Project 2
ESE 545, Data Mining: Learning from Massive Datasets
Team members: Shuairui Yao, Penn ID: 47673675; Zhefu Peng, Penn ID: 29419150

Introduction

The purpose of this project is to learn to process a massive dataset with programming techniques in practice, using Python to implement algorithms, PEGASOS and AdaGrad, which could help us develop better classifier.

Problem1 Data Importing and Splitting

As the problem mentions, we need to import data, and acquire the label of each article, making a vector only contains -1 or 1. What's more, we need to split a part of dataset (first 100,000 articles) as training set, the remaining part of dataset as test sets. The task is achieved by the following steps:

i. Load dataset, just as the project pdf mentioned.

```
from sklearn.datasets import fetch_rcv1
rcv1 = fetch_rcv1()
```

ii. Then, we take the 'target' part from dataset, using function 'todense' to convert it into matrix, using function 'reshape' to make all label become 1 row, then using function 'astype' to change the type to int8. Then the 0 should be 1, converting the label_vector to become a list.

```
target = rcv1['target'].todense()
label_vector = target[:,33].reshape(1,-1)[0].astype(np.int8)
label_vector[np.where(label_vector == 0)] = -1
label_vector = label_vector.tolist()[0]
```

iii. Splitting the dataset, the first 100,000 rows to be training set, the remaining rows to be test set.

```
train_set = rcv1['data'][0:100000,:]
test_set = rcv1['data'][100000:,:]
```

Problem2 Implement PEGASOS method on SVM to develop a classifier, based on training set.

Since we cannot use any library, we should write our own code, based on PEGASOS steps shown in note10, and the selection of parameters such as λ and batch size should be also shown. (The iteration number is 1000, we set) PEGASOS algorithm is shown as below:

INPUT: training set $S = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)\}$,
Regularization parameter λ,
Number of iterations T

INITIALIZE: Choose $\mathbf{w}_1$ s.t. $\|\mathbf{w}_1\| \leq 1/\sqrt{\lambda}$

FOR $t = 1, 2, \ldots, T$

Choose $A_t \subseteq S$

$A_t^+ = \{(\mathbf{x}, y) \in A_t : y\langle \mathbf{w}_t, \mathbf{x} \rangle < 1\}$

$\nabla_t = \lambda \mathbf{w}_t - \frac{\eta_t}{|A_t|} \sum_{(\mathbf{x},y) \in A_t^+} y\,\mathbf{x}$

$\eta_t = \frac{1}{t\lambda}$

$\mathbf{w}'_t = \mathbf{w}_t - \eta_t \nabla_t$

$\mathbf{w}_{t+1} = \min\left\{1, \frac{1/\sqrt{\lambda}}{\|\mathbf{w}'_t\|}\right\} \mathbf{w}'_t$

OUTPUT: $\mathbf{w}_{T+1}$

i. Writing down the code that how to evaluate and calculate the error during the pegasos algorithm, called ‘error_cal’.

```
def error_cal(w,x,y_real):
    y_predict = x.dot(w.transpose()).toarray().reshape(1,-1).tolist()[0]
    y_res = np.array(y_predict)*np.array(y_real)
    error = len(np.where(y_res<0)[0])/len(y_res)
    return error
```

ii. Writing a function called ‘pegasos’, to implement the pegasos algorithm, the most important is writing down the steps that focusing on wrong points and derivation.

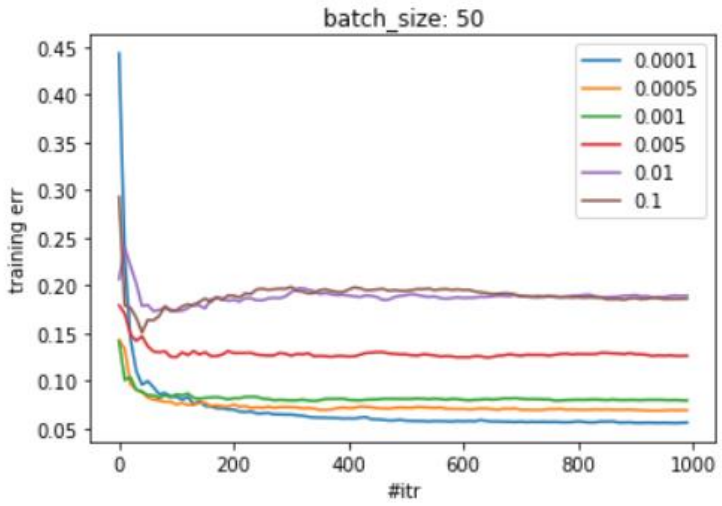
```
def pegasos(lamb,batch_size,T):
    err_all = []
    w = csr_matrix([0]*x_len) # x_len = 47236
    errors = []
    for t in range(1,T+1):
        summ = csr_matrix([0]*x_len) # x_len = 47236
        randidx = random.sample(range(0, 100000), batch_size)
        for i in randidx:
            res = w.dot(train_set[i,:].transpose())*label_vector[i]
            if((res.todense()<1).tolist()[0][0]):
                # at_plus
                summ = summ + train_set[i,:]*label_vector[i]
        grad_t = lamb * w - summ/batch_size
        ita_t = 1/t/lamb
        wt_prime = w - ita_t*grad_t
        w = min(1,1/sqrt(lamb)/norm(wt_prime))*wt_prime
        if(t%10==0):
            error = error_cal(w,train_set,label_vector[0:100000])
            errors.append(error)
    return w, errors
```

iii. This time we should show the how to select value of λ and batch size. But we should fix one of values. First, we fix batch size as 50, and select the value of λ from [0.0001,0.0005,0.001,0.005,0.01,0.1].

```
from scipy.sparse import csr_matrix
import random
from math import sqrt
from scipy.sparse.linalg import norm
import time

start = time.clock()

lambdas = [0.0001,0.0005,0.001,0.005,0.01,0.1]
batch_size = 50
T = 1000
err_all = []
for lamb in lambdas:
    w, err = pegasos(lamb,batch_size,T)
    err_all.append(err)

elapsed = (time.clock()-start)/60
print("minute used: ",elapsed)
```

```
minute used:  3.8875793666666665
```

```
import matplotlib.pyplot as plt
for i in range(len(lambdas)):
    plt.plot(range(0,T,10),err_all[i])
plt.legend(labels = lambdas, loc = 'best')
plt.xlabel('#itr')
plt.ylabel('training err')
plt.title("batch_size: %i" % batch_size)
plt.show()
```



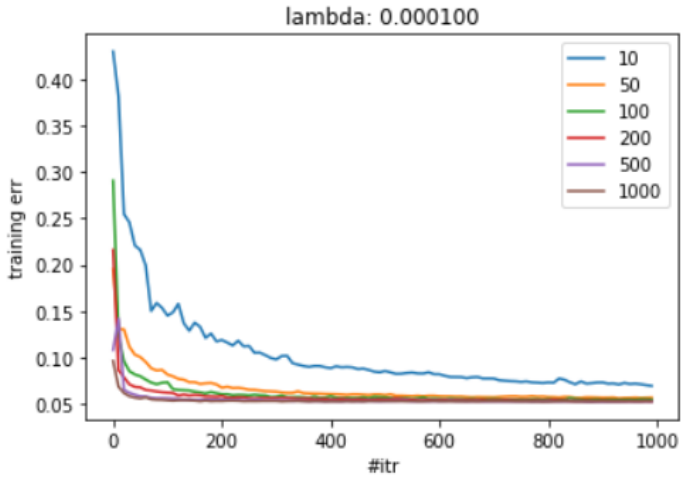
From the plot above, the lowest error could be reached when λ is 0.0001.

iv. Then we need to select the value of batch size, this time we fix the λ as 0.0001, choosing the batch size from [10,50,100,200,500,1000].

```
start = time.clock()

lamb = 0.0001 # select lambda to be 0.0001
batch_size = [10,50,100,200,500,1000]
T = 1000
err_all = []
for b_size in batch_size:
    w, err = pegasos(lamb,b_size,T)
    err_all.append(err)

elapsed = (time.clock()-start)/60
print("minute used: ",elapsed)
```

```
minute used:  17.024597183333334
```

```
import matplotlib.pyplot as plt
for i in range(len(batch_size)):
    plt.plot(range(0,T,10),err_all[i])
plt.legend(labels = batch_size, loc = 'best')
plt.xlabel('#itr')
plt.ylabel('training err')
plt.title("lambda: %f" % lamb)
plt.show()
```

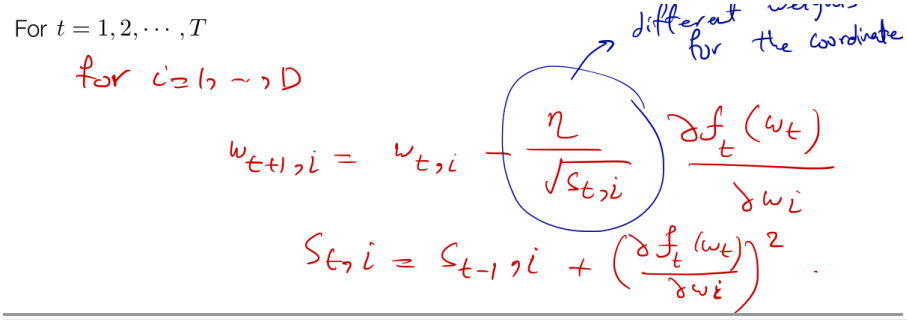


It turns out that the value of batch size should be 50.

Problem3 Implement AdaGrad method on SVM to develop a classifier, based on training set.

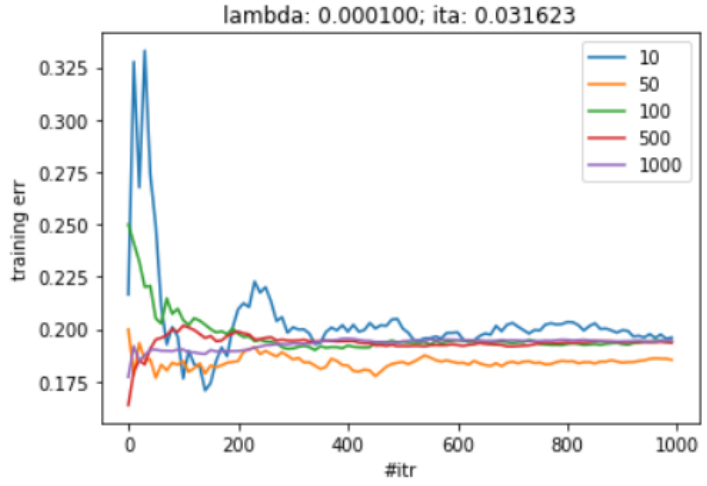
Since we cannot use any library, we should write our own code, based on AdaGrad steps shown in note10, and the selection of parameters such as λ, learning rate η and batch size should be also shown. (The iteration number is 1000, we set).

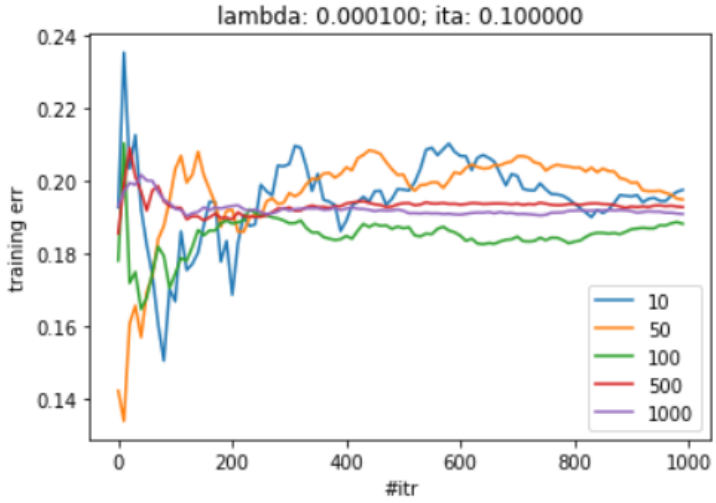
i. This time we could use the function 'error_cal' to evaluate the error among training, so we just need to write a function called 'adagrad' to implement the AdaGrad algorithm.
And the difference between AdaGrad and PEGASOS is :

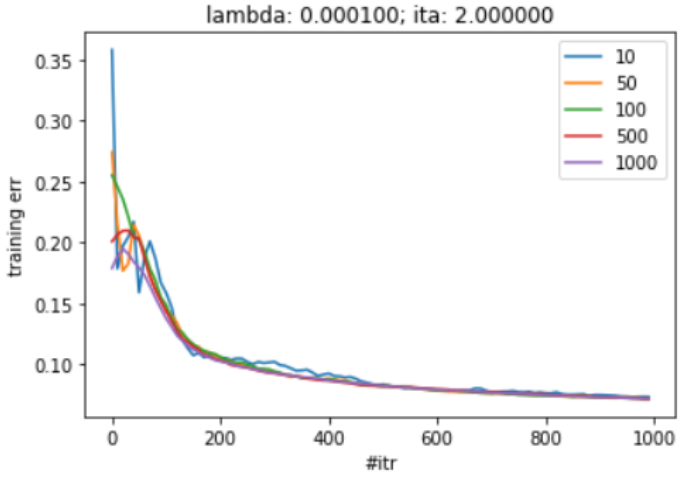



```
def adagrad(ita,lamb,batch_size,T):
    errors = []
    w = csr_matrix([0]*x_len) # x_len = 47236
    s = csr_matrix([1]*x_len)
    for t in range (1,T+1):
        summ = csr_matrix([0]*x_len) # x_len = 47236
        randidx = random.sample(range(0, 100000), batch_size)
        for i in randidx:
            res = w.dot(train_set[i,:].transpose())*label_vector[i]
            if((res.todense()<1).tolist()[0][0]):
                summ = summ + train_set[i,:]*label_vector[i] # at_plus
        grad_t = lamb * w - summ/batch_size
        s_sqrt = np.sqrt(s + grad_t.multiply(grad_t)) # 1*47236 csr_matrix
        wt_prime = w - csr_matrix(csr_matrix([ita]*x_len)/s_sqrt).multiply(grad_t)
        w = min(1,1/sqrt(lamb)/norm(s_sqrt.multiply(wt_prime)))*wt_prime
        if(t%10==0):
            error = error_cal(w,train_set,label_vector[0:100000])
            errors.append(error)
    return w, errors
```

ii. This time we first need to find the best value of batch size. Differing with what we did in problem2, we have to fix the value of λ, learning rate η in three sets, which are [λ = 0.0001, η = 1/sqrt(T)], [λ = 0.0001, η = 0.1], [λ = 0.0001, η = 2], to find the best value of batch size in [10,50,100,500,1000]. The plot would be shown as below:

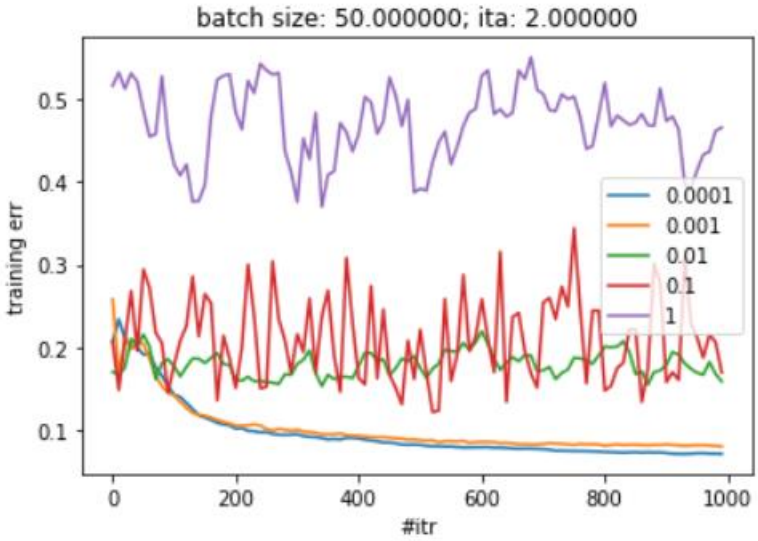






It turns out that we should select 50 to be the value of batch size, and 2 to be the value of η.

iii. Now we should choose the value of λ, we fix 50 to be the value of batch size, and 2 to be the value of η, and the selection of λ in [0.0001,0.001,0.01,0.1,1], the plot would be shown below:



So, the value of λ should be 0.0001.

Problem4 Training Neural Net.

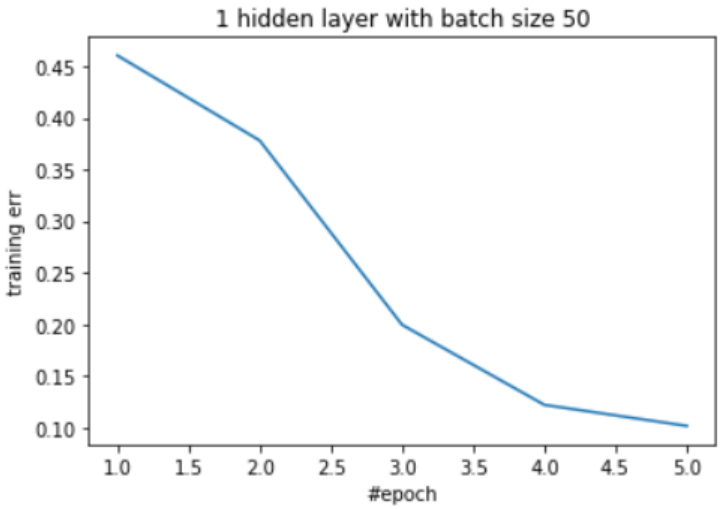
As the problem mentions, we are asked to train a neural net over 5 epochs, with 3 hidden layers, and each layer has 100 hidden units. Then, we are asked to design our own neural net, with appropriate parameters, and the limited time cost.

i. In order to use sigmoid function as output layer, we need to build a new label vector, called 'label_vector4', which only contain [0,1] labels.

```
# change the labels from -1 & 1 to 0 & 1
label_vector4 = np.array(label_vector)
label_vector4[label_vector4==-1]=0
label_vector4
```

ii. Then, for just 1 hidden layer, we used ReLU as the activation function of hidden layer, and the activation function of output layer we used sigmoid.

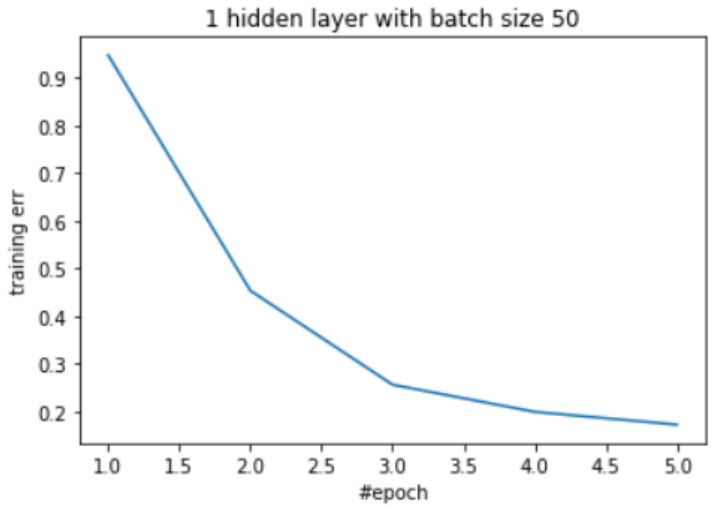
```
from keras.models import Sequential
from keras.layers import Dense
from keras import optimizers
import time

start = time.clock()

# 1 hidden layer (labels 0&1)
# create model
model = Sequential()
model.add(Dense(100, input_dim=x_len, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
# Compile model
# sgd = optimizers.SGD(lr=0.01,decay=0.0001)
model.compile(loss='binary_crossentropy', optimizer='sgd', metrics=['accuracy']) # binary_crossentropy
# Fit the model
h = model.fit(train_set, csr_matrix(label_vector4[0:100000]).transpose(), epochs=5, batch_size=50)

elapsed = (time.clock()-start)/60
print("minute used: ",elapsed)
```



The training accuracy is 89.9%.

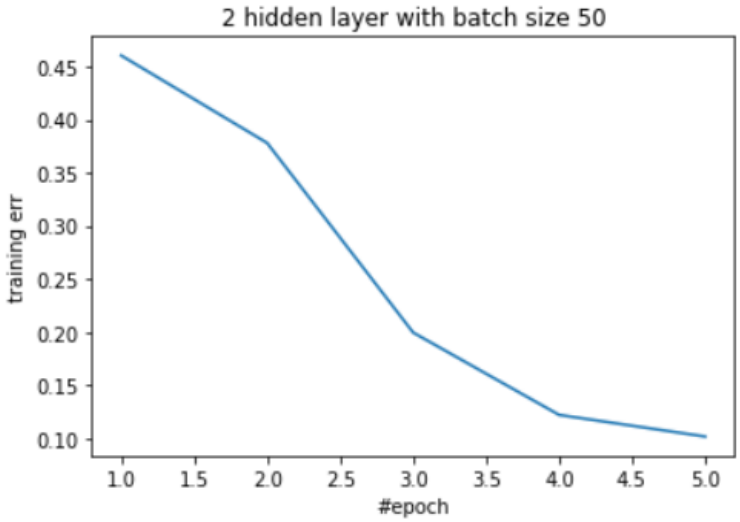
iii. Then, for just 1 hidden layer, we used ReLU as the activation function of hidden layer, and the activation function of output layer we used tanh.

```
start = time.clock()

# 1 hidden layer (labels -1&1)
# create model
model = Sequential()
model.add(Dense(100, input_dim=x_len, activation='relu'))
model.add(Dense(1, activation='tanh'))
# Compile model
# sgd = optimizers.SGD(lr=0.01,decay=0.0001)
model.compile(loss='mse', optimizer='sgd', metrics=['accuracy']) # binary_crossentropy
# Fit the model
h = model.fit(train_set, csr_matrix(label_vector[0:100000]).transpose(), epochs=5, batch_size=50)

elapsed = (time.clock()-start)/60
print("minute used: ",elapsed)
```



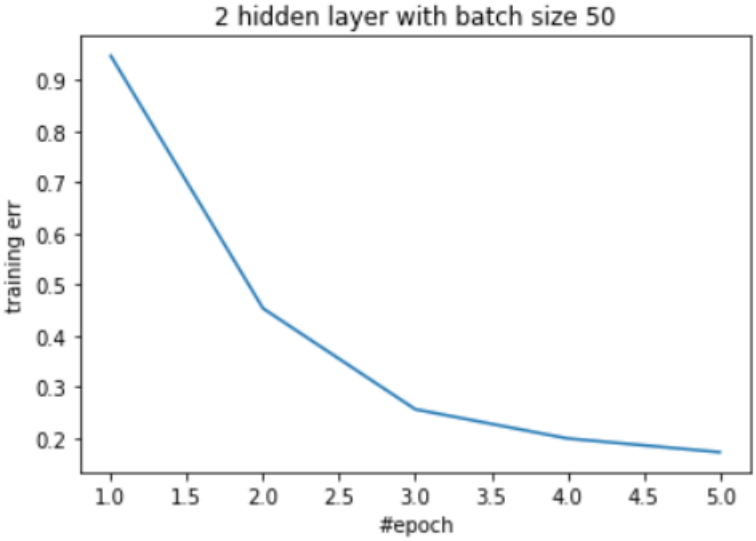
The training accuracy is 82%.

iv. For 2 and 3 hidden layers part, the code is similar as the 1 hidden layer part, so I won't paste the code in the following page. For 2 hidden layers, we used ReLU as the activation function of hidden layer, and the activation function of output layer we used sigmoid.



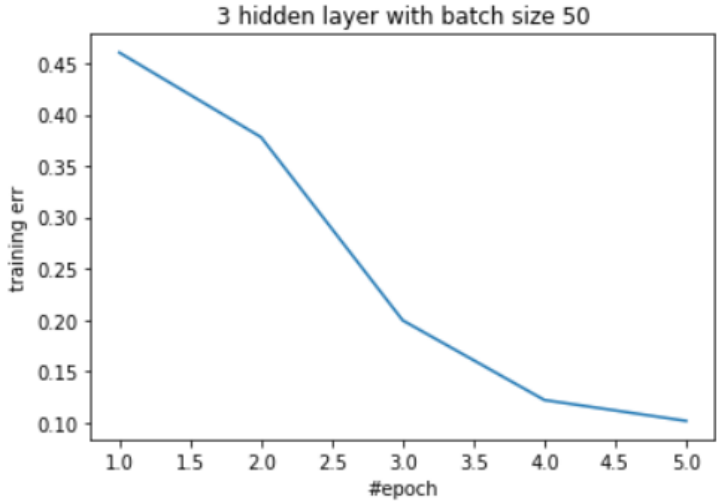
The training accuracy is 93.1%.

v. For 2 hidden layers, we used ReLU as the activation function of hidden layer, and the activation function of output layer we used tanh.



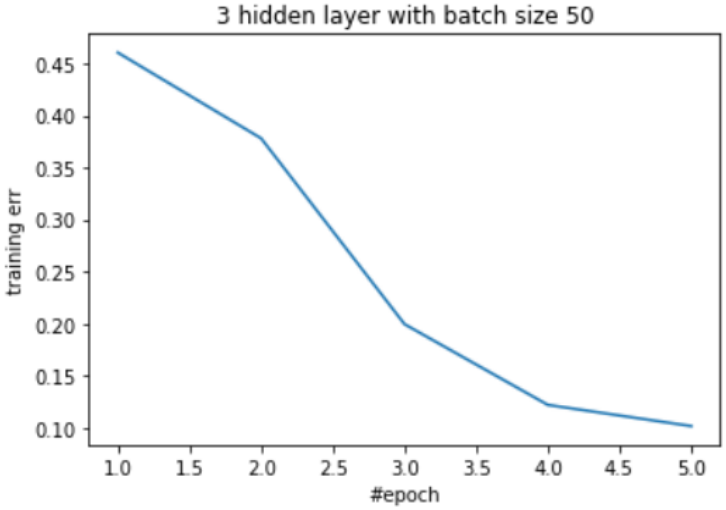
The training accuracy is 89.8%.

vi. For 3 hidden layers, we used ReLU as the activation function of hidden layer, and the activation function of output layer we used tanh.



The training accuracy is 91.6%.

vii. For 3 hidden layers, we used ReLU as the activation function of hidden layer, and the activation function of output layer we used sigmoid.



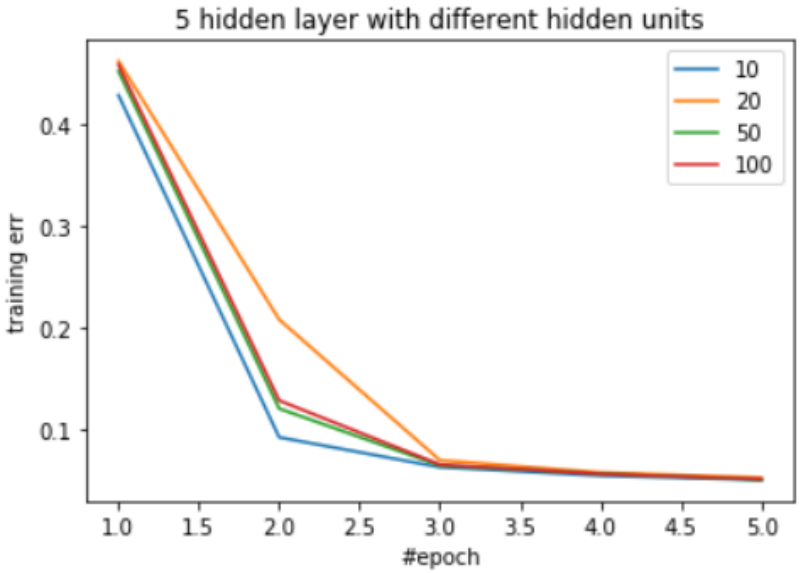
The training accuracy is 94.7%.

viii. Now we need to build our own neural net, and from the solution to question (a), we could find the performance of sigmoid function is better. So, we decided to use the label_vector4, which contains [0,1] labels. As a result, we need to fix the activation function of hidden layer is ReLU, and the activation function of output layer should be Sigmoid. The batch size is 50. The number of layers could be [3,5], and the number of units each hidden layer has [20, 50, 100]. In other words, there would be 6 combinations.

ix. Then, we first fix the number of hidden layers to be 5, and we should choose the number of units in [20,50,100]. The code is as following:

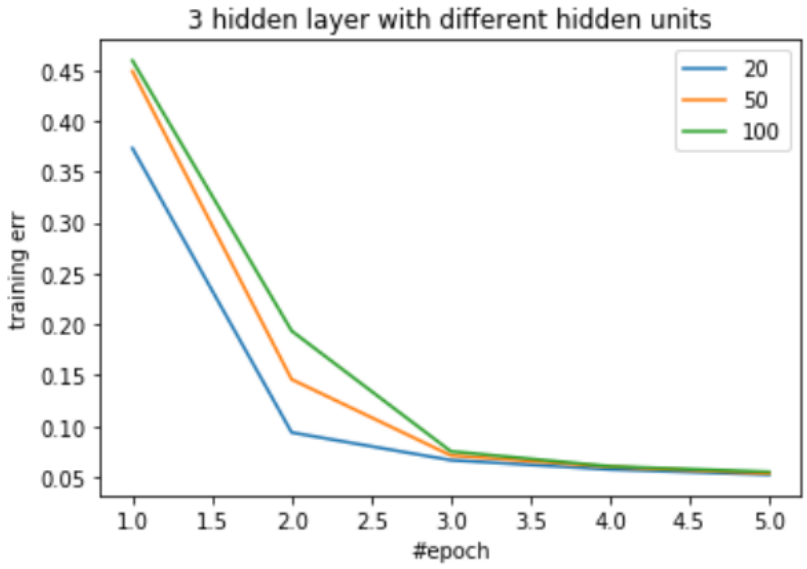
```
# 5 hidden layers
# hidden unit: 20,50,100
# create model
unit_no = [20,50,100]
h3 = []
for hidden in unit_no:
    model = Sequential()
    model.add(Dense(hidden, input_dim=x_len, activation='relu'))
    model.add(Dense(hidden, input_dim=hidden, activation='relu'))
    model.add(Dense(hidden, input_dim=hidden, activation='relu'))
    model.add(Dense(hidden, input_dim=hidden, activation='relu'))
    model.add(Dense(hidden, input_dim=hidden, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    # Compile model
    model.compile(loss='binary_crossentropy', optimizer='sgd', metrics=['accuracy'])
    # Fit the model
    h3.append(model.fit(train_set, label_vector4[0:100000], epochs=5, batch_size=50))
```

The plot is:



And the training accuracy for [20, 50, 100] are: 94.95%, 94.8%, 94.86%, showing that the best selection for the number of units is 20.

x. Now we change the number of hidden layers as 3, and still we should choose the number of units in [20,50,100]. The code is similar as before, so I won't paste here.




```
Epoch 1/5
100000/100000 [==============================] - 42s 425us/step - loss: 0.6859 - acc: 0.5389
Epoch 2/5
100000/100000 [==============================] - 42s 415us/step - loss: 0.4654 - acc: 0.7970
Epoch 3/5
100000/100000 [==============================] - 42s 418us/step - loss: 0.1873 - acc: 0.9303
Epoch 4/5
100000/100000 [==============================] - 45s 454us/step - loss: 0.1571 - acc: 0.9418
Epoch 5/5
100000/100000 [==============================] - 45s 452us/step - loss: 0.1439 - acc: 0.9463
```

The accuracy for 20, 50 and 100 units are: 94.78%, 94.67%, 94.51%. All are lower than five layers, so the number of hidden layers should be 5.

xi. In sum, the number of hidden layers should be 5, and the number of units each hidden layer has should be 20.

Problem5 Evaluate the Effect of Three Methods

We should use the parameter we selected before for three methods, implementing each of them into test sets. And analyze the result.

i. For the best PEGASOS, the λ should be 0.0001, and batch size should be 50.

```
# best PEGASOS
# w, err = pegasos(lamb,b_size,T)
start = time.clock()

w_pegasos, train_err = pegasos(0.0001,50,1000)
test_err = error_cal(w_pegasos,test_set,label_vector[100000:])

elapsed = (time.clock()-start)/60
print("minute used: ",elapsed)
```

```
minute used:  0.6209402333332643
```

```
test_err
```

```
0.06415261479754804
```

The testing error is 6.4%.

ii. For the best AdaGrad, the value of λ should be 0.0001, η should be 2 and batch size is 50.

```
# best AdaGrad
# w, err = adagrad(ita,lamb,b_size,T)
start = time.clock()

w_adagrad, train_err_ada = adagrad(2,0.0001,50,1000)
test_err_ada = error_cal(w_adagrad,test_set,label_vector[100000:])

elapsed = (time.clock()-start)/60
print("minute used: ",elapsed)
print(test_err_ada)
```

```
minute used:  0.8351269666665757
0.07454423109137524
```

The testing error is 7.5%.

iii. For the best NN, the number of hidden layers should be 5, the number of units each layer should have is 20.

```
# best NN
# 5 hidden layers
# hidden unit: 20
# create model
hidden = 20
model = Sequential()
model.add(Dense(hidden, input_dim=x_len, activation='relu'))
model.add(Dense(hidden, input_dim=hidden, activation='relu'))
model.add(Dense(hidden, input_dim=hidden, activation='relu'))
model.add(Dense(hidden, input_dim=hidden, activation='relu'))
model.add(Dense(hidden, input_dim=hidden, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
# Compile model
model.compile(loss='binary_crossentropy', optimizer='sgd', metrics=['accuracy'])
# Fit the model
h_best = model.fit(train_set, label_vector4[0:100000], epochs=5, batch_size=50)
```

```
Epoch 1/5
100000/100000 [==============================] - 40s 404us/step - loss: 0.6706 - acc: 0.5740
Epoch 2/5
100000/100000 [==============================] - 41s 406us/step - loss: 0.2954 - acc: 0.9055
Epoch 3/5
100000/100000 [==============================] - 42s 417us/step - loss: 0.1731 - acc: 0.9369
Epoch 4/5
100000/100000 [==============================] - 42s 416us/step - loss: 0.1540 - acc: 0.9434
Epoch 5/5
100000/100000 [==============================] - 41s 410us/step - loss: 0.1403 - acc: 0.9489
```

```
start = time.clock()

scores = model.evaluate(test_set,label_vector4[100000:])
print('Test loss:', scores[0])
print('Test accuracy:', scores[1])

elapsed = (time.clock()-start)/60
print("minute used: ",elapsed)
```

```
704414/704414 [==============================] - 172s 244us/step
Test loss: 0.15970506702175385
Test accuracy: 0.9401388955931563
minute used:  3.077528766666607
```

The testing error is 5.99%.

In conclusion, the performance between PEGASOS, AdaGrad and NN in this time trial is NN > PEGASOS > AdaGrad.
NN has the best performance is reasonable, since the features and the relation of features to label could be captured and found much more.
AdaGrad and PEGASOS are similar algorithm, both using mini-batch gradient descent, and AdaGrad does more optimization for the learning rate η, and increase the training speed, which should perform better than PEGASOS does in theory. However, this time AdaGrad performs worse than PEGASOS does. The reason for this situation maybe the parameters we select is not as appropriate as the best, or, the progress of optimization goes to a local minimum value, instead of local minimum value.