The Experiment Report of Machine Learning



**SUBJECT:**SOFTWARE ENGINEERING

December 14, 2017

Grade:

Undergraduate

Student ID：

201721045886

Supervisor:

Qingyao Wu

Author:

Siyuan Xiao

**SCHOOL:** SCHOOL OF SOFTWARE ENGINEERING

[[1]](#footnote-0)

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.

# 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.

# 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.

# Experiment

Here I placed the code for the two experiments:

1. Logistic Regression:

|  |
| --- |
| 1. # created by Swain, 2017-12-14, 13:25 2. import math 3. import numpy 4. import matplotlib.pyplot as plot 5. from numpy import random 6. from sklearn.datasets import load\_svmlight\_file 7. #load a9a dataset 8. #training dataset 9. data = load\_svmlight\_file('./a9a') 10. X\_train = data[0].toarray() 11. y\_train = data[1] 12. data = load\_svmlight\_file('./a9a.t') 13. X\_vali = data[0].toarray() 14. y\_vali = data[1] 15. #complete the martrix 16. X\_vali = numpy.column\_stack((X\_vali, numpy.zeros((X\_vali.shape[0])))) 17. #add a constant-bias-column to X 18. X\_train = numpy.column\_stack((X\_train, numpy.ones((X\_train.shape[0])))) 19. X\_vali = numpy.column\_stack((X\_vali, numpy.ones((X\_vali.shape[0])))) 20. #create weight array with initial values in normal distribution 21. d = X\_train.shape[1] 22. W\_init = numpy.random.normal(size=d) 23. #define loss function 24. def loss(X, W, y, \_lambda): 25. y\_predict = numpy.dot(X, W) 26. return numpy.sum(numpy.log(1 + numpy.exp(-y \* y\_predict))) / X.shape[0] + \_lambda / 2 \* numpy.dot(W, W.T) 27. #define gradient function 28. def grad(X, W, y, \_lambda): 29. y\_predict = numpy.dot(X, W) 30. return numpy.dot(((-y) / (1 + numpy.exp(y \* y\_predict))), X) / X.shape[0] + W \* \_lambda 31. #shuffle the array 32. def shuffle\_array(X\_train): 33. randomlist = numpy.arange(X\_train.shape[0]) 34. numpy.random.shuffle(randomlist) 35. X\_random = X\_train[randomlist] 36. y\_random = y\_train[randomlist] 37. return X\_random,y\_random 38. #get the training instance and label in current batch 39. def get\_Batch(runs,X\_random,y\_random,batch\_size,shape): 40. if k == runs - 1: 41. X\_batch = X\_random[k \* batch\_size : shape + 1] 42. y\_batch = y\_random[k \* batch\_size : shape + 1] 43. else: 44. X\_batch = X\_random[k \* batch\_size : (k+1) \* batch\_size] 45. y\_batch = y\_random[k \* batch\_size : (k+1) \* batch\_size] 46. return X\_batch,y\_batch 47. #parameters: learning rate and #iteration 48. lr = 0.05 49. epoch = 5 50. batch\_size = 128 51. runs = math.ceil(X\_train.shape[0] / float(batch\_size)) 52. iteration = epoch \* runs 53. \_lambda = 0.01 54. #NAG/AdaDelta 55. gamma = 0.8 56. #RMSprop/AdaDelta/Adam 57. epsilon = numpy.e\*\*(-8) 58. #Adam 59. beta1 = 0.9 60. beta2 = 0.999 61. #used to save results 62. loss\_train = [] 63. loss\_vali = [] 64. nmethods = 4 65. #use different optimizing method for each i 66. for i in range(0, nmethods): 67. #reset W 68. W = W\_init 69. loss\_train.append(numpy.zeros(iteration)) 70. loss\_vali.append(numpy.zeros(iteration)) 72. #NAG 73. vt = numpy.zeros(X\_train.shape[1]) 75. #RMSprop/AdaDelta 76. g2 = 0 78. #AdaDelta 79. w2 = 0 80. RMS\_g = 0 81. RMS\_w = 0 82. w\_delta = numpy.zeros(X\_train.shape[1]) 84. #Adam 85. mt = numpy.zeros(X\_train.shape[1]) 86. nt = 0 88. for j in range(0, epoch): 89. X\_random, y\_random = shuffle\_array(X\_train) 90. for k in range(0, runs): 91. #get a batch of training data 92. X\_batch, y\_batch = get\_Batch(runs,X\_random,y\_random,batch\_size,X\_train.shape[0]) 94. #calculate gradient 95. #NAG 96. if i == 0: 97. G = grad(X\_batch, W - vt \* gamma, y\_batch, \_lambda) 98. #others 99. else: 100. G = grad(X\_batch, W, y\_batch, \_lambda) 102. #calculate loss on both training and validation datasets 103. loss\_train[i][j \* runs + k] = loss(X\_batch, W, y\_batch, \_lambda) 104. loss\_vali[i][j \* runs + k] = loss(X\_vali, W, y\_vali, \_lambda) 106. #update weight according to optimizing methods 107. #NAG 108. if (i == 0): 109. vt = vt \* gamma + G \* lr 110. W = W - vt 111. #RMSprop 112. elif (i == 1): 113. g2 = g2 \* 0.9 + numpy.dot(G,G.T) \* 0.1 114. W = W - G \*(lr / math.sqrt(g2 + epsilon)) 115. elif (i == 2): 116. g2 = g2 \* gamma + numpy.dot(G,G.T) \* (1-gamma) 117. RMS\_g = math.sqrt(g2 + epsilon) 118. W = W - G \*(RMS\_w / RMS\_g) 119. w\_delta = G \*(- lr / RMS\_g) 120. w2 = w2 \* gamma + numpy.dot(w\_delta, w\_delta.T) \* (1-gamma) 121. RMS\_w = math.sqrt(w2 + epsilon) 122. else: 123. mt = mt \* beta1 + G \* (1-beta1) 124. nt = nt \* beta2 + numpy.dot(G,G.T) \* (1-beta2) 125. hat\_m = mt \* (1/(1-beta1)) 126. hat\_n = nt \* (1/(1-beta2)) 127. W = W - hat\_m \* (lr/(math.sqrt(hat\_n)+epsilon)) 128. names = ['NAG', 'RMSprop', 'AdaDelta', 'Adam'] 129. i = 0 130. plot.plot(loss\_train[i], label="training loss") 131. plot.plot(loss\_vali[i], label="validation loss") 132. plot.legend() 133. plot.xlabel("Iteration") 134. plot.ylabel("Validation Loss") 135. plot.title('Logistic Regression + ' + names[i]) 136. plot.show() |

1. Linear Classification:

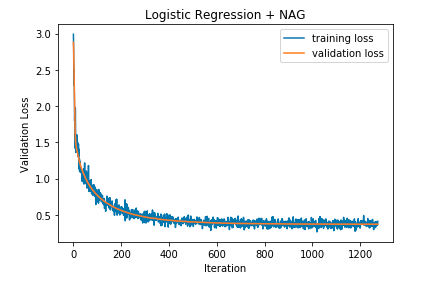
The only difference from the former is the loss function and its corresponding gradient function.

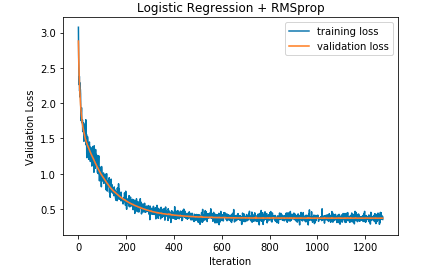
|  |
| --- |
| 1. #define loss function (Hinge loss) 2. def loss(X, W, y, \_lambda): 3. y\_predict = numpy.dot(X,W) 4. diff = numpy.ones(y.shape[0]) - y \* y\_predict 5. diff[diff < 0] = 0 6. W\_0 = W.copy() 7. W\_0[len(W) - 1] = 0 8. return numpy.sum(diff) / X.shape[0] + numpy.dot(W\_0,W\_0.T) / 2 \* \_lambda 9. #define gradient function 10. def grad(X, W, y, \_lambda): 11. y\_predict = numpy.dot(X,W) 12. diff = numpy.ones(y.shape[0]) - y \* y\_predict 13. y\_ = y.copy() 14. y\_[diff <= 0] = 0 15. W\_0 = W.copy() 16. W\_0[len(W) - 1] = 0 17. return -numpy.dot(y\_,X) / X.shape[0] + W\_0 \* \_lambda |

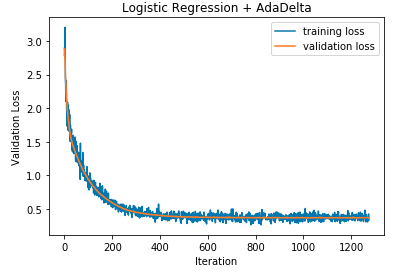
# conclusion

Here is the experiment result gained:

1. Logistic Regression

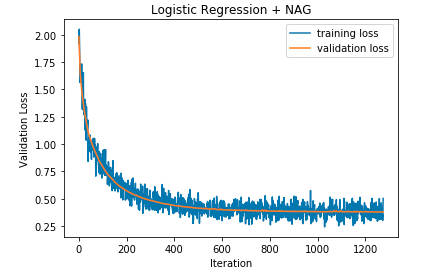


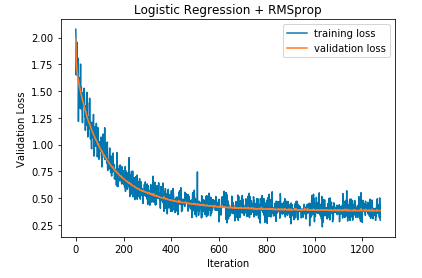


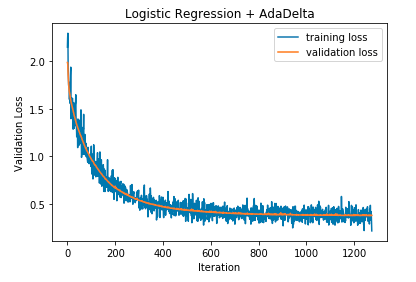


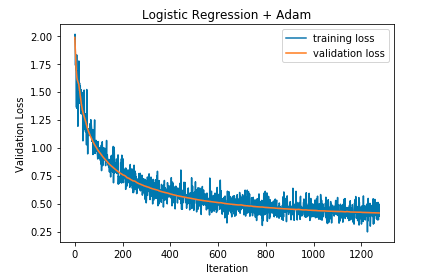


1. Linear Classification









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 difference on loss in these two experiments.

1. [↑](#footnote-ref-0)