The Experiment Report of Machine Learning

SCHOOL: SCHOOL OF SOFTWARE ENGINEERING

SUBJECT: SOFTWARE ENGINEERING

Author:
Li Zhang

Supervisor:
Qingyao Wu

Student ID：
201721045909

Grade:
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.
3. Select the loss function and calculate its derivation, find more detail in PPT.
4. 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,

1) 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_svmlight_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
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

#calculate the loss
def cal_Loss(X,W,y,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)
    return Loss

#calculate the gradient
def cal_G(X,W,y,lambdal):
    preY = np.dot(X,W)
    G = np.dot(((-y)/ (1+ np.exp(y*preY))),X )
    G = G / X.shape[0] + W * lambdal
    return G

#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()

lr = 0.01
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.sh
ape[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

Loss_NAG = Loss_validation
#draw the result
draw_plot(Loss_train, Loss_validation, 'NAG')

lr = 0.03
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.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 * 0.9 + np.dot(G,G.T) * 0.1
        W = W - G *(lr / math.sqrt(G_2 + epsilon))

Loss_RMS = Loss_validation
#draw the result
draw_plot(Loss_train, Loss_validation, 'RMSprop')

lr = 0.02
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.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 = 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)

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 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.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_svmlight_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
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

#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(y.shape[0]) - y * preY
    y_get = y.copy()
    y_get[diifY <= 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()

lr = 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 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.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')

lr = 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
```

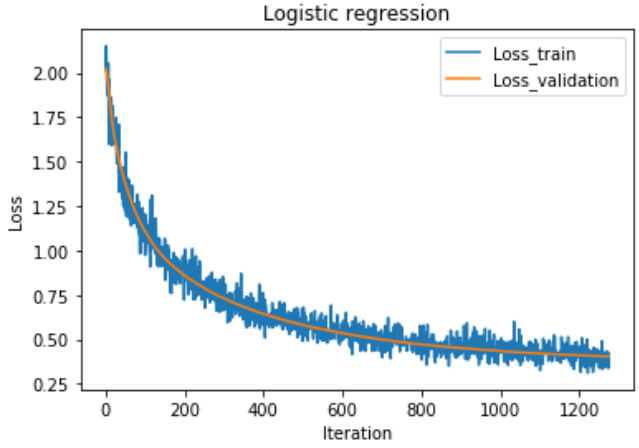
```
#get different kinds of initial data（W_zeros,W_random or W_normal）
W = W_normal
Loss_train = np.zeros(iteration)
Loss_validation = np.zeros(iteration)
Accuracy = np.zeros(iteration)
#the sum of the square of the gradient
G_2 = 0
for j in range(0,epoch):
   #shuffles the array
   X_random,y_random = 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)
      #accuracy
      Accuracy[j*runs+l] = cal_Accuracy(X_validation,W,y_validation)
      #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))

Loss_RMS = Loss_validation
#draw the result
draw_plot(Loss_train, Loss_validation, 'RMSprop')

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)
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,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
      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)

Loss_Ada = Loss_validation
#draw the result
draw_plot(Loss_train, Loss_validation, 'AdaDelta')

lr = 0.07
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)
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)
        #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])
```

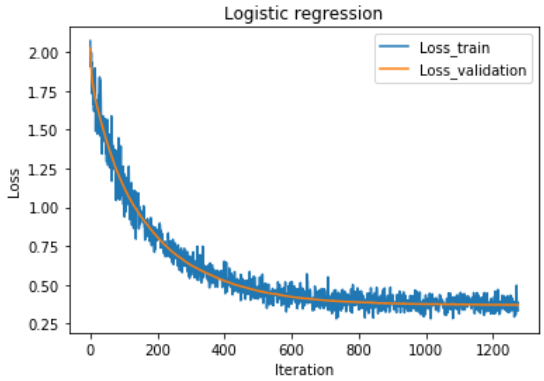
B. Experiment results

1) Logistic Regression

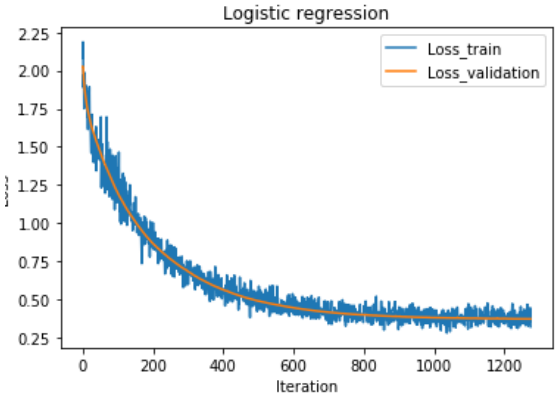
In this set of experiments, we compare different optimized methods that are used in logistic regression. The loss of logistic regression using NAG is shown as follow,



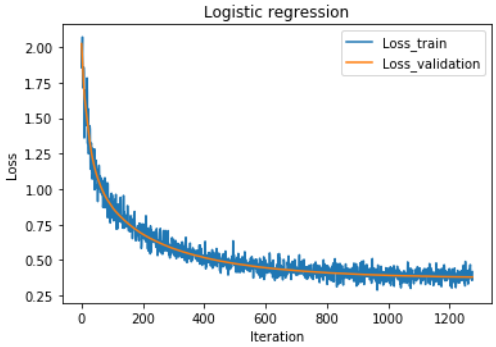
The loss of logistic regression using RMSprop is shown as follow,



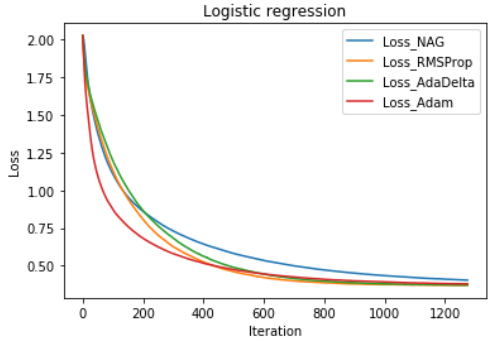
The loss of logistic regression using AdaDelta is shown as follow,



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



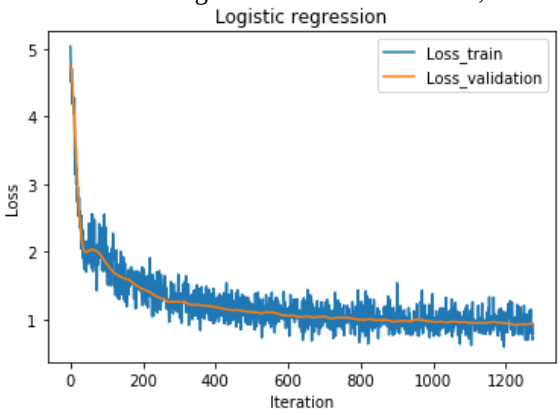
The comparison of these optimized methods is shown as follow,



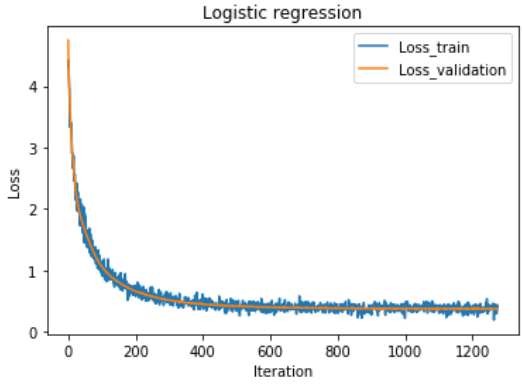
We can see that the loss using NAG is a little higher than others. And the amplitude of train loss is large because of the small simple.

2)Linear Classification

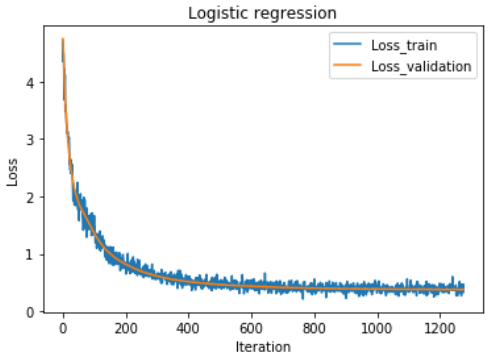
In this set of experiments, we compare different optimized methods that are used in linear classification. The loss of linear classification using NAG is shown as follow,



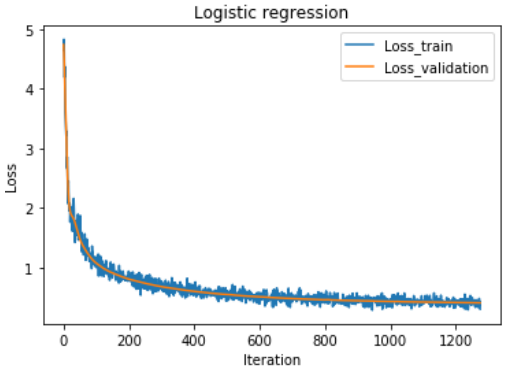
The loss of linear classification using RMSprop is shown as follow,



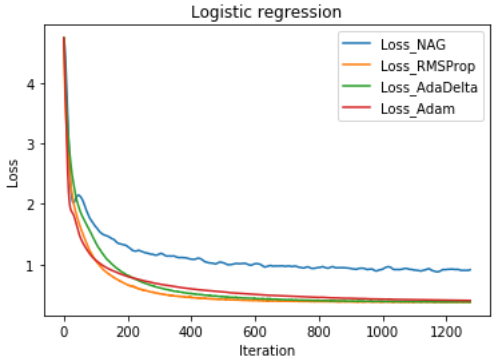
The loss of linear classification using AdaDelta is shown as follow,



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.