The Experiment Report of Machine Learning



Grade:

大三

December 11, 2017

Student ID：

201530613603

Supervisor:

吴庆耀

Author:

张驰

**SUBJECT:**SOFTWARE ENGINEERING

**SCHOOL:** SCHOOL OF SOFTWARE ENGINEERING

[[1]](#footnote-0)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.

# 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.

# 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, , and .
7. Repeat step 4 to 6 for several times, and drawing graph of ，， and 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 ，，and ..
7. Repeat step 4 to 6 for several times, and drawing graph of ，，and . with the number of iterations.

# Experiment

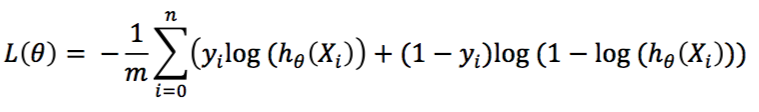
## Data Set

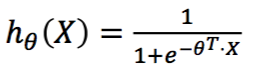
We use a9a of LIBSVM Data, including 32561/16281(testing) samples and each sample has 123/123 (testing) features.

## Implementation

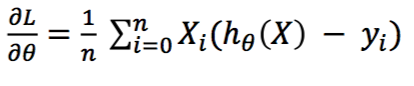
### Logistic regression:

Loss function of logistic regression is as follows.



where .

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



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))])

# hθ(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) ∑ (yi \* log(hθ(Xi)) + (1 - yi) \* log(1 - hθ(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) ∑ (hθ(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')

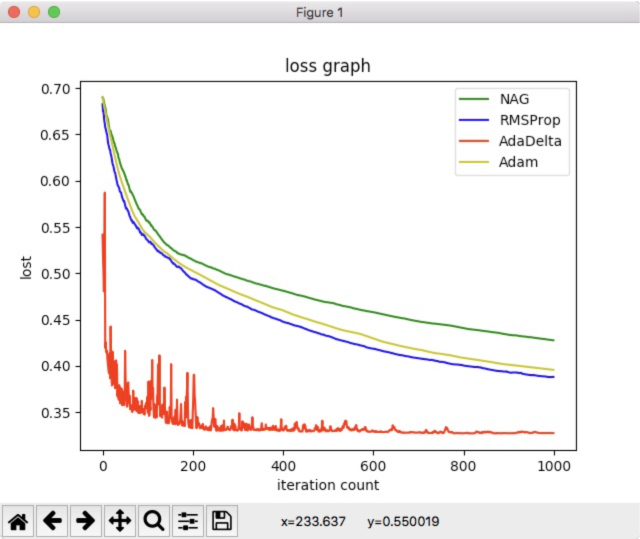
plt.ylabel('lost');

plt.xlabel('iteration count')

plt.title('loss graph')

plt.show()

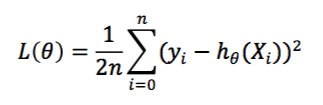
We get the following loss graphs as results after running the program.



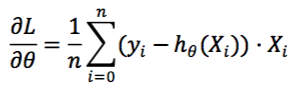
From the graph we find that AdaDelta reaches the local optimal solution fastest, but it also has obvious vibration. By contrast, NAG is slower than AdaDelta with a smoother curve. RMSProp and Adam are closely overlapped, which are slower than AdaDelta and faster than NAG. Four methods reaches optimal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

### Linear classification:

Loss function of logistic regression is as follows.

, where .

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



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')

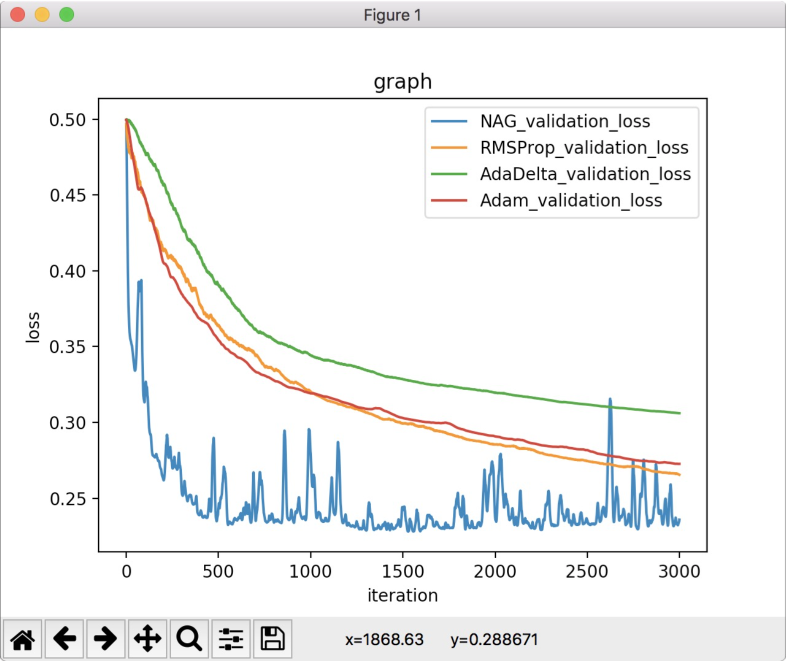
pyplot.ylabel('loss')

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.

# 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.

1. [↑](#footnote-ref-0)