

South China University of Technology

The Experiment Report of Machine Learning

SCHOOL: SCHOOL OF SOFTWARE ENGINEERING

SUBJECT: SOFTWARE ENGINEERING

Author: Supervisor: Shengyan Wen Qingyao Wu

tengyan wen

Student ID: Grade:

201721045893 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 LNAG, LRMSProp, and LAdam.

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

$$\begin{split} RMS[\Delta W]_t &= \sqrt{E\big[\Delta W^2\big]_t + \epsilon} \\ E\big[\Delta W^2\big]_t &= \gamma E\big[\Delta W^2\big]_{t-1} + (1-\gamma) \Delta W_t^2 \\ \Delta W_t &= -\frac{\eta}{RMS[g]_t} g_t \end{split}$$

Update the parameter W use Adam:

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

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:

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

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

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

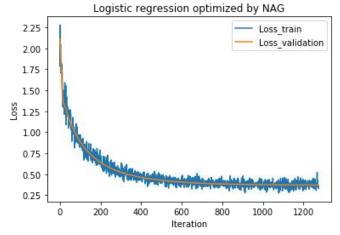
```
# Logistic regression
 import math
 import numpy as np
 import matplotlib.pyplot as plt
 from numpy import random
 from sklearn.externals.joblib import Memory
 from sklearn.datasets import load symlight file
             sklearn.model selection
train test split
  #load dataset
 def get test data():
    data = load symlight file("a9a.test")
    return data[0], data[1]
 def get val data():
    data = load symlight 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])
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 \text{ 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	Loss_train[j*runs+l] =
def cal_G(X,W,y,lambdal):	cal_Loss(X_batch,W,y_batch,lambdal)
preY = np.dot(X,W)	#the gradient of the loss function
G = (np.dot(((-y)/(1+ np.exp(y*preY))),X))/	$G = cal_G(X_batch, W_t, y_batch, lambdal)$
X.shape[0] + W * lambdal	#the validation loss
return G	Loss_validation[j*runs+l] =
#shuffles the array	cal_Loss(X_validation,W,y_validation,lambdal)
def shuffle_array(X_train):	#update the parameter W
randomlist = np.arange(X_train.shape[0])	v_t = v_t * gamma + G * lr
np.random.shuffle(randomlist)	$\overline{W} = W - v t$
X random = X train[randomlist]	#draw the result
y_random = y_train[randomlist]	plt.plot(Loss_train,label="Loss_train")
return X random,y random	plt.plot(Loss validation,label="Loss validation")
#get the training instance and label in current	plt.legend()
batch	plt.xlabel("Iteration")
def	plt.ylabel("Loss")
get_Batch(runs,X_random,y_random,batch_size,s	plt.title("Logistic regression optimized by NAG")
hape):	plt.show()
if I == runs-1:	pic.snow()
X batch =	#RMSprop
X_random[I*batch_size:shape+1]	plt.close()
	lr = 0.062
y_random[l*batch_size:shape+1]	epoch = 5
else:	lambdal = 0.01
X_batch =	epsilon = np.e**(-8)
X_random[I*batch_size:(I+1)*batch_size]	batch_size = 128 # mini-batch gradient descent
y_batch	runs = math.ceil(X_train.shape[0] /
y_random[l*batch_size:(l+1)*batch_size]	float(batch_size))
return X_batch,y_batch	iteration = epoch * runs
#NAG	#get different kinds of initial data
lr = 0.02	(W_zeros,W_random or W_normal)
epoch = 5	W = W_normal
gamma = 0.9	Loss_train = np.zeros(iteration)
lambdal = 0.01	Loss_validation = np.zeros(iteration)
batch_size = 128 # mini-batch gradient descent	#the sum of the square of the gradient
runs = math.ceil(X_train.shape[0] /	$G_2 = 0$
float(batch_size))	for j in range(0,epoch):
iteration = epoch * runs	#shuffles the array
#get different kinds of initial data	X_random,y_random = shuffle_array(X_train)
(W_zeros,W_random or W_normal)	for I in range(0,runs):
W = W normal	#get the training instance and label in
Loss train = np.zeros(iteration)	current batch
Loss validation = np.zeros(iteration)	X batch,y batch =
v t = np.zeros(N)	get_Batch(runs,X_random,y_random,batch_size,X
for j in range(0,epoch):	train.shape[0])
#shuffles the array	#the training loss
X random,y random = shuffle array(X train)	Loss train[j*runs+l] =
for I in range(0,runs):	cal_Loss(X_batch,W,y_batch,lambdal)
#get the training instance and label in	#the gradient of the loss function
current batch	$G = cal_G(X_batch, W, y_batch, lambdal)$
X batch,y batch =	#the validation loss
get_Batch(runs,X_random,y_random,batch_size,X	Loss validation[j*runs+l] =
train.shape[0])	cal_Loss(X_validation,W,y_validation,lambdal)
<u> </u>	
#approximate W in the next time step	#update the parameter W
W_t = W - v_t * gamma #the training loss	$G_2 = G_2 * 0.9 + \text{np.dot}(G,G.T) * 0.1$ W = W - G * (Ir / math.sgrt(G 2 + epsilon))
#IDE ITAIDIDO 1088	vv = vv - G TUI / Math.Soft(G / + enslinn))

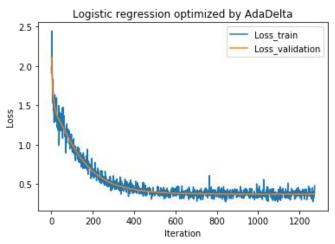
```
#draw the result
                                                         RMS W = math.sqrt(W 2 + epsilon)
plt.plot(Loss train,label="Loss train")
                                                    #draw the result
plt.plot(Loss validation,label="Loss validation")
                                                    plt.plot(Loss train,label="Loss train")
                                                    plt.plot(Loss validation,label="Loss validation")
plt.legend()
plt.xlabel("Iteration")
                                                    plt.legend()
plt.ylabel("Loss")
                                                    plt.xlabel("Iteration")
                                 optimized
                                                    plt.ylabel("Loss")
plt.title("Logistic
                   regression
RMSprop")
                                                    plt.title("Logistic
                                                                        regression
                                                                                      optimized
                                                                                                   by
plt.show()
                                                    AdaDelta")
#AdaDelta
                                                    plt.show()
plt.close()
Ir = 0.04
                                                    #Adam
epoch = 5
                                                    plt.close()
lambdal = 0.01
                                                    Ir = 0.08
                                                    epoch = 5
gamma = 0.9
epsilon = np.e^{**}(-8)
                                                    lambdal = 0.01
batch size = 128 # mini-batch gradient descent
                                                    beta1 = 0.9
                 math.ceil(X train.shape[0]
                                                    beta2 = 0.999
float(batch size))
                                                    epsilon = np.e^{**}(-8)
iteration = epoch * runs
                                                    batch size = 128 # mini-batch gradient descent
#aet
        different
                    kinds
                              of
                                    initial
                                             data
                                                    runs
                                                                     math.ceil(X train.shape[0]
(W zeros,W random or W normal)
                                                    float(batch size))
W = W normal
                                                    iteration = epoch * runs
                                                    #aet
                                                            different
                                                                         kinds
                                                                                  of
                                                                                        initial
                                                                                                 data
Loss train = np.zeros(iteration)
                                                     (W zeros,W random or W normal)
Loss validation = np.zeros(iteration)
#the sum of the square of the gradient
                                                    W = W normal
G 2 = 0
                                                    Loss train = np.zeros(iteration)
W^{2} = 0
                                                    Loss validation = np.zeros(iteration)
RMS g = 0
                                                    #the estimates of the first and second moments
RMS W = 0
                                                    m t = np.zeros(N)
                                                    n t = 0
W delta = np.zeros(N)
for i in range(0,epoch):
                                                    for j in range(0,epoch):
  #shuffles the array
                                                       #shuffles the array
  X random,y random = shuffle array(X train)
                                                      X random,y random = shuffle_array(X_train)
  for I in range(0,runs):
                                                      for I in range(0,runs):
     #get the training instance and label in
                                                         #get the training instance and label in
current batch
                                                    current batch
    X batch,y batch
                                                         X batch,y batch
get_Batch(runs,X_random,y_random,batch_size,X
                                                    get_Batch(runs,X_random,y_random,batch_size,X
train.shape[0])
                                                    train.shape[0])
     #the training loss
                                                         #the training loss
    Loss train[j*runs+l]
                                                         Loss train[j*runs+l]
                                                    cal_Loss(X_batch,W,y_batch,lambdal)
cal Loss(X batch,W,y batch,lambdal)
     #the gradient of the loss function
                                                         #the gradient of the loss function
     G = cal G(X batch, W, y batch, lambdal)
                                                         G = cal G(X batch, W, y batch, lambdal)
     #the validation loss
                                                         #the validation loss
     Loss validation[j*runs+l]
                                                         Loss validation[j*runs+l]
cal Loss(X validation, W, y validation, lambdal)
                                                    cal Loss(X validation, W, y validation, lambdal)
     #update the parameter W
                                                         #update the parameter W
    G 2 = G 2 * gamma + np.dot(G,G.T) *
                                                         m t = m t * beta1 + G * (1-beta1)
                                                         n t = n t * beta2 + np.dot(G,G.T) * (1-beta2)
(1-gamma)
     RMS_g = math.sqrt(G_2 + epsilon)
                                                         hat m = m t * (1/(1-beta1))
     W = W - G *(RMS W / RMS g)
                                                         hat n = n t * (1/(1-beta2))
    W delta = G *(- lr / RMS g)
                                                                                        hat m
    W 2
                    W 2
                                   gamma
                                                    (Ir/(math.sqrt(hat n)+epsilon))
np.dot(W delta,W delta.T) * (1-gamma)
                                                    #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()
```

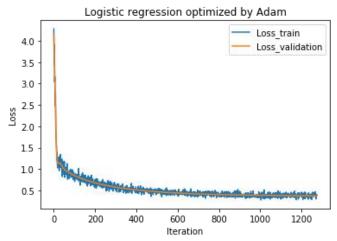
When applying NAG as the optimized method with specific parameters, the results is shown as follows:



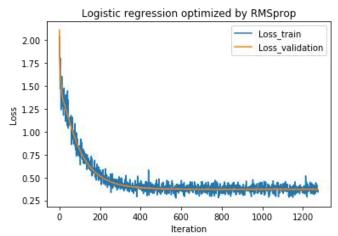
When applying AdaDelta as the optimized method with specific parameters, the results is shown as follows:



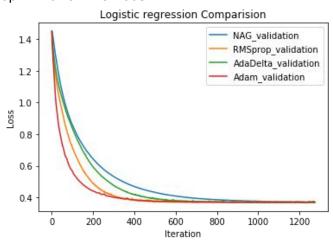
When applying Adam as the optimized method with specific parameters, the results is shown as follows:



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
```

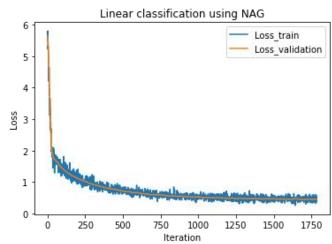
```
def cal Loss(X,W,y,lambdal,W 0):
  preY = np.dot(X,W)
  difY = np.ones(y.shape[0]) - y * preY
  difY[difY < 0] = 0
                                 X.shape[0]
          =np.sum(difY)
np.dot(W 0,W 0.T)/2*lambdal
  return Loss
#calculate the gradient
def cal G(X,W,y,lambdal,W 0):
  preY = np.dot(X,W)
  difY = np.ones(v.shape[0]) - v * preY
  y get = y.copy()
  y get[difY <= 0] =0
  G = -np.dot(y get,X) / X.shape[0] + W 0
*lambdal
  return G
#NAG
Ir = 0.02
epoch = 7
gamma = 0.8
lambdal = 0.01
batch size = 128 # mini-batch gradient descent
                 math.ceil(X train.shape[0]
float(batch_size))
iteration = epoch * runs
                                   initial
#aet
        different
                    kinds
                             of
                                            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 I 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,lambdal,W 0)
     #the gradient of the loss function
                                               =
cal G(X batch,W t,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
     v t = v t * gamma + G * Ir
     W = W - v t
#AdaDelta
Ir = 0.05
epoch = 5
lambdal = 0.01
qamma = 0.9
epsilon = np.e^{**}(-8)
batch size = 128 # mini-batch gradient descent
                math.ceil(X train.shape[0]
float(batch size))
iteration = epoch * runs
        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 q = 0
RMSW = 0
W delta = np.zeros(N)
for j in range(0,epoch):
  #shuffles the array
  X_random,y_random = shuffle_array(X_train)
  for I 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 * 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 *(- Ir / RMS g)
     W 2
                    W 2
                                  gamma
                                               +
np.dot(W delta,W delta.T) * (1-gamma)
     RMS W = math.sqrt(W 2 + epsilon)
```

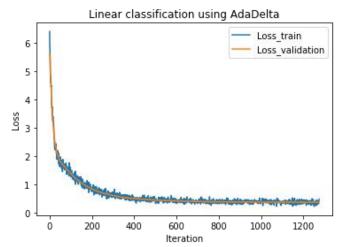
```
#Adam
Ir = 0.07
epoch = 4
lambdal = 0.01
beta1 = 0.9
beta2 = 0.999
epsilon = np.e^{**}(-8)
batch size = 128 # mini-batch gradient descent
                 math.ceil(X train.shape[0]
float(batch size))
iteration = epoch * runs
        different
                     kinds
                              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)
nt = 0
for j in range(0,epoch):
  #shuffles the array
  X random, y random = shuffle array(X train)
  for I 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.copv()
     W 0[N-1] = 0
     #the training loss
     Loss train[j*runs+l]
cal Loss(\overline{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))
                                    hat m
(Ir/(math.sqrt(hat n)+epsilon))
#RMSprop
plt.close()
Ir = 0.08
epoch = 5
lambdal = 0.01
epsilon = np.e^{**}(-8)
batch size = 128 # mini-batch gradient descent
```

```
math.ceil(X train.shape[0]
runs
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 I 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 * (Ir / math.sgrt(G 2 + epsilon))
```

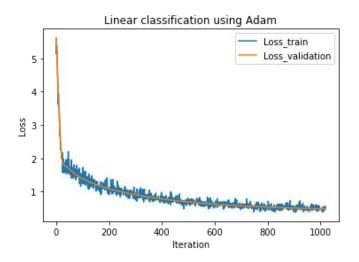
When applying NAG as the optimized method with specific parameters, the results is shown as follows:



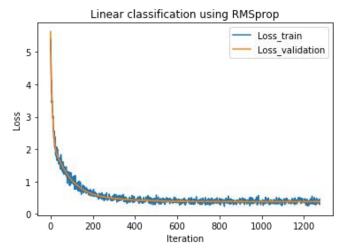
When applying AdaDelta as the optimized method with specific parameters, the results is shown as follows:



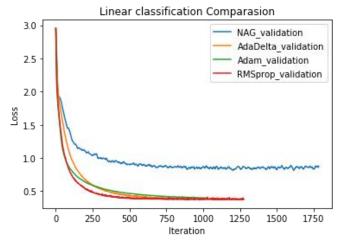
When applying Adam as the optimized method with specific parameters, the results is shown as follows:



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.