MNIST with Naive Bayes and Logistic Regression

Submitted by: Sarala Ravindra

Input data extraction

The input and test dataset consisting of the 28 cross 28 images and corresponding labels are extracted.The MNISt train data consists of 60,000 images of 28 cross 28 pixels and each train image has its corresonding label while the test dataset consists of 10,000 images of 28 cross 28 pixels and each test image has its corresponding label.The train data and test data are loaded from the source files train_data.npz and test_data.npz respectively from http://yann.lecun.com

The dataset is separated into train images, train labels, test images and test labels.The image sets are reshaped to a 2D array (originally 3D array) where the 2D arrangement of the pixel (28*28) in each is reduced to 1D arrangemnt of length 784.

In [5]:

```
%%time


import numpy as np
import mnist

#  train and test data extraction

dataset_train = np.load("train_data.npz")
train_images = dataset_train['x']                      #train images
extraction
train_lab = dataset_train['y']                         #train labels extract
on
dataset_test = np.load("test_data.npz")
test_images = dataset_test['x']                        #test images extracti
n
test_lab = dataset_test['y']                           #test labels extracti
n

train_img = np.reshape(train_images,[60000,28*28])
test_img = np.reshape(test_images,[10000,28*28])
```

```
CPU times: user 948 ms, sys: 460 ms, total: 1.41 s
Wall time: 6.17 s
```

Question 1

In [55]:

```
%%time
```

```
# Question 1 :   Naive Bayes' Classification on MNIST image dataset    ----
---------------------#

# Define required arrays

c_mean = []
c_std  = []
c_img_mean = []
c_img_std = []
prob  = np.zeros((10,10000)).astype('float128')
conditional_prob= []
Disp = np.zeros((10,784))

# loop for 10 possible classes

for c in range(10):
    c_images = []

    # Separate train images with label class 'c' from the rest of the image
s

    for index in range(60000):
        if(train_lab[index]==c):
            c_images.append(index)
    c_img = train_img[c_images]

    # Find the mean and standard deviation of the images features for a
given class

    c_img_mean = np.mean(c_img, axis=0, dtype ='float128')
    c_img_std  = np.std(c_img, axis=0, dtype ='float128') + 12          # a
non-zero constant added to standard deviation for avoiding feature value
being invariant.

    Disp[[c],:] = c_img_mean                                            # s
ring class means to display


    # Calculate the probability of the class for all the test images.

    gauss_term =
np.divide(np.subtract(test_img,c_img_mean,dtype='float128'),c_img_std,dtype
loat128')                          #  (test image - mean of class)/standars d
vition of class

    conditional_prob = np.divide(np.exp(-
np.power(gauss_term,2,dtype='float128')/2,dtype='float128'),c_img_std,dtype
loat128')  #  conditional probability of each image feature for the class =
exp(-(gauss_term^2/2))/standard deviation of the class

    prob[c,:] = np.prod(conditional_prob,axis=1,dtype ='float128')      # p
robability of the class 'c' for the test images


#   Predict test labels from the train model

predict_lab = np.argmax(prob,axis=0)                                    # l
bel for an image is the index (class = index) corresponding to the highest
of the probabilities of the image given a class
```

```
#   Confusion matrix Generation

count = np.zeros([10,10]).astype(int)
# 10 possible classes
for k_actual in range(10):
    for k_predicted in range(10):                                          # 1
possible class prediction
        for l in range(10000):                                             # a
uracy chech over 10,000 test images
            if( test_lab[l]==k_actual and predict_lab[l]==k_predicted):
                count[k_actual,k_predicted]=count[k_actual,k_predicted]+1;

print('The confusion matrix for the classification is ')
print('  ')
print(count)
print('  ')
print('The overall accuracy is ',np.divide(np.trace(count),10000*0.01),'%')
```

```
The confusion matrix for the classification is

[[ 902    1    1    1    1    5   23    1   34   11]
 [    0 1111    2    3    0    0    7    0   11    1]
 [   26   49  603   52   11    0  126    8  144   13]
 [   11   63   16  739    2    9   24   12   64   70]
 [    4   16    6    1  473    8   29    5   21  419]
 [   31   48    5   90   17  347   34   10  231   79]
 [    7   27    6    1    3   12  883    0   18    1]
 [    0   36    5    7   12    1    4  740   24  199]
 [    9  121    4   25    9   11    9    6  673  107]
 [    5   24    3    7   17    0    0   10   16  927]]

The overall accuracy is  73.98 %
CPU times: user 12.1 s, sys: 1.42 s, total: 13.5 s
Wall time: 13.5 s
```

The confusion matrix displayed above gives the actual label versus the predicted label data.The row 1,row 2 ,row 3,row 4 ...till row 10 correspond to digit 0 , 1 , 2 , 3... till 9 respectively.The coloumn spans similarlly giving the predict label outcomes. The trace of this matrix provides th eoveral correct prediction made by this classifier and for the modeled Naive Bayes' classifier the overall prediction accuracy is found to be 73.98 %.

The classification is done based on the conditional probabilities of all the features (that is 784 image pixels) for a given digit.You can observe that the highest misclassification proves the idea behind it.

Say for digit 1 the highest mis classification label is with the digit 9.This is because the features of the images digit 1 and 9 have a similar density function ,this idea can be visualized with the image of probability densities of features for a given digit.

The training of this classifier may takes into consideration of all the features.But a better model (in terms of time and computation) can be modelled by throwing out the insignificant features for a given class.

Though we take into account all the features in our modelling the accuracy is about 75 % ,this is because the Naive Bayes' model assumes that the features (the image pixels) are independent of each other which is not true in reality.

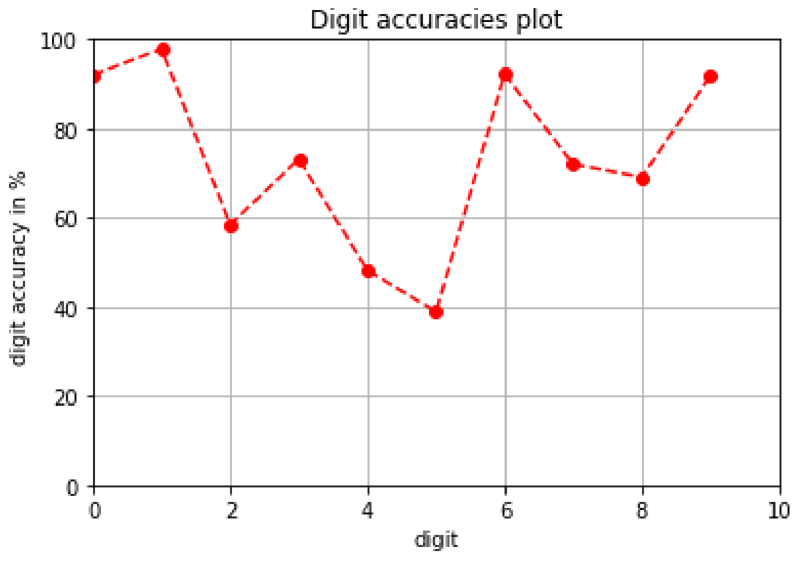
In [20]:

```
# digit accuracies

class_accuracy = np.zeros(10)
for u in range(10):
    class_accuracy[u] = count[u,u]/sum(count[u,:])
    print('Digit = ', u ,'     ',end='')
    print('Digit Accuracy in percent = ',class_accuracy[u]*100 )
```

```
Digit =  0      Digit Accuracy in percent =  96.6326530612
Digit =  1      Digit Accuracy in percent =  93.5682819383
Digit =  2      Digit Accuracy in percent =  83.4302325581
Digit =  3      Digit Accuracy in percent =  89.603960396
Digit =  4      Digit Accuracy in percent =  88.0855397149
Digit =  5      Digit Accuracy in percent =  82.1748878924
Digit =  6      Digit Accuracy in percent =  90.8141962422
Digit =  7      Digit Accuracy in percent =  88.5214007782
Digit =  8      Digit Accuracy in percent =  76.4887063655
Digit =  9      Digit Accuracy in percent =  74.7274529237
```

In [45]:

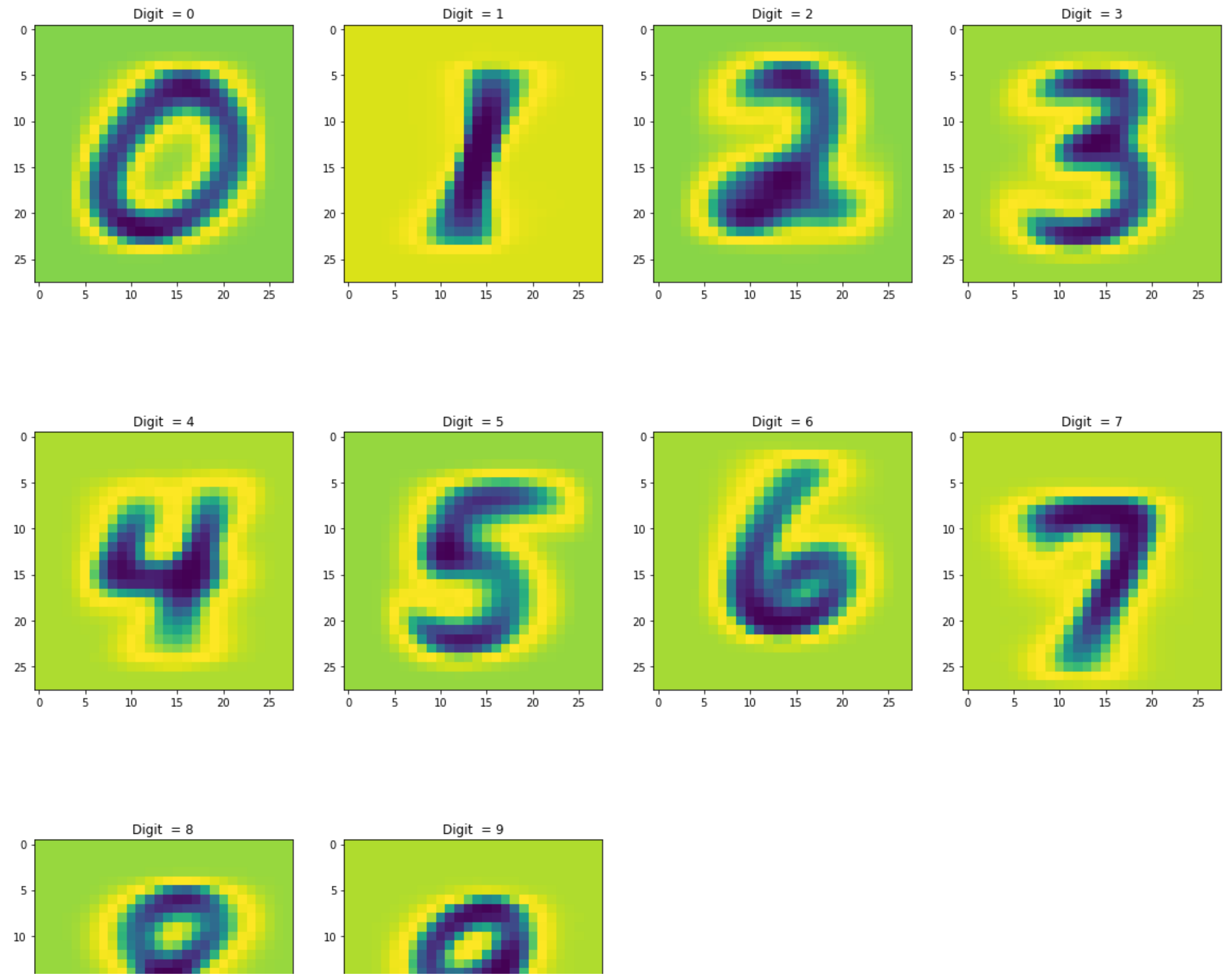
```
## Digit accuracy

import matplotlib.pyplot as plt
plt.plot([0,1,2,3,4,5,6,7,8,9], [92.04081633 , 97.88546256 , 58.43023256 , 7
3.16831683 , 48.16700611,
   38.90134529 , 92.17118998 , 71.9844358   , 69.09650924  ,91.87314172],'--r
o')
plt.axis([0, 10, 0, 100])
plt.xlabel('digit')
plt.ylabel('digit accuracy in % ')
plt.title('Digit accuracies plot')
plt.grid(True)
plt.show()
```

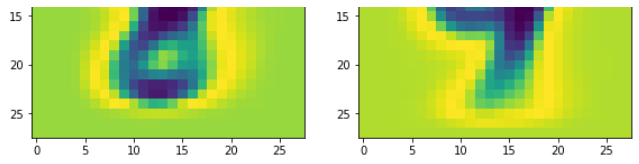


We see that digit 1 has the highest digit accuracy and 5 has the lowest.This means that digit 1 image pixels have got more unique features which easily classifies it from other digits while digit 5 seem to share a lot of features in common with other digits compared to the rest of the digits.

In [157]:

```
# display class mean image conditional probability
import matplotlib.pyplot as plt
import matplotlib.pyplot as pyplot


mean_ = np.mean(Disp,axis=1)
std_ = np.std(Disp,axis = 1)+12
cond_p = np.zeros((10,784))
scaled_p_img=np.zeros((10,28,28))
scaled_p= np.zeros((10,784))
for c in range(10):

    gauss_term =  np.divide(np.subtract(Disp[c,:],mean_[c],dtype='float128'
),std_[c],dtype='float128')  # (test image - mean of class)/standars deviti
on of class
    cond_p[[c],:] =
np.divide(np.exp(-np.power(gauss_term,2,dtype='float128')/2,dtype='float128'
),std_[c],dtype='float128')  #  conditional probability of each image featu
re for the class = exp(-(gauss_term^2/2))/standard deviation of the class
    scaled_p[[c],:] = cond_p[[c],:]*10**(3)

scaled_p_img=scaled_p.reshape(10,28,28)
plt.figure(figsize=(20,20))
for img in range(10):
    plt.subplot(3,4,img+1)
    plt.imshow(scaled_p_img[img,:,:])
    plt.title('Digit  = %i' % img)
pyplot.show()
```





Note that the probability densities of features of each class when nposed as an image appear to have the digit itself itself.This is the probabilities af those features making up the digit is higher compared to the rest of the features.It is based on these probabilities that the classification nmodel is working here.Note in similar 'looking' digits such as digit 1 and 9 the misclassification is high due to this reason.Similarlly 9 is higly misclassified to be digit 4.Thus this model prediction is highly based on the probability dendities . Thus ,this model is a good predictor when the features are mostly independent.

In [ ]:

```
# Question 2 :   Regularized Logistic Regression Classification on MNIST im
age dataset    --------------------------#
```

Question 2

The log liklihood function for the logistic regression classifier model regularized with ridge(l_2) regularizer is

$$L(w) = \sum_{i=1}^{N} log(1 + exp(-y_i w^T x_i)) + \frac{\lambda}{2}||w||_2$$

Using maximum liklihood method the opyimum weight can be achieved by calculating the gradient of log liklihood function.that is the log gunction is diffrentiated w.r.t weight vector 'w' and then equated to zero which results in equation below to compute optimum weights.

$$\nabla L(w) = \sum_{i=1}^{N} \frac{-y_i x_i \exp(-y_i w^T x_i)}{1 + \exp(-y_i w^T x_i)} + \lambda w$$

The update function of the l2 *regularizer is $$w*{t+1}=w_t-\eta{_t\nabla{L(w)}}$$

In [92]:

```
%%time

from scipy.stats import mode
from pandas import *

# normalize image data (both train and test to lie between 0 and 1 instead
of 0 and 255 interval)

train_norm = train_img/255
test_norm  = test_img/255

# insert a coloumn of ones in the train image matrix to weigh the bias term
w_0

one_col_t = np.ones((60000,1))       # one col for train
one_col_l = np.ones((10000,1))       # one col for test
```

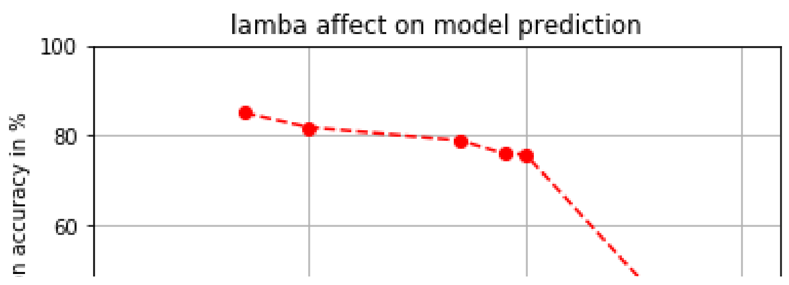
```
train_norm = np.hstack((train_norm,one_col_t))
test_norm = np.hstack((test_norm,one_col_l))
train_new_lab = np.zeros((60000)).astype(int)

 # different regularisation coefficient values

o =[0,0.0005,0.001,0.005,0.008,0.01,0.05,0.1]
for o_index in range(8):
    lam =o[o_index]
    eta=0.00022                     # update rate

    w=np.zeros((10,785))            # define weight matrix for each class
    for c in range(10):

        # change train labels as positive or negative one based on class

        for i in range(60000):
            if train_lab[i]==c:
                train_new_lab[i]=1
            else:
                train_new_lab[i]=-1
        # calculate weights  and optimise using iteration(Newton's update)

        w_final=np.ones((1,785))*0.01   # initial non-zero weights
        # 35 iterations

        for iteration in range(35):
            l=np.zeros((1,785))
            # determine change in loss function

            for sample in range(10000):

                # regularized logistic regression

                term=-
train_new_lab[sample]*np.matmul(train_norm[[sample],:],np.transpose(w_final

                regularization = np.add((-
train_new_lab[sample]*train_norm[[sample],:]*np.exp(term)/(1+np.exp(term)))
(lam*w_final))
                l=np.add(l,(regularization))

            # Gradient descent method
            w_final=np.subtract(w_final,(eta*l))
        # store for each class

        w[[c],:]=w_final

    #  Classification

    cond_prob=np.zeros((10,10000))
    for image in range(10000):
        for digit in range(10):
            # Compute liklihood

            d=1+np.exp(-
np.matmul(test_norm[[image],:],np.transpose(w[[digit],:])).astype('float128
astype('float128'))
            cond_prob[digit,image]=1/d
    # predict label (label corresponding to max likly class)
    predict_label=np.argmax(cond_prob,axis=0)

    # generate confusion matrix

    count = np.zeros([10,10]).astype(int)
# 10 possible classes
    for k_actual in range(10):
        for k_predicted in range(10):
10 possible class prediction
            for l in range(10000):
accuracy chech over 10,000 test images
                if( test_lab[l]==k_actual and
predict_label[l]==k_predicted):

count[k_actual,k_predicted]=count[k_actual,k_predicted]+1;

    print('Regularization coeff = ',lam,',   Predicton Accuracy = ',100*np.
trace(count)/np.sum(count),'%')
```

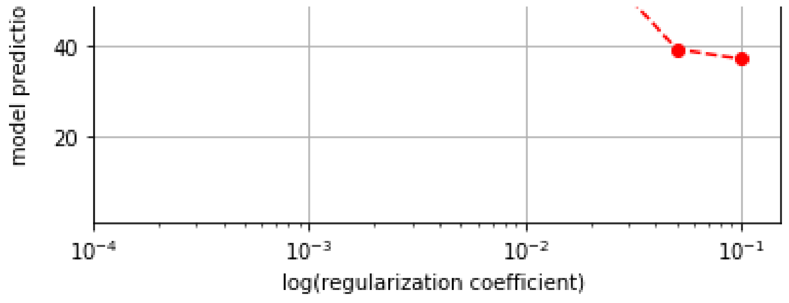
```
Regularization coeff =  0 ,   Predicton Accuracy =  81.14 %
Regularization coeff =  0.0005 ,   Predicton Accuracy =  85.06 %
Regularization coeff =  0.001 ,   Predicton Accuracy =  81.81 %
Regularization coeff =  0.005 ,   Predicton Accuracy =  78.92 %
Regularization coeff =  0.008 ,   Predicton Accuracy =  75.94 %
Regularization coeff =  0.01 ,   Predicton Accuracy =  75.85 %
Regularization coeff =  0.05 ,   Predicton Accuracy =  39.41 %
Regularization coeff =  0.1 ,   Predicton Accuracy =  37.24 %
CPU times: user 13min 45s, sys: 192 ms, total: 13min 45s
Wall time: 13min 46s
```

so 0.005 is choosen to be our lambda for analysis

In [88]:

```
### plot model accuracy over regularisation coefficient (lambda)
import matplotlib.pyplot as plt
plt.plot(o, [81.14 , 85.06,81.81,78.92,75.94,75.85,39.41,37.24],'--ro')
plt.axis([0.0001, 0.15, 1, 100.0])
plt.xscale('log')
plt.xlabel('log(regularization coefficient) ')
plt.ylabel('model prediction accuracy in % ')
plt.title('lamba affect on model prediction')

plt.grid(True)

plt.show()
```





The above result gives the model prediction accuracy for different regularization coefficient.It can be seen that the model has a less accuracy when there is no regularization (that is 0 lamdba).Also the plot of model regularization constant versus model accuracy can be seen in the graph above.The regularization constant corresponding to best accuracy is chosen(which is 0.0005) and the model analysis is done.

With increase in lambda value the prediction error is going down (non-regularization model not considered).

We want our model to have a generalised approach instead of being a perfect fit for the train data provided.Regularization ensures that the minimum variance in prediction is acheived with certain bias.that is the model is capable to produce weights so as to cutdown the effect of high order coefficients of the feature. Note that the regularisation is done in order to overcome the issue of model overfit but prediction accuracy is not the only parameter to chose the regularization coeffient.

In this logistic regression classifier,a ridge regulariser is used which results in an extra term in the change in the loss function which is provided in derivation part.

In [97]:

```
## generate confusion matrix for lambda = 0.005

lam =0.0005
eta=0.00022                         # update rate

w=np.zeros((10,785))                # define weight matrix for each class
for c in range(10):

    # change train labels as positive or negative one based on class

    for i in range(60000):
        if train_lab[i]==c:
            train_new_lab[i]=1
        else:
            train_new_lab[i]=-1
    # calculate weights  and optimise using iteration(Newton's update)

    w_final=np.ones((1,785))*0.01    # initial non-zero weights
    # 35 iterations

    for iteration in range(35):
        l=np.zeros((1,785))
        # determine change in loss function

        for sample in range(10000):

            # regularized logistic regression
            term=-train_new_lab[sample]*np.matmul(train_norm[[sample],:],np
.transpose(w_final))
            regularization = np.add((-train_new_lab[sample]*train_norm[[samp
le],:]*np.exp(term)/(1+np.exp(term))),(lam*w_final))
            l=np.add(l,(regularization))

        # Gradient descent method
        w_final=np.subtract(w_final,(eta*l))
    # store for each class

    w[[c],:]=w_final

#  Classification

cond_prob=np.zeros((10,10000))
for image in range(10000):
    for digit in range(10):
        # Compute liklihood


d=1+np.exp(-np.matmul(test_norm[[image],:],np.transpose(w[[digit],:])).asty
pe('float128').astype('float128'))
        cond_prob[digit,image]=1/d
# predict label (label corresponding to max likly class)

predict_label=np.argmax(cond_prob,axis=0)

# generate confusion matrix

count = np.zeros([10,10]).astype(int)
# 10 possible classes
for k_actual in range(10):
    for k_predicted in range(10):                                        # 1
possible class prediction
        for l in range(10000):                                           # ac
uracy chech over 10,000 test images
            if( test_lab[l]==k_actual and predict_label[l]==k_predicted):
                count[k_actual,k_predicted]=count[k_actual,k_predicted]+1;

# print confusion matrix

print('The confusion matrix for the classification with lambda = 0.0005 is
')
print(count)
print('The correct prediction  is ',100*np.trace(count)/np.sum(count),'%')
```

```
The confusion matrix for the classification with lambda = 0.0005 is
[[ 959    0    2    2    1    0    9    0    5    2]
 [    0 1087   12    8    1    1    4    1   19    2]
 [   11   12  880   23   13    0   19   19   37   18]
 [    6    1   19  919    1    4    8    9    7   36]
 [    1    2    6    0  771    0   14    2    5  181]
 [   34    8    7  138   29  488   33   14   57   84]
 [   20    3    8    3   16    4  902    0    1    1]
 [    3   15   36    5    7    0    2  874    0   86]
 [   21    5   19   67    9    5   14   12  698  124]
 [   11    6    9   19   24    1    1    9    1  928]]
The correct prediction  is  85.06 %
```

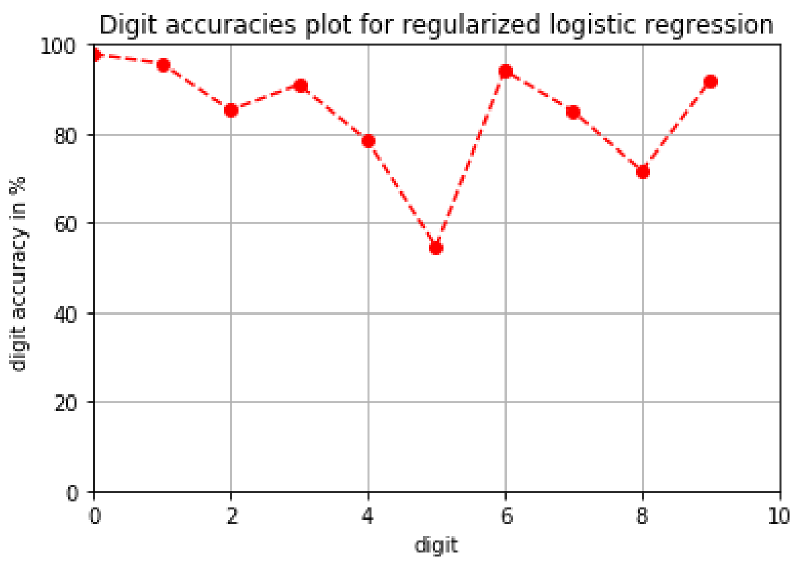
The confusion matrix reading is same as described in question 1.The digit accuracies for this model is given below graph

Note that the logistic model for classification produces better accuracy on the test compared to the Naive Bayes, ,this is because unlike the previuos model logistic does not assume independency with the features.Also it does not assume any probability distribution on the dataset .It tries to model a distribution based on the data.

In [100]:

```
## digit accuracy and plot

class_accuracy = np.zeros(10)
for u in range(10):
    class_accuracy[u] = count[u,u]/sum(count[u,:])
    print('Digit = ', u ,'     ',end='')
    print('Digit Accuracy in percent = ',class_accuracy[u]*100 )
u = [0,1,2,3,4,5,6,7,8,9]
import matplotlib.pyplot as plt
plt.plot(u, class_accuracy*100,'--ro')
plt.axis([0, 10, 0, 100])
plt.xlabel('digit')
plt.ylabel('digit accuracy in % ')
plt.title('Digit accuracies plot for regularized logistic regression')
plt.grid(True)
plt.show()
```

```
Digit =  0      Digit Accuracy in percent =  97.8571428571
Digit =  1      Digit Accuracy in percent =  95.7709251101
Digit =  2      Digit Accuracy in percent =  85.2713178295
Digit =  3      Digit Accuracy in percent =  90.9900990099
Digit =  4      Digit Accuracy in percent =  78.5132382892
Digit =  5      Digit Accuracy in percent =  54.7085201794
Digit =  6      Digit Accuracy in percent =  94.1544885177
Digit =  7      Digit Accuracy in percent =  85.0194552529
Digit =  8      Digit Accuracy in percent =  71.6632443532
Digit =  9      Digit Accuracy in percent =  91.9722497522
```

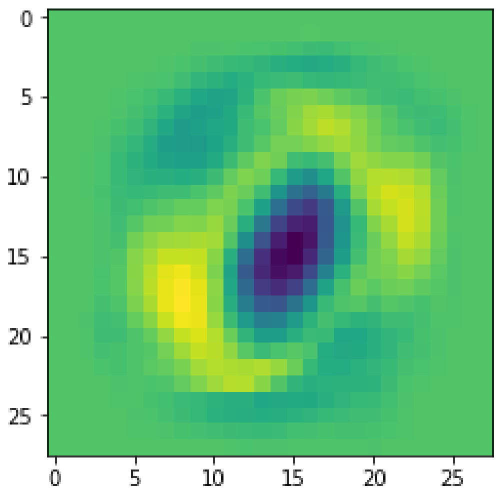


As already discussed ,the logistic model provides better digit accuracies as optimum weights are used

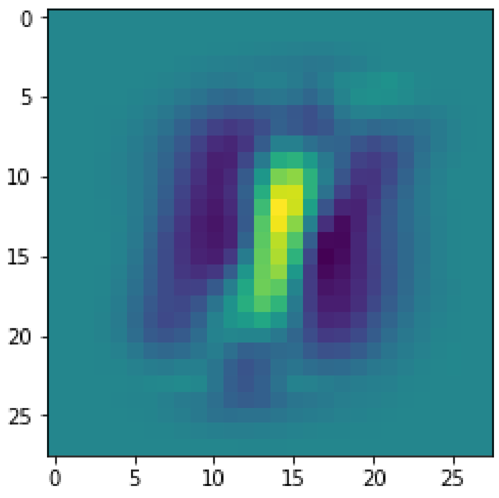
In [111]:

```
## display weights

w_disp = np.delete(w, -1, 1)
w_reshape = np.zeros((10,28,28))
w_reshape = np.reshape(w_disp,(10,28,28))
for img in range(10):
    print('Digit',img)
    plt.imshow(w_reshape[img,:,:])
    pyplot.show()
```

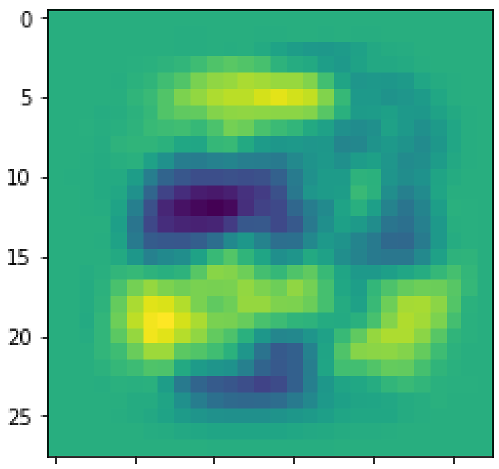
```
Digit 0
```



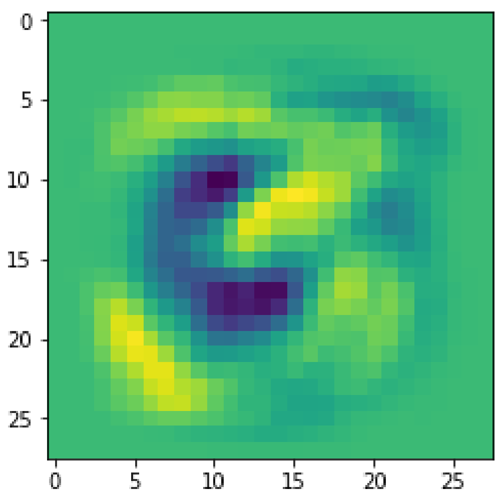
```
Digit 1
```



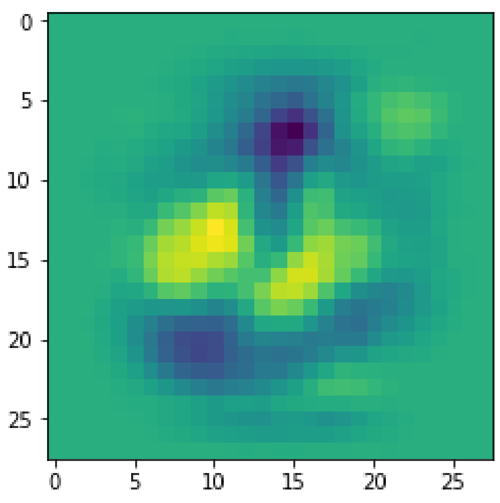
```
Digit 2
```



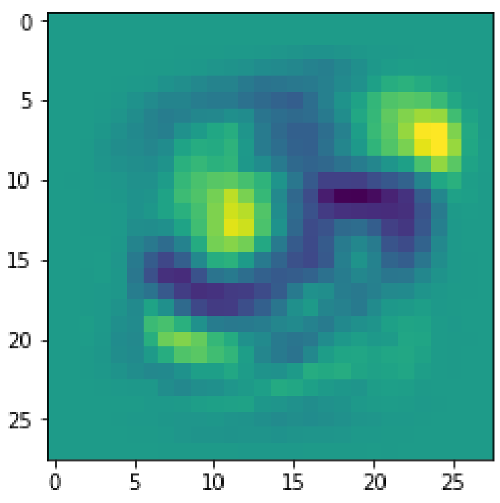
```
Digit 3
```



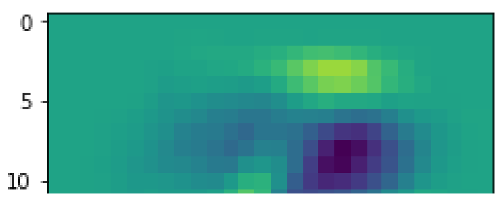
```
Digit 4
```

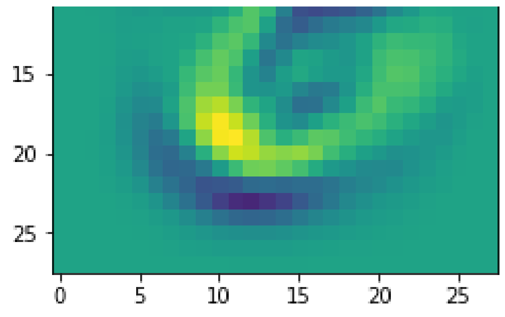


```
Digit 5
```

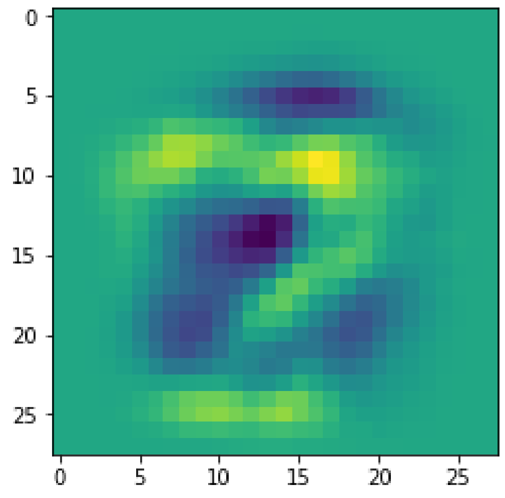


```
Digit 6
```

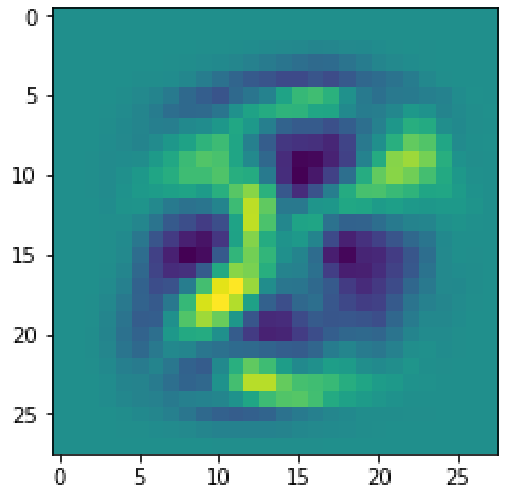




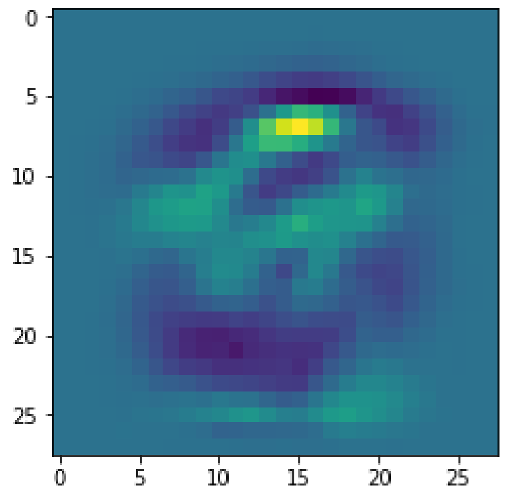
```
Digit 7
```



```
Digit 8
```



```
Digit 9
```



The above weight images for each class makes it evident that the feature influence is determined by an assumed probablistic distribution on the dataset but on the weights calculated by maximum log liklihood function.The iterative updates helps us to reach at better weights for each classification.The class weight images also seem to weigh the featues so as to look like the class digit.But the weights are not simply heavy at those features like the Naive Bayes' model but also give a range of weights within that 'likly digit' area.

Thus this model is better compared to Naive Bayes when a good regularizer coefficient and regularizer rate is chosen.