The Experiment Report of Machine Learning



**SUBJECT:**SOFTWARE ENGINEERING

December 14, 2017

Grade:

Post Graduate

Student ID：

201721045893

Supervisor:

 Qingyao Wu

Author:

Shengyan Wen

**SCHOOL:** SCHOOL OF SOFTWARE ENGINEERING

[[1]](#footnote-0)

Logistic Regression, Linear Classification and Stochastic Gradient Descent

Abstract— **In this experiment, we aim to compare and understand the difference between gradient descent and stochastic gradient descent. We compare logistic regression and linear classification. We aim to further understand the principles of SVM and practice on larger data.**

# INTRODUCTION

The motivation of the experiment is compare and understand the difference between gradient descent and stochastic gradient descent,

and compare Logistic regression and linear classification, and finally understand the principles of SVM and practice on larger data.

Logic regression uses 'a9a' in LIBSVM Data, including 32561/16281(testing) samples and each sample has 123/123 (testing) features. This time we load train set and validation set separately .

For logistic regression and stochastic gradient descent, all the experiment steps is as follows. After downloading, initialize the logistic regression model parameters with normal distribution. Secondly, select the Log-like-hood loss as the loss function of logic regression with calculating its derivation

Thirdly compute the gradient of the loss function from partial samples.

Fourthly update the parameters using different optimized methods(NAG，RMSProp，AdaDelta and Adam).

Fifthly select the appropriate threshold, mark the sample whose predict scores greater than the threshold as positive, on the contrary as negative, and predict under validation set and get the different optimized method loss LNAG, LRMSProp, and LAdam.

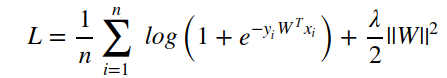
Finally repeat step 3 to 5 for several times, and drawing graph of LNAG, LRMSProp, and LAdam with the number of iterations.

For linear classification and stochastic gradient descent, the experiment steps is almost as same as logistic regression. The difference is that at step 1 initialize SVM model parameters with normal distribution, and at step 2 select the Hinge loss as the loss function of linear classification with calculating its derivation.

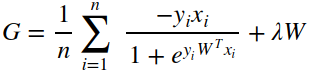
# METHODS AND THEORY

*A)Logistic regression and stochastic gradient descent*

We defined the loss function of the linear regression to be Log-likehood loss:

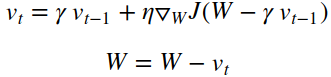


The gradient of the loss function is:



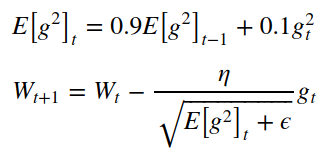
Where λ is the regularization parameter.

Update the parameter W use nesterov accelerated gradient(NAG):



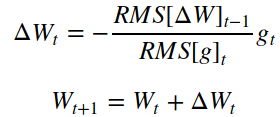
Where η is the learning rate, γ is the momentum.

Update the parameter W use RMSprop:

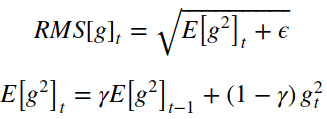


Where η is the learning rate, ϵ is the smoothing term.

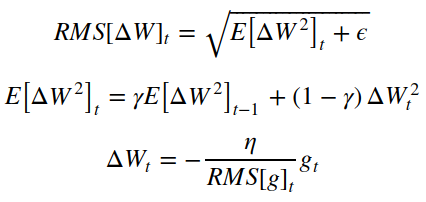
Update the parameter W use AdaDelta:

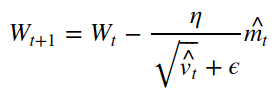


Where

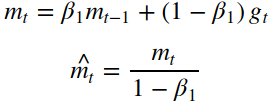


And

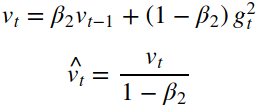
Update the parameter W use Adam:



Where

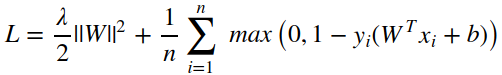


And

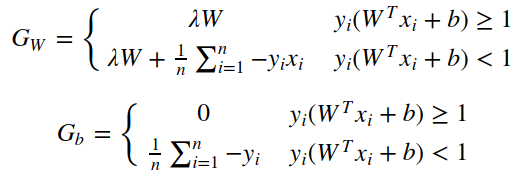


*B)Linear classification and stochastic gradient descent*

We defined the loss function of the linear classification to be Hinge loss:



The gradient of the loss function is:

Update the parameters W use four optimized methods (NAG, RMSProp，AdaDelta and Adam), which are same as the logic classification.

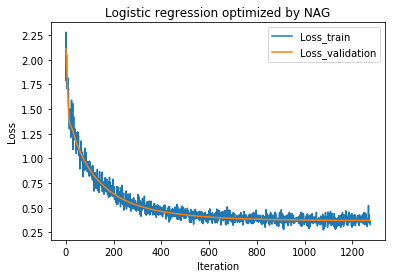
# Experiment

1. *Logic regression and stochastic gradient descent*

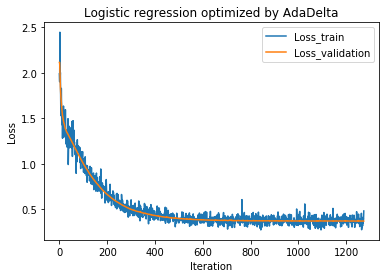
The code is listed as follows

|  |
| --- |
| # 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  #load dataset  def get\_test\_data():  data = load\_svmlight\_file("a9a.test")  return data[0], data[1]  def get\_val\_data():  data = load\_svmlight\_file("a9a.validation")  return data[0], data[1]  X\_train, y\_train = get\_test\_data()  X\_train = X\_train.toarray()  X\_validation, y\_validation = get\_val\_data()  X\_validation = X\_validation.toarray()  print(X\_train.shape)  print(X\_validation.shape)  (32561, 123)  (16281, 122)  In [64]:  #X\_train = [X\_train, 1]  addone\_train = np.ones( X\_train.shape[0])  X\_train = np.column\_stack((X\_train,addone\_train))  print(X\_train.shape)  #X\_validation = [X\_validation,0,1]  addzero = np.zeros(( X\_validation.shape[0]))  X\_validation = np.column\_stack((X\_validation,addzero))  addone = np.ones( X\_validation.shape[0])  X\_validation = np.column\_stack((X\_validation,addone))  print(X\_validation.shape)  (32561, 124)  (16281, 124)  In [65]:  # Initialize with normal distribution  N = X\_train.shape[1]  W\_normal = np.random.normal(size=N)  In [66]:  #Log-likehood loss function  def cal\_Loss(X,W,y,lambdal):  preY = np.dot(X,W)  Loss = (np.sum(np.log(1 + np.exp(-y \* preY))))/ X.shape[0] + lambdal / 2 \* np.dot(W,W.T)  return Loss  #calculate gradient  def cal\_G(X,W,y,lambdal):  preY = np.dot(X,W)  G = (np.dot(((-y)/ (1+ np.exp(y\*preY))),X ))/ X.shape[0] + W \* lambdal  return G  #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  #NAG  lr = 0.02  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.shape[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  #draw the result  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 optimized by NAG")  plt.show()  #RMSprop  plt.close()  lr = 0.062  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.shape[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))  #draw the result  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 optimized by RMSprop")  plt.show()  #AdaDelta  plt.close()  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)  #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])  #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)  #draw the result  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 optimized by AdaDelta")  plt.show()  #Adam  plt.close()  lr = 0.08  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.shape[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))  #draw the result  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 optimized by Adam")  plt.show() |

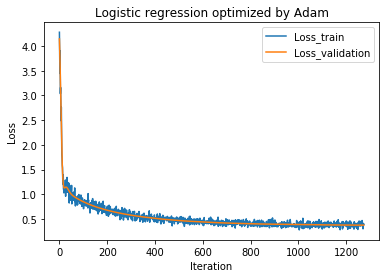
When applying NAG as the optimized method with specific parameters, the results is shown as follows:



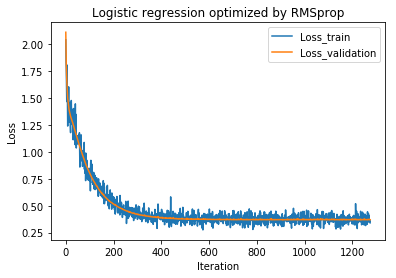
When applying AdaDelta as the optimized method with specific parameters, the results is shown as follows:



When applying Adam as the optimized method with specific parameters, the results is shown as follows:

**

When applying RMSProp as the optimized method with specific parameters, the results is shown as follows:

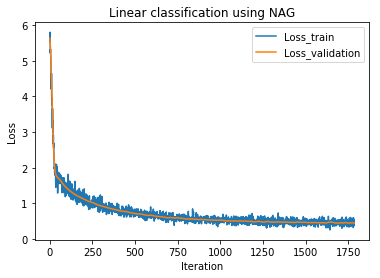


*B) Linear classification and stochastic gradient descent*

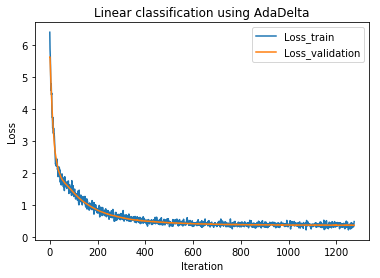
The code is listed as follows

|  |
| --- |
| ...  ...  #Hinge loss function  def cal\_Loss(X,W,y,lambdal,W\_0):  preY = np.dot(X,W)  difY = np.ones(y.shape[0]) - y \* preY  difY[difY < 0] =0  Loss =np.sum(difY) / 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)  difY = np.ones(y.shape[0]) - y \* preY  y\_get = y.copy()  y\_get[difY <= 0] =0  G = -np.dot(y\_get,X) / X.shape[0] + W\_0 \*lambdal  return G  ...  ...  #NAG  lr = 0.02  epoch = 7  gamma = 0.8  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.shape[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)  #update the parameter W,b  v\_t = v\_t \* gamma + G \* lr  W = W - v\_t  #AdaDelta  lr = 0.05  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)  #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)  #Adam  lr = 0.07  epoch = 4  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)  #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))  #RMSprop  plt.close()  lr = 0.08  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)  #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)) |

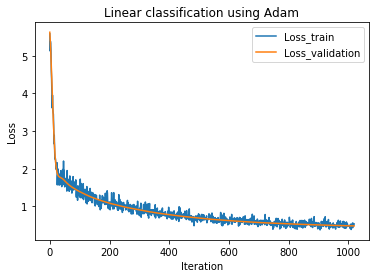
When applying NAG as the optimized method with specific parameters, the results is shown as follows:



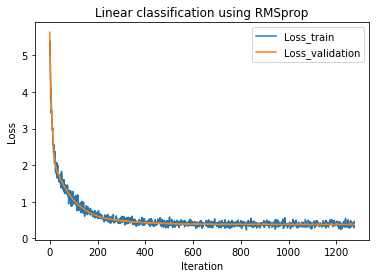
When applying AdaDelta as the optimized method with specific parameters, the results is shown as follows:



When applying Adam as the optimized method with specific parameters, the results is shown as follows:



When applying RMSProp as the optimized method with specific parameters, the results is shown as follows:



# conclusion

For logic regression and stochastic gradient descent, we select Log-likehood loss. We compare the results under four different optimization methods. It looks like the result of using Adam method with learning rate 0.08 is the best.

For the linear classification and stochastic gradient descent, we select Hinge loss. We still compare the results under four different optimization methods. The results shows that linear classification and stochastic gradient descent is better than logic regression under each kind of optimization method.

1. [↑](#footnote-ref-0)