EE5841 MACHINE LEARNING

Classification Project 2 - Report

--

Submitted by : Ashwini Nikumbh, Vrushaketu Mali, Ponkrshnan Thiagarajan ¶

Date: 20-Mar-2020

Introduction

This project aims at developing a framework to perform classification on a given dataset. Therefore, this project includes the implementation of Naive-Bayes classifier & Logistic Regression on the MNIST data set. The MNIST database of handwritten digits has a training set of 60,000 examples and a test set of 10,000 examples. It is a subset of a larger set available from NIST. The digits have been size-normalized and centered in a fixed-size image of 28 x 28 pixels.

Import the libraries

Below imported libraries will be used in the various functions developed in the different codes of this project.

In [1]:
```
import numpy as np
import idx2numpy as id
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import math

from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import classification_report
from sklearn.decomposition import PCA
from sklearn import preprocessing
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import confusion_matrix
%matplotlib inline
```

Extracting the dataset

In [2]:
```
train_img_read = open('train-images.idx3-ubyte', 'rb')   # open the f
ile of training images
train_img = id.convert_from_file(train_img_read)

train_lab_read = open('train-labels.idx1-ubyte', 'rb')  # open the fi
le of training labels
train_lab = id.convert_from_file(train_lab_read)

test_img_read = open('t10k-images.idx3-ubyte', 'rb')    # open the fi
le of testing images
test_img = id.convert_from_file(test_img_read)

test_lab_read = open('t10k-labels.idx1-ubyte', 'rb')    # open the fi
le of testing labels
test_lab = id.convert_from_file(test_lab_read)

train=train_img.flatten()
X_train=train.reshape(60000,784)     # reshape the training data as 6
0000x784
test=test_img.flatten()
x_test=test.reshape(10000,784)       # reshape the testing data as 10
000x784
y_train=train_lab[:]
y_test=test_lab[:]

X_train_sort = X_train[np.argsort(y_train,axis=0)]
```

Part 1 - Naive Bayes

Naive Bayes classifier is based on the Bayesian probability theoerem which is used for predictive modeling. This theorem provides the way of calculating the posterior probability P(c|x), from P(c), P(x), and P(x|c).

$$\mathbb{P}(C|X) = \frac{\mathbb{P}(\mathrm{x}|\mathrm{c})\mathbb{P}(\mathrm{c})}{\mathbb{P}(x)}$$

$\mathbb{P}(C|x)$ is the posterior probability of the class.

$\mathbb{P}(C)$ is the prior probability of the class

$\mathbb{P}(x|c)$ is the likelihood which is the probability of predictor given class.

$\mathbb{P}(x)$ is the prior probability of predictor.

In this problem, we have class labels from 0 to 9 (C takes any value from 0 to 9). We have considered the prior probability of all these classes as 0.1. Probability for each feature is modeled a Gaussian distribution. Naive Bayes classifier is trained on the training data and is applied to test data. Below is the function developed for building and applying the Naive Bayes Classifier on the MNIST data set.

1.1 Function for probability of predictor given class and predicting the class of input features

Below code has two functions developed. Function prob_x_c calculates the probability of a feature for every class and function nb_predict returns the predicted class of that feature which takes the results from the previous function. That means, the maximum of probabilities of all classes is calculated and then its corresponding class is assigned to that particular feature.

In [3]:
```
count_dig = [0]
for i in range(10):
    count_dig.append(np.count_nonzero(y_train == i))
indx = np.cumsum(count_dig)

mean_dig = []
std_dig = []
for i in range(10):
    mean_dig= np.append(mean_dig,np.mean(X_train_sort[indx[i]:indx[i+
1]-1,:],axis=0),axis=0)
    std_dig = np.append(std_dig,np.std(X_train_sort[indx[i]:indx[i+1]
-1,:],axis=0),axis=0)
mean_dig = mean_dig.reshape(10,784)
std_dig = std_dig.reshape(10,784)
std_dig = std_dig+10

def prob_x_c(x,mu,sigma):
    p_x_c = 0
    sz=mu.shape
    for i in range(sz[0]):
        #print(p_x_c)
        p_x_c = p_x_c + math.log(math.exp(-0.5*((x[i]-mu[i])/sigma[i
])**2)/sigma[i])
    return(p_x_c)

sz = x_test.shape
def nb_predict(x_test):
    y=[]
    for i in range(sz[0]):
        x = x_test[i,:]
        den = []
        num = []
        for i in range(10):
            den.append(prob_x_c(x,mean_dig[i,:],std_dig[i,:]))
        y.append(np.argmax(den))
    return(y)
```

1.2 Implementing the function on test features and calcuting overall accuracy

The above coded function is then implemented on the test features and the class of all 10000 test features is predicted. It is then compared with the true class of those test features for calculating the overall accuracy of our Naive Bayes function.

In [4]:
```
y=nb_predict(x_test)
accuracy_overall = 1-np.count_nonzero(y-y_test)/sz[0]
print('The overall accuracy of the classifier is',accuracy_overall*10
0,'%')
```

```
The overall accuracy of the classifier is 73.8 %
```

The overall accuracy of the Naive-Bayes classifier on MNIST data set is 73.8 %

1.3 Results and conclusion of part 1

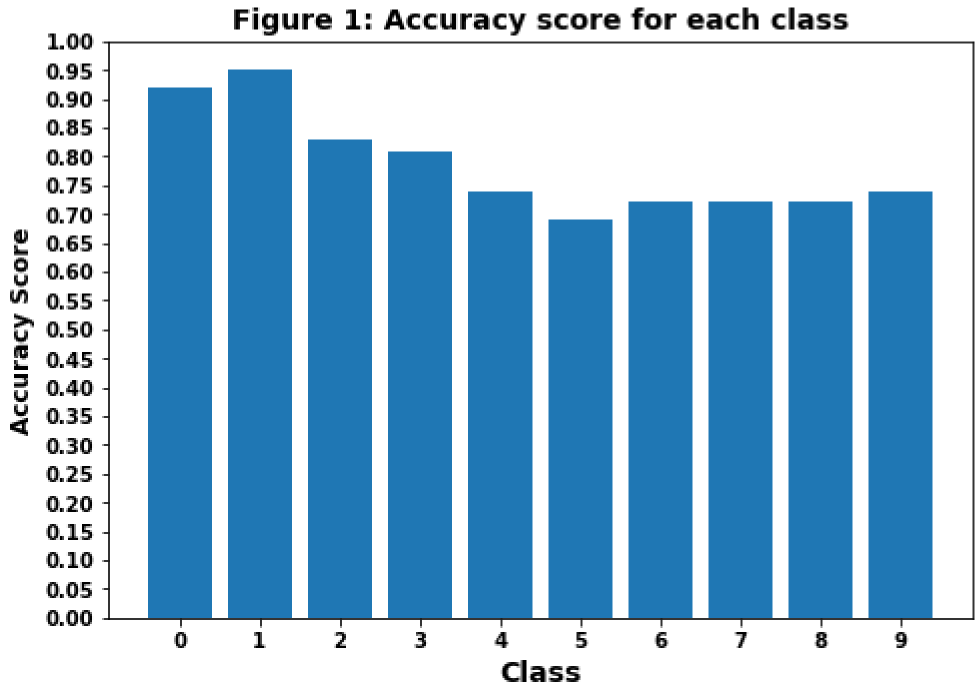
Below codes are developed to plot the confusion matrix, as well as accuracy score for each class.

In [5]:
```
# Calculate the accuracy for each class
accuracy=[]
accuracy_digit=[]
for i in range(10):
    Test_class=y_test
    test_data=pd.DataFrame(Test_class)
    i1=test_data[test_data[0]==i]
    indices=list(i1.index)
    for j in indices:
        acc=((y[j] ==i).sum()/((y_test[j]==i).sum()))
        accuracy.append(acc)
    accuracy_avg = np.mean(accuracy)
    accuracy_digit.append(accuracy_avg)
accuracy_table=pd.DataFrame(accuracy_digit, columns=['Table 1:Accurac
y for each class'])

my = np.array(accuracy_digit).round(2)
fig = plt.figure()
ax = fig.add_axes([0,0,1,1])
labels = ['0','1','2','3','4','5','6','7','8','9',]
values = [my[0],my[1],my[2],my[3],my[4],my[5],my[6],my[7],my[8],my[9
]]
ax.bar(labels,values)
plt.xlabel('Class', fontweight='bold', fontsize='14')
plt.ylabel('Accuracy Score', fontweight='bold', fontsize='12')
plt.xticks(np.arange(0, 10, 1),fontweight='bold', fontsize='10')
plt.yticks(np.arange(0.0,1.05, 0.05),fontweight='bold', fontsize='10'
)
plt.title('Figure 1: Accuracy score for each class', fontweight='bol
d', fontsize='14')
plt.figure(figsize=(20,20))
plt.show()

accuracy_table.round(2)

#plt.figure(figsize=(10,5))
#C = range(0, 10)
#plt.plot(C, accuracy_digit, 'bo')
#plt.xlabel('Class', fontweight='bold', fontsize='14')
#plt.ylabel('Accuracy Score', fontweight='bold', fontsize='14')
#plt.xticks(np.arange(0, 10, 1),fontweight='bold', fontsize='10')
#plt.yticks(np.arange(0.5,1.05, 0.05),fontweight='bold', fontsize='1
0')
#plt.title('Figure 1:Accuracy score for class', fontweight='bold', fo
ntsize='16')
#accuracy_table.round(2)
```



<Figure size 1440x1440 with 0 Axes>

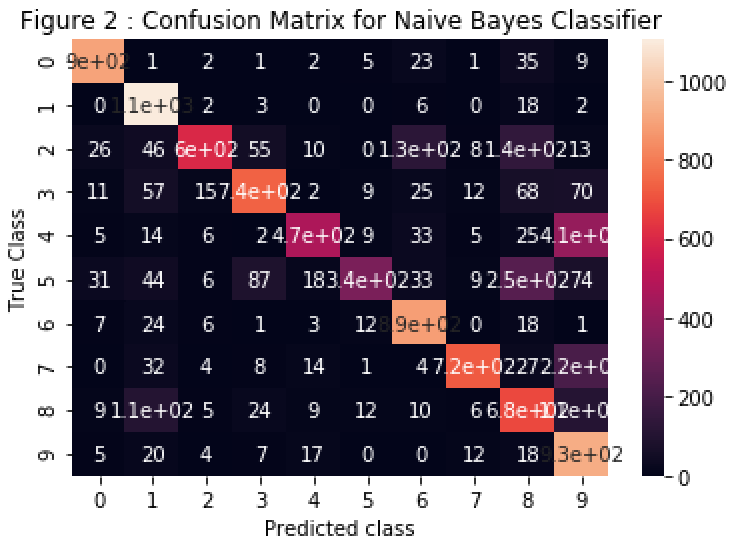
Out[5]:

	Table 1:Accuracy for each class
0	0.92
1	0.95
2	0.83
3	0.81
4	0.74
5	0.69
6	0.72
7	0.72
8	0.72
9	0.74

1.3.1 Comments

The above plot shows the accuracy score for each class. From the figure, it can be inferred that the classifier has least accuracy in correctly predicting the test features with class 5 and most accuracy in correctly predicting the test features with class 1.

In [29]:
```
# Plot the confusion matrix
conf=confusion_matrix(y_test, y)
ax= plt.subplot()
df_cm = pd.DataFrame(conf, index = [i for i in "0123456789"],columns
= [i for i in "0123456789"])
sns.heatmap(df_cm, annot=True, ax = ax); #annot=True to annotate cell
s
ax.set_xlabel('Predicted class');ax.set_ylabel('True Class');
ax.set_title('Figure 2 : Confusion Matrix for Naive Bayes Classifier'
);
ax.xaxis.set_ticklabels(['0','1','2','3','4','5','6','7','8','9']); a
x.yaxis.set_ticklabels(['0','1','2','3','4','5','6','7','8','9']);
```



1.3.2 Comments

The above plot shows the confusion matrix which demonstrates the performance of our classification algorithm. Both axes have class labels however the X-axis has the predicted classes and Y-axis has true classes. The values inside the cells in this matrix show the number of correctly predicted class for any particular true class. From the confusion matrix, it can be inferred that (1,8), (6,2), (8,2), (8,5) are the combinations of the classes where the classifier has less accuracy in identifying the correct label.

In [7]:
```
plt.figure(figsize=(5,10))
plt.subplot(1,2,1)
image = np.array(mean_dig[0][:].reshape(28,28,1)).squeeze()
plt.imshow(image)
plt.title('Fig-1: Mean of digit Zero',fontweight='bold',fontsize='8')
plt.imshow(image)

plt.subplot(1,2,2)
image = np.array(std_dig[0][:].reshape(28,28,1)).squeeze()
plt.imshow(image)
plt.title('Fig-2: Standard deviation of digit Zero',fontweight='bold'
,fontsize='8')
plt.imshow(image)

plt.figure(figsize=(5,10))
plt.subplot(1,2,1)
image = np.array(mean_dig[1][:].reshape(28,28,1)).squeeze()
plt.imshow(image)
plt.title('Fig-1: Mean of digit One',fontweight='bold',fontsize='8')
plt.imshow(image)

plt.subplot(1,2,2)
image = np.array(std_dig[1][:].reshape(28,28,1)).squeeze()
plt.imshow(image)
plt.title('Fig-2: Standard deviation of digit One',fontweight='bold',
fontsize='8')
plt.imshow(image)

plt.figure(figsize=(5,10))
plt.subplot(1,2,1)
image = np.array(mean_dig[2][:].reshape(28,28,1)).squeeze()
plt.imshow(image)
plt.title('Fig-1: Mean of digit Two',fontweight='bold',fontsize='8')
plt.imshow(image)

plt.subplot(1,2,2)
image = np.array(std_dig[2][:].reshape(28,28,1)).squeeze()
plt.imshow(image)
plt.title('Fig-2: Standard deviation of digit Two',fontweight='bold',
fontsize='8')
plt.imshow(image)

plt.figure(figsize=(5,10))
plt.subplot(1,2,1)
image = np.array(mean_dig[3][:].reshape(28,28,1)).squeeze()
plt.imshow(image)
plt.title('Fig-1: Mean of digit Three',fontweight='bold',fontsize='8'
)
plt.imshow(image)

plt.subplot(1,2,2)
image = np.array(std_dig[3][:].reshape(28,28,1)).squeeze()
plt.imshow(image)
plt.title('Fig-2: Standard deviation of digit Three',fontweight='bol
d',fontsize='8')
plt.imshow(image)
```

```
plt.figure(figsize=(5,10))
plt.subplot(1,2,1)
image = np.array(mean_dig[4][:].reshape(28,28,1)).squeeze()
plt.imshow(image)
plt.title('Fig-1: Mean of digit Four',fontweight='bold',fontsize='8')
plt.imshow(image)

plt.subplot(1,2,2)
image = np.array(std_dig[4][:].reshape(28,28,1)).squeeze()
plt.imshow(image)
plt.title('Fig-2: Standard deviation of digit Four',fontweight='bold'
,fontsize='8')
plt.imshow(image)

plt.figure(figsize=(5,10))
plt.subplot(1,2,1)
image = np.array(mean_dig[5][:].reshape(28,28,1)).squeeze()
plt.imshow(image)
plt.title('Fig-1: Mean of digit Five',fontweight='bold',fontsize='8')
plt.imshow(image)

plt.subplot(1,2,2)
image = np.array(std_dig[5][:].reshape(28,28,1)).squeeze()
plt.imshow(image)
plt.title('Fig-2: Standard deviation of digit Five',fontweight='bold'
,fontsize='8')
plt.imshow(image)

plt.figure(figsize=(5,10))
plt.subplot(1,2,1)
image = np.array(mean_dig[6][:].reshape(28,28,1)).squeeze()
plt.imshow(image)
plt.title('Fig-1: Mean of digit Six',fontweight='bold',fontsize='8')
plt.imshow(image)

plt.subplot(1,2,2)
image = np.array(std_dig[6][:].reshape(28,28,1)).squeeze()
plt.imshow(image)
plt.title('Fig-2: Standard deviation of digit Six',fontweight='bold',
fontsize='8')
plt.imshow(image)

plt.figure(figsize=(5,10))
plt.subplot(1,2,1)
image = np.array(mean_dig[7][:].reshape(28,28,1)).squeeze()
plt.imshow(image)
plt.title('Fig-1: Mean of digit Seven',fontweight='bold',fontsize='8'
)
plt.imshow(image)

plt.subplot(1,2,2)
image = np.array(std_dig[7][:].reshape(28,28,1)).squeeze()
plt.imshow(image)
plt.title('Fig-2: Standard deviation of digit Seven',fontweight='bol
d',fontsize='8')
plt.imshow(image)
```

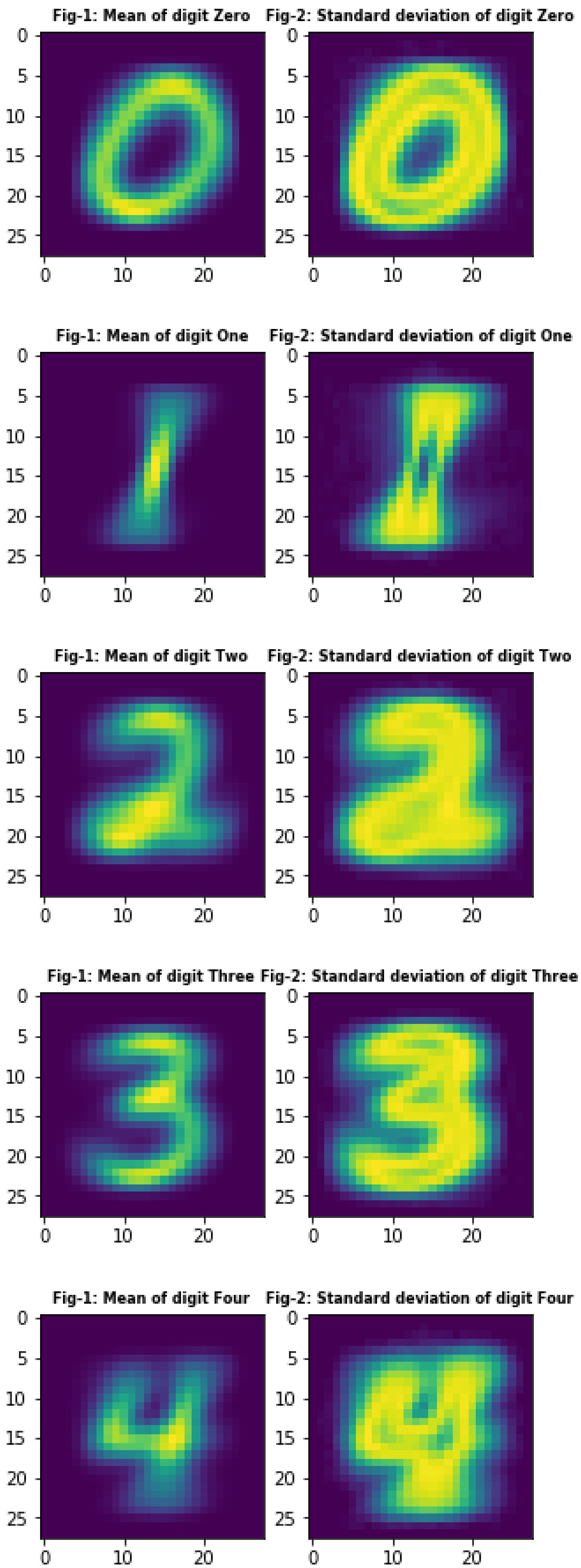
```
plt.figure(figsize=(5,10))
plt.subplot(1,2,1)
image = np.array(mean_dig[8][:].reshape(28,28,1)).squeeze()
plt.imshow(image)
plt.title('Fig-1: Mean of digit Eight',fontweight='bold',fontsize='8'
)
plt.imshow(image)

plt.subplot(1,2,2)
image = np.array(std_dig[8][:].reshape(28,28,1)).squeeze()
plt.imshow(image)
plt.title('Fig-2: Standard deviation of digit Eight',fontweight='bol
d',fontsize='8')
plt.imshow(image)

plt.figure(figsize=(5,10))
plt.subplot(1,2,1)
image = np.array(mean_dig[9][:].reshape(28,28,1)).squeeze()
plt.imshow(image)
plt.title('Fig-1: Mean of digit Nine',fontweight='bold',fontsize='8')
plt.imshow(image)

plt.subplot(1,2,2)
image = np.array(std_dig[9][:].reshape(28,28,1)).squeeze()
plt.imshow(image)
plt.title('Fig-2: Standard deviation of digit Nine',fontweight='bold'
,fontsize='8')
plt.imshow(image)
```

```
Out[7]: <matplotlib.image.AxesImage at 0x7f03805ad278>
```



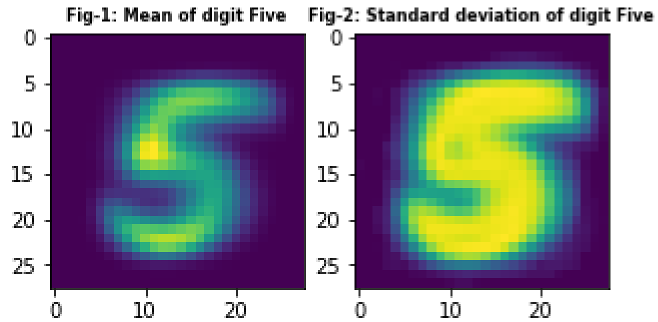


Fig-1: Mean of digit Five
Fig-2: Standard deviation of digit Five

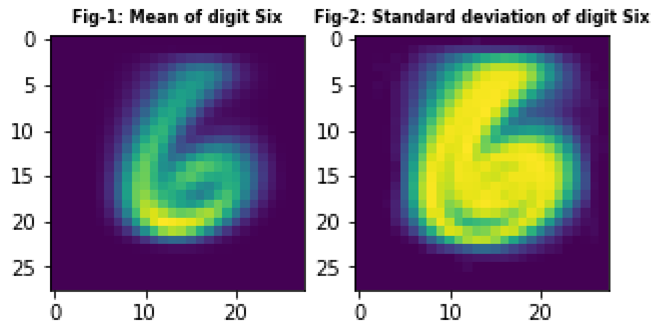


Fig-1: Mean of digit Six
Fig-2: Standard deviation of digit Six

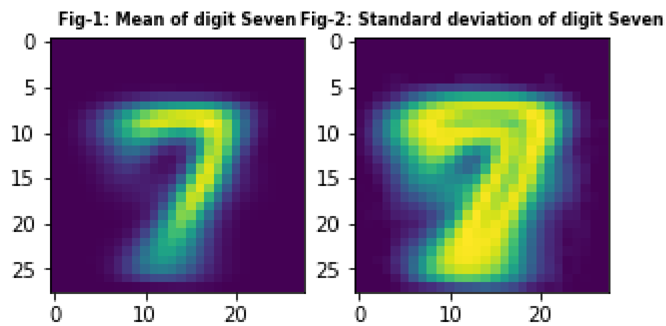


Fig-1: Mean of digit Seven
Fig-2: Standard deviation of digit Seven

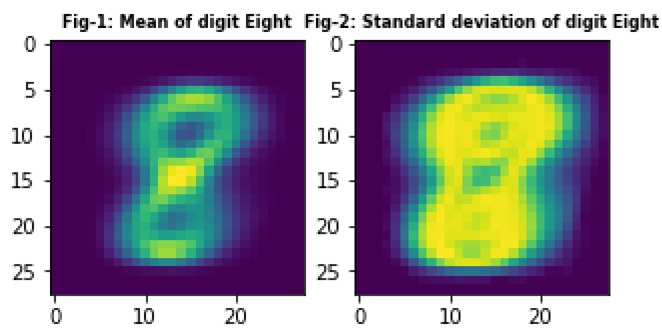


Fig-1: Mean of digit Eight
Fig-2: Standard deviation of digit Eight

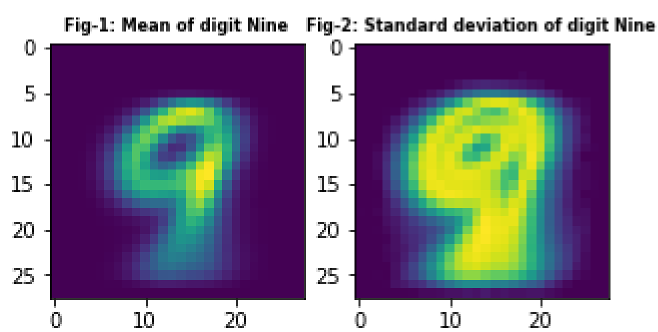


Fig-1: Mean of digit Nine
Fig-2: Standard deviation of digit Nine

Comment

The above plots show the images of mean and standard deviation of every class. These values represent the probability of feature given a particular class. These images show how a naive base classifier makes predictions on the basis of the mean and standard deviations of gaussians fit for each feature.

Part 2 - Regularized Logistic Regression

Logistic regression is the technique for binary classification of the multi-class data set. This method makes the use of logistic function which is also called as sigmoid function.

In [14]:
```
train_min = np.min(X_train,axis=0)
train_max = np.max(X_train,axis=0)
train_den = train_max-train_min
train_den[np.argwhere(train_den==0)]=1
X_train_mod = (X_train-train_min)/train_den
X_train_mod = np.append(np.ones([60000,1]),X_train_mod,axis=1)
x_test_mod = (x_test-train_min)/train_den
x_test_mod = np.append(np.ones([10000,1]),x_test_mod,axis=1)
```

2.1 Function for calculating sigmoid, gradient descent & predicting the class

Below code has three functions developed which are sigmoid function, the gradient descent function and the function for predicting the class of input feature.

In [15]:
```
def sigmoid(x,y,w):
    p = 1/(np.exp(-y*np.dot(w,x))+1)
    return(p)

def grad_desc(x,y,lamda,eps):
    sz = x.shape
    diff = 1
    #lamda =1
    ita = 0.0001
    w=np.zeros(sz[1])
    count = 0
    max_itr = 1000
    while diff>eps and count <max_itr:
        sum_grad = np.zeros(sz[1])
        for i in range(sz[0]):
            sum_grad = sum_grad + -y[i]*x[i,:]*(1-sigmoid(x[i,:],y[i
],w))
        grad = sum_grad + lamda*w
        w_new = w - ita*grad
        diff = max(abs(w_new-w))
        w = w_new
        count = count+1
    if count==max_itr:
        print('Maximum iterations reached in gradient descent and the
difference is ', diff)
    return(w)

def predict_class(w,testx):
    Y=np.matmul(w,np.transpose(testx))
    ytest = np.argmax(Y,axis=0)
    return(ytest)
```

2.2 Calculation of accuracy for different regularization parameter

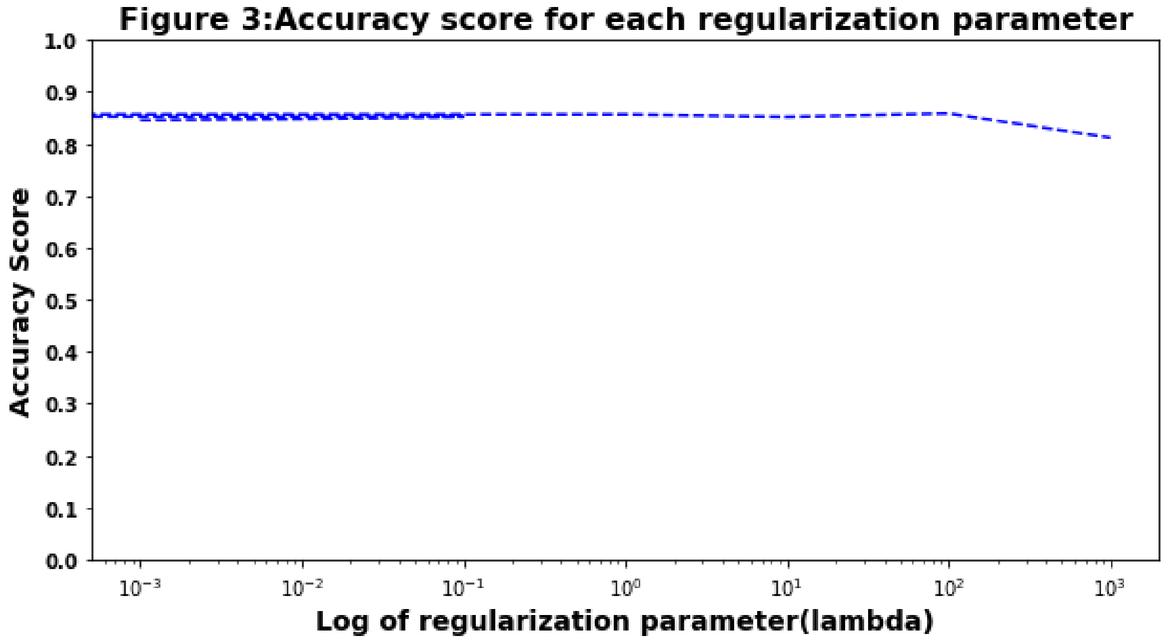
In [16]:
```
sz = y_train.shape
acc=[]
eps = 0.1
for lamda in [0.001,0.01,0.1,0,1,10,100,1000]:
    print('Performing gradient descent for lamda ',lamda)
    w=[]
    for i in range(10):
        #print('calculating w for digit',i)
        y_train_mod = -1*np.ones(sz[0])
        y_train_mod[np.argwhere(y_train==i)] = 1
        w=np.append(w,grad_desc(X_train_mod,y_train_mod,lamda,eps),ax
is=0)
    w=w.reshape(10,785)
    ytest = predict_class(w,x_test_mod)
    acc.append(np.count_nonzero(ytest-y_test==0)/10000)
accuracy_table1=pd.DataFrame(acc, columns=['Table 2:Accuracy for each
regularization parameter'])

plt.figure(figsize=(10,5))
lam = np.array([0.001,0.01,0.1,0,1,10,100,1000])
plt.plot(lam, acc, 'b--')
plt.xscale('log')
plt.xlabel('Log of regularization parameter(lambda)', fontweight='bol
d', fontsize='14')
plt.ylabel('Accuracy Score', fontweight='bold', fontsize='14')
#plt.xticks(lam,fontweight='bold', fontsize='10')
plt.yticks(np.arange(0.0,1.1, 0.1),fontweight='bold', fontsize='10')
plt.title('Figure 3:Accuracy score for each regularization parameter'
, fontweight='bold', fontsize='16')
accuracy_table1.round(3)
```

```
Performing gradient descent for lamda  0.001
Maximum iterations reached in gradient descent and the difference is
0.41644333526244637
Performing gradient descent for lamda  0.01
Maximum iterations reached in gradient descent and the difference is
0.41236451529613305
Performing gradient descent for lamda  0.1
Maximum iterations reached in gradient descent and the difference is
0.37833057572380824
Performing gradient descent for lamda  0
Maximum iterations reached in gradient descent and the difference is
0.4167395587887839
Performing gradient descent for lamda  1
Maximum iterations reached in gradient descent and the difference is
0.3013852303323219
Performing gradient descent for lamda  10
Performing gradient descent for lamda  100
Performing gradient descent for lamda  1000
Maximum iterations reached in gradient descent and the difference is
0.6173483430584975
Maximum iterations reached in gradient descent and the difference is
0.8244077076628427
```

Out[16]:

	Table 2:Accuracy for each regularization parameter
0	0.845
1	0.848
2	0.852
3	0.847
4	0.856
5	0.852
6	0.858
7	0.812



Comments on the result:

1. From the results table it can be seen that highest accuracy is obtained for the regularizer value of 100.
2. It is also observed that the accuracy is not changing much for the given values of lambda

Note: λ = 100 did not converge when a stricter convergence (eps =0.01) criteria was implemented. Therefore λ = 1 was used for the remaining results. It is also seen from the previous table that the accuracy was almost the same for λ =1 and λ =100

2.3 Calculating the weights of features for every class

In [23]:
```
w=[]
lamda = 1
eps = 0.01
for i in range(10):
    print('calculating w for digit',i)
    y_train_mod = -1*np.ones(sz[0])
    y_train_mod[np.argwhere(y_train==i)] = 1
    w=np.append(w,grad_desc(X_train_mod,y_train_mod,lamda,eps),axis=0
)
w=w.reshape(10,785)
print('Calculation Complete')
```

```
calculating w for digit 0
calculating w for digit 1
calculating w for digit 2
calculating w for digit 3
calculating w for digit 4
calculating w for digit 5
Maximum iterations reached in gradient descent and the difference is
0.08060448394405784
calculating w for digit 6
calculating w for digit 7
calculating w for digit 8
Maximum iterations reached in gradient descent and the difference is
0.3013852303323219
calculating w for digit 9
Maximum iterations reached in gradient descent and the difference is
0.01928269793202908
Calculation Complete
```

2.4 Predicting the class of all test features and calculating overall accuracy

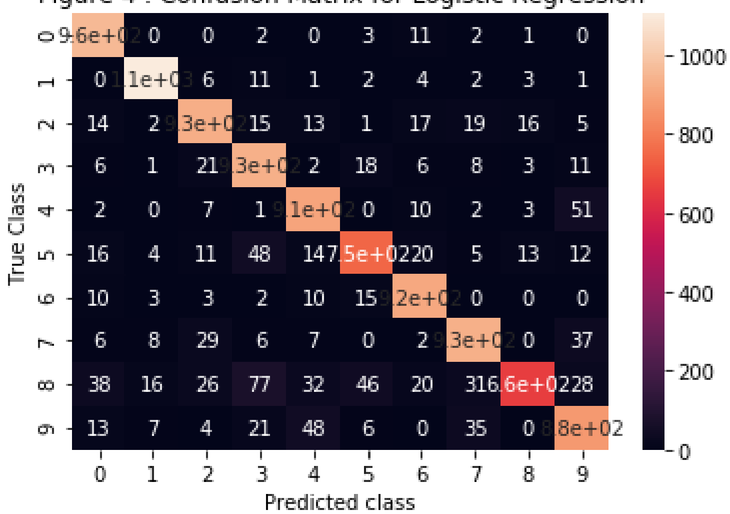
In [24]:
```
ytest = predict_class(w,x_test_mod)
ac = np.count_nonzero(ytest-y_test==0)/10000
print('Accuracy of the logistic regression classifier is', ac*100,'%'
)
```

```
Accuracy of the logistic regression classifier is 89.68 %
```

2.5 Plotting the confusion matrix and calculating the accuracy score for every class

In [28]:
```
# Plot the confusion matrix
conf1=confusion_matrix(y_test, ytest)
ax1= plt.subplot()
df_cm1 = pd.DataFrame(conf1, index = [i for i in "0123456789"],column
s = [i for i in "0123456789"])
sns.heatmap(df_cm1, annot=True, ax = ax1); #annot=True to annotate ce
lls
ax1.set_xlabel('Predicted class');ax1.set_ylabel('True Class');
ax1.set_title('Figure 4 : Confusion Matrix for Logistic Regression');
ax1.xaxis.set_ticklabels(['0','1','2','3','4','5','6','7','8','9']);
ax1.yaxis.set_ticklabels(['0','1','2','3','4','5','6','7','8','9']);
```

Figure 4 : Confusion Matrix for Logistic Regression



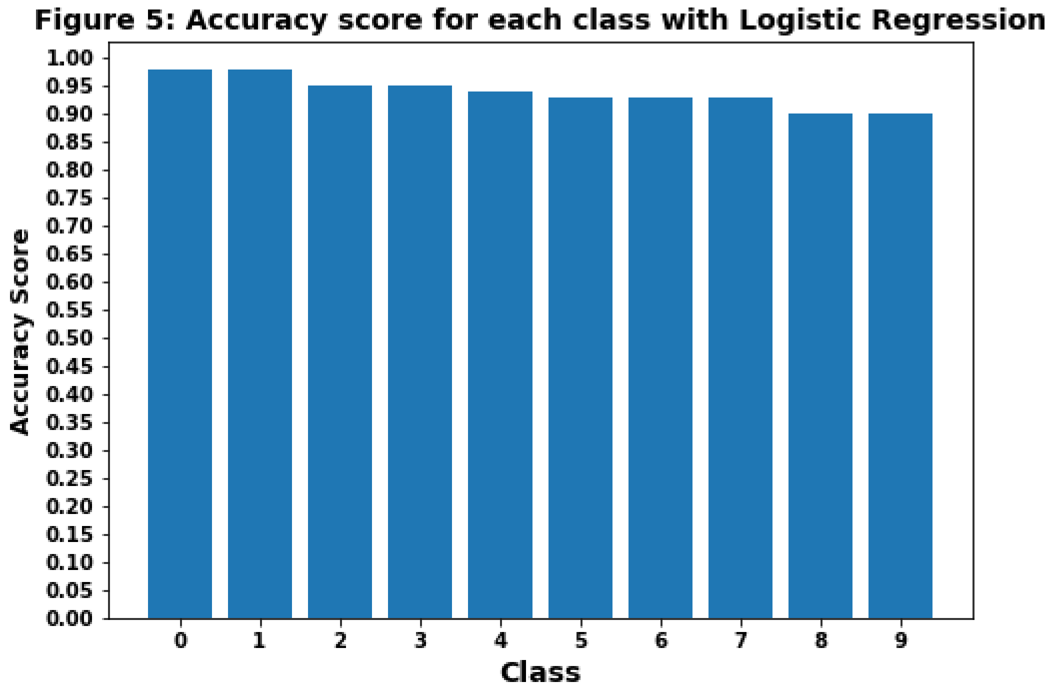
Comment

The above plot shows the confusion matrix which demonstrates the performance of our classification algorithm. Both axes have class labels however the X-axis has the predicted classes and Y-axis has true classes. The values inside the cells in this matrix show the number of correctly predicted class for any particular true class. From the confusion matrix, it can be inferred that (7,8) is the combination of the classes where the classifier has less accuracy in identifying the correct label.

In [26]:
```
# Calculate the accuracy for each class
accuracy1=[]
accuracy_digit1=[]
for i in range(10):
    Test_class1=y_test
    test_data1=pd.DataFrame(Test_class1)
    i11=test_data1[test_data1[0]==i]
    indices1=list(i11.index)
    for j in indices1:
        acc1=((ytest[j] ==i).sum()/((y_test[j]==i).sum()))
        accuracy1.append(acc1)
    accuracy_avg1 = np.mean(accuracy1)
    accuracy_digit1.append(accuracy_avg1)
accuracy_table1=pd.DataFrame(accuracy_digit1, columns=['Table 3 :Accu
racy for each class with Logistic Regression'])

my1 = np.array(accuracy_digit1).round(2)
fig = plt.figure()
ax1 = fig.add_axes([0,0,1,1])
labels = ['0','1','2','3','4','5','6','7','8','9',]
values = [my1[0],my1[1],my1[2],my1[3],my1[4],my1[5],my1[6],my1[7],my1
[8],my1[9]]
ax1.bar(labels,values)
plt.xlabel('Class', fontweight='bold', fontsize='14')
plt.ylabel('Accuracy Score', fontweight='bold', fontsize='12')
plt.xticks(np.arange(0, 10, 1),fontweight='bold', fontsize='10')
plt.yticks(np.arange(0.0,1.05, 0.05),fontweight='bold', fontsize='10'
)
plt.title('Figure 5: Accuracy score for each class with Logistic Regr
ession', fontweight='bold', fontsize='14')
plt.figure(figsize=(20,20))
plt.show()

accuracy_table1.round(2)
```




```
<Figure size 1440x1440 with 0 Axes>
```

Out[26]:

	Table 3 :Accuracy for each class with Logistic Regression
0	0.98
1	0.98
2	0.95
3	0.95
4	0.94
5	0.93
6	0.93
7	0.93
8	0.90
9	0.90

Comment

The above plot shows the accuracy score for each class. From the figure, it can be inferred that the classifier has least accuracy in correctly predicting the test features with class 8 & 9 and most accuracy in correctly predicting the test features with class 0 & 1.

2.6 Plotting the images of weights for all classes

In [27]:
```
t=np.array(w)

plt.figure(figsize=(40,40))
plt.subplot(10,1,1)
image = np.array(t[0][1:].reshape(28,28,1)).squeeze()
plt.imshow(image)
plt.title('Fig-1: Weight of image zero',fontweight='bold',fontsize='1
6')
plt.show


plt.figure(figsize=(40,40))
plt.subplot(10,1,2)
image = np.array(t[1][1:].reshape(28,28,1)).squeeze()
plt.imshow(image)
plt.title('Fig-2: Weight of image one',fontweight='bold',fontsize='1
6')
plt.show

plt.figure(figsize=(40,40))
plt.subplot(10,1,3)
image = np.array(t[2][1:].reshape(28,28,1)).squeeze()
plt.imshow(image)
plt.title('Fig-3: Weight of image two',fontweight='bold',fontsize='1
6')
plt.show


plt.figure(figsize=(40,40))
plt.subplot(10,1,4)
image = np.array(t[3][1:].reshape(28,28,1)).squeeze()
plt.imshow(image)
plt.title('Fig-4: Weight of image three',fontweight='bold',fontsize=
'16')
plt.show

plt.figure(figsize=(40,40))
plt.subplot(10,1,5)
image = np.array(t[4][1:].reshape(28,28,1)).squeeze()
plt.imshow(image)
plt.title('Fig-5: Weight of image four',fontweight='bold',fontsize='1
6')
plt.show


plt.figure(figsize=(40,40))
plt.subplot(10,1,6)
image = np.array(t[5][1:].reshape(28,28,1)).squeeze()
plt.imshow(image)
plt.title('Fig-6: Weight of image five',fontweight='bold',fontsize='1
6')
plt.show


plt.figure(figsize=(40,40))
plt.subplot(10,1,7)
```

```
image = np.array(t[6][1:].reshape(28,28,1)).squeeze()
plt.imshow(image)
plt.title('Fig-7: Weight of image six',fontweight='bold',fontsize='16')
plt.show

plt.figure(figsize=(40,40))
plt.subplot(10,1,8)
image = np.array(t[7][1:].reshape(28,28,1)).squeeze()
plt.imshow(image)
plt.title('Fig-8: Weight of image seven',fontweight='bold',fontsize='16')
plt.show


plt.figure(figsize=(40,40))
plt.subplot(10,1,9)
image = np.array(t[8][1:].reshape(28,28,1)).squeeze()
plt.imshow(image)
plt.title('Fig-8: Weight of image eight',fontweight='bold',fontsize='16')
plt.show

plt.figure(figsize=(40,40))
plt.subplot(10,1,10)
image = np.array(t[9][1:].reshape(28,28,1)).squeeze()
plt.imshow(image)
plt.title('Fig-9: Weight of image nine',fontweight='bold',fontsize='16')
plt.show
```

Out[27]: <function matplotlib.pyplot.show(*args, **kw)>

Fig-1: Weight of image zero

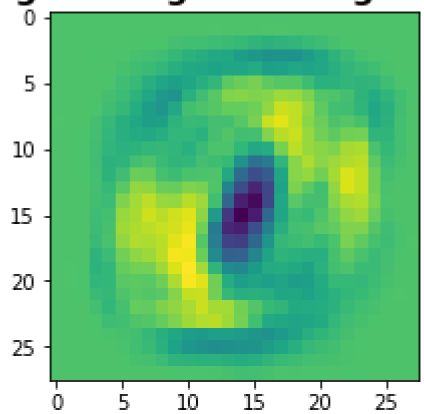


Fig-2: Weight of image one

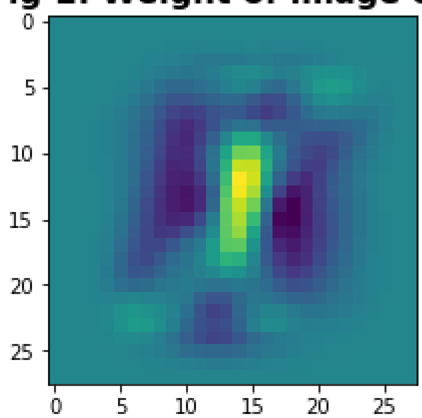


Fig-3: Weight of image two

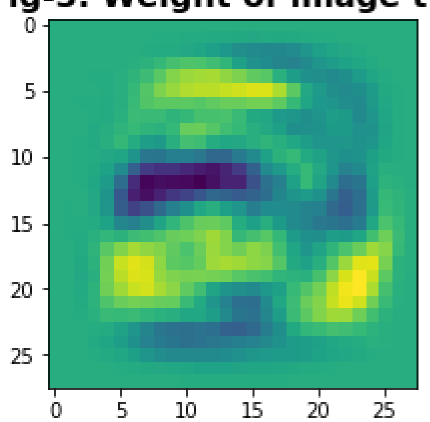


Fig-4: Weight of image three

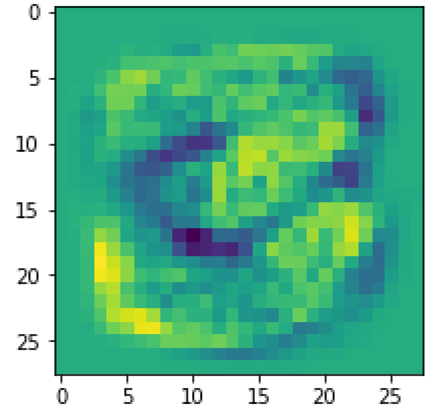


Fig-5: Weight of image four

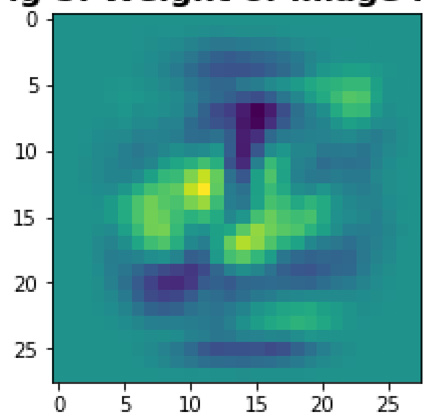


Fig-6: Weight of image five

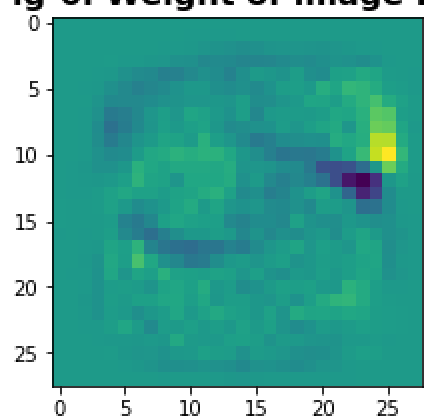


Fig-7: Weight of image six

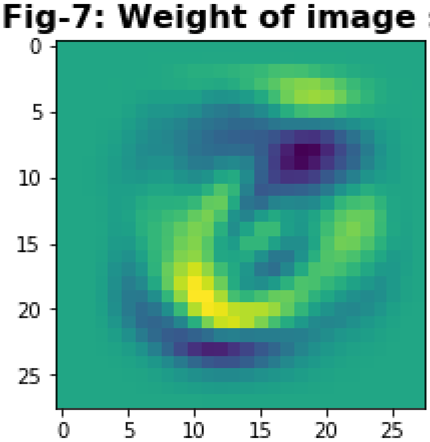


Fig-8: Weight of image seven

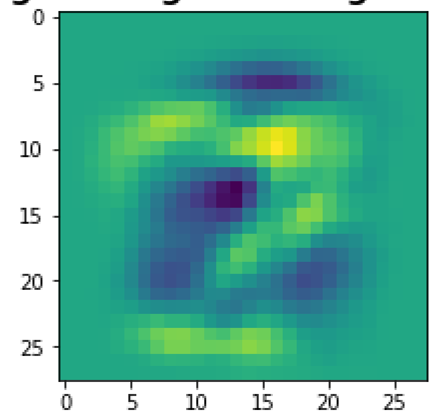


Fig-8: Weight of image eight

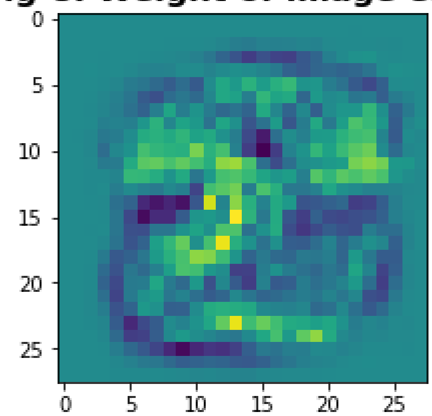
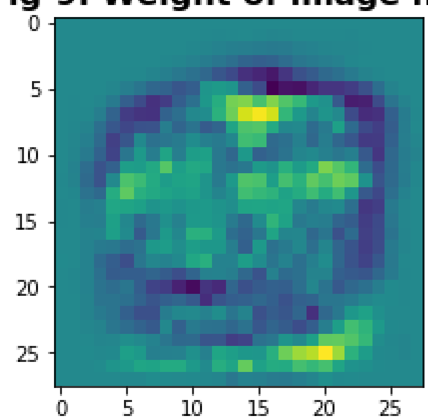


Fig-9: Weight of image nine



Comment

The above plots show the images of weights of every class. These weights determine the importance of features in predicting a label. The more important the feature is the higher weight it has. Thus, plotting the weights enables us to see the pixels that play a significant role in predicting the class of that feature.

2.7 Derivation of update equation

We know that likelihood function is given as

==> $L(w) = \sum_{i=1}^{N} \ln(1 + e^{(-yw^T x_i)}) + \frac{\lambda}{2}|w|^2$

If we take gradient of this function at w, then the function will become

==> $\bigtriangledown L(w) = \sum_{i=1}^{N} \frac{-yx_i e^{-yw^T x_i}}{1+e^{-yw^T x_i}} + \lambda|w|$

The update equation for W can be written as:

==> $W_{t+1} = W_t - \eta_t \bigtriangledown L(w)$

Putting the gradient in above equation, we get

==> $W_{t+1} = W_t - \eta_t[sum_{i=1}^{N} \frac{-yx_i e^{-yw^T x_i}}{1+e^{-yw^T x_i}} + \lambda|w|]$