

**The Experiment Report of**

**Deep *Learning***

**College Software College**

**Subject Software Engineering**

**Members**  Tengyun Wang

**Student ID 201710106574**

**E-mail wty9391@gmail.com**

**Tutor**   **Mingkui Tang**

**Date submitted** **2017.12 .12**

1. **Topic:**

Logistic Regression, Linear Classification and Stochastic Gradient Descent

**2. Time:**

2017.12.12

**3. Reporter:**

Tengyun Wang

**4. Purposes:**

* 1. Compare and understand the difference between gradient descent and stochastic gradient descent.
  2. Compare and understand the differences and relationships between Logistic regression and linear classification.
  3. Further understand the principles of SVM and practice on larger data.

**5. Data sets and data analysis:**

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.

**6. Experimental steps:**

The experimental code and drawing are completed on jupyter.

* **Logistic Regression and Stochastic Gradient Descent**

1. Load the training set and validation set.
2. Initalize logistic regression model parameters, you can consider initalizing 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 ，， and .
7. Repeate 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. Initalize SVM model parameters, you can consider initalizing 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 ，， and .
7. Repeate step 4 to 6 for several times, and drawing graph of ，， and **with the number of iterations**.

**7. Code:**

(Fill in the contents of 8-11 respectively for logistic regression and linear classification)

optimizer.py

1. #!/usr/bin/env python3
2. # -\*- coding: utf-8 -\*-
3. """
4. Created on Wed Dec  6 21:08:36 2017
6. @author: wty
7. """
8. **import** numpy as np
10. epsilon = 1e-8
12. **def** gradient\_decent(w,gradient,eta):
13. **return** w - eta \* gradient
15. **def** NAG(w,gradient,last\_delta,eta,gamma):
16. delta = gamma \* last\_delta + eta \* gradient
17. **return** w-delta,delta
19. **def** Adadelta(w,gradient,last\_E\_g\_2,last\_E\_delta\_2,gamma):
20. E\_g\_2 = gamma \* last\_E\_g\_2 + (1-gamma) \* (gradient\*gradient)
21. RMS\_g = np.sqrt(E\_g\_2+epsilon)
22. RMS\_last\_delta = np.sqrt(last\_E\_delta\_2+epsilon)
23. delta = RMS\_last\_delta / RMS\_g \* gradient
24. **return** w - delta,\
25. E\_g\_2,\
26. gamma \* last\_E\_delta\_2 + (1-gamma) \* (delta\*delta)
28. **def** RMSprop(w,gradient,last\_E\_g\_2,eta,gamma):
29. E\_g\_2 = gamma \* last\_E\_g\_2 + (1-gamma) \* (gradient\*gradient)
30. **return** w - (eta/np.sqrt(E\_g\_2+epsilon))\*gradient,\
31. E\_g\_2
33. **def** Adam(w,gradient,last\_m,last\_v,eta,beta1,beta2,epoch,counteract\_bias=True):
34. m = beta1 \* last\_m + (1-beta1) \* gradient
35. v = beta2 \* last\_v + (1-beta2) \* (gradient\*gradient)
36. **if** counteract\_bias:
37. m = m/(1-beta1\*\*epoch)
38. v = v/(1-beta2\*\*epoch)
40. **return** w - (eta/(np.sqrt(v)+epsilon))\*m,\
41. m\*(1-beta1\*\*epoch),v\*(1-beta2\*\*epoch)

* **Logistic Regression and Stochastic Gradient Descent**

LogiticRegressoinClassifier.py

1. #!/usr/bin/env python3
2. # -\*- coding: utf-8 -\*-
3. """
4. Created on Mon Dec 11 13:40:41 2017
6. @author: wty
7. """

10. **import** numpy as np
11. **from** sklearn.base **import** BaseEstimator,ClassifierMixin
12. **import** math
14. **import** optimizer as op
16. **class** Classifier(BaseEstimator,ClassifierMixin):
17. """A Logistic regression Classifier for ML2017-lab-02"""
19. **def** \_\_init\_\_(self, w=0, lamda=0.1, eta=0.05, gamma=0.9,\
20. threshold=0.5, max\_iterate=100, batch\_size=10000,\
21. Adam\_beta1=0.9, Adam\_beta2=0.999,\
22. Adadelta\_last\_E\_delta\_2\_init=1e-4, optimizer='GD'):
23. """
24. Called when initializing the classifier,
25. optimizer expectes {'NAG','Adadelta','RMSprop','Adam','GD'}
26. """
27. self.w = w
28. self.lamda = lamda
29. self.eta = eta
30. self.gamma = gamma
31. self.threshold = threshold
32. self.max\_iterate = max\_iterate
33. self.batch\_size = batch\_size
34. self.Adam\_beta1 = Adam\_beta1
35. self.Adam\_beta2 = Adam\_beta2
36. self.Adadelta\_last\_E\_delta\_2\_init = Adadelta\_last\_E\_delta\_2\_init
37. self.optimizer = optimizer
39. self.w\_history = []
41. **def** sigmoid(self,inX):
42. **return** 1.0 / (1 + np.exp(-inX))
44. **def** \_\_h(self,w,X):
45. **return** self.sigmoid(X.dot(w))
47. **def** h(self,X):
48. **return** self.\_\_h(self.w,X)
50. **def** L(self,X,Y):
51. **return** self.\_\_L(self.w,X,Y)
53. **def** \_\_L(self,w,X,Y):
54. num\_records,num\_features  = np.shape(X)
55. lamda = self.lamda
57. hx = self.\_\_h(w,X)
58. regulation\_loss = 1.0/2 \* lamda \* w.transpose().dot(w)
59. loss = 1.0/2 \* 1.0/(1 - -1) \* 1.0/num\_records \* (-np.log(hx).transpose().dot(1+Y) - np.log(1-hx).transpose().dot(1-Y))\
60. + regulation\_loss
62. **return** loss[0][0]
64. **def** g(self,X,Y):
65. **return** self.\_\_g(self.w,X,Y)
67. **def** \_\_g(self,w,X,Y):
68. num\_records,num\_features  = np.shape(X)
69. lamda = self.lamda
71. # L2 norm
72. **return** 1.0/num\_records \* X.transpose().dot(2\*self.\_\_h(w,X)-(Y+1)) \
73. + lamda \* w

76. **def** fit(self, X, Y):
77. """
78. A reference implementation of a fitting function
79. Parameters
80. ----------
81. X : array-like or sparse matrix of shape = [n\_samples, n\_features]
82. The training input samples.
83. y : array-like, shape = [n\_samples] or [n\_samples, n\_outputs]
84. The target values (class labels in classification, real numbers in
85. regression).
86. Returns
87. -------
88. self : object
89. Returns self.
90. """
91. self.classes\_, \_ = np.unique(Y, return\_inverse=True)
92. train\_size,num\_features  = np.shape(X)
93. batch\_size = self.batch\_size
94. optimizer = self.optimizer
95. eta = self.eta
96. gamma = self.gamma
98. NAG\_last\_delta = np.zeros((num\_features,1))
100. Adadelta\_last\_E\_g\_2 = np.zeros((num\_features,1))
101. Adadelta\_last\_E\_delta\_2 = np.zeros((num\_features,1)) + self.Adadelta\_last\_E\_delta\_2\_init
103. RMSprop\_last\_E\_g\_2 = np.zeros((num\_features,1))
105. Adam\_last\_m = np.zeros((num\_features,1))
106. Adam\_last\_v = np.zeros((num\_features,1))
108. epoch = 0
109. self.w\_history.append(self.w)
110. **for** counter **in** range(self.max\_iterate):
111. starts = [i\*batch\_size **for** i **in** range(math.ceil(train\_size/batch\_size))]
112. ends = [i\*batch\_size **for** i **in** range(1,math.ceil(train\_size/batch\_size))]
113. ends.append(train\_size)
114. **for** start, end **in** zip(starts, ends):
116. **if** optimizer == 'NAG':
117. # Nesterov accelerated gradient decent
118. self.w,NAG\_last\_delta =\
119. op.NAG(self.w,
120. self.g(X[start:end,:],Y[start:end,:]),
121. NAG\_last\_delta,eta,gamma)
122. self.w\_history.append(self.w)
123. **elif** optimizer == 'Adadelta':
124. # Adadelta gradient decent
125. self.w, Adadelta\_last\_E\_g\_2, Adadelta\_last\_E\_delta\_2 =\
126. op.Adadelta(self.w,
127. self.g(X[start:end,:],Y[start:end,:]),
128. Adadelta\_last\_E\_g\_2,
129. Adadelta\_last\_E\_delta\_2,
130. gamma)
131. self.w\_history.append(self.w)
132. **elif** optimizer == 'RMSprop':
133. # RMSprop gradient decent
134. self.w, RMSprop\_last\_E\_g\_2 =\
135. op.RMSprop(self.w,
136. self.g(X[start:end,:],Y[start:end,:]),
137. RMSprop\_last\_E\_g\_2,
138. eta,gamma)
139. self.w\_history.append(self.w)
140. **elif** optimizer == 'Adam':
141. # Adaptive Moment Estimation
142. self.w, Adam\_last\_m, Adam\_last\_v =\
143. op.Adam(self.w,
144. self.g(X[start:end,:],Y[start:end,:]),
145. Adam\_last\_m,
146. Adam\_last\_v,
147. eta,self.Adam\_beta1,self.Adam\_beta2,epoch+1)
148. self.w\_history.append(self.w)
149. **elif** optimizer == 'GD':
150. # mini-batch gradient decent
151. self.w = op.gradient\_decent(self.w,
152. self.g(X[start:end,:],Y[start:end,:]),
153. eta)
154. self.w\_history.append(self.w)
156. **else**:
157. **raise** ValueError("Optimizer error, expected {'NAG','Adadelta','RMSprop','Adam','GD'}, got %s" % optimizer)
159. epoch += 1
161. **return** self
163. **def** \_\_predict(self,w,X):
164. threshold = self.threshold
165. raw = self.\_\_h(w,X)
166. raw[raw<=threshold] = self.classes\_[0]
167. raw[raw>threshold] = self.classes\_[1]
168. **return** raw
170. **def** predict(self, X):
171. """ A reference implementation of a predicting function.
172. Parameters
173. ----------
174. X : array-like of shape = [n\_samples, n\_features]
175. The input samples.
176. Returns
177. -------
178. y : array of shape = [n\_samples]
179. Returns :math:`x^2` where :math:`x` is the first column of `X`.
180. """
181. **return** self.\_\_predict(self.w,X)
183. **def** \_\_score(self,w,X,Y):
184. num\_records,num\_features  = np.shape(X)
185. P = self.\_\_predict(w,X)
187. is\_right = P \* Y
188. is\_right[is\_right < 0] = 0
190. **return** 1.0/num\_records \* np.count\_nonzero(is\_right)
192. **def** score(self, X, Y):
193. # RMSE
194. **return** self.\_\_score(self.w,X,Y)
196. **def** getLossHistory(self,X,Y):
197. **return** [self.\_\_L(w,X,Y) **for** w **in** self.w\_history]
199. **def** getScoreHistory(self,X,Y):
200. **return** [self.\_\_score(w,X,Y) **for** w **in** self.w\_history]

RegressionExperiment.py

1. #!/usr/bin/env python3
2. # -\*- coding: utf-8 -\*-
3. """
4. Created on Sat Dec  2 19:17:35 2017
6. @author: wty
7. """
8. **import** numpy as np
9. **import** scipy
10. **import** matplotlib.pyplot as plt
11. **import** pandas as pd
13. **from** sklearn.datasets **import** load\_svmlight\_file
14. **from** sklearn.model\_selection **import** GridSearchCV
16. **import** LogiticRegressionClassifier as LRC
18. **def** plotFigure(GD,NAG,Adadelta,RMSprop,Adam,X,Y,tuned='tuned'):
19. GD\_loss\_test = GD.getLossHistory(X,Y)
20. NAG\_loss\_test = NAG.getLossHistory(X,Y)
21. Adadelta\_loss\_test = Adadelta.getLossHistory(X,Y)
22. RMSprop\_loss\_test = RMSprop.getLossHistory(X,Y)
23. Adam\_loss\_test = Adam.getLossHistory(X,Y)
25. \_, ax = plt.subplots()
26. ax.plot(range(len(GD\_loss\_test)),GD\_loss\_test,label=\
27. r'GD,$\lambda$=%.2f,$\eta$=%.2f'\
28. %(GD.get\_params()['lamda'],GD.get\_params()['eta']))
29. ax.plot(range(len(NAG\_loss\_test)),NAG\_loss\_test,label=\
30. r'NAG,$\lambda$=%.2f,$\eta$=%.2f,$\gamma$=%.2f'\
31. %(NAG.get\_params()['lamda'],NAG.get\_params()['eta'],NAG.get\_params()['gamma']))
32. ax.plot(range(len(Adadelta\_loss\_test)),Adadelta\_loss\_test,label=\
33. r'Adadelta,$\lambda$=%.2f,$\gamma$=%.2f'\
34. %(Adadelta.get\_params()['lamda'],Adadelta.get\_params()['gamma']))
35. ax.plot(range(len(RMSprop\_loss\_test)),RMSprop\_loss\_test,label=\
36. r'RMSprop,$\lambda$=%.2f,$\eta$=%.2f,$\gamma$=%.2f'\
37. %(RMSprop.get\_params()['lamda'],RMSprop.get\_params()['eta'],RMSprop.get\_params()['gamma']))
38. ax.plot(range(len(Adam\_loss\_test)),Adam\_loss\_test,label=\
39. r'Adam,$\lambda$=%.2f,$\eta$=%.2f,$\beta\_1$=%.2f,$\beta\_2$=%.3f'\
40. %(Adam.get\_params()['lamda'],Adam.get\_params()['eta'],Adam.get\_params()['Adam\_beta1'],Adam.get\_params()['Adam\_beta2']))
42. plt.legend()
43. plt.title('Different %s estimators\' performance'%tuned)
44. ax.set(xlabel='Epoch', ylabel='Loss in testset with l2 norm')
45. plt.show()
46. plt.close('all')
48. result\_path = './results/regression/grid\_search'
50. X\_train, Y\_train = load\_svmlight\_file("./resources/a9a")
51. X\_test, Y\_test = load\_svmlight\_file("./resources/a9a.t")
53. X\_train = scipy.sparse.hstack(\
54. (scipy.sparse.csr\_matrix(np.ones((len(Y\_train),1))),X\_train))
55. X\_test = scipy.sparse.hstack(\
56. (scipy.sparse.csr\_matrix(np.ones((len(Y\_test),1))),X\_test))
57. #Something wrong with this dataset
58. X\_test = scipy.sparse.hstack(\
59. (X\_test, scipy.sparse.csr\_matrix(np.zeros((len(Y\_test),1)))))
61. X\_train = X\_train.tocsr()
62. X\_test = X\_test.tocsr()
64. Y\_train = Y\_train.reshape((len(Y\_train),1))
65. Y\_test = Y\_test.reshape((len(Y\_test),1))
67. train\_size,num\_features  = np.shape(X\_train)
69. max\_iterate = 50
70. batch\_size = 8000
72. figure\_num = 2
74. init\_w = np.random.normal(size=(num\_features,1))
76. optimizers = ['NAG','Adadelta','RMSprop','Adam','GD']
77. o = {}
79. param\_grid = {
80. 'NAG': {'lamda': [0.01, 0.1],
81. 'eta': [0.01, 0.05],
82. 'gamma': [0.8, 0.9, 0.95],
83. 'threshold': [0.5,0.6]},
84. 'Adadelta' : {'lamda': [0.01, 0.1],
85. 'gamma': [0.8, 0.9, 0.95],
86. 'threshold': [0.5,0.6]},
87. 'RMSprop' : {'lamda': [0.01, 0.1],
88. 'eta': [0.01, 0.05],
89. 'gamma': [0.8, 0.9, 0.95],
90. 'threshold': [0.5,0.6]},
91. 'Adam' : {'lamda': [0.01, 0.1],
92. 'eta': [0.01, 0.05],
93. 'Adam\_beta1': [0.9, 0.95],
94. 'Adam\_beta2' : [0.99, 0.999],
95. 'threshold': [0.5,0.6]},
96. 'GD' : {'lamda': [0.01, 0.1, 0.5],
97. 'eta': [0.1, 0.2, 0.3, 0.4, 0.5],
98. 'threshold': [0.4,0.5,0.6]}}
100. **print** ("===========================")
101. **print** ("Start to execute exhaustive grid search")
102. **for** i **in** range(len(optimizers)):
103. optimizer\_name = optimizers[i]
105. cls = GridSearchCV(LRC.Classifier(init\_w,max\_iterate=max\_iterate,batch\_size=batch\_size,optimizer=optimizer\_name), param\_grid[optimizer\_name],return\_train\_score=True,n\_jobs=4)
106. cls.fit(X\_train,Y\_train)
107. result = pd.DataFrame(cls.cv\_results\_)
108. result.sort\_values('rank\_test\_score',inplace=True)
109. result = result.reset\_index(drop = True)
111. # Best optimizer
112. o[optimizer\_name] = cls.best\_estimator\_
114. **print** ("Exhaustive Grid Search Result of %s"%optimizer\_name)
115. **print** ("The best estimator's parameter is",cls.best\_params\_)
116. **print** (result.loc[0:5,['rank\_test\_score','mean\_test\_score','mean\_train\_score','mean\_fit\_time','params']])
117. path = result\_path+'\_'+optimizer\_name+'.csv'
118. result.to\_csv(path)
119. **print** ("Result has been saved in",path)

122. **print** ("Printing the best %d models loss curves"%figure\_num)
123. **for** j **in** range(figure\_num):
124. params = result.loc[i,'params']
125. **print** ("Figure of",params)
126. cls = LRC.Classifier(init\_w,max\_iterate=max\_iterate,batch\_size=batch\_size,optimizer=optimizer\_name,\*\*params)
127. cls.fit(X\_train,Y\_train)
128. loss\_train = cls.getLossHistory(X\_train,Y\_train)
129. loss\_test = cls.getLossHistory(X\_test,Y\_test)
130. accuracy\_train = cls.getScoreHistory(X\_train,Y\_train)
131. accuracy\_test = cls.getScoreHistory(X\_test,Y\_test)
133. plt.figure(j)
134. \_, ax = plt.subplots()
135. ax\_e = ax.twinx()
136. ax.plot(range(len(loss\_train)),loss\_train,label='train loss')
137. ax.plot(range(len(loss\_test)),loss\_test,label='test loss')
138. ax\_e.plot(range(len(accuracy\_train)),accuracy\_train,'r',label='train accuracy')
139. ax\_e.plot(range(len(accuracy\_test)),accuracy\_test,'g',label='test accuracy')
141. ax.set(xlabel='Epoch', ylabel='Loss with l2 norm')
142. ax\_e.set\_ylabel('Accuracy with threshold=%s'%str(cls.get\_params()['threshold']))
144. ax.legend(loc=4)
145. ax\_e.legend(loc=1)
146. plt.show()
148. plt.close('all')

151. **print** ("===========================")
152. **print** ("Start to figure the accuracy **and** loss curves of\
153. estimators of different optimized algorithms with tuned hyperparameter")
154. **for** i **in** range(len(optimizers)):
155. cls\_name = optimizers[i]
156. cls = o[cls\_name]
158. **print**("Optimizer %s, parameters:"%cls\_name)
159. params = cls.get\_params()
160. params.pop('w')
161. **print**(params)
163. loss\_train = cls.getLossHistory(X\_train,Y\_train)
164. loss\_test = cls.getLossHistory(X\_test,Y\_test)
165. accuracy\_train = cls.getScoreHistory(X\_train,Y\_train)
166. accuracy\_test = cls.getScoreHistory(X\_test,Y\_test)
168. plt.figure(i)
169. \_, ax = plt.subplots()
170. ax\_e = ax.twinx()
171. ax.plot(range(len(loss\_train)),loss\_train,label='train loss')
172. ax.plot(range(len(loss\_test)),loss\_test,label='test loss')
173. ax\_e.plot(range(len(accuracy\_train)),accuracy\_train,'r',label='train accuracy')
174. ax\_e.plot(range(len(accuracy\_test)),accuracy\_test,'g',label='test accuracy')
176. ax.set(xlabel='Epoch', ylabel='Loss with l2 norm')
177. ax\_e.set\_ylabel('Accuracy with threshold=%s'%str(cls.get\_params()['threshold']))
179. ax.legend(loc=4)
180. ax\_e.legend(loc=1)
181. plt.title(cls\_name+' Gradient Decent')
182. plt.show()
183. plt.close('all')
185. **print** ("===========================")
186. **print** ("Start to figure the loss curves of\
187. tuned estimators **in** one figure")
189. GD = o['GD']
190. NAG = o['NAG']
191. Adadelta = o['Adadelta']
192. RMSprop = o['RMSprop']
193. Adam = o['Adam']
195. plotFigure(GD,NAG,Adadelta,RMSprop,Adam,X\_test,Y\_test,'tuned')
197. init\_w = np.random.normal(size=(num\_features,1))
198. GD = LRC.Classifier(w=init\_w,optimizer='GD')
199. NAG = LRC.Classifier(w=init\_w,optimizer='NAG')
200. Adadelta = LRC.Classifier(w=init\_w,optimizer='Adadelta')
201. RMSprop = LRC.Classifier(w=init\_w,optimizer='RMSprop')
202. Adam = LRC.Classifier(w=init\_w,optimizer='Adam')
204. GD.fit(X\_train,Y\_train)
205. NAG.fit(X\_train,Y\_train)
206. Adadelta.fit(X\_train,Y\_train)
207. RMSprop.fit(X\_train,Y\_train)
208. Adam.fit(X\_train,Y\_train)
210. plotFigure(GD,NAG,Adadelta,RMSprop,Adam,X\_test,Y\_test,'untuned')

* **Linear Classification and Stochastic Gradient Descent**

LinearClassifier.py

1. #!/usr/bin/env python3
2. # -\*- coding: utf-8 -\*-
3. """
4. Created on Tue Dec 12 16:06:02 2017
6. @author: wty
7. """

10. **import** numpy as np
11. **from** sklearn.base **import** BaseEstimator,ClassifierMixin
12. **import** math
14. **import** optimizer as op
16. **class** Classifier(BaseEstimator,ClassifierMixin):
17. """A Linear Classifier for ML2017-lab-02"""
19. **def** \_\_init\_\_(self, w=0, lamda=0.1, eta=0.05, C=1.0, gamma=0.9,\
20. threshold=0.5, max\_iterate=100, batch\_size=10000,\
21. Adam\_beta1=0.9, Adam\_beta2=0.999,\
22. Adadelta\_last\_E\_delta\_2\_init=1e-4, optimizer='GD'):
23. """
24. Called when initializing the classifier,
25. optimizer expectes {'NAG','Adadelta','RMSprop','Adam','GD'}
26. """
27. self.w = w
28. self.lamda = lamda
29. self.eta = eta
30. self.C = C
31. self.gamma = gamma
32. self.threshold = threshold
33. self.max\_iterate = max\_iterate
34. self.batch\_size = batch\_size
35. self.Adam\_beta1 = Adam\_beta1
36. self.Adam\_beta2 = Adam\_beta2
37. self.Adadelta\_last\_E\_delta\_2\_init = Adadelta\_last\_E\_delta\_2\_init
38. self.optimizer = optimizer
40. self.w\_history = []
42. **def** \_\_h(self,w,X):
43. **return** X.dot(w)
45. **def** h(self,X):
46. **return** self.\_\_h(self.w,X)
48. **def** \_\_hinge\_loss(self,w,X,Y):
49. num\_records,num\_features  = np.shape(X)
50. C = self.C
51. zero = np.zeros((num\_records,1))
52. margin = 1 - C \* Y \* self.\_\_h(w,X)
53. **return** np.max([zero,margin],axis=0)
55. **def** hinge\_loss(self,X,Y):
56. **return** self.\_\_hinge\_loss(self.w,X,Y)
58. **def** L(self,X,Y):
59. **return** self.\_\_L(self.w,X,Y)
61. **def** \_\_L(self,w,X,Y):
62. num\_records,num\_features  = np.shape(X)
63. lamda = self.lamda
64. e = self.\_\_hinge\_loss(w,X,Y)
65. regulation\_loss = 1.0/2 \* lamda \* w.transpose().dot(w)
66. loss = 1.0/float(num\_records) \* e.sum()  + regulation\_loss
67. **return** loss[0][0]
69. **def** g(self,X,Y):
70. **return** self.\_\_g(self.w,X,Y)
72. **def** \_\_g(self,w,X,Y):
73. num\_records,num\_features  = np.shape(X)
74. C = self.C
75. lamda = self.lamda
76. e = self.\_\_hinge\_loss(w,X,Y)
77. indicator = np.zeros((num\_records,1))
78. indicator[np.nonzero(e)] = 1
80. **return** - 1.0/float(num\_records) \* C \
81. \* X.transpose().dot(Y \* indicator).sum(axis=1).reshape((num\_features,1)) \
82. + lamda \* w

85. **def** fit(self, X, Y):
86. """
87. A reference implementation of a fitting function
88. Parameters
89. ----------
90. X : array-like or sparse matrix of shape = [n\_samples, n\_features]
91. The training input samples.
92. y : array-like, shape = [n\_samples] or [n\_samples, n\_outputs]
93. The target values (class labels in classification, real numbers in
94. regression).
95. Returns
96. -------
97. self : object
98. Returns self.
99. """
100. self.classes\_, \_ = np.unique(Y, return\_inverse=True)
101. train\_size,num\_features  = np.shape(X)
102. batch\_size = self.batch\_size
103. optimizer = self.optimizer
104. eta = self.eta
105. gamma = self.gamma
107. NAG\_last\_delta = np.zeros((num\_features,1))
109. Adadelta\_last\_E\_g\_2 = np.zeros((num\_features,1))
110. Adadelta\_last\_E\_delta\_2 = np.zeros((num\_features,1)) + self.Adadelta\_last\_E\_delta\_2\_init
112. RMSprop\_last\_E\_g\_2 = np.zeros((num\_features,1))
114. Adam\_last\_m = np.zeros((num\_features,1))
115. Adam\_last\_v = np.zeros((num\_features,1))
117. epoch = 0
118. self.w\_history.append(self.w)
119. **for** counter **in** range(self.max\_iterate):
120. starts = [i\*batch\_size **for** i **in** range(math.ceil(train\_size/batch\_size))]
121. ends = [i\*batch\_size **for** i **in** range(1,math.ceil(train\_size/batch\_size))]
122. ends.append(train\_size)
123. **for** start, end **in** zip(starts, ends):
125. **if** optimizer == 'NAG':
126. # Nesterov accelerated gradient decent
127. self.w,NAG\_last\_delta =\
128. op.NAG(self.w,
129. self.g(X[start:end,:],Y[start:end,:]),
130. NAG\_last\_delta,eta,gamma)
131. self.w\_history.append(self.w)
132. **elif** optimizer == 'Adadelta':
133. # Adadelta gradient decent
134. self.w, Adadelta\_last\_E\_g\_2, Adadelta\_last\_E\_delta\_2 =\
135. op.Adadelta(self.w,
136. self.g(X[start:end,:],Y[start:end,:]),
137. Adadelta\_last\_E\_g\_2,
138. Adadelta\_last\_E\_delta\_2,
139. gamma)
140. self.w\_history.append(self.w)
141. **elif** optimizer == 'RMSprop':
142. # RMSprop gradient decent
143. self.w, RMSprop\_last\_E\_g\_2 =\
144. op.RMSprop(self.w,
145. self.g(X[start:end,:],Y[start:end,:]),
146. RMSprop\_last\_E\_g\_2,
147. eta,gamma)
148. self.w\_history.append(self.w)
149. **elif** optimizer == 'Adam':
150. # Adaptive Moment Estimation
151. self.w, Adam\_last\_m, Adam\_last\_v =\
152. op.Adam(self.w,
153. self.g(X[start:end,:],Y[start:end,:]),
154. Adam\_last\_m,
155. Adam\_last\_v,
156. eta,self.Adam\_beta1,self.Adam\_beta2,epoch+1)
157. self.w\_history.append(self.w)
158. **elif** optimizer == 'GD':
159. # mini-batch gradient decent
160. self.w = op.gradient\_decent(self.w,
161. self.g(X[start:end,:],Y[start:end,:]),
162. eta)
163. self.w\_history.append(self.w)
165. **else**:
166. **raise** ValueError("Optimizer error, expected {'NAG','Adadelta','RMSprop','Adam','GD'}, got %s" % optimizer)
168. epoch += 1
170. **return** self
172. **def** \_\_predict(self,w,X):
173. threshold = self.threshold
174. raw = self.\_\_h(w,X)
175. raw[raw<=threshold] = self.classes\_[0]
176. raw[raw>threshold] = self.classes\_[1]
177. **return** raw
179. **def** predict(self, X):
180. """ A reference implementation of a predicting function.
181. Parameters
182. ----------
183. X : array-like of shape = [n\_samples, n\_features]
184. The input samples.
185. Returns
186. -------
187. y : array of shape = [n\_samples]
188. Returns :math:`x^2` where :math:`x` is the first column of `X`.
189. """
190. **return** self.\_\_predict(self.w,X)
192. **def** \_\_score(self,w,X,Y):
193. num\_records,num\_features  = np.shape(X)
194. P = self.\_\_predict(w,X)
196. is\_right = P \* Y
197. is\_right[is\_right < 0] = 0
199. **return** 1.0/num\_records \* np.count\_nonzero(is\_right)
201. **def** score(self, X, Y):
202. # RMSE
203. **return** self.\_\_score(self.w,X,Y)
205. **def** getLossHistory(self,X,Y):
206. **return** [self.\_\_L(w,X,Y) **for** w **in** self.w\_history]
208. **def** getScoreHistory(self,X,Y):
209. **return** [self.\_\_score(w,X,Y) **for** w **in** self.w\_history]

ClassificationExperiment.py

1. #!/usr/bin/env python3
2. # -\*- coding: utf-8 -\*-
3. """
4. Created on Wed Dec  6 20:58:35 2017
6. @author: wty
7. """
8. **import** numpy as np
9. **import** scipy
10. **import** matplotlib.pyplot as plt
11. **import** pandas as pd
13. **from** sklearn.datasets **import** load\_svmlight\_file
14. **from** sklearn.model\_selection **import** GridSearchCV
16. **import** LinearClassifier as LC
18. **def** plotFigure(GD,NAG,Adadelta,RMSprop,Adam,X,Y,tuned='tuned'):
19. GD\_loss\_test = GD.getLossHistory(X,Y)
20. NAG\_loss\_test = NAG.getLossHistory(X,Y)
21. Adadelta\_loss\_test = Adadelta.getLossHistory(X,Y)
22. RMSprop\_loss\_test = RMSprop.getLossHistory(X,Y)
23. Adam\_loss\_test = Adam.getLossHistory(X,Y)
25. \_, ax = plt.subplots()
26. ax.plot(range(len(GD\_loss\_test)),GD\_loss\_test,label=\
27. r'GD,$\lambda$=%.2f,$\eta$=%.2f'\
28. %(GD.get\_params()['lamda'],GD.get\_params()['eta']))
29. ax.plot(range(len(NAG\_loss\_test)),NAG\_loss\_test,label=\
30. r'NAG,$\lambda$=%.2f,$\eta$=%.2f,$\gamma$=%.2f'\
31. %(NAG.get\_params()['lamda'],NAG.get\_params()['eta'],NAG.get\_params()['gamma']))
32. ax.plot(range(len(Adadelta\_loss\_test)),Adadelta\_loss\_test,label=\
33. r'Adadelta,$\lambda$=%.2f,$\gamma$=%.2f'\
34. %(Adadelta.get\_params()['lamda'],Adadelta.get\_params()['gamma']))
35. ax.plot(range(len(RMSprop\_loss\_test)),RMSprop\_loss\_test,label=\
36. r'RMSprop,$\lambda$=%.2f,$\eta$=%.2f,$\gamma$=%.2f'\
37. %(RMSprop.get\_params()['lamda'],RMSprop.get\_params()['eta'],RMSprop.get\_params()['gamma']))
38. ax.plot(range(len(Adam\_loss\_test)),Adam\_loss\_test,label=\
39. r'Adam,$\lambda$=%.2f,$\eta$=%.2f,$\beta\_1$=%.2f,$\beta\_2$=%.3f'\
40. %(Adam.get\_params()['lamda'],Adam.get\_params()['eta'],Adam.get\_params()['Adam\_beta1'],Adam.get\_params()['Adam\_beta2']))
42. plt.legend()
43. plt.title('Different %s estimators\' performance'%tuned)
44. ax.set(xlabel='Epoch', ylabel='Loss in testset with l2 norm')
45. plt.show()
46. plt.close('all')

49. result\_path = './results/classification/grid\_search'
51. X\_train, Y\_train = load\_svmlight\_file("./resources/a9a")
52. X\_test, Y\_test = load\_svmlight\_file("./resources/a9a.t")
54. X\_train = scipy.sparse.hstack(\
55. (scipy.sparse.csr\_matrix(np.ones((len(Y\_train),1))),X\_train))
56. X\_test = scipy.sparse.hstack(\
57. (scipy.sparse.csr\_matrix(np.ones((len(Y\_test),1))),X\_test))
58. #Something wrong with this dataset
59. X\_test = scipy.sparse.hstack(\
60. (X\_test, scipy.sparse.csr\_matrix(np.zeros((len(Y\_test),1)))))
62. X\_train = X\_train.tocsr()
63. X\_test = X\_test.tocsr()
65. Y\_train = Y\_train.reshape((len(Y\_train),1))
66. Y\_test = Y\_test.reshape((len(Y\_test),1))
68. train\_size,num\_features  = np.shape(X\_train)
70. max\_iterate = 50
71. batch\_size = 8000
73. figure\_num = 2
75. init\_w = np.random.normal(size=(num\_features,1))
77. optimizers = ['NAG','Adadelta','RMSprop','Adam','GD']
78. o = {}
80. param\_grid = {
81. 'NAG': {'lamda': [0.01, 0.1],
82. 'eta': [0.01, 0.05],
83. 'gamma': [0.8, 0.9, 0.95],
84. 'threshold': [0.5,0.6]},
85. 'Adadelta' : {'lamda': [0.01, 0.1],
86. 'gamma': [0.8, 0.9, 0.95],
87. 'threshold': [0.5,0.6]},
88. 'RMSprop' : {'lamda': [0.01, 0.1],
89. 'eta': [0.01, 0.05],
90. 'gamma': [0.8, 0.9, 0.95],
91. 'threshold': [0.5,0.6]},
92. 'Adam' : {'lamda': [0.01, 0.1],
93. 'eta': [0.01, 0.05],
94. 'Adam\_beta1': [0.9, 0.95],
95. 'Adam\_beta2' : [0.99, 0.999],
96. 'threshold': [0.5,0.6]},
97. 'GD' : {'lamda': [0.01, 0.1, 0.5],
98. 'eta': [0.1, 0.2, 0.3, 0.4, 0.5],
99. 'threshold': [0.4,0.5,0.6]}}
101. **print** ("===========================")
102. **print** ("Start to execute exhaustive grid search")
103. **for** i **in** range(len(optimizers)):
104. optimizer\_name = optimizers[i]
106. cls = GridSearchCV(LC.Classifier(init\_w,max\_iterate=max\_iterate,batch\_size=batch\_size,optimizer=optimizer\_name), param\_grid[optimizer\_name],return\_train\_score=True,n\_jobs=4)
107. cls.fit(X\_train,Y\_train)
108. result = pd.DataFrame(cls.cv\_results\_)
109. result.sort\_values('rank\_test\_score',inplace=True)
110. result = result.reset\_index(drop = True)
112. # Best optimizer
113. o[optimizer\_name] = cls.best\_estimator\_
115. **print** ("Exhaustive Grid Search Result of %s"%optimizer\_name)
116. **print** ("The best estimator's parameter is",cls.best\_params\_)
117. **print** (result.loc[0:5,['rank\_test\_score','mean\_test\_score','mean\_train\_score','mean\_fit\_time','params']])
118. path = result\_path+'\_'+optimizer\_name+'.csv'
119. result.to\_csv(path)
120. **print** ("Result has been saved in",path)

123. **print** ("Printing the best %d models loss curves"%figure\_num)
124. **for** j **in** range(figure\_num):
125. params = result.loc[i,'params']
126. **print** ("Figure of",params)
127. cls = LC.Classifier(init\_w,max\_iterate=max\_iterate,batch\_size=batch\_size,optimizer=optimizer\_name,\*\*params)
128. cls.fit(X\_train,Y\_train)
129. loss\_train = cls.getLossHistory(X\_train,Y\_train)
130. loss\_test = cls.getLossHistory(X\_test,Y\_test)
131. accuracy\_train = cls.getScoreHistory(X\_train,Y\_train)
132. accuracy\_test = cls.getScoreHistory(X\_test,Y\_test)
134. plt.figure(j)
135. \_, ax = plt.subplots()
136. ax\_e = ax.twinx()
137. ax.plot(range(len(loss\_train)),loss\_train,label='train loss')
138. ax.plot(range(len(loss\_test)),loss\_test,label='test loss')
139. ax\_e.plot(range(len(accuracy\_train)),accuracy\_train,'r',label='train accuracy')
140. ax\_e.plot(range(len(accuracy\_test)),accuracy\_test,'g',label='test accuracy')
142. ax.set(xlabel='Epoch', ylabel='Loss with l2 norm')
143. ax\_e.set\_ylabel('Accuracy with threshold=%s'%str(cls.get\_params()['threshold']))
145. ax.legend(loc=4)
146. ax\_e.legend(loc=1)
147. plt.show()
149. plt.close('all')

152. **print** ("===========================")
153. **print** ("Start to figure the accuracy **and** loss curves of\
154. estimators of different optimized algorithms with tuned hyperparameter")
155. **for** i **in** range(len(optimizers)):
156. cls\_name = optimizers[i]
157. cls = o[cls\_name]
159. **print**("Optimizer %s, parameters:"%cls\_name)
160. params = cls.get\_params()
161. params.pop('w')
162. **print**(params)
164. loss\_train = cls.getLossHistory(X\_train,Y\_train)
165. loss\_test = cls.getLossHistory(X\_test,Y\_test)
166. accuracy\_train = cls.getScoreHistory(X\_train,Y\_train)
167. accuracy\_test = cls.getScoreHistory(X\_test,Y\_test)
169. plt.figure(i)
170. \_, ax = plt.subplots()
171. ax\_e = ax.twinx()
172. ax.plot(range(len(loss\_train)),loss\_train,label='train loss')
173. ax.plot(range(len(loss\_test)),loss\_test,label='test loss')
174. ax\_e.plot(range(len(accuracy\_train)),accuracy\_train,'r',label='train accuracy')
175. ax\_e.plot(range(len(accuracy\_test)),accuracy\_test,'g',label='test accuracy')
177. ax.set(xlabel='Epoch', ylabel='Loss with l2 norm')
178. ax\_e.set\_ylabel('Accuracy with threshold=%s'%str(cls.get\_params()['threshold']))
180. ax.legend(loc=4)
181. ax\_e.legend(loc=1)
182. plt.title(cls\_name+' Gradient Decent')
183. plt.show()
184. plt.close('all')
186. **print** ("===========================")
187. **print** ("Start to figure the loss curves of\
188. tuned estimators **in** one figure")
190. GD = o['GD']
191. NAG = o['NAG']
192. Adadelta = o['Adadelta']
193. RMSprop = o['RMSprop']
194. Adam = o['Adam']
196. plotFigure(GD,NAG,Adadelta,RMSprop,Adam,X\_test,Y\_test,'tuned')
198. init\_w = np.random.normal(size=(num\_features,1))
199. GD = LC.Classifier(w=init\_w,optimizer='GD')
200. NAG = LC.Classifier(w=init\_w,optimizer='NAG')
201. Adadelta = LC.Classifier(w=init\_w,optimizer='Adadelta')
202. RMSprop = LC.Classifier(w=init\_w,optimizer='RMSprop')
203. Adam = LC.Classifier(w=init\_w,optimizer='Adam')
205. GD.fit(X\_train,Y\_train)
206. NAG.fit(X\_train,Y\_train)
207. Adadelta.fit(X\_train,Y\_train)
208. RMSprop.fit(X\_train,Y\_train)
209. Adam.fit(X\_train,Y\_train)
211. plotFigure(GD,NAG,Adadelta,RMSprop,Adam,X\_test,Y\_test,'untuned')

**8. The initialization method of model parameters:**

* **Logistic Regression and Stochastic Gradient Descent**

w = np.random.normal(size=(num\_features,1))

* **Linear Classification and Stochastic Gradient Descent**

w = np.random.normal(size=(num\_features,1))

**9. The selected loss function and its derivatives:**

* **Logistic Regression and Stochastic Gradient Descent**

hypothesis

loss function

loss function’s derivatives

* **Linear Classification and Stochastic Gradient Descent**

hypothesis

loss function

loss function’s derivatives

* **Different gradient decent variants**
  + **Nesterov accelerated gradient**

Nesterov accelerated gradient (NAG) is a way to give our momentum term this kind of prescience. We know that we will use our momentum term to move the parameters . Computing thus gives us an approximation of the next position of the parameters (the gradient is missing for the full update), a rough idea where our parameters are going to be. We can now effectively look ahead by calculating the gradient not w.r.t. to our current parameters but w.r.t. the approximate future position of our parameters:

* + **Adadelta**

Adadelta is an extension of Adagrad that seeks to reduce its aggressive, monotonically decreasing learning rate.

Instead of inefficiently storing previous squared gradients, the sum of gradients is recursively defined as a decaying average of all past squared gradients. The running average at time step t then depends (as a fraction similarly to the Momentum term) only on the previous average and the current gradient:

We set to a similar value as the momentum term, around 0.9. For clarity, we now rewrite our vanilla SGD update in terms of the parameter update vector :

The parameter update vector of Adagrad that we derived previously thus takes the form:

We now simply replace the diagonal matrix with the decaying average over past squared gradients :

As the denominator is just the root mean squared (RMS) error criterion of the gradient, we can replace it with the criterion short-hand:

The authors note that the units in this update (as well as in SGD, Momentum, or Adagrad) do not match, i.e. the update should have the same hypothetical units as the parameter. To realize this, they first define another exponentially decaying average, this time not of squared gradients but of squared parameter updates:

The root mean squared error of parameter updates is thus:

Since is unknown, we approximate it with the RMS of parameter updates until the previous time step. Replacing the learning rate in the previous update rule with finally yields the Adadelta update rule:

* + **RMSprop**

RMSprop is an unpublished, adaptive learning rate method proposed by Geoff Hinton in Lecture 6e of his Coursera Class.

RMSprop and Adadelta have both been developed independently around the same time stemming from the need to resolve Adagrad's radically diminishing learning rates. RMSprop in fact is identical to the first update vector of Adadelta that we derived above:

* + **Adam**

Adaptive Moment Estimation (Adam) is another method that computes adaptive learning rates for each parameter. In addition to storing an exponentially decaying average of past squared gradients like Adadelta and RMSprop, Adam also keeps an exponentially decaying average of past gradients , similar to momentum:

and are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively, hence the name of the method. As and are initialized as vectors of 0's, the authors of Adam observe that they are biased towards zero, especially during the initial time steps, and especially when the decay rates are small.

They counteract these biases by computing bias-corrected first and second moment estimates:

They then use these to update the parameters just as we have seen in Adadelta and RMSprop, which yields the Adam update rule:

**10. Experimental results and curve:**(Fill in this content for various methods of gradient descent respectively)

* **Logistic Regression and Stochastic Gradient Descent**

## Hyper-parameter selection:

I fix some hyper-parameters and I use exhaustive grid search to tuning the hyper parameter.

Fixed hyper-parameters are:

max\_iterate=50, means that I use the whole training set 50 times to perform the gradient decent.

batch\_size = 8000

The hyper-parameters’ candidate are as follows

param\_grid = {

'NAG': {'lamda': [0.01, 0.1],

'eta': [0.01, 0.05],

'gamma': [0.8, 0.9, 0.95],

'threshold': [0.5,0.6]},

'Adadelta' : {'lamda': [0.01, 0.1],

'gamma': [0.8, 0.9, 0.95],

'threshold': [0.5,0.6]},

'RMSprop' : {'lamda': [0.01, 0.1],

'eta': [0.01, 0.05],

'gamma': [0.8, 0.9, 0.95],

'threshold': [0.5,0.6]},

'Adam' : {'lamda': [0.01, 0.1],

'eta': [0.01, 0.05],

'Adam\_beta1': [0.9, 0.95],

'Adam\_beta2' : [0.99, 0.999],

'threshold': [0.5,0.6]},

'GD' : {'lamda': [0.01, 0.1, 0.5],

'eta': [0.1, 0.2, 0.3, 0.4, 0.5],

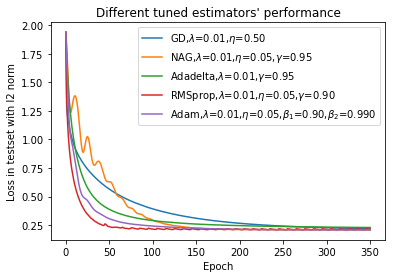
'threshold': [0.4,0.5,0.6]}}

## Predicted Results (Best Results):

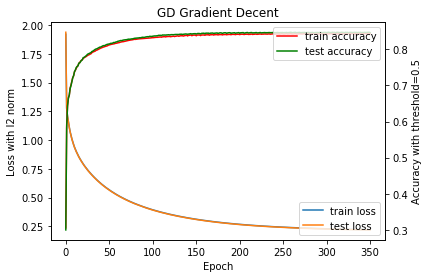
The performance of best estimators with five different optimize algorithm are as follows:

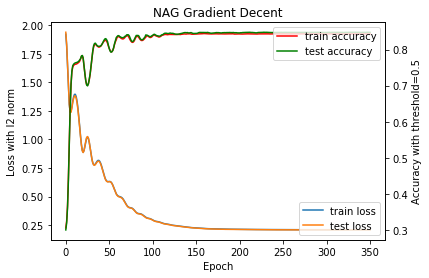
|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** | **Test accuracy** | **Train accuracy** | **Hyper-parameters** |
| GD | 0.841927 | 0.842265 | {'eta': 0.5, 'lamda': 0.01, 'threshold': 0.5} |
| Adadelta | 0.842357 | 0.842189 | {'gamma': 0.95, 'lamda': 0.01, 'threshold': 0.5} |
| Adam | 0.844139 | **0.844477** | {'Adam\_beta1': 0.9, 'Adam\_beta2': 0.99, 'eta': 0.05, 'lamda': 0.01, 'threshold': 0.5} |
| NAG | 0.843954 | 0.844369 | {'eta': 0.05, 'gamma': 0.95, 'lamda': 0.01, 'threshold': 0.5} |
| RMSprop | **0.844354** | 0.844139 | {'eta': 0.05, 'gamma': 0.9, 'lamda': 0.01, 'threshold': 0.6} |

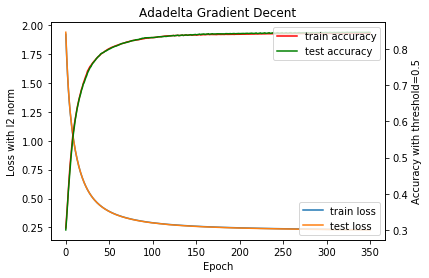
## Loss curve:

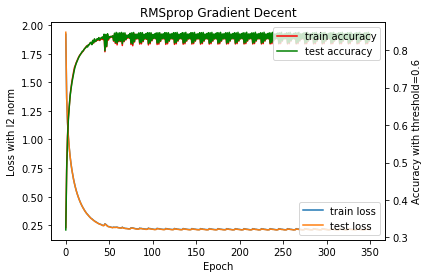


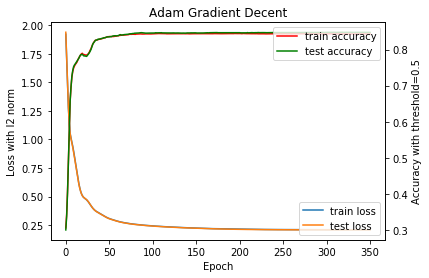
Metrics evolution process details:



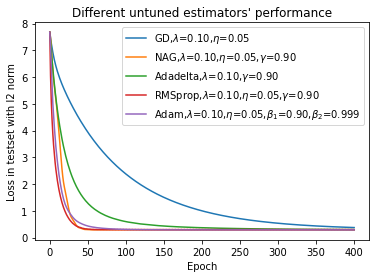








I append a experiment to show the untuned estimators performance



* **Linear Classification and Stochastic Gradient Descent**

## Hyper-parameter selection:

I fix some hyper-parameters and I use exhaustive grid search to tuning the hyper parameter.

Fixed hyper-parameters are:

max\_iterate=50, means that I use the whole training set 50 times to perform the gradient decent.

batch\_size = 8000

The hyper-parameters’ candidate are as follows

param\_grid = {

'NAG': {'lamda': [0.01, 0.1],

'eta': [0.01, 0.05],

'gamma': [0.8, 0.9, 0.95],

'threshold': [0.5,0.6]},

'Adadelta' : {'lamda': [0.01, 0.1],

'gamma': [0.8, 0.9, 0.95],

'threshold': [0.5,0.6]},

'RMSprop' : {'lamda': [0.01, 0.1],

'eta': [0.01, 0.05],

'gamma': [0.8, 0.9, 0.95],

'threshold': [0.5,0.6]},

'Adam' : {'lamda': [0.01, 0.1],

'eta': [0.01, 0.05],

'Adam\_beta1': [0.9, 0.95],

'Adam\_beta2' : [0.99, 0.999],

'threshold': [0.5,0.6]},

'GD' : {'lamda': [0.01, 0.1, 0.5],

'eta': [0.1, 0.2, 0.3, 0.4, 0.5],

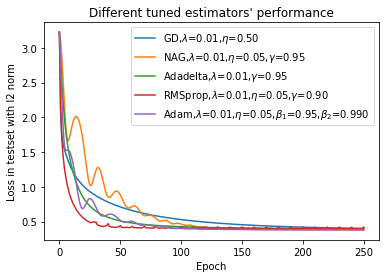
'threshold': [0.4,0.5,0.6]}}

## Predicted Results (Best Results):

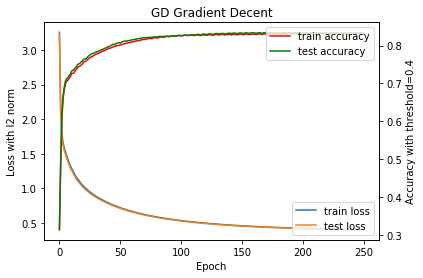
The performance of best estimators with five different optimize algorithm are as follows:

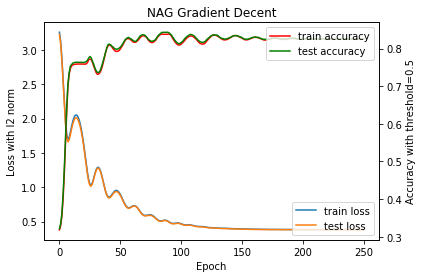
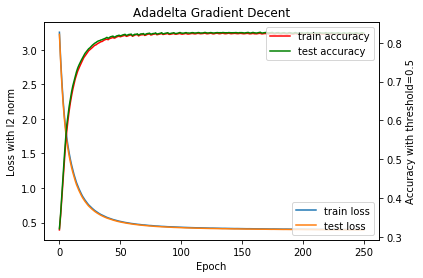
|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** | **Test accuracy** | **Train accuracy** | **Hyper-parameters** |
| GD | **0.8313319615** | **0.8317465648** | {'eta': 0.5, 'lamda': 0.01, 'threshold': 0.4} |
| Adadelta | 0.8255888946 | 0.8259728123 | {'gamma': 0.95, 'lamda': 0.01, 'threshold': 0.5} |
| Adam | 0.8307177298 | 0.8315162541 | {'Adam\_beta1': 0.95, 'Adam\_beta2': 0.99, 'eta': 0.05, 'lamda': 0.01, 'threshold': 0.5} |
| NAG | 0.8268480698 | 0.827800148 | {'eta': 0.05, 'gamma': 0.95, 'lamda': 0.01, 'threshold': 0.5} |
| RMSprop | 0.8290285925 | 0.8297654828 | {'eta': 0.05, 'gamma': 0.9, 'lamda': 0.01, 'threshold': 0.5} |

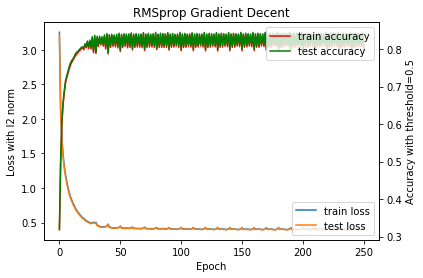
## Loss curve:

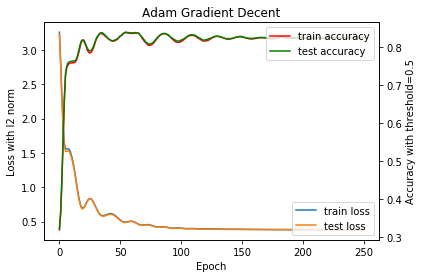


Metrics evolution process details:

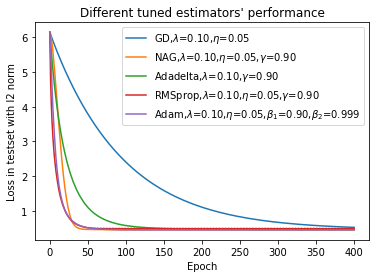






I append a experiment to show the untuned estimators performance



**11. Results analysis:**

Logistic Regression which accuracy is 0.8444 is slightly better than Linear Classification which accuracy is 0.8313.

The performance of different optimization algorithms are ranked as **RMSprop, Adam, Adadelta, GD, NAG** with exhaustive grid search tuning hyper-parameters. And they are ranked as **RMSprop, Adam, NAG, Adadelta, GD**.

With mini-bath gradient decent and no tuning, **RMSprop, Adam, NAG** reach convergence within 50 epochs. Adadelta needs 100+ epochs to converge and GD needs 400+ epochs.

**12. Similarities and differences between logistic regression and linear classification：**

They both solve the classification problems. Thus, for binary classification, they both need threshold to give a final prediction. The raw prediction of classifier is a float between 0 and 1. The greater raw prediction the higher probability of positive.

The loss of linear classification is hinge loss, but the logistic regression is negative log likelihood.

**13. Summary:**

In practice, we prefer logistic regression rather than linear classification if we face a classification problem. And if we can’t tuning the hyper-parameters due to the resource limits, we can propose RMSprop or Adam. Both of the algorithm are not so sensitive to hyper-parameters especially learning rate.