is used for all the mathematical calculations.
Fisher’s formula in brief:

$$J(\vec{w}) = \frac{\vec{w}^T S_B \vec{w}}{\vec{w}^T S_W \vec{w}}$$

Fisher's Linear Discriminant method uses the idea of dimension reduction causing information loss, by adjusting the weight vector ω for projection to minimize the amount of overlapping data in reduced dimension.

Algorithm:
Suppose we have two classes and D dimensional samples with x1,x2,x3........xn, where n1 is the samples came from the first class and n2 is the sample that came from the second class. To measure the separation between the projection of different classes we need to find the mean of the respective classes. If larger the separation of the mean between the two classes the good the result. Therefore Fisher Linear Discriminant will maximize the function that will give a large separation between the given class means, also which gives the small variance within the class. By this, we can come to the conclusion that it maximize the class overlap. Fisher Linear Discriminant maintains two properties the first is a small variance within each of the classes and the second is a large variance within the class datasets.

The maximization function which gives the Eigenvalue is given my the below equation,

$$S_B W_i = \lambda_i S_W W_i$$

Conclusion:

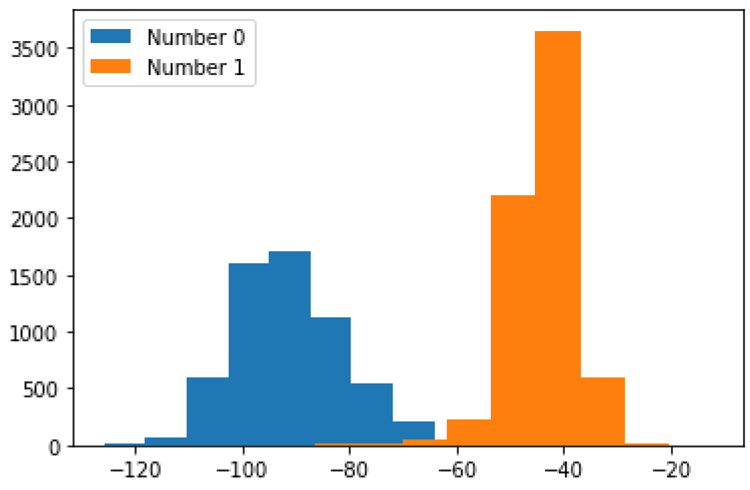
In the given problem the mean of the two-class is given by the below figure, when the handwritten value is 0 separated from 1 and the required SW and SB was calculated from the above equation. Numpy library from python helps in the mathematical computation.

The outputs were as follows:
```
Mean of Class 0:
 [161.60864427  95.04012192]
Mean of Class 1:
 [71.65114209 45.75754109]
Total Mean Vector:
 [[161.60864427  95.04012192]
 [ 71.65114209  45.75754109]]

Within class Matrix:
 SW [[9118674.32361935 1833914.22071991]
 [1833914.22071991  928636.43601096]]
Between class Matrix:
 SB [[25622410.14484183 14037056.03685114]
 [14037056.03685114  7690101.79244851]]
```

The eigenvalues are calculated from the above equation and the maximum eigenvalue is taken from the eigenvector. With the eigenvector the weight matrix was also calculated. The outputs are as follows:

```
Eigenvectors
[[-0.48046566  0.02404997]
 [ 0.87701354 -0.99971076]]

Eigenvalues
[1.77635684e-15 8.31233799e+00]

Eigenvalues in decreasing order:

8.312337988279975
1.7763568394002505e-15

Mat W:
 [[ 0.02404997]
 [-0.99971076]]
```

The histogram plot gives the distribution of the two handwritten images 0 and 1 and the threshold was set between the common points as -60 and the decision boundary was calculated with the thresholded value. The balanced accuracy was also calculated from the sensitivity and the specificity. The training accuracy was calculated to be nearly 99 percentage and the testing accuracy was found to be nearly 99.4 percentage as the testing accuracy was more the model performs well during the testing datasets which were given in the coed appendix. The same logic is implemented for 5 from 6 but the balanced accuracy was comparatively less. Therefore the model performs well in the handwritten 0 from 1. The parameter is considered to be the area and the perimeter. Suppose if we consider the other features the model may perform well as the accuracy can also be increased.



```
The number of 0s in training sample:  5923
True Negative TN :  5903
False Positive FP :  20

The number of 1s in training sample:  6742
True Positive TP :  6650
False Negative FN :  92
Accuracy :  99.11567311488353
The Balanced accuracy:  99.14887651690213


The number of 0s in test sample:  980
True Negative TN :  978
False Positive FP:  2

The number of 1s in test sample:  1135
True Positive TP :  1125
False Negative FN :  10
The Balanced accuracy:  99.45743054931225
```

Logistic Regression:

Logistic regression is a classification algorithm used to assign observations to a discrete set of classes. Unlike linear regression which outputs continuous number values, logistic regression transforms its output using the logistic sigmoid function to return a probability value which can then be mapped to two or more discrete classes. Logistic Regression could help us predict whether the student passed or failed. Logistic regression predictions are discrete (only specific values or categories are allowed). We can also view probability scores underlying the model's classifications. Logistic regression is the appropriate regression analysis to conduct when the dependent variable is dichotomous (binary). Like all regression analyses, logistic regression is a predictive analysis. Logistic regression is used to describe data and to explain the relationship between one dependent binary variable and one or more nominal, ordinal, interval, or ratio-level independent variables.
In this project we will be focusing upon the binary classification problem using the MNIST dataset from the first section of the project. We use 80% of the images as training data set and the rest are considered as the testing data set.

Sigmoid Function:
A sigmoid activation function is used because the value should be between 0 and 1, here sigmoid is used to predict the probabilities.In order to map for the true or false class a threshold is set to separate all the values and taken to the specific class. In this problem statement the threshold was set to 0.5. The value about 0.5 belongs to true value which is one and the value below 0.5 is considered to be false which is 0.

Algorithm:
The MNIST data set has images, each image has a pixel density of 28 * 28 which gets converted into 784 vector values which feed to the neural network. In this neural network there is no hidden layer to be considered. As the algorithm proceeds the weights are initialized to zero and the predicted Y vector is calculated. Using the Y vector and the true Y the cost function can be calculated. The given equation below calculates the cost function.

$$J = \frac{1}{m}\sum_{i=1}^{m} L\left(\hat{y}^{(i)}, y^{(i)}\right)$$

Where the loss function or error function will be:

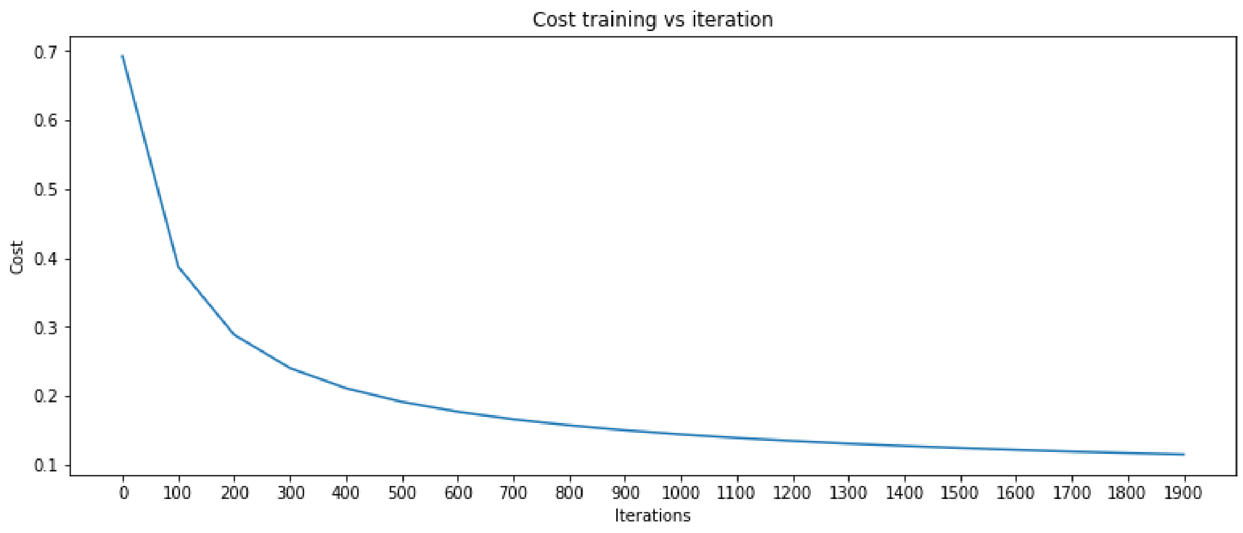
$$L\left(\hat{y}^{(i)}, y^{(i)}\right) = -y^{(i)}\, log\left(\hat{y}^{(i)}\right) - \left(1 - y^{(i)}\right) log\left(1 - \hat{y}^{(i)}\right)$$

Conclusion:

Initially, the cost function will be high, to minimize the cost Gradient descent algorithm was implemented. The change was only due to the bias and the weights, therefore the respective dw and db were calculated using the Gradient descent algorithm. This process is known as backward propagation. Calculating the error function or loss function is known as Forward propagation. The iteration was set to be 2000 and the learning rate was set to be 0.005 which was given by the initial code. By the experimental value, we can see the cost function will be decreasing steeply as the number of iterations increases.

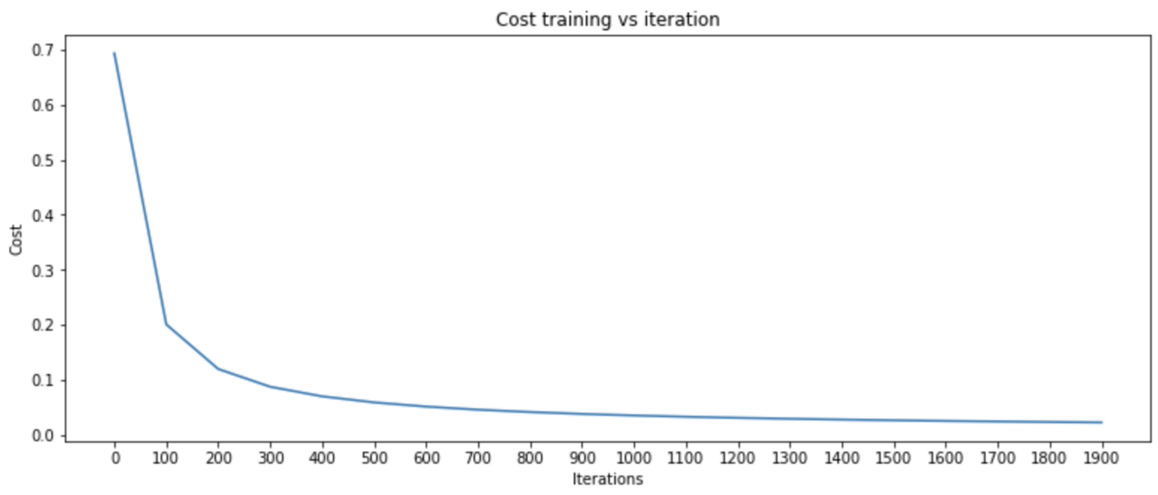
Shown below is the graph for differenciating handwritten 5's from 6's , From the above figure we can comes to a conclusion that the number of iterations increases the cost value or the error value was minimized.The training accuracy was found to be 96 percent and the testing accuracy was found to be 97 percent. This shows that the model performa well for the new datasets. The balanced accuracy was calculated to be 97.22 percent.

```
Balanced Accuracy = 0.9722621399215481
```



The graph shown below is for differentiating the handwritten 0 from 1, From the above figure we can come to a conclusion that the number of iterations increases the cost value or the error value was minimized. The training and the testing accuracy were found to be 99.73943939992104 % and 99.90543735224587 %. Here the testing accuracy was more when compared to the training accuracy therefore the model performs well for the test data.

```
Balanced Accuracy = 0.9990543735224587
```



From the following experiment we can say the following things:

- This algorithm performs well in differentiating the handwritten 0 from 1
- The balanced accuracy was approximately 1.5 times higher than the handwritten 5 from 6.
- Fisher Discriminant can perform well only in the small datasets where the neural networks can apply for large datasets having more neurons and thus more hidden layers.

APPENDIX:

Fisher Discriminant:

Load and visualize MNIST da t a

```
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 to show a clean
output with no warnings but not necessary
```

Choose number to visualize (from 0 to 9):

```
(x_train, y_train), (x_test, y_test) = mnist.load_data()

number = 0
x = x_train[y_train==number,:,:]
print('The shape of x is:')
print(x.shape)
print('which means:')
print('Number '+str(number)+' has '+str(x.shape[0])+' images of size
'+str(x.shape[1])+'x'+str(x.shape[2]))
```

Plot average image:

```
m = np.mean(x, axis=0) # IMPORTANT: indexes in python start at "0", not
"1", so the first element of array "a" would be a[0]

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

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

From a thresholded image, we can use the regionprops function from skimage.measure

```
mt_props = measure.regionprops(mt)
num_regions = len(mt_props)
print(str(num_regions)+' region/s were found')
print('')

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

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

print('Centroid (pixel coordinates):')
centroid = mt_props[0].centroid
print(centroid)

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

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

Example: Scatter plot of Area vs Perimeter for all images of numbers "number" and "number+1"

Are "Area" and "Perimeter" good features to classify "number" and "number+1"?

```
x0 = x_train[y_train==number,:,:]
x1 = x_train[y_train==number+1,:,:]
buf0 = "Number %d" % number
buf1 = "Number %d" % (number+1)
# Threshold images
t0 = 1*(x0 > 60)
t1 = 1*(x1 > 60)

# Region properties
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

plt.figure(figsize=(20,5))
plt.subplot(1,2,1)
plt.scatter(area0,perimeter0, label=buf0)
plt.scatter(area1,perimeter1, label=buf1)
plt.title('All images for both classes')
plt.legend()

plt.subplot(1,2,2)
plt.scatter(area0[0:100],perimeter0[0:100], label=buf0)
plt.scatter(area1[0:100],perimeter1[0:100], label=buf1)
plt.title('100 images of each class')
plt.legend()
```

```
# Function to calculate mean vector of features for different classes
def mean_vect(area_x, perimeter_x):
 mean = np.array([np.mean(area_x), np.mean(perimeter_x)])
 return mean


# Mean Vector Array
mean0 = mean_vect(area0, perimeter0)
print("Mean of Class 0:\n", mean0)
mean1 = mean_vect(area1, perimeter1)
print("Mean of Class 1:\n", mean1)

#Mean Array of Class 0 and Class 1 features
mean_vector = np.array([mean0, mean1])
print("Total Mean Vector: \n", mean_vector)

shape_matrix = np.array([t0.shape[0], t1.shape[0]])
area_matrix = np.array([area0,area1])
perimeter_matrix = np.array([perimeter0,perimeter1])
matrix_0 = np.array([area0,perimeter0]).T
matrix_1 = np.array([area1, perimeter1]).T
overall_mean = (mean0+mean1)/2

Sw = np.zeros((2,2))
Sw_mat_0 = np.zeros((2,2))
Sw_mat_1= np.zeros((2,2))
Sb = np.zeros((2,2))
for i in range(t0.shape[0]):
 row, mv =
np.array([matrix_0[i][:2]]).reshape(2,1),mean_vector[0].reshape(2,1)
 Sw_mat_0 += (row-mv).dot((row-mv).T)
for i in range(t1.shape[0]):
 row, mv =
np.array([matrix_1[i][:2]]).reshape(2,1),mean_vector[1].reshape(2,1)
 Sw_mat_1 += (row-mv).dot((row-mv).T)

Sw = Sw_mat_0 + Sw_mat_1
print('Within class Matrix:\n SW', Sw)

for i in range (2):
 n=shape_matrix[i]
 mv=mean_vector[i].reshape(2,1)
 overall_mean = overall_mean.reshape(2,1)
 Sb += n*(overall_mean-mv).dot((overall_mean-mv).T)

print('\n Between class Matrix:\n SB', Sb)

e_vals, e_vecs = np.linalg.eig(np.linalg.inv(Sw).dot(Sb))
print('Eigenvectors \n%s' %e_vecs)
print('\nEigenvalues \n%s' %e_vals)

e_pairs=[(np.abs(e_vals[i]),e_vecs[:,i]) for i in range(len(e_vals))]
e_pairs=sorted(e_pairs,key=lambda k: k[0], reverse=True)
print('\nEigenvalues in decreasing order:\n')
for i in e_pairs:
 print(i[0])

W = np.hstack(e_pairs[0][1].reshape(2,1))

W= W.reshape(2,1)
print('Mat W:\n', W.real)

X_0 =np.dot(np.array([area0 ,perimeter0]).T,W)
X_1 =np.dot(np.array([area1 ,perimeter1]).T,W)
lin0=np.zeros((1,shape_matrix[0]))
lin1=np.ones((1,shape_matrix[1]))
plt.figure()
plt.hist(X_0,label = 'Number 0')
plt.hist(X_1, label = 'Number 1')
plt.legend()


threshold = -60
values0 = 0
a0 = 0
failed0 = {}
for i,x in enumerate(X_0) :
 if x < threshold :
   values0 = values0 + 1
 else :
   failed0[a0] = i
   a0 = a0 + 1
print ("The number of 0s in training sample: ", (X_0.shape[0]))
print ("True Negative TN : ", values0)
print ("False Positive FP : ", a0)

values1 = 0
a1 = 0
failed1 = {}
for i,x in enumerate(X_1) :
 if x > threshold :
   values1 = values1 + 1
 else :
   failed1[a1] = i
   a1 = a1 + 1

print ("\nThe number of 1s in training sample: ", (X_1.shape[0]))
print ("True Positive TP : ", values1)
print ("False Negative FN : ", a1)
print("\n Accuracy : ",((values0+values1)/((values0+values1+a1+a0))*100))

Sensitivity = values1/(a1+values1)
Specificity = values0/(a0+values0)
balanced_accuracy = (Sensitivity + Specificity)/2
print ("\n The Balanced accuracy: ",(balanced_accuracy*100))


x_test_0 = x_test[y_test==0,:,:]
x_test_1 = x_test[y_test==1,:,:]
t0 = 1*(x_test_0 > 60)
t1 = 1*(x_test_1 > 60)
area_test0 = np.zeros(t0.shape[0])
perimeter_test0 = np.zeros(t0.shape[0])
for i in range(0,t0.shape[0]):
 props = measure.regionprops(t0[i,:,:])
 area_test0[i] = props[0].area
 perimeter_test0[i] = props[0].perimeter
area_test1 = np.zeros(t1.shape[0])
perimeter_test1 = np.zeros(t1.shape[0])
for i in range(0,t1.shape[0]):
 props = measure.regionprops(t1[i,:,:])
 area_test1[i] = props[0].area
 perimeter_test1[i] = props[0].perimeter

test_matrix_shape = np.array([t0.shape[0],t1.shape[0]])
X_test_lda_0 = np.dot(np.array([area_test0,perimeter_test0]).T,W)
X_test_lda_1 = np.dot(np.array([area_test1,perimeter_test1]).T,W)
lin0=np.zeros((1,test_matrix_shape[0]))
lin1=np.ones((1,test_matrix_shape[1]))

plt.figure()
plt.hist(X_test_lda_0,label = 'Number 0')
plt.hist(X_test_lda_1, label = 'Number 1')
plt.legend()
plt.show()

print ("The number of 0s in test sample: ", (x_test_0.shape[0]))

threshold = -60
count0 = 0
k0 = 0
failed0 = {}
for i,x in enumerate(X_test_lda_0) :
   if x < threshold :
     count0 = count0 + 1
   else :
     failed0[k0] = i
     k0 = k0 + 1
print ("True Negative TN : ", count0)
print ("False Positive FP: ", k0)
print ("\nThe number of 1s in test sample: " , (x_test_1.shape[0]))
threshold = -60
count1 = 0
k1 = 0
failed1 = {}
for i,x in enumerate(X_test_lda_1) :
 if x > threshold :
   count1 = count1 + 1
 else :
   failed1[k1] = i
```

```
    k1 = k1 + 1
print ("True Positive TP : ", count1)
print ("False Negative FN : ", k1)
Sensitivity = count1/(k1+count1)
Specificity = count0/(k0+count0)
balanced_accuracy = (Sensitivity + Specificity)/2
print ("\n The Balanced accuracy: ", (balanced_accuracy*100))
```

Logistic Regression:

Load packages

```
import numpy as np
import matplotlib.pyplot as plt
from keras.datasets import mnist
```

Define functions

Sigmoid

```
def sigmoid(z):
    """
    Compute the sigmoid of z

    Arguments:
    x -- A scalar or numpy array of any size.

    Return:
    s -- sigmoid(z)
    """

    s=1/(1+np.exp(-z))

    return s
sigmoid(np.array([2,3,4]))
```

Initialize weights

```
def initialize_weights(dim):
   """
   This function creates a vector of zeros of shape (dim, 1) for w and
initializes b to 0.

   Argument:
   dim -- size of the w vector we want (or number of parameters in this
case)

   Returns:
   w -- initialized vector of shape (dim, 1)
   b -- initialized scalar (corresponds to the bias)
   """

   w=np.zeros([dim,1])
   b=0
    assert(w.shape == (dim, 1))
   assert(isinstance(b, float) or isinstance(b, int))

   return w, b
```

Forward and backward propagation

```
def propagate(w, b, X, Y):
   """
   Implement the cost function and its gradient for the propagation
explained in the assignment

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

   Return:
   cost -- negative log-likelihood cost for logistic regression
   dw -- gradient of the loss with respect to w, thus same shape as w
   db -- gradient of the loss with respect to b, thus same shape as b
```

```
    """

    m = X.shape[0]

    # FORWARD PROPAGATION (FROM X TO COST)

    y_pred=sigmoid(np.dot(w.T, X.T) + b)
    cost=-1/m * np.sum( np.dot(np.log(y_pred), Y.T) + np.dot(np.log(1-
    y_pred), (1-Y.T)))

    # BACKWARD PROPAGATION (TO FIND GRAD)

    dw=np.dot(X.T,(y_pred-Y).T)/m
    db=np.sum((y_pred-Y))/m

    assert(dw.shape == w.shape)
    assert(db.dtype == float)
    cost = np.squeeze(cost)
    assert(cost.shape == ())

    grads = {"dw": dw,
             "db": db}

    return grads, cost
```

Gradient descent

```
def gradient_descent(w, b, X, Y, num_iterations, learning_rate):
    """
    This function optimizes w and b by running a gradient descent algorithm

    Arguments:
    w -- weights, a numpy array of size (num_px * num_px, 1)
    b -- bias, a scalar
    X -- data of shape (num_px * num_px, number of examples)
    Y -- true "label" vector of shape (1, number of examples)
    num_iterations -- number of iterations of the optimization loop
    learning_rate -- learning rate of the gradient descent update rule
    Returns:
    params -- dictionary containing the weights w and bias b
    grads -- dictionary containing the gradients of the weights and bias
with respect to the cost function
    costs -- list of all the costs computed during the optimization, this
will be used to plot the learning curve.

    Tips:
    You basically need to write down two steps and iterate through them:
        1) Calculate the cost and the gradient for the current parameters.
Use propagate().
        2) Update the parameters using gradient descent rule for w and b.
    """

    costs = []

    for i in range(num_iterations):


        # Cost and gradient calculation

        grads, cost = propagate(w, b, X, Y)

        # Retrieve derivatives from grads
        dw = grads["dw"]
        db = grads["db"]

        # update rule

        w=w-(dw*learning_rate)
        b=b-(db*learning_rate)
        # Record the costs
        if i % 100 == 0:
            costs.append(cost)
            # Print the cost every 100 training examples
            print ("Cost after iteration %i: %f" % (i, cost))

    params = {"w": w,
              "b": b}
    grads = {"dw": dw,
             "db": db}

    return params, grads, costs
```

Make predictions

```
def predict(w, b, X):
    '''
    Predict whether the label is 0 or 1 using learned logistic regression
parameters (w, b)

    Arguments:
    w -- weights, a numpy array of size (num_px * num_px, 1)
    b -- bias, a scalar
    X -- data of size (num_px * num_px, number of examples)

    Returns:
    Y_prediction -- a numpy array (vector) containing all predictions (0/1)
for the examples in X
    '''

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

    # Compute vector "A" predicting the probabilities of the picture
containing a 1

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


    for i in range(A.shape[1]):
        # Convert probabilities A[0,i] to actual predictions p[0,i]

      if A[0][i]<0.5:
        Y_prediction[0][i]=0
      else:
        Y_prediction[0][i]=1
    assert(Y_prediction.shape == (1, m))

    return Y_prediction
```

Merge functions and run your model

```
# LOAD DATA
class0 = 5
class1 = 6

(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])]>class0)
x_test = x_test[np.isin(y_test,[class0,class1]),:,:]
y_test = 1*(y_test[np.isin(y_test,[class0,class1])]>class0)

x_train.shape[0]

plt.imshow(x_train[6],cmap="gray")

# 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
'+str(x_train_flat.shape[1])+' neurons \n')

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
'+str(x_test_flat.shape[1])+' neurons \n')

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

Train the model (in training set)

```
# Initialize parameters with zeros (≈ 1 line of code)
```

```
w, b = 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_train_flat, y_train,
2000, 0.005)
```

Test the model (in testing set)

```
# Retrieve parameters w and b from dictionary "parameters"
w = parameters["w"]
b = parameters["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_prediction_train - y_train)) * 100))
print("test accuracy: {} %".format(100 - np.mean(np.abs(y_prediction_test
- y_test)) * 100))
print('')

plt.figure(figsize=(13,5))
plt.plot(range(0,2000,100),costs)
plt.title('Cost training vs iteration')
plt.xlabel('Iterations')
plt.ylabel('Cost')
plt.xticks(range(0,2000,100))


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


x_test_flat.shape
```

```
w = parameters["w"]
b = parameters["b"]

def TP(y,y_pred):
 count=0
 for i in range(0,len(y)):
   if y[i]==1 and y_pred[i]==1:
     count=count+1
 return count
def TN(y,y_pred):
 count=0
 for i in range(0,len(y)):
   if y[i]==0 and y_pred[i]==0:
     count=count+1
 return count
def FN(y,y_pred):
 count=0
 for i in range(0,len(y)):
   if y[i]==1 and y_pred[i]==0:
     count=count+1
 return count
def FP(y,y_pred):
 count=0
 for i in range(0,len(y)):
   if y[i]==0 and y_pred[i]==1:
     count=count+1
 return count
```

```
y_prediction_test.shape
```

```
y_test.shape
```

```
y_prediction_test=y_prediction_test.reshape(y_test.shape[0],)
```

```
y_prediction_test=np.array(list(map(int,y_prediction_test)))
```

```
y_prediction_test
```

```
tp=TP(y_test,y_prediction_test)
```

```
tn=TN(y_test,y_prediction_test)
```

```
fn=FN(y_test,y_prediction_test)

fp=FP(y_test,y_prediction_test)

balanced_accuracy=((tp/(tp+fn))+(tn/(tn+fp)))/2
balanced_accuracy
```