South China University of Technology

The Experiment Report of Machine Learning

SCHOOL: SCHOOL OF SOFTWARE ENGINEERING

SUBJECT: SOFTWARE ENGINEERING

Author:
张驰

Supervisor:
吴庆耀

Student ID：
201530613603

Grade:
大三

December 11, 2017

Logistic Regression, Linear Classification and Stochastic Gradient Descent

Abstract—This report tends to illustrate the experiments we have done about logistic regression, linear classification and stochastic gradient descent(SGD), with respect to understanding and comprehending the core of this mentioned topics.

I. Introduction

Logistic regression and linear classification are the two of most fundamental machine learning models. Additionally, gradient decent(GD) is one of the most widely-used optimizing methods to reach local optimal solution. Stochastic gradient descent(SGD), an improved version of traditional GD, accelerates the process reaching the solution. This experiment aims to compare GD to SGD, to help understanding the differences and relations between them. What's more, we also compare logistic regression to linear classification, figuring out what is and is not similar to each other. Lastly, we practice SVM on larger data to have a better command of its principles.

II. Methods and Theory

Logistic Regression and Stochastic Gradient Descent

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 G 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 L_{NAG}, $L_{RMSProp}$, $L_{AdaDelta}$ and L_{Adam}.
7. Repeat step 4 to 6 for several times, and drawing graph of L_{NAG}, $L_{RMSProp}$, $L_{AdaDelta}$ and L_{Adam} with the number of iterations.

Linear Classification and Stochastic Gradient Descent

1. Load the training set and validation set.
2. Initialize SVM 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 G 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 L_{NAG}, $L_{RMSProp}$, $L_{AdaDelta}$and L_{Adam}..
7. Repeat step 4 to 6 for several times, and drawing graph of L_{NAG}, $L_{RMSProp}$, $L_{AdaDelta}$and L_{Adam}. with the number of iterations.

III. Experiment

A. Data Set

We use a9a of LIBSVM Data, including 32561/16281(testing) samples and each sample has 123/123 (testing) features.

B. Implementation

1) Logistic regression:

Loss function of logistic regression is as follows.

$$L(\theta) = -\frac{1}{m}\sum_{i=0}^{n}\left(y_i \log\left(h_\theta(X_i)\right) + (1-y_i)\log\left(1-\log\left(h_\theta(X_i)\right)\right)\right)$$

where
$$h_\theta(X) = \frac{1}{1+e^{-\theta^T \cdot X}}$$
.

Compute the gradient of L(θ), we obtain,

$$\frac{\partial L}{\partial \theta} = \frac{1}{n}\sum_{i=0}^{n} X_i\left(h_\theta(X) - y_i\right)$$

Having defined loss function and its gradient, we can use SGD to get the final solution. We use four method respectively to reach the local optimal solution including NAG, RMSProp, AdaDelta and Adam. The super parameters we select are as follows.

NAG	learning rate	0.005
	gamma	0.9
	iteration	1000
RMSProp	learning rate	0.005
	gamma	0.9
	epsilon	1e-8
	iteration	1000
AdaDelta	learning rate	0.005
	gamma	0.9

	epsilon	1e-8
	iteration	1000
Adam	learning rate	0.005
	beta1	0.9
	beta2	0.999
	epsilon	1e-8
	iteration	1000

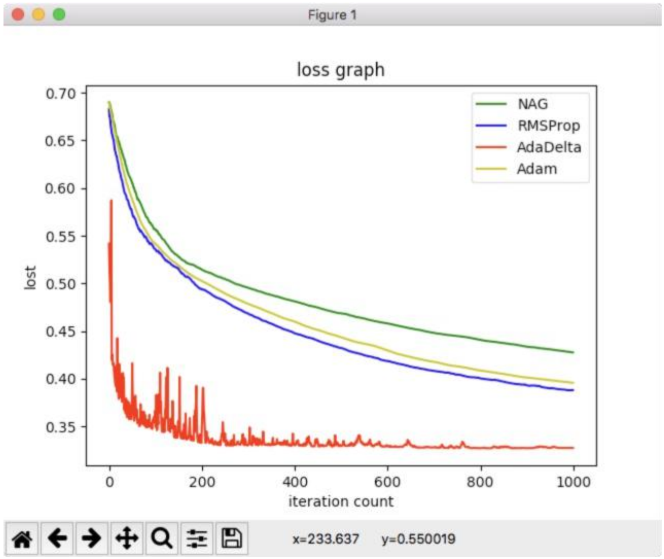
Next, we program to implement the above methods.

```
from sklearn import datasets, model_selection, linear_model

import numpy as np

import jupyter

import matplotlib.pyplot as plt

import math

import random


X_train, y_train = datasets.load_svmlight_file("a9atrain.txt")


# turn the csr_matrix into array for futher processing

x_ = np.array(X_train.toarray(), np.float32).reshape((-1, 123))

y_ = np.array(y_train, np.float32).reshape((-1, 1))


for i in range(y_.shape[0]):

    if y_[i,0] == -1.0 : y_[i,0] = 0.


X_ = np.hstack([x_, np.ones((x_.shape[0], 1))])


X_test, y_test = datasets.load_svmlight_file("a9atest.txt")


xt_ = np.array(X_test.toarray(), np.float32).reshape((-1, 122))

yt_ = np.array(y_test, np.float32).reshape((-1, 1))

for i in range(yt_.shape[0]):

    if yt_[i,0] == -1.0 : yt_[i,0] = 0.

# X_ = (x_;1) to fit the constent item


Xt_ = np.hstack([xt_, np.zeros((xt_.shape[0], 1)),np.ones((xt_.shape[0], 1))])


# h θ (X) = e^(Theta * X) / (1 + e^(Theta * X)) = 1 / (1 + e^(-Theta * X))

def h_theta(Xi, Theta):

    e_t = math.exp(Xi.dot(Theta.T))

    return e_t / (1 + e_t)


# L( θ ) = - (1/m) Σ (yi * log(h θ (Xi)) + (1 - yi) * log(1 - h θ (Xi)))

def compute_loss(X, y, Theta):

    m = y.shape[0]

    loss = 0.

    for i in range(m):

        loss +=  (y[i] * math.log(h_theta(X[i, :], Theta))) + ((1 - y[i]) * math.log(1 - h_theta(X[i,:], Theta)))

    loss /= - m

    return loss


# ∂L/∂ θ = (1/m) Σ (h θ (Xi) - yi) * Xi

def compute_gradient(X, y, Theta):

    m = y.shape[0]

    gradient = np.zeros(Theta.shape)

    for i in range(m):

        gradient += (h_theta(X[i, :], Theta) - y[i]) * (X[i, :])

    gradient /= m
```

```
    return gradient

def train_model_nag(X, y, Theta, learning_rate, gamma,
iteration = 10000):

    test_loss_history = np.zeros((iteration, 1))

    v = np.zeros(Theta.shape)

    Theta_gradient = np.zeros(Theta.shape)

    for iter in range(iteration):

        index = random.randint(0, y.shape[0]-10)

        Theta = Theta - gamma * v

        v = gamma * v - learning_rate *
compute_gradient(X[index:index+10,:], y[index:index+10],
Theta)

        Theta = Theta + v

        test_loss_history[iter] = compute_loss(Xt_, yt_,
Theta)[-1:]

    return test_loss_history, Theta

def train_model_rmsprop(X, y, Theta, learning_rate, gamma,
epsilon, iteration = 10000):

    test_loss_history = np.zeros((iteration, 1))

    G_t = 0.

    Theta_gradient = np.zeros(Theta.shape)

    for iter in range(iteration):

        index = random.randint(0, y.shape[0]-10)

        Theta_gradient = compute_gradient(X[index:index+10,:],
y[index:index+10], Theta)

        G_t = gamma * G_t + (1 - gamma) *
Theta_gradient.dot(Theta_gradient.T)

        Theta = Theta - (learning_rate / np.sqrt(G_t + epsilon)) *
Theta_gradient

        test_loss_history[iter] = compute_loss(Xt_, yt_,
Theta)[-1:]

    return test_loss_history, Theta

def train_model_adadelta(X, y, Theta, gamma, epsilon,
iteration):

    test_loss_history = np.zeros((iteration, 1))

    Theta_gradient = np.zeros(Theta.shape)

    G_t = 0.

    delta_theta = np.zeros(Theta.shape)

    delta_t = 0.03

    for iter in range(iteration):

        index = random.randint(0, y.shape[0]-10)

        Theta_gradient = compute_gradient(X[index:index+10,:],
y[index:index+10], Theta)

        G_t = gamma * G_t + (1 - gamma) *
Theta_gradient.dot(Theta_gradient.T)

        delta_theta = - (np.sqrt(delta_t + epsilon) / np.sqrt(G_t +
epsilon)) * Theta_gradient

        Theta = Theta + delta_theta

        delta_t = gamma * delta_t + (1 - gamma) *
(delta_theta.dot(delta_theta.T))

        test_loss_history[iter] = compute_loss(Xt_, yt_,
Theta)[-1:]

    return test_loss_history, Theta

def train_model_adam(X, y, Theta, learning_rate, beta1, beta2,
epsilon, iteration):

    test_loss_history = np.zeros((iteration, 1))

    Theta_gradient = np.zeros(Theta.shape)

    v_t = 0.

    m_t = np.zeros(Theta.shape)

    for iter in range(iteration):
```

```
        index = random.randint(0, y.shape[0]-10)

        Theta_gradient = compute_gradient(X[index:index+10,:], y[index:index+10], Theta)

        m_t = beta1 * m_t + (1 - beta1) * Theta_gradient

        v_t = beta2 * v_t + (1 - beta2) * Theta_gradient.dot(Theta_gradient.T)

        mt_estimate = m_t / (1 - pow(beta1, iter + 1))

        vt_estimate = v_t / (1 - pow(beta2, iter + 1))

        Theta = Theta - learning_rate * mt_estimate / (np.sqrt(vt_estimate) + epsilon)

        test_loss_history[iter] = compute_loss(Xt_, yt_, Theta)[-1:]

    return test_loss_history, Theta



iteration = 1000



be1 = 0.9

be2 = 0.999

ep = 1e-8



t_nag = np.zeros((1, 124))

t_rmsprop = np.zeros((1, 124))

t_adadelta = np.zeros((1, 124))

t_adam = np.zeros((1, 124))

# for i in range(t.shape[0]):

    # t[i] = [-0.03]



nag_loss_history, t_nag = train_model_nag(X_, y_, t_nag, 0.005, be1, iteration)

rmsprop_loss_history, t_rmsprop = train_model_rmsprop(X_, y_, t_rmsprop, 0.005, be1, ep, iteration)

adadelta_loss_history, t_adadelta = train_model_adadelta(X_, y_, t_adadelta, be1, ep, iteration)

adam_loss_history, t_adam = train_model_adam(X_, y_, t_adam, 0.005, be1, be2, ep, iteration)



plt.plot(nag_loss_history, 'g', label='NAG')

plt.plot(rmsprop_loss_history, 'b', label='RMSProp')

plt.plot(adadelta_loss_history, 'r', label='AdaDelta')

plt.plot(adam_loss_history, 'y', label='Adam')



plt.legend(loc='upper right')


plt.ylabel('lost');


plt.xlabel('iteration count')


plt.title('loss graph')


plt.show()
```

We get the following loss graphs as results after running the program.



From the graph we find that AdaDelta reaches the local optimal solution fastest, but it also has obvious vibration. By contrast, NAG is slower than AdaDelta with a smoother curve. RMSProp and Adam are closely overlapped, which are slower than AdaDelta and faster than NAG. Four methods reaches optimal solution far faster than traditional GD.

2) Linear classification:

Loss function of logistic regression is as follows.

$$L(\theta) = \frac{1}{2n}\sum_{i=0}^{n}(y_i - h_\theta(X_i))^2$$

, where $h_\theta(X) = \sum_{i=0}^{n}\theta_i X_i$.

Compute the gradient of L(θ), we obtain,

$$\frac{\partial L}{\partial \theta} = \frac{1}{n}\sum_{i=0}^{n}(y_i - h_\theta(X_i)) \cdot X_i$$

Having defined loss function and its gradient, we can use SGD to get the final solution. We use four method respectively to reach the local optimal solution including NAG, RMSProp, AdaDelta and Adam. The super parameters we select are as follows.

NAG	learning rate	0.005
	gamma	0.9
	iteration	3000
RMSProp	learning rate	0.005
	gamma	0.9
	epsilon	1e-8
	iteration	3000
AdaDelta	learning rate	0.005
	gamma	0.9
	epsilon	1e-8
	iteration	3000
Adam	learning rate	0.005
	beta1	0.9
	beta2	0.999
	epsilon	1e-8
	iteration	3000

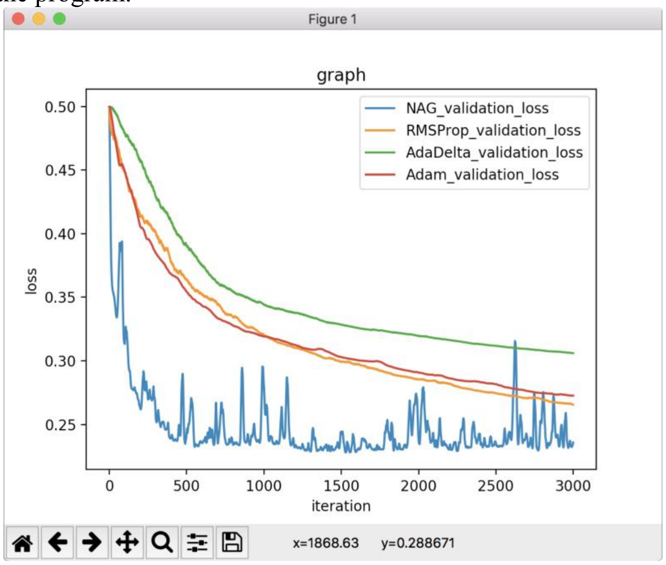
Next, we program to implement the above methods.

```
import numpy
import random
import jupyter
import math
from sklearn.datasets import load_svmlight_file
from sklearn.model_selection import train_test_split
from matplotlib import pyplot

x, y_train = load_svmlight_file("a9atrain.txt")
x_train = x.toarray()
x, y_test = load_svmlight_file("a9atest.txt")
x_test = x.toarray()

X_train = numpy.hstack([x_train,
numpy.ones((x_train.shape[0], 1))])
X_test = numpy.hstack([x_test, numpy.zeros((x_test.shape[0],
1))])
X_test = numpy.hstack([X_test, numpy.ones((x_test.shape[0],
1))])

def compute_grad(x, y, w):
    gradient = x * (y - x.dot(w.T))
    return gradient

def compute_loss(x, y, w, random_i):
    loss = 0
    a = len(random_i)
    for m in range(a):
        loss += 0.5 * ((y[random_i[m]] -
x[random_i[m],:].dot(w.T)) ** 2)
    return loss/a

def NAG_train(x, y, x_test, y_test, w, C, lr, gamma, threshold,
iteration):
    vt = numpy.zeros(w.shape)
    loss_history = []
    test_loss_history = []
    random_index = []
    random_test_index = []
    for i in range(iteration):
        random_num = random.randint(0, x.shape[0]-1)
        random_test_num = random.randint(0, x_test.shape[0]-1)
        random_index.append(random_num)
        random_test_index.append(random_test_num)
    for i in range(iteration):
        gradient = compute_grad(x[random_index[i],:],
y[random_index[i]], w-gamma*vt)
        vt = gamma*vt - lr*gradient
        w -= vt
        loss = compute_loss(x, y, w, random_index)
        loss_history.append(loss)
        test_loss_history.append(compute_loss(x_test, y_test, w,
random_test_index))
        if loss < threshold :
```

```
        break
    return w, loss_history, test_loss_history

def RMSProp_train(x, y, x_test, y_test, w, C, lr, gamma,
threshold, iteration):
    Gt = 0
    loss_history = []
    test_loss_history = []
    random_index = []
    random_test_index = []
    for i in range(iteration):
        random_num = random.randint(0, x.shape[0]-1)
        random_test_num = random.randint(0, x_test.shape[0]-1)
        random_index.append(random_num)
        random_test_index.append(random_test_num)
    for i in range(iteration):
        gradient = compute_grad(x[random_index[i],:],
y[random_index[i]], w)
        Gt = gamma*Gt + (1-gamma)*gradient.dot(gradient.T)
        w += lr * gradient / math.sqrt(Gt+1e-8)
        loss = compute_loss(x, y, w, random_index)
        loss_history.append(loss)
        test_loss_history.append(compute_loss(x_test, y_test, w,
random_test_index))
        if loss < threshold :
            break
    return w, loss_history, test_loss_history

def AdaDelta_train(x, y, x_test, y_test, w, C, lr, gamma,
threshold, iteration):
    Gt = 0
    variable_t = 0
    loss_history = []
    test_loss_history = []
    random_index = []
    random_test_index = []
    for i in range(iteration):
        random_num = random.randint(0, x.shape[0]-1)
        random_test_num = random.randint(0, x_test.shape[0]-1)
        random_index.append(random_num)
        random_test_index.append(random_test_num)
    for i in range(iteration):
        gradient = compute_grad(x[random_index[i],:],
y[random_index[i]], w)
        Gt = gamma*Gt + (1-gamma)*gradient.dot(gradient.T)
        variable_w = - math.sqrt(variable_t + 1e-8) * gradient /
math.sqrt(Gt + 1e-8)
        w -= variable_w
        variable_t = gamma*variable_t +
(1-gamma)*variable_w.dot(variable_w.T)
        loss = compute_loss(x, y, w, random_index)
        loss_history.append(loss)
        test_loss_history.append(compute_loss(x_test, y_test, w,
random_test_index))
        if loss < threshold :
            break
    return w, loss_history, test_loss_history

def Adam_train(x, y, x_test, y_test, w, C, lr, gamma, threshold,
iteration):
    Gt = 0
    moment = numpy.zeros((1, x.shape[1]))
    B = 0.9
    loss_history = []
    test_loss_history = []
    random_index = []
    random_test_index = []
    for i in range(iteration):
        random_num = random.randint(0, x.shape[0]-1)
        random_test_num = random.randint(0, x_test.shape[0]-1)
        random_index.append(random_num)
        random_test_index.append(random_test_num)
    for i in range(iteration):
        gradient = compute_grad(x[random_index[i],:],
y[random_index[i]], w)
        moment = B*moment + (1-B)*gradient
        Gt = gamma*Gt + (1-gamma)*gradient.dot(gradient.T)
        a = lr * math.sqrt(1 - pow(gamma, iteration)) / (1-pow(B,
iteration))
        w += a * moment / math.sqrt(Gt + 1e-8)
        loss = compute_loss(x, y, w, random_index)
        loss_history.append(loss)
        test_loss_history.append(compute_loss(x_test, y_test, w,
random_test_index))
        if loss < threshold :
            break
    return w, loss_history, test_loss_history

iteration = 3000
# NAG
NAG_w = numpy.zeros((1, X_train.shape[1]))
NAG_w, NAG_loss_history, NAG_test_loss_history =
NAG_train(X_train, y_train, X_test, y_test, NAG_w, 0.3,
0.001, 0.9, 0.001, iteration)

# RMSProp
RMS_w = numpy.zeros((1, X_train.shape[1]))
RMS_w, RMS_loss_history, RMS_test_loss_history =
RMSProp_train(X_train, y_train, X_test, y_test, RMS_w, 0.3,
0.001, 0.9, 0.001, iteration)

# AdaDelta
AdaDelta_w = numpy.zeros((1, X_train.shape[1]))
AdaDelta_w, AdaDelta_loss_history,
AdaDelta_test_loss_history = AdaDelta_train(X_train, y_train,
X_test, y_test, AdaDelta_w, 0.3, 0.001, 0.9, 0.001, iteration)

#Adam
Adam_w = numpy.zeros((1, X_train.shape[1]))
Adam_w, Adam_loss_history, Adam_test_loss_history =
Adam_train(X_train, y_train, X_test, y_test, Adam_w, 0.3,
0.001, 0.9, 0.001, iteration)


pyplot.plot(NAG_test_loss_history, label =
'NAG_validation_loss')
pyplot.plot(RMS_test_loss_history, label =
'RMSProp_validation_loss')
pyplot.plot(AdaDelta_test_loss_history, label =
'AdaDelta_validation_loss')
```

```
pyplot.plot(Adam_test_loss_history, label =
'Adam_validation_loss')
pyplot.legend(loc='upper right')
pyplot.ylabel('loss')
pyplot.xlabel('iteration')
pyplot.title('graph')
pyplot.show()
```

We get the following loss graphs as results after running the program.



From the graph we find that NAG reaches the local optimal solution fastest, but it also has obvious vibration. By contrast, AdaDelta is slower than NAG with a smoother curve. RMSProp and Adam are closely overlapped, which are slower than NAG and faster than AdaDelta. Four methods reaches optimal solution far faster than traditional GD.

IV. CONCLUSION

The experiment I learned a lot of experience in the practice of using the knowledge learned in the course of practical problems. And let me have a deeper grasp of python, in addition, let me consolidate the knowledge of linear algebra.