

### **South China University of Technology**

## The Experiment Report of Machine Learning

**SCHOOL: SCHOOL OF SOFTWARE ENGINEERING** 

**SUBJECT: SOFTWARE ENGINEERING** 

Author: Supervisor: Li Zhang Qingyao Wu

Student ID: Grade:

201721045909 Postgraduate

December 14, 2017

# Logistic Regression, Linear Classification and Stochastic Gradient Descent

Abstract—Logistic regression and linear classification are the widely used classification algorithm in the field of machine learning. Stochastic gradient descent is an effective way to find the good model parameters for above algorithm. The experiments compare the effectiveness of different optimized methods.

#### I. INTRODUCTION

Logistic regression and linear classification are the widely used classification algorithm in the field of machine learning. Logistic regression fits the data into a logit function (or logistic function) so that it can predict the probability of an event occurring. Stochastic gradient descent updates the parameters using the small sample of dataset. The experiments compare the effectiveness of different optimized methods that are used in Logistic regression and linear classification.

The rest paper is organized as follow. Section II contains the experiment steps. Section III describes the code and result for the two experiments. Section IV concludes the report.

#### II. METHODS AND THEORY

Experiment uses a9a of LIBSVM Data, including 32561/16281(testing) samples and each sample has 123/123 (testing) features. Please download the training set and validation set.

The experiment will be performed by the following steps:

- 1. Load the training set and validation set.
- 2. Initialize logistic regression model parameters, you can consider initializing zeros, random numbers or normal distribution.
- Select the loss function and calculate its derivation, find more detail in PPT.
- Calculate gradient toward loss function from partial samples.
- 5. Update model parameters using different optimized methods(NAG, RMSProp, AdaDelta and Adam).
- 6. Select the appropriate threshold, mark the sample whose predict scores greater than the threshold as positive, on the contrary as negative. Predict under validation set and get the different optimized method loss.
- 7. Repeat step 4 to 6 for several times, and drawing graph of the different optimized method loss and with the number of iterations.

#### III. EXPERIMENT

A. Codes of experiments

```
The codes of two experiments are shown as follow,
        Logistic Regression
# 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
path = './a9a.txt'
path_validation = './a9a.t'
#get the dataset
def get data(path):
  data = load symlight file(path)
  return data[0], data[1]
X_train, y_train = get_data(path)
X_{train} = X_{train.toarray}()
X validation, y_validation = get_data(path_validation)
X_{validation} = X_{validation.toarray()}
#complete the matrix
column = np.zeros(( X_validation.shape[0]))
X_{validation} = np.column_{stack}((X_{validation}, column))
column train = np.ones(( X train.shape[0]))
column \ validation = np.ones((X \ validation.shape[0]))
X_{train} = np.column_{stack}((X_{train}, column_{train}))
X validation =
np.column stack((X validation,column validation))
N = X_{train.shape}[1]
W_zeros = np.zeros(N)
W_{random} = random.random(size=N)
W normal = np.random.normal(size=N)
#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
get Batch(runs,X random,y random,batch size,shape):
  if 1 == runs-1:
```

X\_batch = X\_random[1\*batch\_size:shape+1] y batch = y random[1\*batch\_size:shape+1]

 $X_batch = X_random[1*batch_size:(l+1)*batch_size]$ 

else:

```
y batch = y random[l*batch size:(l+1)*batch size]
                                                                     v t = v t * gamma + G * lr
                                                                     W = W - v t
  return X batch, v batch
                                                                 Loss NAG = Loss validation
#calculate the loss
                                                                 #draw the result
def cal Loss(X,W,v,lambdal):
  preY = np.dot(X, W)
  Loss = np.sum(np.log(1 + np.exp(-y * preY)))
  Loss = Loss/X.shape[0] + lambdal/2 * np.dot(W,W.T)
                                                                 lr = 0.03
  return Loss
                                                                 epoch = 5
                                                                 lambdal = 0.01
#calculate the gradient
                                                                 epsilon = np.e**(-8)
def cal_G(X,W,y,lambdal):
  preY = np.dot(X,W)
  G = np.dot(((-y)/(1+np.exp(y*preY))),X)
                                                                 iteration = epoch * runs
  G = G / X.shape[0] + W * lambdal
  return G
                                                                 W_normal)
                                                                 W = W normal
#draw the result
                                                                 Loss_train = np.zeros(iteration)
def draw plot(Loss train, Loss validation, name):
  plt.plot(Loss_train,label="Loss_train")
  plt.plot(Loss_validation,label="Loss_validation")
                                                                 G_{2} = 0
  plt.legend()
                                                                 for j in range(0,epoch):
  plt.xlabel("Iteration")
                                                                   #shuffles the array
  plt.ylabel("Loss")
  plt.title("Logistic regression")
                                                                   for 1 in range(0,runs):
  plt.show()
                                                                     X_batch,y_batch =
1r = 0.01
epoch = 5
                                                                 ape[0])
gamma = 0.9
                                                                     #the training loss
lambdal = 0.01
                                                                     Loss_train[j*runs+l] =
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
                                                                     #the validation loss
W normal)
                                                                     Loss_validation[j*runs+l] =
W = W normal
Loss train = np.zeros(iteration)
                                                                     #update the parameter W
Loss_validation = np.zeros(iteration)
v t = np.zeros(N)
for j in range(0,epoch):
  #shuffles the array
                                                                 Loss\_RMS = Loss\_validation
  X_random, y_random = shuffle_array(X_train)
                                                                 #draw the result
  for 1 in range(0,runs):
    #get the training instance and label in current batch
    X_batch,y_batch =
                                                                 1r = 0.02
get_Batch(runs,X_random,y_random,batch_size,X_train.sh
                                                                 epoch = 5
ape[0])
                                                                 lambdal = 0.01
    #approximate W in the next time step
                                                                 gamma = 0.9
    W_t = W - v_t * gamma
                                                                 epsilon = np.e**(-8)
    #the training loss
    Loss train[j*runs+l] =
cal_Loss(X_batch,W,y_batch,lambdal)
                                                                 iteration = epoch * runs
    #the gradient of the loss function
    G = cal\_G(X\_batch, W\_t, y\_batch, lambdal)
                                                                 W normal)
    #the validation loss
                                                                 W = W_normal
    Loss_validation[j*runs+l] =
                                                                 Loss_train = np.zeros(iteration)
cal Loss(X validation, W, y validation, lambdal)
    #update the parameter W
```

```
draw_plot(Loss_train, Loss_validation, 'NAG')
batch_size = 128 # mini-batch gradient descent
runs = math.ceil(X train.shape[0] / float(batch size))
#get different kinds of initial data (W zeros,W random or
Loss validation = np.zeros(iteration)
#the sum of the square of the gradient
  X_{random,y_{random}} = shuffle_{array}(X_{train})
     #get the training instance and label in current batch
get Batch(runs,X random,y random,batch size,X train.sh
cal_Loss(X_batch,W,y_batch,lambdal)
     #the gradient of the loss function
     G = cal\_G(X\_batch, W, y\_batch, lambdal)
cal Loss(X validation, W, y validation, lambdal)
     G_2 = G_2 * 0.9 + np.dot(G,G.T) * 0.1
     W = W - G *(lr / math.sqrt(G 2 + epsilon))
draw_plot(Loss_train, Loss_validation, 'RMSprop')
batch size = 128 # mini-batch gradient descent
runs = math.ceil(X train.shape[0] / float(batch size))
#get different kinds of initial data (W_zeros,W_random or
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 1 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.sh
ape[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 = \text{math.sqrt}(G \ 2 + \text{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)
Loss Ada = Loss validation
#draw the result
draw_plot(Loss_train, Loss_validation, 'AdaDelta')
lr = 0.05
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 1 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.sh
ape[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))
Loss Adam = Loss validation
#draw the result
draw_plot(Loss_train, Loss_validation, 'Adam')
plt.plot(Loss_NAG,label="Loss_NAG")
plt.plot(Loss_RMS,label="Loss_RMSProp")
plt.plot(Loss Ada,label="Loss AdaDelta")
plt.plot(Loss_Adam,label="Loss_Adam")
plt.legend()
plt.xlabel("Iteration")
plt.ylabel("Loss")
plt.title("Logistic regression")
plt.show()
print(Loss_NAG[iteration-1])
print(Loss RMS[iteration-1])
print(Loss_Ada[iteration-1])
print(Loss_Adam[iteration-1])
```

#### 2) Linear Classification

```
# Linear classification
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
#mem = Memory("./mycache")
path = './a9a.txt'
path validation = './a9a.t'
#@mem.cache
def get_data(path):
  data = load_symlight_file(path)
  return data[0], data[1]
X_train, y_train = get_data(path)
X_{train} = X_{train.toarray}()
X validation, y validation = get data(path validation)
X_{validation} = X_{validation.toarray}()
#complete the matrix
column = np.zeros(( X validation.shape[0]))
X_{validation} = np.column_{stack}((X_{validation,column}))
column_train = np.ones(( X_train.shape[0]))
```

```
column \ validation = np.ones((X \ validation.shape[0]))
X train = np.column stack((X train,column train))
X validation =
np.column stack((X validation,column validation))
N=X_train.shape[1]
W_zeros = np.zeros(N)
W_random = random.random(size=N)
W normal = np.random.normal(size=N)
#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
get_Batch(runs,X_random,y_random,batch_size,shape):
  if 1 == runs-1:
    X batch = X random[1*batch size:shape+1]
    y_batch = y_random[1*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
#calculate the loss
def cal_Loss(X,W,y,lambdal,W_0):
  preY = np.dot(X,W)
  diifY = np.ones(y.shape[0]) - y * preY
  diifY[diifY < 0] = 0
  Loss = np.sum(diifY) / X.shape[0] +
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)
  diifY = np.ones(v.shape[0]) - v * preY
  y get = y.copy()
  y_get[diifY \le 0] = 0
  G = -np.dot(y\_get,X) / X.shape[0] + W 0 *lambdal
  return G
#calculate the accuracy
def cal_Accuracy(X,W,y):
  preY = np.dot(X,W)
  count = np.sum(preY * y > 0)
  Accuracy = count / X.shape[0]
  return Accuracy
#draw the result
def draw plot(Loss train, Loss validation, name):
  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")
  plt.show()
1r = 0.005
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)
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 1 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.sh
ape[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
    G = 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)
    #accuracy
    Accuracy[j*runs+l] =
cal_Accuracy(X_validation,W,y_validation)
    #update the parameter W,b
    v_t = v_t * gamma + G * lr
    W = W - v_t
Loss_NAG = Loss_validation
#draw the result
draw_plot(Loss_train, Loss_validation, 'NAG')
1r = 0.06
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
```

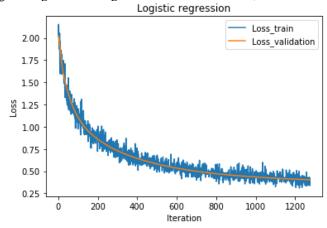
```
X random, y random = shuffle array(X train)
#get different kinds of initial data (W zeros,W random or
                                                                   for 1 in range(0,runs):
W_normal)
                                                                     #get the training instance and label in current batch
W = W_normal
                                                                     X batch,y batch =
Loss train = np.zeros(iteration)
                                                                get_Batch(runs,X_random,y_random,batch_size,X_train.sh
Loss validation = np.zeros(iteration)
                                                                ape[0])
Accuracy = np.zeros(iteration)
                                                                     W_0 = W.copy()
#the sum of the square of the gradient
                                                                     W_0[N-1] = 0
G 2 = 0
                                                                     #the training loss
for j in range(0,epoch):
                                                                     Loss_train[j*runs+l] =
  #shuffles the array
                                                                cal Loss(X batch, W, v batch, lambdal, W 0)
  X_{random,y_{random}} = shuffle_{array}(X_{train})
                                                                     #the gradient of the loss function
  for 1 in range(0,runs):
                                                                     G = cal G(X batch, W, y batch, lambdal, W 0)
    #get the training instance and label in current batch
                                                                     #the validation loss
    X_batch, y_batch =
                                                                     Loss_validation[j*runs+l] =
get_Batch(runs,X_random,y_random,batch_size,X_train.sh
                                                                cal Loss(X validation, W, y validation, lambdal, W 0)
ape[0])
                                                                     #accuracy
    W_0 = W.copy()
                                                                     Accuracy[j*runs+l] =
    W_0[N-1] = 0
                                                                cal_Accuracy(X_validation,W,y_validation)
    #the training loss
                                                                     #update the parameter W,b
    Loss_train[j*runs+l] =
                                                                     G_2 = G_2 * gamma + np.dot(G,G.T) * (1-gamma)
cal_Loss(X_batch,W,y_batch,lambdal,W_0)
                                                                     RMS g = math.sqrt(G_2 + epsilon)
    #the gradient of the loss function
                                                                     W = W - G *(RMS W / RMS g)
    G = cal\_G(X\_batch, W, y\_batch, lambdal, W\_0)
                                                                     W delta = G *(- lr / RMS g)
    #the validation loss
                                                                     W_2 = W_2 * gamma + np.dot(W_delta, W_delta.T) *
    Loss_validation[j*runs+l] =
                                                                (1-gamma)
cal_Loss(X_validation,W,y_validation,lambdal,W_0)
                                                                     RMS_W = math.sqrt(W_2 + epsilon)
    #accuracy
    Accuracy[j*runs+l] =
                                                                Loss_Ada = Loss_validation
cal_Accuracy(X_validation, W, y_validation)
                                                                #draw the result
    #update the parameter W,b
                                                                draw_plot(Loss_train, Loss_validation, 'AdaDelta')
    G = G = 0.9 + np.dot(G,G.T) * 0.1
    W = W - G *(lr / math.sqrt(G_2 + epsilon))
                                                                lr = 0.07
                                                                epoch = 5
Loss RMS = Loss validation
                                                                lambdal = 0.01
#draw the result
                                                                beta1 = 0.9
draw_plot(Loss_train, Loss_validation, 'RMSprop')
                                                                beta2 = 0.999
                                                                epsilon = np.e^{**}(-8)
1r = 0.04
                                                                batch_size = 128 # mini-batch gradient descent
epoch = 5
                                                                runs = math.ceil(X_train.shape[0] / float(batch_size))
lambdal = 0.01
                                                                iteration = epoch * runs
gamma = 0.9
                                                                #get different kinds of initial data (W zeros,W random or
epsilon = np.e^{**}(-8)
                                                                W normal)
batch size = 128 # mini-batch gradient descent
runs = math.ceil(X_train.shape[0] / float(batch_size))
                                                                W = W_normal
iteration = epoch * runs
                                                                Loss_train = np.zeros(iteration)
                                                                Loss_validation = np.zeros(iteration)
#get different kinds of initial data (W_zeros,W_random or
                                                                Accuracy = np.zeros(iteration)
W normal)
                                                                #the estimates of the first and second moments
W = W_normal
                                                                m t = np.zeros(N)
Loss train = np.zeros(iteration)
                                                                n_t = 0
Loss_validation = np.zeros(iteration)
                                                                for j in range(0,epoch):
Accuracy = np.zeros(iteration)
                                                                   #shuffles the array
#the sum of the square of the gradient
                                                                   X_{random,y_{random}} = shuffle_{array}(X_{train})
G_2 = 0
                                                                   for 1 in range(0,runs):
W_{2} = 0
                                                                     #get the training instance and label in current batch
RMS g = 0
                                                                     X_batch,y_batch =
RMS W = 0
                                                                get_Batch(runs,X_random,y_random,batch_size,X_train.sh
W_{delta} = np.zeros(N)
                                                                ape[0])
for j in range(0,epoch):
                                                                     W_0 = W.copy()
  #shuffles the array
```

```
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)
    #accuracy
     Accuracy[j*runs+l] =
cal_Accuracy(X_validation,W,y_validation)
    #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))
Loss\_Adam = Loss\_validation
#draw the result
draw plot(Loss train, Loss validation, 'Adam')
plt.plot(Loss NAG,label="Loss NAG")
plt.plot(Loss RMS,label="Loss RMSProp")
plt.plot(Loss_Ada,label="Loss_AdaDelta")
plt.plot(Loss_Adam,label="Loss_Adam")
plt.legend()
plt.xlabel("Iteration")
plt.ylabel("Loss")
plt.title("Logistic regression")
plt.show()
print(Loss_NAG[iteration-1])
print(Loss_RMS[iteration-1])
print(Loss_Ada[iteration-1])
print(Loss_Adam[iteration-1])
```

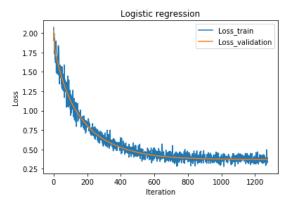
#### B. Experiment results

#### 1) Logistic Regression

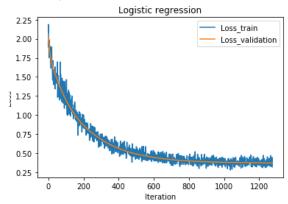
In this set of experiments, we compare different optimized methods that are used in logistic regression. The loss of logistic regression using NAG is shown as follow,



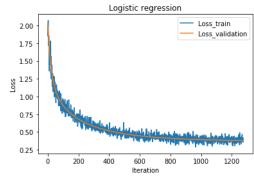
The loss of logistic regression using RMSprop is shown as follow,



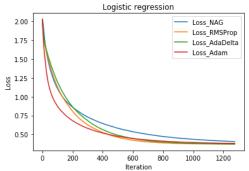
The loss of logistic regression using AdaDelta is shown as follow,



The loss of logistic regression using Adam is shown as follow,



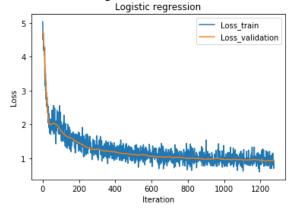
The comparison of these optimized methods is shown as follow,



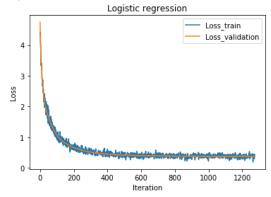
We can see that the loss using NAG is a little higher than others. And the amplitude of train loss is large because of the small simple.

2)Linear Classification

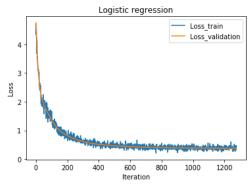
In this set of experiments, we compare different optimized methods that are used in linear classification. The loss of linear classification using NAG is shown as follow,



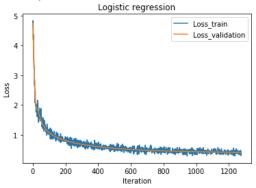
The loss of linear classification using RMSprop is shown as follow,



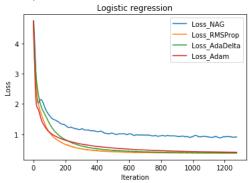
The loss of linear classification using AdaDelta is shown as follow,



The loss of linear classification using Adam is shown as follow,



The comparison of these optimized methods is shown as follow,



We can see that the loss using NAG is a much higher than others. And the amplitude of train loss is large because of the small simple.

#### IV. CONCLUSION

After the experiments, we know the algorithm using the different optimized methods can cause different effectiveness. For each algorithm, we need to select the right optimized methods to update its parameters. For example, if we want to train the linear classification, don't use the NAG method.