The Experiment Report of Machine Learning

SCHOOL: SCHOOL OF SOFTWARE ENGINEERING

SUBJECT: SOFTWARE ENGINEERING

Author:
Siyuan Xiao

Supervisor:
Qingyao Wu

Student ID：
201721045886

Grade:
Postgraduate

December 14, 2017

Logistic Regression, Linear Classification and Stochastic Gradient Descent

Abstract—Logistic Regression is a method using regression techniques to achieve classification. Stochastic Gradient Descent is an advanced way to solve optimizing problems, and is very popular in the field of machine learning. This experiment combines these two.

I. Introduction

This report will talk about the whole experiment I have made on Logistic Regression and Linear classification, which are based on Stochastic Gradient Descent. Its content is organized as follow:

1) Section II contains the experiment steps.
2) Section III contains the code for the two experiments.
3) Section IV makes conclusion for the experiment result.

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.

Then the experiment will be performed by the following steps:

1) Download the dataset to local host machine.
2) Load dataset into memory.
3) Split dataset into training set and validation set.
4) Create and fill necessary data structures according to different optimizing methods.
5) Write functions for calculating loss and gradient (different in regression and classification).
6) Set parameters for different optimizing methods (learning rate and the number of iterations).
7) Initiate weights (using normal distribution).
8) Calculate the gradient and update weights according to the optimizing methods.
9) Switch to another optimizing method and run again.
10) Draw plot for experiment result.

III. Experiment

Here I placed the code for the two experiments:

1) Logistic Regression:

```
# created by Swain, 2017-12-14, 13:25

import math
import numpy
import matplotlib.pyplot as plot
from numpy import random
from sklearn.datasets import load_svmlight_file

#load a9a dataset
#training dataset
data = load_svmlight_file('./a9a')
X_train = data[0].toarray()
y_train = data[1]
data = load_svmlight_file('./a9a.t')
X_vali = data[0].toarray()
y_vali = data[1]

#complete the martrix
X_vali = numpy.column_stack((X_vali, numpy.zeros((X_vali.shape[0]))))

#add a constant-bias-column to X
X_train = numpy.column_stack((X_train, numpy.ones((X_train.shape[0]))))
X_vali = numpy.column_stack((X_vali, numpy.ones((X_vali.shape[0]))))

#create weight array with initial values in normal distribution
d = X_train.shape[1]
W_init = numpy.random.normal(size=d)

#define loss function
def loss(X, W, y, _lambda):
    y_predict = numpy.dot(X, W)
    return numpy.sum(numpy.log(1 + numpy.exp(-y * y_predict))) / X.shape[0] + _lambda / 2 * numpy.dot(W, W.T)

#define gradient function
```

```
def grad(X, W, y, _lambda):
    y_predict = numpy.dot(X, W)
    return numpy.dot(((-y) / (1 + numpy.exp(y * y_predict))), X) / X.shape[0] + W * _lambda

#shuffle the array
def shuffle_array(X_train):
    randomlist = numpy.arange(X_train.shape[0])
    numpy.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 k == runs - 1:
        X_batch = X_random[k * batch_size : shape + 1]
        y_batch = y_random[k * batch_size : shape + 1]
    else:
        X_batch = X_random[k * batch_size : (k+1) * batch_size]
        y_batch = y_random[k * batch_size : (k+1) * batch_size]
    return X_batch,y_batch

#parameters: learning rate and #iteration
lr = 0.05
epoch = 5
batch_size = 128
runs = math.ceil(X_train.shape[0] / float(batch_size))
iteration = epoch * runs

_lambda = 0.01

#NAG/AdaDelta
gamma = 0.8

#RMSprop/AdaDelta/Adam
epsilon = numpy.e**(-8)

#Adam
beta1 = 0.9
beta2 = 0.999

#used to save results
loss_train = []
loss_vali = []

nmethods = 4
#use different optimizing method for each i
for i in range(0, nmethods):
    #reset W
    W = W_init
    loss_train.append(numpy.zeros(iteration))
    loss_vali.append(numpy.zeros(iteration))

    #NAG
    vt = numpy.zeros(X_train.shape[1])

    #RMSprop/AdaDelta
    g2 = 0

    #AdaDelta
    w2 = 0
    RMS_g = 0
    RMS_w = 0
    w_delta = numpy.zeros(X_train.shape[1])

    #Adam
    mt = numpy.zeros(X_train.shape[1])
    nt = 0

    for j in range(0, epoch):
        X_random, y_random = shuffle_array(X_train)
        for k in range(0, runs):
            #get a batch of training data
            X_batch, y_batch = get_Batch(runs,X_random,y_random,batch_size,X_train.shape[0])

            #calculate gradient
            #NAG
            if i == 0:
                G = grad(X_batch, W - vt * gamma, y_batch, _lambda)
            #others
            else:
                G = grad(X_batch, W, y_batch, _lambda)

            #calculate loss on both training and validation datasets
            loss_train[i][j * runs + k] = loss(X_batch, W, y_batch, _lambda)
            loss_vali[i][j * runs + k] = loss(X_vali, W, y_vali, _lambda)

            #update weight according to optimizing methods
            #NAG
            if (i == 0):
                vt = vt * gamma + G * lr
                W = W - vt
            #RMSprop
            elif (i == 1):
                g2 = g2 * 0.9 + numpy.dot(G,G.T) * 0.1
                W = W - G *(lr / math.sqrt(g2 + epsilon))
            elif (i == 2):
                g2 = g2 * gamma + numpy.dot(G,G.T) * (1-gamma)
                RMS_g = math.sqrt(g2 + epsilon)
                W = W - G *(RMS_w / RMS_g)
                w_delta = G *(- lr / RMS_g)
                w2 = w2 * gamma + numpy.dot(w_delta, w_delta.T) * (1-gamma)
                RMS_w = math.sqrt(w2 + epsilon)
            else:
                mt = mt * beta1 + G * (1-beta1)
                nt = nt * beta2 + numpy.dot(G,G.T) * (1-beta2)
                hat_m = mt * (1/(1-beta1))
                hat_n = nt * (1/(1-beta2))
                W = W - hat_m * (lr/(math.sqrt(hat_n)+epsilon))

names = ['NAG', 'RMSprop', 'AdaDelta', 'Adam']
i = 0

plot.plot(loss_train[i], label="training loss")
plot.plot(loss_vali[i], label="validation loss")
plot.legend()
plot.xlabel("Iteration")
plot.ylabel("Validation Loss")
plot.title('Logistic Regression + ' + names[i])
plot.show()
```

2) Linear Classification:

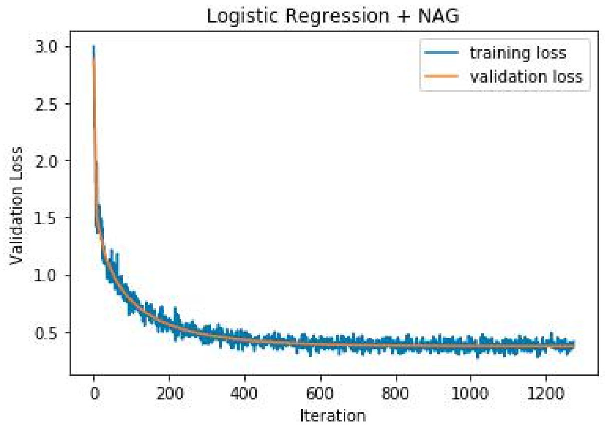
The only difference from the former is the loss function and its corresponding gradient function.

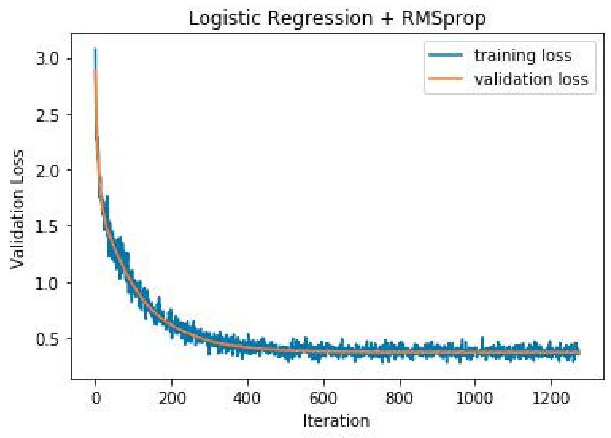
```
#define loss function (Hinge loss)
def loss(X, W, y, _lambda):
    y_predict = numpy.dot(X,W)
    diff = numpy.ones(y.shape[0]) - y * y_predict
    diff[diff < 0] = 0
    W_0 = W.copy()
    W_0[len(W) - 1] = 0
    return numpy.sum(diff) / X.shape[0] + numpy.dot(W_0,W_0.T) / 2 * _lambda

#define gradient function
def grad(X, W, y, _lambda):
    y_predict = numpy.dot(X,W)
    diff = numpy.ones(y.shape[0]) - y * y_predict
    y_ = y.copy()
    y_[diff <= 0] = 0
    W_0 = W.copy()
    W_0[len(W) - 1] = 0
    return -numpy.dot(y_,X) / X.shape[0] + W_0 * _lambda
```

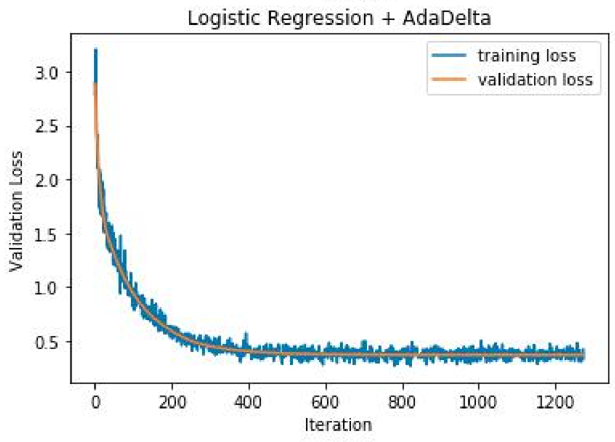
IV. CONCLUSION

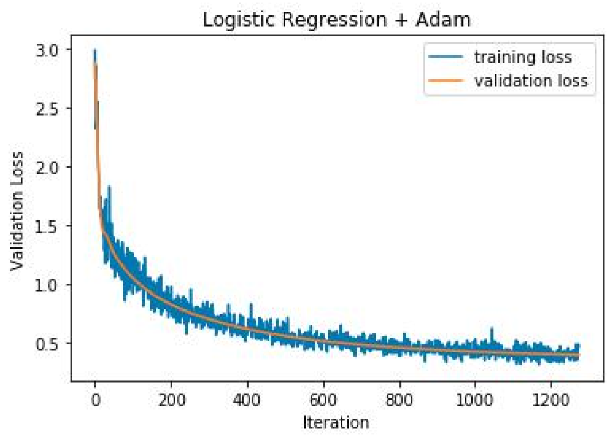
Here is the experiment result gained:

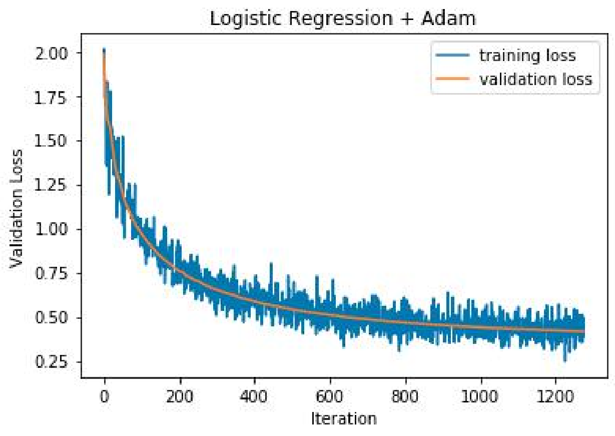
1) Logistic Regression



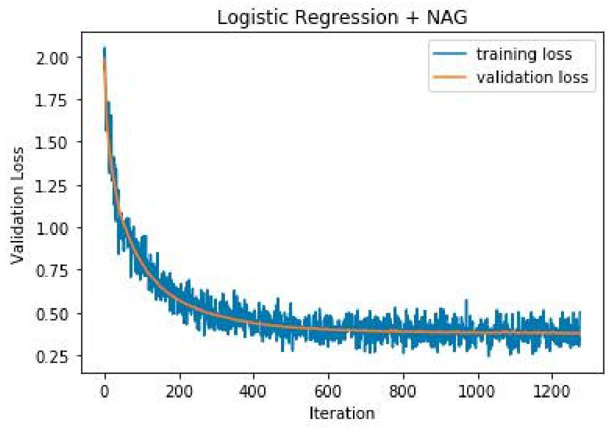


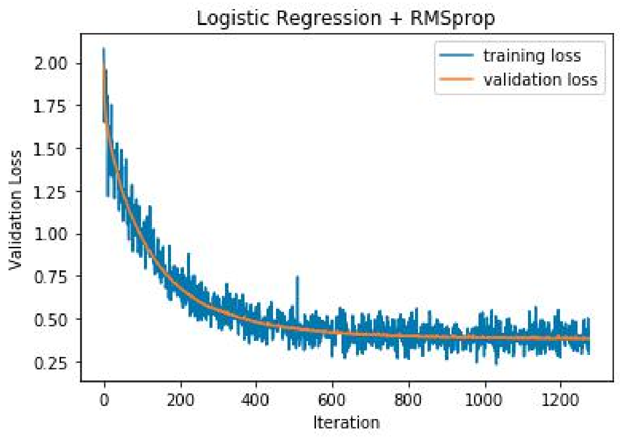


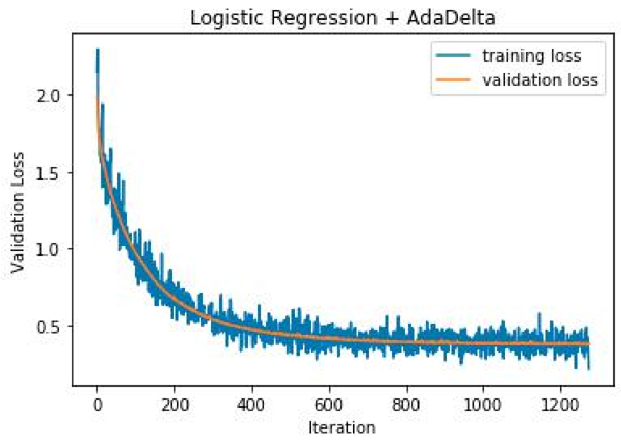




2) Linear Classification







Then we can draw a conclusion according to the two experiments:

1) Using batches to train the model is faster and more effective.
2) The 4 optimizing methods show slight

ference on loss in these two experiments.