华南理工大学

South China University of Technology

The Experiment Report of Machine Learning

SCHOOL: SCHOOL OF SOFTWARE ENGINEERING

SUBJECT: SOFTWARE ENGINEERING

Author:
Shengyan Wen

Supervisor:
Qingyao Wu

Student ID:
201721045893

Grade:
Post Graduate

December 14, 2017

Logistic Regression, Linear Classification and Stochastic Gradient Descent

Abstract— In this experiment, we aim to compare and understand the difference between gradient descent and stochastic gradient descent. We compare logistic regression and linear classification. We aim to further understand the principles of SVM and practice on larger data.

I. Introduction

The motivation of the experiment is compare and understand the difference between gradient descent and stochastic gradient descent, and compare Logistic regression and linear classification, and finally understand the principles of SVM and practice on larger data.

Logic regression uses 'a9a' in LIBSVM Data, including 32561/16281(testing) samples and each sample has 123/123 (testing) features. This time we load train set and validation set separately .

For logistic regression and stochastic gradient descent, all the experiment steps is as follows. After downloading, initialize the logistic regression model parameters with normal distribution.

Secondly, select the Log-like-hood loss as the loss function of logic regression with calculating its derivation

Thirdly compute the gradient of the loss function from partial samples.

Fourthly update the parameters using different optimized methods(NAG, RMSProp, AdaDelta and Adam).

Fifthly select the appropriate threshold, mark the sample whose predict scores greater than the threshold as positive, on the contrary as negative, and predict under validation set and get the different optimized method loss L_{NAG}, $L_{RMSProp}$, and L_{Adam}.

Finally repeat step 3 to 5 for several times, and drawing graph of L_{NAG}, $L_{RMSProp}$, and L_{Adam} with the number of iterations.

For linear classification and stochastic gradient descent, the experiment steps is almost as same as logistic regression. The difference is that at step 1 initialize SVM model parameters with normal distribution, and at step 2 select the Hinge loss as the loss function of linear classification with calculating its derivation.

II. Methods and Theory

A)Logistic regression and stochastic gradient descent

We defined the loss function of the linear regression to be Log-likehood loss:

$$L = \frac{1}{n}\sum_{i=1}^{n} log\left(1 + e^{-y_i W^T x_i}\right) + \frac{\lambda}{2}\|W\|^2$$

The gradient of the loss function is:

$$G = \frac{1}{n}\sum_{i=1}^{n} \frac{-y_i x_i}{1 + e^{y_i W^T x_i}} + \lambda W$$

Where λ is the regularization parameter.

Update the parameter W use nesterov accelerated gradient(NAG):

$$v_t = \gamma\, v_{t-1} + \eta \nabla_W J(W - \gamma\, v_{t-1})$$

$$W = W - v_t$$

Where η is the learning rate, γ is the momentum.

Update the parameter W use RMSprop:

$$E[g^2]_t = 0.9E[g^2]_{t-1} + 0.1g_t^2$$

$$W_{t+1} = W_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

Where η is the learning rate, ϵ is the smoothing term.

Update the parameter W use AdaDelta:

$$\Delta W_t = -\frac{RMS[\Delta W]_{t-1}}{RMS[g]_t} g_t$$

$$W_{t+1} = W_t + \Delta W_t$$

Where

$$RMS[g]_t = \sqrt{E[g^2]_t + \epsilon}$$

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2$$

And

$$RMS[\Delta W]_t = \sqrt{E[\Delta W^2]_t + \epsilon}$$

$$E[\Delta W^2]_t = \gamma E[\Delta W^2]_{t-1} + (1-\gamma) \Delta W_t^2$$

$$\Delta W_t = -\frac{\eta}{RMS[g]_t} g_t$$

Update the parameter W use Adam:

$$W_{t+1} = W_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

Where

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1}$$

And

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2}$$

B)Linear classification and stochastic gradient descent

We defined the loss function of the linear classification to be Hinge loss:

$$L = \frac{\lambda}{2}\|W\|^2 + \frac{1}{n}\sum_{i=1}^{n} max\left(0, 1 - y_i(W^T x_i + b)\right)$$

The gradient of the loss function is:

$$G_W = \begin{cases} \lambda W & y_i(W^T x_i + b) \geq 1 \\ \lambda W + \frac{1}{n}\sum_{i=1}^{n} -y_i x_i & y_i(W^T x_i + b) < 1 \end{cases}$$

$$G_b = \begin{cases} 0 & y_i(W^T x_i + b) \geq 1 \\ \frac{1}{n}\sum_{i=1}^{n} -y_i & y_i(W^T x_i + b) < 1 \end{cases}$$

Update the parameters W use four optimized methods (NAG, RMSProp, AdaDelta and Adam), which are same as the logic classification.

III. Experiment

A) Logic regression and stochastic gradient descent

The code is listed as follows

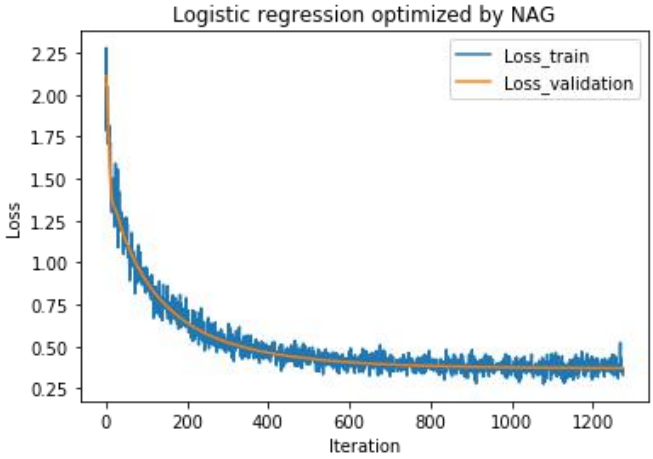
```
# Logistic regression
import math
import numpy as np
import matplotlib.pyplot as plt
from numpy import random
from sklearn.externals.joblib import Memory
from sklearn.datasets import load_svmlight_file
from sklearn.model_selection import train_test_split

#load dataset
def get_test_data():
    data = load_svmlight_file("a9a.test")
    return data[0], data[1]
def get_val_data():
    data = load_svmlight_file("a9a.validation")
    return data[0], data[1]

X_train, y_train = get_test_data()
X_train = X_train.toarray()
X_validation, y_validation = get_val_data()
X_validation = X_validation.toarray()
print(X_train.shape)
print(X_validation.shape)
(32561, 123)
(16281, 122)
In [64]:
#X_train = [X_train, 1]
addone_train = np.ones( X_train.shape[0])
X_train = np.column_stack((X_train,addone_train))
print(X_train.shape)

#X_validation = [X_validation,0,1]
addzero = np.zeros(( X_validation.shape[0]))
X_validation = np.column_stack((X_validation,addzero))
addone = np.ones( X_validation.shape[0])
X_validation = np.column_stack((X_validation,addone))
print(X_validation.shape)
(32561, 124)
(16281, 124)
In [65]:
# Initialize with normal distribution
N = X_train.shape[1]
W_normal = np.random.normal(size=N)
In [66]:
#Log-likehood loss function
def cal_Loss(X,W,y,lambdal):
    preY = np.dot(X,W)
    Loss = (np.sum(np.log(1 + np.exp(-y * preY))))/ X.shape[0] + lambdal / 2 * np.dot(W,W.T)
    return Loss
```

```
  #calculate gradient
  def cal_G(X,W,y,lambdal):
      preY = np.dot(X,W)
      G = (np.dot(((-y)/ (1+ np.exp(y*preY))),X ))/ X.shape[0] + W * lambdal
      return G
#shuffles the array
  def shuffle_array(X_train):
      randomlist = np.arange(X_train.shape[0])
      np.random.shuffle(randomlist)
      X_random = X_train[randomlist]
      y_random = y_train[randomlist]
      return X_random,y_random
  #get the training instance and label in current batch
  def get_Batch(runs,X_random,y_random,batch_size,shape):
      if l == runs-1:
          X_batch = X_random[l*batch_size:shape+1]
          y_batch = y_random[l*batch_size:shape+1]
      else:
          X_batch = X_random[l*batch_size:(l+1)*batch_size]
          y_batch = y_random[l*batch_size:(l+1)*batch_size]
      return X_batch,y_batch
#NAG
lr = 0.02
epoch = 5
gamma = 0.9
lambdal = 0.01
batch_size = 128 # mini-batch gradient descent
runs = math.ceil(X_train.shape[0] / float(batch_size))
iteration = epoch * runs
#get different kinds of initial data （W_zeros,W_random or W_normal）
W = W_normal
Loss_train = np.zeros(iteration)
Loss_validation = np.zeros(iteration)
v_t = np.zeros(N)
for j in range(0,epoch):
   #shuffles the array
   X_random,y_random = shuffle_array(X_train)
   for l in range(0,runs):
      #get the training instance and label in current batch
      X_batch,y_batch = get_Batch(runs,X_random,y_random,batch_size,X_train.shape[0])
      #approximate W in the next time step
      W_t = W - v_t * gamma
      #the training loss
      Loss_train[j*runs+l] = cal_Loss(X_batch,W,y_batch,lambdal)
      #the gradient of the loss function
      G = cal_G(X_batch,W_t,y_batch,lambdal)
      #the validation loss
      Loss_validation[j*runs+l] = cal_Loss(X_validation,W,y_validation,lambdal)
      #update the parameter W
      v_t = v_t * gamma + G * lr
      W = W - v_t
#draw the result
plt.plot(Loss_train,label="Loss_train")
plt.plot(Loss_validation,label="Loss_validation")
plt.legend()
plt.xlabel("Iteration")
plt.ylabel("Loss")
plt.title("Logistic regression optimized by NAG")
plt.show()

#RMSprop
plt.close()
lr = 0.062
epoch = 5
lambdal = 0.01
epsilon = np.e**(-8)
batch_size = 128 # mini-batch gradient descent
runs = math.ceil(X_train.shape[0] / float(batch_size))
iteration = epoch * runs
#get different kinds of initial data （W_zeros,W_random or W_normal）
W = W_normal
Loss_train = np.zeros(iteration)
Loss_validation = np.zeros(iteration)
#the sum of the square of the gradient
G_2 = 0
for j in range(0,epoch):
   #shuffles the array
   X_random,y_random = shuffle_array(X_train)
   for l in range(0,runs):
      #get the training instance and label in current batch
      X_batch,y_batch = get_Batch(runs,X_random,y_random,batch_size,X_train.shape[0])
      #the training loss
      Loss_train[j*runs+l] = cal_Loss(X_batch,W,y_batch,lambdal)
      #the gradient of the loss function
      G = cal_G(X_batch,W,y_batch,lambdal)
      #the validation loss
      Loss_validation[j*runs+l] = cal_Loss(X_validation,W,y_validation,lambdal)
      #update the parameter W
      G_2 = G_2 * 0.9 + np.dot(G,G.T) * 0.1
      W = W - G *(lr / math.sqrt(G_2 + epsilon))
```

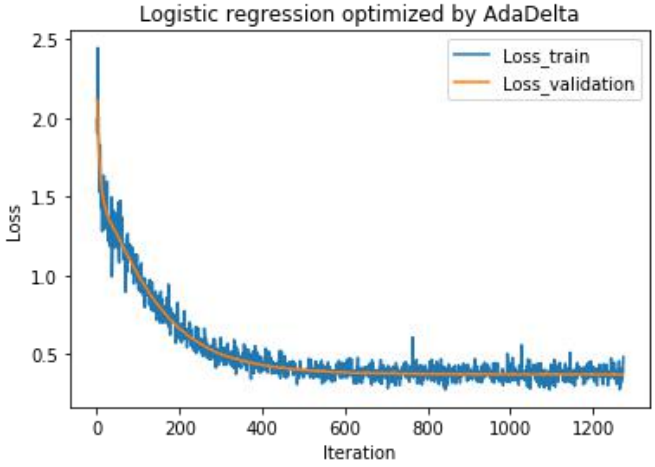
```
#draw the result
plt.plot(Loss_train,label="Loss_train")
plt.plot(Loss_validation,label="Loss_validation")
plt.legend()
plt.xlabel("Iteration")
plt.ylabel("Loss")
plt.title("Logistic regression optimized by RMSprop")
plt.show()
#AdaDelta
plt.close()
lr = 0.04
epoch = 5
lambdal = 0.01
gamma = 0.9
epsilon = np.e**(-8)
batch_size = 128 # mini-batch gradient descent
runs = math.ceil(X_train.shape[0] / float(batch_size))
iteration = epoch * runs
#get different kinds of initial data （W_zeros,W_random or W_normal）
W = W_normal
Loss_train = np.zeros(iteration)
Loss_validation = np.zeros(iteration)
#the sum of the square of the gradient
G_2 = 0
W_2 = 0
RMS_g = 0
RMS_W = 0
W_delta = np.zeros(N)
for j in range(0,epoch):
    #shuffles the array
    X_random,y_random = shuffle_array(X_train)
    for l in range(0,runs):
        #get the training instance and label in current batch
        X_batch,y_batch = get_Batch(runs,X_random,y_random,batch_size,X_train.shape[0])
        #the training loss
        Loss_train[j*runs+l] = cal_Loss(X_batch,W,y_batch,lambdal)
        #the gradient of the loss function
        G = cal_G(X_batch,W,y_batch,lambdal)
        #the validation loss
        Loss_validation[j*runs+l] = cal_Loss(X_validation,W,y_validation,lambdal)
        #update the parameter W
        G_2 = G_2 * gamma + np.dot(G,G.T) * (1-gamma)
        RMS_g = math.sqrt(G_2 + epsilon)
        W = W - G *(RMS_W / RMS_g)
        W_delta = G *(- lr / RMS_g)
        W_2 = W_2 * gamma + np.dot(W_delta,W_delta.T) * (1-gamma)
        RMS_W = math.sqrt(W_2 + epsilon)
#draw the result
plt.plot(Loss_train,label="Loss_train")
plt.plot(Loss_validation,label="Loss_validation")
plt.legend()
plt.xlabel("Iteration")
plt.ylabel("Loss")
plt.title("Logistic regression optimized by AdaDelta")
plt.show()

#Adam
plt.close()
lr = 0.08
epoch = 5
lambdal = 0.01
beta1 = 0.9
beta2 =0.999
epsilon = np.e**(-8)
batch_size = 128 # mini-batch gradient descent
runs = math.ceil(X_train.shape[0] / float(batch_size))
iteration = epoch * runs
#get different kinds of initial data （W_zeros,W_random or W_normal）
W = W_normal
Loss_train = np.zeros(iteration)
Loss_validation = np.zeros(iteration)
#the estimates of the first and second moments
m_t = np.zeros(N)
n_t = 0
for j in range(0,epoch):
    #shuffles the array
    X_random,y_random = shuffle_array(X_train)
    for l in range(0,runs):
        #get the training instance and label in current batch
        X_batch,y_batch = get_Batch(runs,X_random,y_random,batch_size,X_train.shape[0])
        #the training loss
        Loss_train[j*runs+l] = cal_Loss(X_batch,W,y_batch,lambdal)
        #the gradient of the loss function
        G = cal_G(X_batch,W,y_batch,lambdal)
        #the validation loss
        Loss_validation[j*runs+l] = cal_Loss(X_validation,W,y_validation,lambdal)
        #update the parameter W
        m_t = m_t * beta1 + G * (1-beta1)
        n_t = n_t * beta2 + np.dot(G,G.T) * (1-beta2)
        hat_m = m_t * (1/(1-beta1))
        hat_n = n_t * (1/(1-beta2))
        W = W - hat_m * (lr/(math.sqrt(hat_n)+epsilon))
#draw the result
```

```
plt.plot(Loss_train,label="Loss_train")
plt.plot(Loss_validation,label="Loss_validation")
plt.legend()
plt.xlabel("Iteration")
plt.ylabel("Loss")
plt.title("Logistic regression optimized by Adam")
plt.show()
```

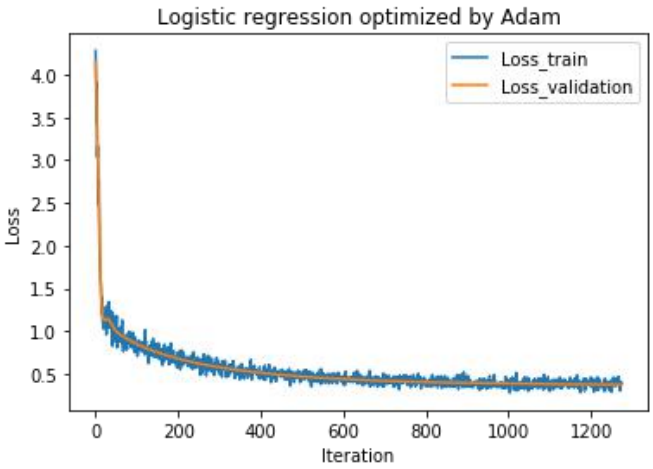
When applying NAG as the optimized method with specific parameters, the results is shown as follows:



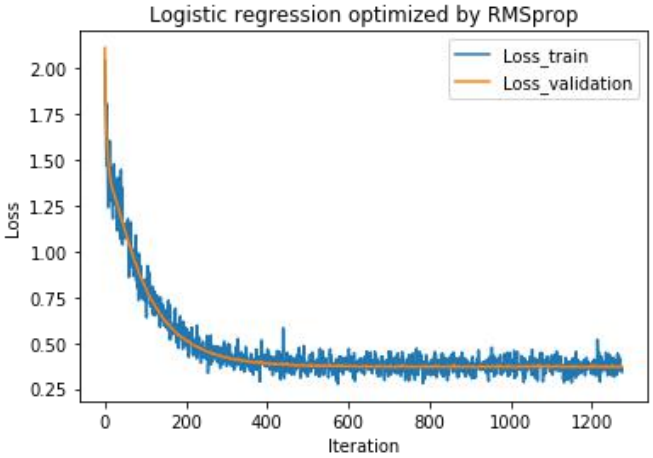
When applying AdaDelta as the optimized method with specific parameters, the results is shown as follows:



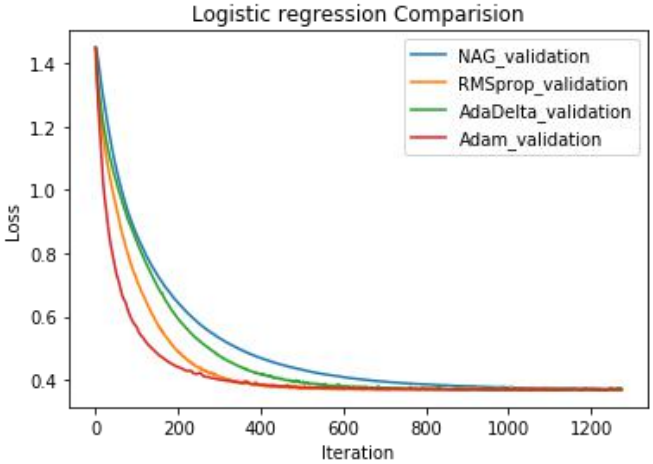
When applying Adam as the optimized method with specific parameters, the results is shown as follows:



When applying RMSProp as the optimized method with specific parameters, the results is shown as follows:



We compare the validation loss of four different optimization methods.



B) Linear classification and stochastic gradient descent

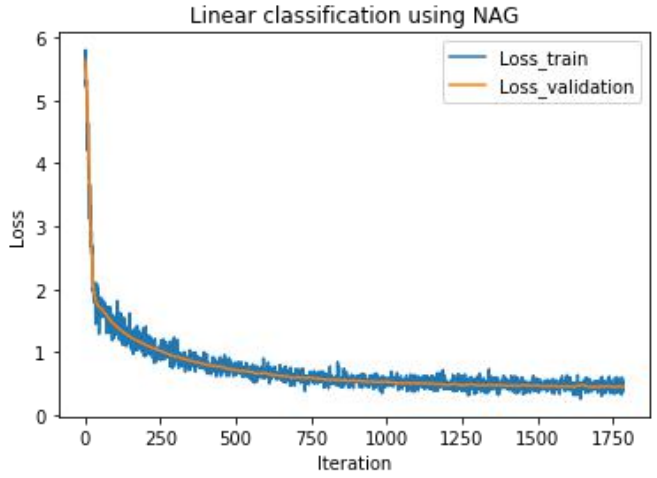
The code is listed as follows

```
...
...
#Hinge loss function
```

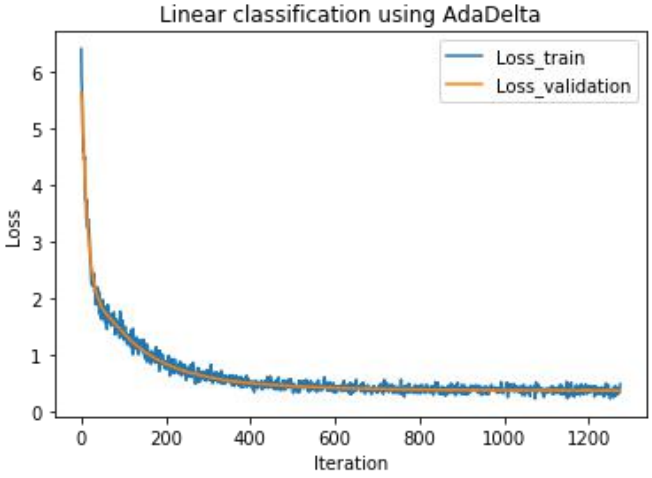
```
def cal_Loss(X,W,y,lambdaI,W_0):
    preY = np.dot(X,W)
    difY = np.ones(y.shape[0]) - y * preY
    difY[difY < 0] =0
    Loss =np.sum(difY) / X.shape[0] + np.dot(W_0,W_0.T)/2*lambdaI
    return Loss
#calculate the gradient
def cal_G(X,W,y,lambdaI,W_0):
    preY = np.dot(X,W)
    difY = np.ones(y.shape[0]) - y * preY
    y_get = y.copy()
    y_get[difY <= 0] =0
    G = -np.dot(y_get,X) / X.shape[0] + W_0 *lambdaI
    return G
...
...
#NAG
lr = 0.02
epoch = 7
gamma = 0.8
lambdaI = 0.01
batch_size = 128 # mini-batch gradient descent
runs = math.ceil(X_train.shape[0] / float(batch_size))
iteration = epoch * runs
#get different kinds of initial data （W_zeros,W_random or W_normal）
W = W_normal
Loss_train = np.zeros(iteration)
Loss_validation = np.zeros(iteration)
Accuracy = np.zeros(iteration)
v_t = np.zeros(N)
for j in range(0,epoch):
    #shuffles the array
    X_random,y_random = shuffle_array(X_train)
    for l in range(0,runs):
        #get the training instance and label in current batch
        X_batch,y_batch = get_Batch(runs,X_random,y_random,batch_size,X_train.shape[0])
        W_0 = W.copy()
        W_0[N-1]= 0
        #approximate W in the next time step
        W_t = W_0 - v_t * gamma
        #the training loss
        Loss_train[j*runs+l] = cal_Loss(X_batch,W,y_batch,lambdaI,W_0)
        #the gradient of the loss function
        G = cal_G(X_batch,W_t,y_batch,lambdaI,W_0)
        #the validation loss
        Loss_validation[j*runs+l] = cal_Loss(X_validation,W,y_validation,lambdaI,W_0)
        #update the parameter W,b
        v_t = v_t * gamma + G * lr
        W = W - v_t

#AdaDelta
lr = 0.05
epoch = 5
lambdaI = 0.01
gamma = 0.9
epsilon = np.e**(-8)
batch_size = 128 # mini-batch gradient descent
runs = math.ceil(X_train.shape[0] / float(batch_size))
iteration = epoch * runs
#get different kinds of initial data （W_zeros,W_random or W_normal）
W = W_normal
Loss_train = np.zeros(iteration)
Loss_validation = np.zeros(iteration)
Accuracy = np.zeros(iteration)
#the sum of the square of the gradient
G_2 = 0
W_2 = 0
RMS_g = 0
RMS_W = 0
W_delta = np.zeros(N)
for j in range(0,epoch):
    #shuffles the array
    X_random,y_random = shuffle_array(X_train)
    for l in range(0,runs):
        #get the training instance and label in current batch
        X_batch,y_batch = get_Batch(runs,X_random,y_random,batch_size,X_train.shape[0])
        W_0 = W.copy()
        W_0[N-1]= 0
        #the training loss
        Loss_train[j*runs+l] = cal_Loss(X_batch,W,y_batch,lambdaI,W_0)
        #the gradient of the loss function
        G = cal_G(X_batch,W,y_batch,lambdaI,W_0)
        #the validation loss
        Loss_validation[j*runs+l] = cal_Loss(X_validation,W,y_validation,lambdaI,W_0)
        #update the parameter W,b
        G_2 = G_2 * gamma + np.dot(G,G.T) * (1-gamma)
        RMS_g = math.sqrt(G_2 + epsilon)
        W = W - G *(RMS_W / RMS_g)
        W_delta = G *(- lr / RMS_g)
        W_2 = W_2 * gamma + np.dot(W_delta,W_delta.T) * (1-gamma)
        RMS_W = math.sqrt(W_2 + epsilon)
```

```
 #Adam
lr = 0.07
epoch = 4
lambdal = 0.01
beta1 = 0.9
beta2 =0.999
epsilon = np.e**(-8)
batch_size = 128 # mini-batch gradient descent
runs = math.ceil(X_train.shape[0] / float(batch_size))
iteration = epoch * runs
#get different kinds of initial data （W_zeros,W_random or W_normal）
W = W_normal
Loss_train = np.zeros(iteration)
Loss_validation = np.zeros(iteration)
Accuracy = np.zeros(iteration)
#the estimates of the first and second moments
m_t = np.zeros(N)
n_t = 0
for j in range(0,epoch):
    #shuffles the array
    X_random,y_random = shuffle_array(X_train)
    for l in range(0,runs):
        #get the training instance and label in current batch
        X_batch,y_batch = get_Batch(runs,X_random,y_random,batch_size,X_train.shape[0])
        W_0 = W.copy()
        W_0[N-1]= 0
        #the training loss
        Loss_train[j*runs+l] = cal_Loss(X_batch,W,y_batch,lambdal,W_0)
        #the gradient of the loss function
        G = cal_G(X_batch,W,y_batch,lambdal,W_0)
        #the validation loss
        Loss_validation[j*runs+l] = cal_Loss(X_validation,W,y_validation,lambdal,W_0)
        #update the parameter W,b
        m_t = m_t * beta1 + G * (1-beta1)
        n_t = n_t * beta2 + np.dot(G,G.T) * (1-beta2)
        hat_m = m_t * (1/(1-beta1))
        hat_n = n_t * (1/(1-beta2))
        W = W - hat_m * (lr/(math.sqrt(hat_n)+epsilon))

#RMSprop
plt.close()
lr = 0.08
epoch = 5
lambdal = 0.01
epsilon = np.e**(-8)
batch_size = 128 # mini-batch gradient descent
runs = math.ceil(X_train.shape[0] / float(batch_size))
iteration = epoch * runs
#get different kinds of initial data （W_zeros,W_random or W_normal）
W = W_normal
Loss_train = np.zeros(iteration)
Loss_validation = np.zeros(iteration)
Accuracy = np.zeros(iteration)
#the sum of the square of the gradient
G_2 = 0
for j in range(0,epoch):
    #shuffles the array
    X_random,y_random = shuffle_array(X_train)
    for l in range(0,runs):
        #get the training instance and label in current batch
        X_batch,y_batch = get_Batch(runs,X_random,y_random,batch_size,X_train.shape[0])
        W_0 = W.copy()
        W_0[N-1]= 0
        #the training loss
        Loss_train[j*runs+l] = cal_Loss(X_batch,W,y_batch,lambdal,W_0)
        #the gradient of the loss function
        G = cal_G(X_batch,W,y_batch,lambdal,W_0)
        #the validation loss
        Loss_validation[j*runs+l] = cal_Loss(X_validation,W,y_validation,lambdal,W_0)
        #update the parameter W,b
        G_2 = G_2 * 0.9 + np.dot(G,G.T) * 0.1
        W = W - G *(lr / math.sqrt(G_2 + epsilon))
```

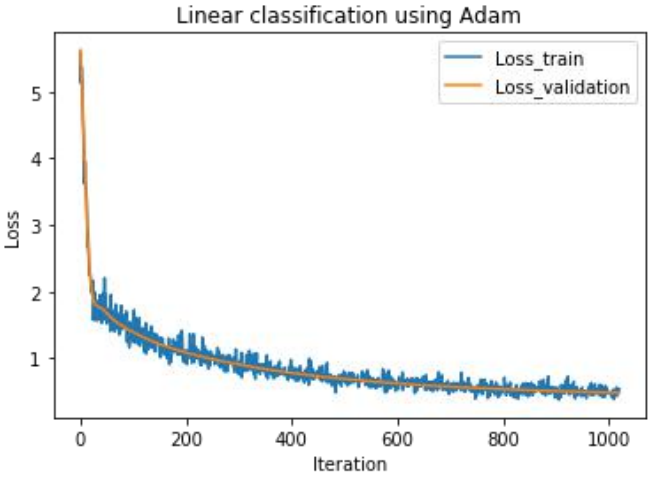
When applying NAG as the optimized method with specific parameters, the results is shown as follows:



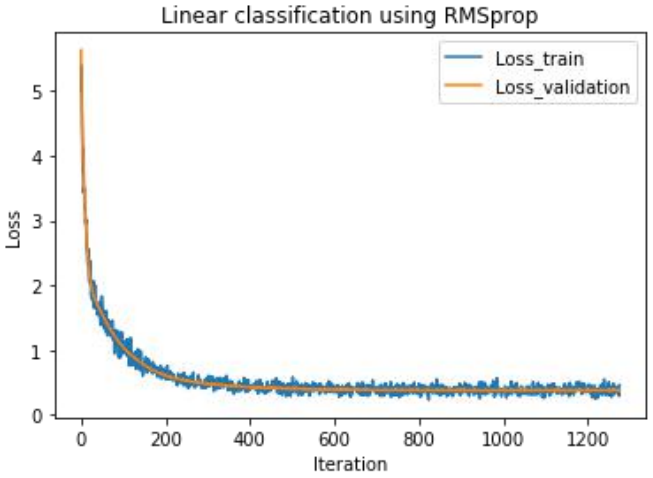
When applying AdaDelta as the optimized method with specific parameters, the results is shown as follows:



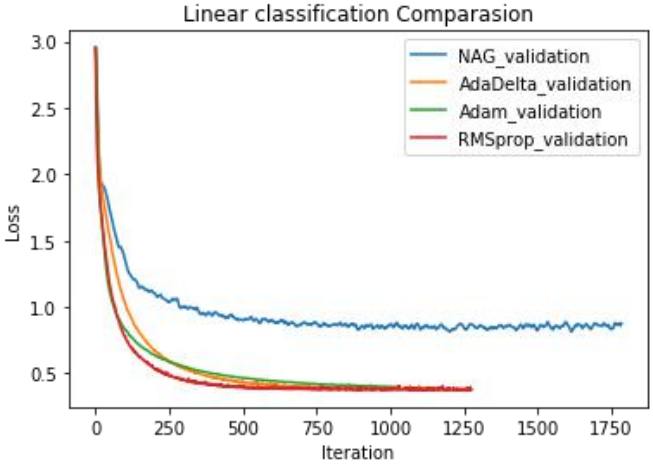
When applying Adam as the optimized method with specific parameters, the results is shown as follows:



When applying RMSProp as the optimized method with specific parameters, the results is shown as follows:



We compare the validation loss of four different optimization methods.



IV. CONCLUSION

For logic regression and stochastic gradient descent, we select Log-likehood loss. We compare the results under four different optimization methods. It seems that the result of using Adam method with learning rate 0.08 is the best.

For the linear classification and stochastic gradient descent, we select Hinge loss. We still compare the results under four different optimization methods. The results shows that linear classification and stochastic gradient descent is better than logic regression under each kind of optimization method. It seems that the result of using RMSProp method with learning rate 0.08 is the best.