I tried multiple different ways at the multi digit classification including for loop method, matrix method and a combination of both. My for loop method was extremely logical and was built directly from the slides and everything made so much sense and it flowed really well but it was so slow. It took about 60 seconds to do one single iteration for one beta value. The matrix version was much faster but I couldn't get above a classification accuracy of around %94.

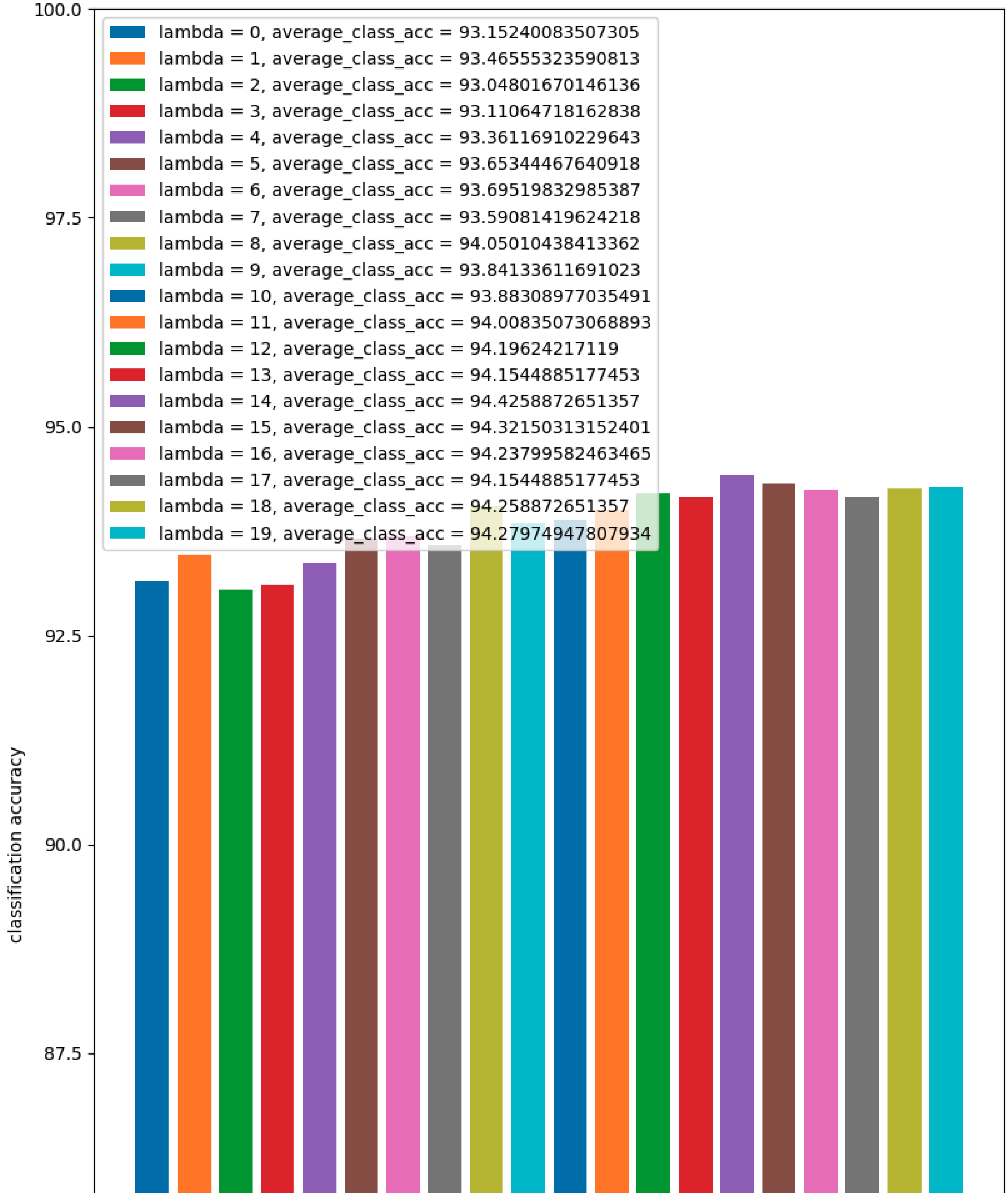
It was also unclear whether that meant my algorithm was working or not. I ended up assuming that it was but there was no final output or example prediction once the algorithm was complete. Below is the algorithm snippet

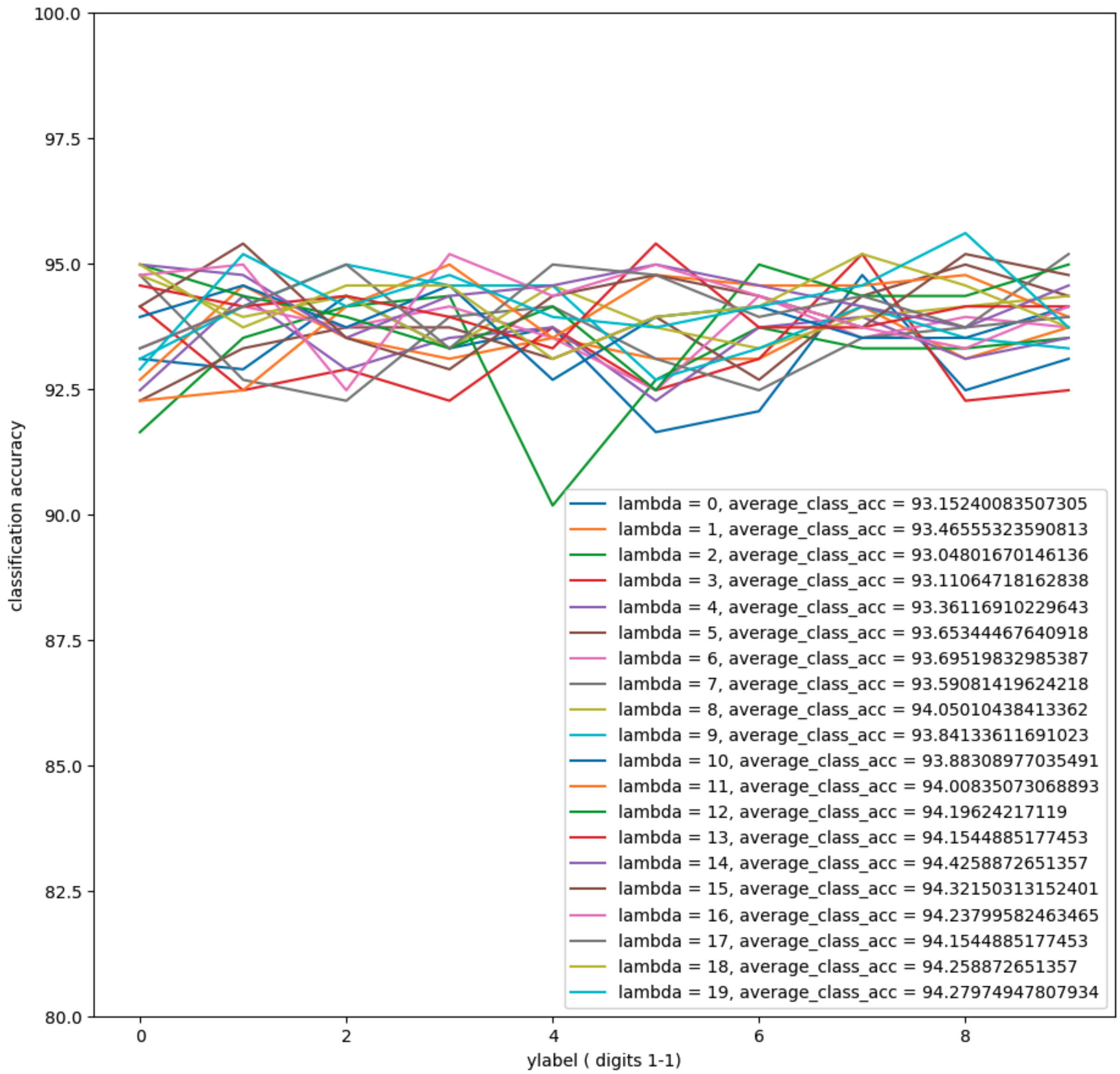
```python
##############################################################################
# The logistic function
def logistic(x,beta):
    y_pred = np.matmul(x,beta)  #no use of bias term as indicated in assignment instructions
    logistic_prob = 1/(1 + np.exp(-y_pred))
    return logistic_prob

##############################################################################
# Here you create you own LOGISTIC REGRESSION algorith for the logistic regression model
# Hint: please check the slides of the logistic regression
##############################################################################

def logistic_regression(beta, lr, x_batch, y_batch,lambda1):
    start = time.time()
    cost = -np.sum(np.dot(y_batch , np.log(logistic(x_batch[:,:-1] , beta[:-1]))) + np.dot((1-y_batch) , np.log(1 - logistic(x_batch[:,:-1] , beta[:-1]))))

    #FOR LOOP APPROACH
    #this could work...
    '''DNNL_list = []
    for j in range(len(beta)):
        current_DNNL = 0
        for i in range(len(x_batch)):
            current_DNNL +=  x_batch[i,j] * (y_batch[i] - logistic(x_batch[i,:-1] , beta[:-1])) + lambda1 * beta[j]

            print("{}          {}".format(time.time() - start, current_DNNL))
        DNNL_list.append(-current_DNNL)


    DNNL_list = np.array(DNNL_list)

    beta_next = beta - learning_rate * DNNL_list
    print("beta_next shape = ", beta_next.shape)'''
    ##############################################################
    """#MATRIX APPROACH
    DNNL_list = []
    for j in range(len(beta)):
        c_DNNL = np.sum(x_batch[:,j] * (y_batch - logistic(x_batch, beta)))  +  lambda1 *beta[j]
        DNNL_list.append(-c_DNNL)

    DNNL_list = np.array(DNNL_list)

    beta_next = beta - learning_rate * DNNL_list
    print("beta_next shape = ", beta_next.shape)"""

    #different approach
    #cost = -np.sum(np.dot(y_batch , np.log(logistic(x_batch[:,:-1] , beta[:-1]))) + np.dot((1-y_batch) , np.log(1 - logistic(x_batch[:,:-1] , beta[:-1]))))
    p = logistic(x_batch,beta.T)

    cost = -np.sum(y_batch * np.log(p) + (1-y_batch)* np.log(1-p))
    regularization = (lambda1/2) * np.sum(beta**2)
    cost += regularization

    dcost =   -np.sum(x_batch.T * (y_batch - p), axis = 1) + lambda1 * beta
    beta_next = beta - learning_rate * dcost


    return cost, beta_next
```

I was also forced to edit the main function in order to store the classification accuracies from each digit as well as to perform the algorithm 20 times for 20 different lambdas. As well as create a plotting function to visually understand the output of my algorithm. That code is shown below:

```python
##############################################################################
#                       Main Function
##############################################################################

# Hyper-parameters
training_epochs = 100
learning_rate = 0.0005          # The optimization initial learning rate
lambda1 = 1                     # The regularization parameter
cost = 0
FINAL = []
for i in range(0,20): #loop through lambda
    lambda1 = i
    classification_list = []
    for i in range(10):
        current_label = data_y[:,0]
        print("current label (1-10)", current_label.shape)
        beta = np.random.randn(dim_images + 1)
        for epoch in range(training_epochs):
            cost, beta_next = logistic_regression(beta,learning_rate,data_x,current_label, lambda1)
            ratio = classifcation_ratio(beta,data_x,current_label)

            print('Class %d Epoch %3d, cost %.3f, the classification accuracy is %.2f%%' % (i+1, epoch+1,cost,ratio*100))
            beta = beta_next
        classification_list.append(ratio*100)
    FINAL.append(classification_list)

##############################################################################
# Visualizing five images of the MNIST dataset
fig, _axs = plt.subplots(nrows=1, ncols=5)
axs = _axs.flatten()

for i in range(5):
    axs[i].grid(False)
    axs[i].set_xticks([])
    axs[i].set_yticks([])
    image = data_x[i*10,:784]
    image = np.reshape(image,(28,28))
    aa = axs[i].imshow(image,cmap=plt.get_cmap("gray"))

fig.tight_layout()


#MY DISPLAY
fig, ax = plt.subplots()

for i in range(len(FINAL)): #should be 20 lists for each lambda
    ax.bar(i, np.mean(FINAL[i]), label = "lambda = {}, average_class_acc = {}".format(i,np.mean(FINAL[i])))
    ax.set_ylim([80,100])



ax.set_ylabel("classification accuracy")
ax.set_xlabel("ylabel ( digits 1-1)")
ax.legend()
plt.show()
```

I found the best way to visualize this was to average all 10 classification accuracies for each lambda. One can see from the plot that as we increase the lambda regularzation term the accuracy of the classification slightly increases.This plot is shown below: Also shown below is the plot but not averaged…


100.0
97.5
95.0
92.5
90.0
87.5
classification accuracy
lambda = 0, average_class_acc = 93.15240083507305
lambda = 1, average_class_acc = 93.46555323590813
lambda = 2, average_class_acc = 93.04801670146136
lambda = 3, average_class_acc = 93.11064718162838
lambda = 4, average_class_acc = 93.36116910229643
lambda = 5, average_class_acc = 93.65344467640918
lambda = 6, average_class_acc = 93.69519832985387
lambda = 7, average_class_acc = 93.59081419624218
lambda = 8, average_class_acc = 94.05010438413362
lambda = 9, average_class_acc = 93.84133611691023
lambda = 10, average_class_acc = 93.88308977035491
lambda = 11, average_class_acc = 94.00835073068893
lambda = 12, average_class_acc = 94.19624217119
lambda = 13, average_class_acc = 94.1544885177453
lambda = 14, average_class_acc = 94.4258872651357
lambda = 15, average_class_acc = 94.32150313152401
lambda = 16, average_class_acc = 94.23799582463465
lambda = 17, average_class_acc = 94.1544885177453
lambda = 18, average_class_acc = 94.258872651357
lambda = 19, average_class_acc = 94.27974947807934



100.0
97.5
95.0
92.5
90.0
87.5
85.0
82.5
80.0
classification accuracy
0
2
4
6
8
ylabel (digits 1-1)
lambda = 0, average_class_acc = 93.15240083507305
lambda = 1, average_class_acc = 93.46555323590813
lambda = 2, average_class_acc = 93.04801670146136
lambda = 3, average_class_acc = 93.11064718162838
lambda = 4, average_class_acc = 93.36116910229643
lambda = 5, average_class_acc = 93.65344467640918
lambda = 6, average_class_acc = 93.69519832985387
lambda = 7, average_class_acc = 93.59081419624218
lambda = 8, average_class_acc = 94.05010438413362
lambda = 9, average_class_acc = 93.84133611691023
lambda = 10, average_class_acc = 93.88308977035491
lambda = 11, average_class_acc = 94.00835073068893
lambda = 12, average_class_acc = 94.19624217119
lambda = 13, average_class_acc = 94.1544885177453
lambda = 14, average_class_acc = 94.4258872651357
lambda = 15, average_class_acc = 94.32150313152401
lambda = 16, average_class_acc = 94.23799582463465
lambda = 17, average_class_acc = 94.1544885177453
lambda = 18, average_class_acc = 94.258872651357
lambda = 19, average_class_acc = 94.27974947807934