The Experiment Report of Machine Learning



**SUBJECT:**SOFTWARE ENGINEERING

December 14, 2017

Grade:

Postgraduate

Student ID：

201721045909

Supervisor:

Qingyao Wu

Author:

Li Zhang

**SCHOOL:** SCHOOL OF SOFTWARE ENGINEERING

Logistic Regression, Linear Classification and Stochastic Gradient Descent

Abstract—Logistic regression and linear classification are the widely used classification algorithm in the field of machine learning. Stochastic gradient descent is an effective way to find the good model parameters for above algorithm. The experiments compare the effectiveness of different optimized methods.

# INTRODUCTION

Logistic regression and linear classification are the widely used classification algorithm in the field of machine learning. Logistic regression fits the data into a logit function (or logistic function) so that it can predict the probability of an event occurring. Stochastic gradient descent updates the parameters using the small sample of dataset. The experiments compare the effectiveness of different optimized methods that are used in Logistic regression and linear classification.

The rest paper is organized as follow. Section II contains the experiment steps. Section III describes the code and result for the two experiments. Section IV concludes the report.

# 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.

The experiment will be performed by the following steps:

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 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.
7. Repeat step 4 to 6 for several times, and drawing graph of the different optimized method loss and with the number of iterations.

# Experiment

1. Codes of experiments

The codes of two experiments are shown as follow,

1. Logistic Regression

|  |
| --- |
| # 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  path = './a9a.txt'  path\_validation = './a9a.t'  #get the dataset  def get\_data(path):  data = load\_svmlight\_file(path)  return data[0], data[1]  X\_train, y\_train = get\_data(path)  X\_train = X\_train.toarray()  X\_validation, y\_validation = get\_data(path\_validation)  X\_validation = X\_validation.toarray()  #complete the matrix  column = np.zeros(( X\_validation.shape[0]))  X\_validation = np.column\_stack((X\_validation,column))  column\_train = np.ones(( X\_train.shape[0]))  column\_validation = np.ones(( X\_validation.shape[0]))  X\_train = np.column\_stack((X\_train,column\_train))  X\_validation = np.column\_stack((X\_validation,column\_validation))  N = X\_train.shape[1]  W\_zeros = np.zeros(N)  W\_random = random.random(size=N)  W\_normal = np.random.normal(size=N)  #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  #calculate the loss  def cal\_Loss(X,W,y,lambdal):  preY = np.dot(X,W)  Loss = np.sum(np.log(1 + np.exp(-y \* preY)))  Loss = Loss/ X.shape[0] + lambdal / 2 \* np.dot(W,W.T)  return Loss  #calculate the gradient  def cal\_G(X,W,y,lambdal):  preY = np.dot(X,W)  G = np.dot(((-y)/ (1+ np.exp(y\*preY))),X )  G = G / X.shape[0] + W \* lambdal  return G  #draw the result  def draw\_plot(Loss\_train, Loss\_validation, name):  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")  plt.show()  lr = 0.01  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  Loss\_NAG = Loss\_validation  #draw the result  draw\_plot(Loss\_train, Loss\_validation, 'NAG')  lr = 0.03  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))    Loss\_RMS = Loss\_validation  #draw the result  draw\_plot(Loss\_train, Loss\_validation, 'RMSprop')  lr = 0.02  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)  Loss\_Ada = Loss\_validation  #draw the result  draw\_plot(Loss\_train, Loss\_validation, 'AdaDelta')  lr = 0.05  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))  Loss\_Adam = Loss\_validation  #draw the result  draw\_plot(Loss\_train, Loss\_validation, 'Adam')  plt.plot(Loss\_NAG,label="Loss\_NAG")  plt.plot(Loss\_RMS,label="Loss\_RMSProp")  plt.plot(Loss\_Ada,label="Loss\_AdaDelta")  plt.plot(Loss\_Adam,label="Loss\_Adam")  plt.legend()  plt.xlabel("Iteration")  plt.ylabel("Loss")  plt.title("Logistic regression")  plt.show()  print(Loss\_NAG[iteration-1])  print(Loss\_RMS[iteration-1])  print(Loss\_Ada[iteration-1])  print(Loss\_Adam[iteration-1]) |

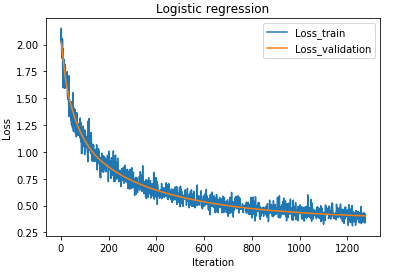
1. Linear Classification

|  |
| --- |
| # Linear classification  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  #mem = Memory("./mycache")  path = './a9a.txt'  path\_validation = './a9a.t'  #@mem.cache  def get\_data(path):  data = load\_svmlight\_file(path)  return data[0], data[1]  X\_train, y\_train = get\_data(path)  X\_train = X\_train.toarray()  X\_validation, y\_validation = get\_data(path\_validation)  X\_validation = X\_validation.toarray()  #complete the matrix  column = np.zeros(( X\_validation.shape[0]))  X\_validation = np.column\_stack((X\_validation,column))  column\_train = np.ones(( X\_train.shape[0]))  column\_validation = np.ones(( X\_validation.shape[0]))  X\_train = np.column\_stack((X\_train,column\_train))  X\_validation = np.column\_stack((X\_validation,column\_validation))  N=X\_train.shape[1]  W\_zeros = np.zeros(N)  W\_random = random.random(size=N)  W\_normal = np.random.normal(size=N)  #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  #calculate the loss  def cal\_Loss(X,W,y,lambdal,W\_0):  preY = np.dot(X,W)  diifY = np.ones(y.shape[0]) - y \* preY  diifY[diifY < 0] =0  Loss =np.sum(diifY) / 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)  diifY = np.ones(y.shape[0]) - y \* preY  y\_get = y.copy()  y\_get[diifY <= 0] =0  G = -np.dot(y\_get,X) / X.shape[0] + W\_0 \*lambdal  return G  #calculate the accuracy  def cal\_Accuracy(X,W,y):  preY = np.dot(X,W)  count = np.sum(preY \* y >0)  Accuracy = count / X.shape[0]  return Accuracy  #draw the result  def draw\_plot(Loss\_train, Loss\_validation, name):  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")  plt.show()  lr = 0.005  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)  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)  #accuracy  Accuracy[j\*runs+l] = cal\_Accuracy(X\_validation,W,y\_validation)  #update the parameter W,b  v\_t = v\_t \* gamma + G \* lr  W = W - v\_t  Loss\_NAG = Loss\_validation  #draw the result  draw\_plot(Loss\_train, Loss\_validation, 'NAG')  lr = 0.06  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)  #accuracy  Accuracy[j\*runs+l] = cal\_Accuracy(X\_validation,W,y\_validation)  #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))    Loss\_RMS = Loss\_validation  #draw the result  draw\_plot(Loss\_train, Loss\_validation, 'RMSprop')  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)  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)  #accuracy  Accuracy[j\*runs+l] = cal\_Accuracy(X\_validation,W,y\_validation)  #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)    Loss\_Ada = Loss\_validation  #draw the result  draw\_plot(Loss\_train, Loss\_validation, 'AdaDelta')  lr = 0.07  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)  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)  #accuracy  Accuracy[j\*runs+l] = cal\_Accuracy(X\_validation,W,y\_validation)  #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))    Loss\_Adam = Loss\_validation  #draw the result  draw\_plot(Loss\_train, Loss\_validation, 'Adam')  plt.plot(Loss\_NAG,label="Loss\_NAG")  plt.plot(Loss\_RMS,label="Loss\_RMSProp")  plt.plot(Loss\_Ada,label="Loss\_AdaDelta")  plt.plot(Loss\_Adam,label="Loss\_Adam")  plt.legend()  plt.xlabel("Iteration")  plt.ylabel("Loss")  plt.title("Logistic regression")  plt.show()  print(Loss\_NAG[iteration-1])  print(Loss\_RMS[iteration-1])  print(Loss\_Ada[iteration-1])  print(Loss\_Adam[iteration-1]) |

1. Experiment results

1) Logistic Regression

In this set of experiments, we compare different optimized methods that are used in logistic regression. The loss of logistic regression using NAG is shown as follow,



The loss of logistic regression using RMSprop is shown as follow,



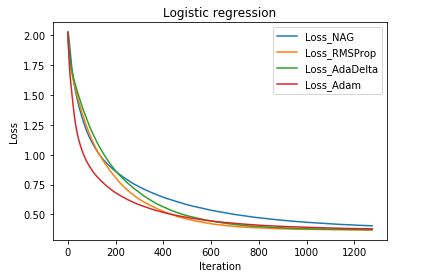
The loss of logistic regression using AdaDelta is shown as follow,



The loss of logistic regression using Adam is shown as follow,



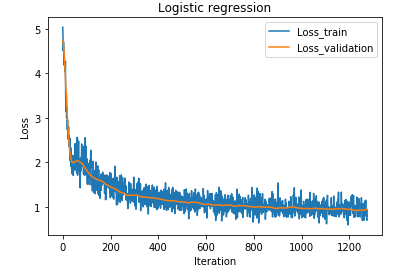
The comparison of these optimized methods is shown as follow,



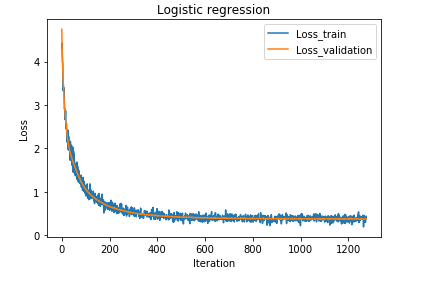
We can see that the loss using NAG is a little higher than others. And the amplitude of train loss is large because of the small simple.

2)Linear Classification

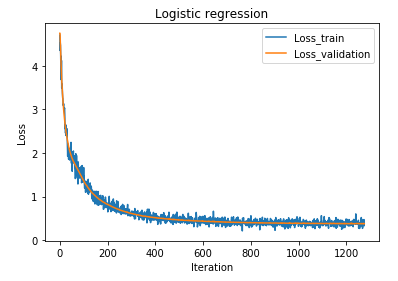
In this set of experiments, we compare different optimized methods that are used in linear classification. The loss of linear classification using NAG is shown as follow,



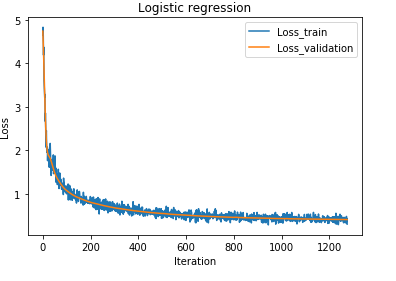
The loss of linear classification using RMSprop is shown as follow,



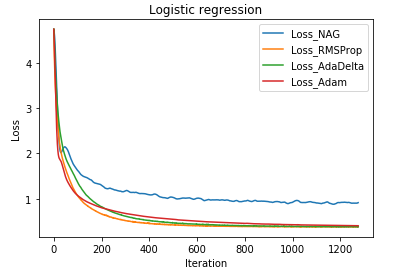
The loss of linear classification using AdaDelta is shown as follow,



The loss of linear classification using Adam is shown as follow,



The comparison of these optimized methods is shown as follow,



We can see that the loss using NAG is a much higher than others. And the amplitude of train loss is large because of the small simple.

# conclusion

After the experiments, we know the algorithm using the different optimized methods can cause different effectiveness. For each algorithm, we need to select the right optimized methods to update its parameters. For example, if we want to train the linear classification, don’t use the NAG method.