

**The Experiment Report of**

***Deep Learning***

**College: Software Engineering**

**Subject: Deep Learning**

**E-mail: israrshmad@yahoo.com**

**Tutor: Ming Kui tan**

**Date-submitted:2017.12.25**

**1. Topic:** Logistic Regression, Linear Classification and stochastic gradient descent

**2. Time:** 25-12-2017

**3. Reporter: Israr Ahmad**

**4. Purposes:**

1. Further understanding of logistic regression, linear classification and stochastic gradient decent.
2. Compare and understand the relationship and difference between gradient descent and stochastic gradient descent, as well as the logistic regression and linear classification under large scale data-set.
3. Understand the principles of the SVM and practice this process on large scale data.

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

1. a9a Data (Experiment one)
2. a9a.t Data (Experiment two)

**6. Experimental steps:**

**Experiment: 01**

***Logistic Regression and Stochastic Gradient Descent***

1. Load the training set and validation set.
2. Initialize logistic regression model parameters, you can consider initializing zeros, random numbers or normal distribution.
3. Select the loss function and calculate its derivation.
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 different output methods and with the number of iterations.

**Experiment: 02**

***Linear Classification and Stochastic Gradient Descent***

1. Load the training set and validation set.
2. Initialize SVM model parameters, you can consider initializing zeros, random numbers or normal distribution.
3. Select the loss function and calculate its derivation, find more detail in PPT.
4. Calculate gradient toward loss function from **partial samples**.
5. **Update model parameters using different optimized methods(NAG，RMSProp，AdaDelta and Adam).**
6. Select the appropriate threshold, mark the sample whose predict scores greater than the threshold as positive, on the contrary as negative. Predict under validation set and get the different optimized method loss ，， and .
7. Repeat step 4 to 6 for several times, and drawing graph of different methods and with the number of iterations.

**8. Selection of validation (hold-out, cross-validation, k-folds cross-validation, etc.):**

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

The SVM model is being updated by using different optimizing methods like NAG, RMSProp, AdaDelta and Adam.

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

**11. Experimental results and curve**

**import** numpy  
**from** collections **import** defaultdict  
  
  
**class** Model(object):  
 **def** \_\_init\_\_(self, n\_features):  
 self.params = numpy.random.random(size=(n\_features, 1))  
 self.diffs = numpy.zeros((n\_features, 1))  
 self.recorder = defaultdict(list)  
  
 **def** train(self, X, y):  
 **pass  
  
 def** validate(self, X, y):  
 self.\_\_loss\_\_(X, y, **"validation"**)  
  
 **def** predict(self, X):  
 **pass  
  
 def** \_\_calculate\_gradient\_\_(self, params=None):  
 **pass  
  
 def** \_\_loss\_\_(self, X, y, key):  
 **pass  
  
  
class** SVMClassifier(Model):  
 **def** \_\_init\_\_(self, n\_features, C):  
 super(SVMClassifier, self).\_\_init\_\_(n\_features=n\_features)  
 self.C = C  
 self.X\_train = None  
 self.y\_train = None  
  
 **def** train(self, X, y):  
 self.X\_train = X  
 self.y\_train = y  
  
 **def** predict(self, X):  
 **return** numpy.where(numpy.dot(X, self.params) > 0, 1, -1)  
  
 **def** \_\_calculate\_gradient\_\_(self, params=None):  
 **if** params **is** None:  
 params = self.params  
 h = 1 - self.y\_train \* numpy.dot(self.X\_train, params)  
 y\_mask = numpy.where(h > 0, self.y\_train, 0)  
 self.diffs = params - self.C \* numpy.dot(self.X\_train.transpose(), y\_mask)  
  
 **def** \_\_loss\_\_(self, X, y, key):  
 loss = numpy.sum(self.params \* self.params) \  
 + self.C \* numpy.sum(numpy.maximum(1 - y \* numpy.dot(X, self.params), 0))  
 self.recorder[key].append(loss)  
  
  
**class** LogisticRegressionClassifier(Model):  
 **def** \_\_init\_\_(self, n\_features):  
 super(LogisticRegressionClassifier, self).\_\_init\_\_(n\_features=n\_features)  
 self.X\_train = None  
 self.y\_train = None  
  
 **def** train(self, X, y):  
 self.X\_train = X  
 self.y\_train = y  
  
 **def** predict(self, X):  
 **return** numpy.where(numpy.dot(X, self.params) > 0, 1, 0)  
  
 **def** \_\_calculate\_gradient\_\_(self, params=None):  
 **if** params **is** None:  
 params = self.params  
 y\_hat = 1 / (1 + numpy.exp(-numpy.dot(self.X\_train, params)))  
 self.diffs = numpy.dot(self.X\_train.transpose(), (y\_hat - self.y\_train))  
  
 **def** \_\_loss\_\_(self, X, y, key):  
 y\_hat = 1 / (1 + numpy.exp(-numpy.dot(X, self.params)))  
 loss = -numpy.average(y \* numpy.log(y\_hat) + (1 - y) \* numpy.log(1 - y\_hat))  
 self.recorder[key].append(loss)  
  
  
**class** Optimizer(object):  
 **def** \_\_init\_\_(self, model):  
 self.model = model  
 self.color = None  
  
 **def** step(self):  
 **pass  
  
  
class** SGD(Optimizer):  
 **def** \_\_init\_\_(self, model, learning\_rate, momentum=None):  
 super(SGD, self).\_\_init\_\_(model=model)  
 self.color = **"r"** self.learning\_rate = learning\_rate  
 self.momentum = momentum  
 **if** momentum **is not** None:  
 self.v = numpy.zeros\_like(self.model.diffs)  
  
 **def** step(self):  
 self.model.\_\_calculate\_gradient\_\_()  
 **if** self.momentum **is** None:  
 self.model.params -= self.learning\_rate \* self.model.diffs  
 **else**:  
 self.v = self.momentum \* self.v + self.learning\_rate \* self.model.diffs  
 self.model.params -= self.v  
  
  
**class** NAG(Optimizer):  
 **def** \_\_init\_\_(self, model, learning\_rate, momentum):  
 super(NAG, self).\_\_init\_\_(model=model)  
 self.color = **"y"** self.learning\_rate = learning\_rate  
 self.momentum = momentum  
 self.v = numpy.zeros\_like(self.model.diffs)  
  
 **def** step(self):  
 self.model.\_\_calculate\_gradient\_\_(params=self.model.params - self.momentum \* self.v)  
 self.v = self.momentum \* self.v + self.learning\_rate \* self.model.diffs  
 self.model.params -= self.v  
  
  
**class** AdaGrad(Optimizer):  
 **def** \_\_init\_\_(self, model, learning\_rate):  
 super(AdaGrad, self).\_\_init\_\_(model=model)  
 self.color = **"g"** self.G = numpy.zeros\_like(self.model.diffs)  
 self.learning\_rate = learning\_rate  
 self.epsilon = 1e-8  
  
 **def** step(self):  
 self.model.\_\_calculate\_gradient\_\_()  
 self.G += self.model.diffs \* self.model.diffs  
 self.model.params -= self.learning\_rate / numpy.sqrt(self.G + self.epsilon) \* self.model.diffs  
  
  
**class** RMSProP(Optimizer):  
 **def** \_\_init\_\_(self, model, leaning\_rate, weight\_decay):  
 self.color = **"c"** super(RMSProP, self).\_\_init\_\_(model=model)  
 self.G = numpy.zeros\_like(self.model.diffs)  
 self.learning\_rate = leaning\_rate  
 self.weight\_decay = weight\_decay  
 self.epsilon = 1e-8  
  
 **def** step(self):  
 self.model.\_\_calculate\_gradient\_\_()  
 self.G = self.weight\_decay \* self.G + (1 - self.weight\_decay) \* self.model.diffs \* self.model.diffs  
 self.model.params -= self.learning\_rate / numpy.sqrt(self.G + self.epsilon) \* self.model.diffs  
  
  
**class** AdaDelta(Optimizer):  
 **def** \_\_init\_\_(self, model, gamma):  
 super(AdaDelta, self).\_\_init\_\_(model=model)  
 self.color = **"b"** self.gamma = gamma  
 self.G = numpy.zeros\_like(self.model.diffs)  
 self.delta = numpy.zeros\_like(self.model.diffs)  
 self.delta\_theta = numpy.zeros\_like(self.model.diffs)  
 self.epsilon = 1e-4  
  
 **def** step(self):  
 self.model.\_\_calculate\_gradient\_\_()  
 self.G = self.gamma \* self.G + (1 - self.gamma) \* self.model.diffs \* self.model.diffs  
 self.delta\_theta = -(numpy.sqrt(self.delta + self.epsilon)  
 / numpy.sqrt(self.G + self.epsilon)) \* self.model.diffs  
 self.model.params += self.delta\_theta  
 self.delta = self.gamma \* self.delta + (1 - self.gamma) \* self.delta\_theta \* self.delta\_theta  
  
  
**class** Adam(Optimizer):  
 **def** \_\_init\_\_(self, model, beta, gamma, eta):  
 super(Adam, self).\_\_init\_\_(model=model)  
 self.color = **"m"** self.beta = beta  
 self.gamma = gamma  
 self.eta = eta  
 self.m = numpy.zeros\_like(self.model.diffs)  
 self.G = numpy.zeros\_like(self.model.diffs)  
 self.epsilon = 1e-8  
  
 **def** step(self):  
 self.model.\_\_calculate\_gradient\_\_()  
 self.m = self.beta \* self.m + (1 - self.beta) \* self.model.diffs  
 self.G = self.gamma \* self.G + (1 - self.gamma) \* self.model.diffs \* self.model.diffs  
 alpha = self.eta \* (numpy.sqrt(1 - self.gamma)) / (1 - self.beta)  
 self.model.params -= alpha \* self.m / numpy.sqrt(self.G + self.epsilon)  
  
**import** requests  
  
train\_set = requests.get(**"https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary/a9a"**)  
validation\_set = requests.get(**"https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary/a9a.t"**)  
  
**from** io **import** BytesIO  
**from** sklearn.datasets **import** load\_svmlight\_file  
  
X\_train, y\_train = load\_svmlight\_file(BytesIO(train\_set.content), n\_features=123)  
X\_val, y\_val = load\_svmlight\_file(BytesIO(validation\_set.content), n\_features=123)  
X\_train = X\_train.toarray()  
X\_val = X\_val.toarray()  
  
n\_samples\_train, n\_features\_train = X\_train.shape  
X\_train = numpy.concatenate((X\_train, numpy.ones(shape=(n\_samples\_train, 1))), axis=1)  
y\_train = y\_train.reshape((n\_samples\_train, 1))  
  
n\_samples\_val, n\_features\_val = X\_val.shape  
X\_val = numpy.concatenate((X\_val, numpy.ones(shape=(n\_samples\_val, 1))), axis=1)  
y\_val = y\_val.reshape((n\_samples\_val, 1))  
  
  
optimizers = [  
 SGD(model=SVMClassifier(n\_features=123 + 1, C=1), learning\_rate=0.00001, momentum=0.5),  
 NAG(model=SVMClassifier(n\_features=123 + 1, C=1), learning\_rate=0.0001, momentum=0.9),  
 AdaGrad(model=SVMClassifier(n\_features=123 + 1,C=1),learning\_rate=0.1),  
 RMSProP(model=SVMClassifier(n\_features=123 + 1,C=1),leaning\_rate=0.1,weight\_decay=0.9),  
 AdaDelta(model=SVMClassifier(n\_features=123 + 1,C=1), gamma=0.95),  
 Adam(model=SVMClassifier(n\_features=123 + 1,C=1),beta=0.9,gamma=0.999,eta=0.1)  
]  
  
max\_epoch = 100  
batch\_size = 11000  
**for** epoch **in** range(max\_epoch):  
 indexes = numpy.random.randint(0, n\_samples\_train, size=batch\_size)  
 **for** optimizer **in** optimizers:  
 optimizer.model.train(X\_train[indexes], y\_train[indexes])  
 optimizer.step()  
 optimizer.model.validate(X\_val, y\_val)  
  
**from** sklearn.metrics **import** classification\_report  
  
**print**(**"-"** \* 20 + optimizers[0].model.\_\_class\_\_.\_\_name\_\_ + **"-"** \* 20)  
**for** optimizer **in** optimizers:  
 **print**(**"-"** \* 24 + optimizer.\_\_class\_\_.\_\_name\_\_ + **"-"** \* 24)  
 **print**(classification\_report(y\_val,  
 optimizer.model.predict(X\_val),  
 target\_names=[**"positive"**, **"negative"**],  
 digits=3))  
   
 **import** matplotlib.pyplot **as** plt  
%matplotlib inline  
plt.figure(figsize=(16,11))  
plt.xlabel(**"epoch"**)  
plt.ylabel(**"loss"**)  
plt.title(optimizers[0].model.\_\_class\_\_.\_\_name\_\_)  
**for** optimizer **in** optimizers:  
 plt.plot(optimizer.model.recorder[**"validation"**], color=optimizer.color, label=optimizer.\_\_class\_\_.\_\_name\_\_)  
plt.legend()  
plt.show()

**import** numpy  
**from** collections **import** defaultdict  
  
  
**class** Model(object):  
 **def** \_\_init\_\_(self, n\_features):  
 self.params = numpy.random.random(size=(n\_features, 1))  
 self.diffs = numpy.zeros((n\_features, 1))  
 self.recorder = defaultdict(list)  
  
 **def** train(self, X, y):  
 **pass  
  
 def** validate(self, X, y):  
 self.\_\_loss\_\_(X, y, **"validation"**)  
  
 **def** predict(self, X):  
 **pass  
  
 def** \_\_calculate\_gradient\_\_(self, params=None):  
 **pass  
  
 def** \_\_loss\_\_(self, X, y, key):  
 **pass  
  
  
class** SVMClassifier(Model):  
 **def** \_\_init\_\_(self, n\_features, C):  
 super(SVMClassifier, self).\_\_init\_\_(n\_features=n\_features)  
 self.C = C  
 self.X\_train = None  
 self.y\_train = None  
  
 **def** train(self, X, y):  
 self.X\_train = X  
 self.y\_train = y  
  
 **def** predict(self, X):  
 **return** numpy.where(numpy.dot(X, self.params) > 0, 1, -1)  
  
 **def** \_\_calculate\_gradient\_\_(self, params=None):  
 **if** params **is** None:  
 params = self.params  
 h = 1 - self.y\_train \* numpy.dot(self.X\_train, params)  
 y\_mask = numpy.where(h > 0, self.y\_train, 0)  
 self.diffs = params - self.C \* numpy.dot(self.X\_train.transpose(), y\_mask)  
  
 **def** \_\_loss\_\_(self, X, y, key):  
 loss = numpy.sum(self.params \* self.params) \  
 + self.C \* numpy.sum(numpy.maximum(1 - y \* numpy.dot(X, self.params), 0))  
 self.recorder[key].append(loss)  
  
  
**class** LogisticRegressionClassifier(Model):  
 **def** \_\_init\_\_(self, n\_features):  
 super(LogisticRegressionClassifier, self).\_\_init\_\_(n\_features=n\_features)  
 self.X\_train = None  
 self.y\_train = None  
  
 **def** train(self, X, y):  
 self.X\_train = X  
 self.y\_train = y  
  
 **def** predict(self, X):  
 **return** numpy.where(numpy.dot(X, self.params) > 0, 1, 0)  
  
 **def** \_\_calculate\_gradient\_\_(self, params=None):  
 **if** params **is** None:  
 params = self.params  
 y\_hat = 1 / (1 + numpy.exp(-numpy.dot(self.X\_train, params)))  
 self.diffs = numpy.dot(self.X\_train.transpose(), (y\_hat - self.y\_train))  
  
 **def** \_\_loss\_\_(self, X, y, key):  
 y\_hat = 1 / (1 + numpy.exp(-numpy.dot(X, self.params)))  
 loss = -numpy.average(y \* numpy.log(y\_hat) + (1 - y) \* numpy.log(1 - y\_hat))  
 self.recorder[key].append(loss)  
  
  
**class** Optimizer(object):  
 **def** \_\_init\_\_(self, model):  
 self.model = model  
 self.color = None  
  
 **def** step(self):  
 **pass  
  
  
class** SGD(Optimizer):  
 **def** \_\_init\_\_(self, model, learning\_rate, momentum=None):  
 super(SGD, self).\_\_init\_\_(model=model)  
 self.color = **"r"** self.learning\_rate = learning\_rate  
 self.momentum = momentum  
 **if** momentum **is not** None:  
 self.v = numpy.zeros\_like(self.model.diffs)  
  
 **def** step(self):  
 self.model.\_\_calculate\_gradient\_\_()  
 **if** self.momentum **is** None:  
 self.model.params -= self.learning\_rate \* self.model.diffs  
 **else**:  
 self.v = self.momentum \* self.v + self.learning\_rate \* self.model.diffs  
 self.model.params -= self.v  
  
  
**class** NAG(Optimizer):  
 **def** \_\_init\_\_(self, model, learning\_rate, momentum):  
 super(NAG, self).\_\_init\_\_(model=model)  
 self.color = **"y"** self.learning\_rate = learning\_rate  
 self.momentum = momentum  
 self.v = numpy.zeros\_like(self.model.diffs)  
  
 **def** step(self):  
 self.model.\_\_calculate\_gradient\_\_(params=self.model.params - self.momentum \* self.v)  
 self.v = self.momentum \* self.v + self.learning\_rate \* self.model.diffs  
 self.model.params -= self.v  
  
  
**class** AdaGrad(Optimizer):  
 **def** \_\_init\_\_(self, model, learning\_rate):  
 super(AdaGrad, self).\_\_init\_\_(model=model)  
 self.color = **"g"** self.G = numpy.zeros\_like(self.model.diffs)  
 self.learning\_rate = learning\_rate  
 self.epsilon = 1e-8  
  
 **def** step(self):  
 self.model.\_\_calculate\_gradient\_\_()  
 self.G += self.model.diffs \* self.model.diffs  
 self.model.params -= self.learning\_rate / numpy.sqrt(self.G + self.epsilon) \* self.model.diffs  
  
  
**class** RMSProP(Optimizer):  
 **def** \_\_init\_\_(self, model, leaning\_rate, weight\_decay):  
 self.color = **"c"** super(RMSProP, self).\_\_init\_\_(model=model)  
 self.G = numpy.zeros\_like(self.model.diffs)  
 self.learning\_rate = leaning\_rate  
 self.weight\_decay = weight\_decay  
 self.epsilon = 1e-8  
  
 **def** step(self):  
 self.model.\_\_calculate\_gradient\_\_()  
 self.G = self.weight\_decay \* self.G + (1 - self.weight\_decay) \* self.model.diffs \* self.model.diffs  
 self.model.params -= self.learning\_rate / numpy.sqrt(self.G + self.epsilon) \* self.model.diffs  
  
  
**class** AdaDelta(Optimizer):  
 **def** \_\_init\_\_(self, model, gamma):  
 super(AdaDelta, self).\_\_init\_\_(model=model)  
 self.color = **"b"** self.gamma = gamma  
 self.G = numpy.zeros\_like(self.model.diffs)  
 self.delta = numpy.zeros\_like(self.model.diffs)  
 self.delta\_theta = numpy.zeros\_like(self.model.diffs)  
 self.epsilon = 1e-4  
  
 **def** step(self):  
 self.model.\_\_calculate\_gradient\_\_()  
 self.G = self.gamma \* self.G + (1 - self.gamma) \* self.model.diffs \* self.model.diffs  
 self.delta\_theta = -(numpy.sqrt(self.delta + self.epsilon)  
 / numpy.sqrt(self.G + self.epsilon)) \* self.model.diffs  
 self.model.params += self.delta\_theta  
 self.delta = self.gamma \* self.delta + (1 - self.gamma) \* self.delta\_theta \* self.delta\_theta  
  
  
**class** Adam(Optimizer):  
 **def** \_\_init\_\_(self, model, beta, gamma, eta):  
 super(Adam, self).\_\_init\_\_(model=model)  
 self.color = **"m"** self.beta = beta  
 self.gamma = gamma  
 self.eta = eta  
 self.m = numpy.zeros\_like(self.model.diffs)  
 self.G = numpy.zeros\_like(self.model.diffs)  
 self.epsilon = 1e-8  
  
 **def** step(self):  
 self.model.\_\_calculate\_gradient\_\_()  
 self.m = self.beta \* self.m + (1 - self.beta) \* self.model.diffs  
 self.G = self.gamma \* self.G + (1 - self.gamma) \* self.model.diffs \* self.model.diffs  
 alpha = self.eta \* (numpy.sqrt(1 - self.gamma)) / (1 - self.beta)  
 self.model.params -= alpha \* self.m / numpy.sqrt(self.G + self.epsilon)  
  
**import** requests  
  
train\_set = requests.get(**"https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary/a9a"**)  
validation\_set = requests.get(**"https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary/a9a.t"**)  
  
**from** io **import** BytesIO  
**from** sklearn.datasets **import** load\_svmlight\_file  
  
X\_train, y\_train = load\_svmlight\_file(BytesIO(train\_set.content), n\_features=123)  
X\_val, y\_val = load\_svmlight\_file(BytesIO(validation\_set.content), n\_features=123)  
X\_train = X\_train.toarray()  
X\_val = X\_val.toarray()  
  
n\_samples\_train, n\_features\_train = X\_train.shape  
X\_train = numpy.concatenate((X\_train, numpy.ones(shape=(n\_samples\_train, 1))), axis=1)  
y\_train = y\_train.reshape((n\_samples\_train, 1))  
  
n\_samples\_val, n\_features\_val = X\_val.shape  
X\_val = numpy.concatenate((X\_val, numpy.ones(shape=(n\_samples\_val, 1))), axis=1)  
y\_val = y\_val.reshape((n\_samples\_val, 1))  
  
  
  
y\_train = numpy.where(y\_train == -1, 0, y\_train)  
y\_val = numpy.where(y\_val == -1, 0, y\_val)  
optimizers = [  
 SGD(model=LogisticRegressionClassifier(n\_features=123 + 1), learning\_rate=0.00001, momentum=0.5),  
 NAG(model=LogisticRegressionClassifier(n\_features=123 + 1), learning\_rate=0.00001, momentum=0.5),  
 AdaGrad(model=LogisticRegressionClassifier(n\_features=123 + 1), learning\_rate=0.1),  
 RMSProP(model=LogisticRegressionClassifier(n\_features=123 + 1), leaning\_rate=0.1, weight\_decay=0.9),  
 AdaDelta(model=LogisticRegressionClassifier(n\_features=123 + 1), gamma=0.95),  
 Adam(model=LogisticRegressionClassifier(n\_features=123 + 1), beta=0.9, gamma=0.999, eta=0.1)  
]  
  
max\_epoch = 100  
batch\_size = 11000  
**for** epoch **in** range(max\_epoch):  
 indexes = numpy.random.randint(0, n\_samples\_train, size=batch\_size)  
 **for** optimizer **in** optimizers:  
 optimizer.model.train(X\_train[indexes], y\_train[indexes])  
 optimizer.step()  
 optimizer.model.validate(X\_val, y\_val)  
  
**from** sklearn.metrics **import** classification\_report  
  
**print**(**"-"** \* 20 + optimizers[0].model.\_\_class\_\_.\_\_name\_\_ + **"-"** \* 20)  
**for** optimizer **in** optimizers:  
 **print**(**"-"** \* 24 + optimizer.\_\_class\_\_.\_\_name\_\_ + **"-"** \* 24)  
 **print**(classification\_report(y\_val,  
 optimizer.model.predict(X\_val),  
 target\_names=[**"positive"**, **"negative"**],  
 digits=3))  
   
 **import** matplotlib.pyplot **as** plt  
%matplotlib inline  
plt.figure(figsize=(14,11))  
plt.xlabel(**"epoch"**)  
plt.ylabel(**"loss"**)  
plt.title(optimizers[0].model.\_\_class\_\_.\_\_name\_\_)  
**for** optimizer **in** optimizers:  
 plt.plot(optimizer.model.recorder[**"validation"**], color=optimizer.color, label=optimizer.\_\_class\_\_.\_\_name\_\_)  
plt.legend()  
plt.show()

1. **Results analysis:**

For second experiment the estimated updated methods have different curves at different , which indicates its SVM gradient descent and at different values:

max\_epoch = 100

batch\_size = 10000

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

The logistic regression is called as a linear classifier because it produces a decision boundary which is linear in nature. So, the classification makes by logistic regression is linear classification only.

**Regression:** given a set of data, find the best relationship that represents the set of data.

**Classification:** given a known relationship, identify the class that the data belongs to.

We can see that regression and classification start from opposing ends: to find a pattern or to find the pattern that it belongs to.

**14. Summary:**

Logistic regression measures the relationship between the categorical dependent variable and one or more independent variables by estimating probabilities using a logistic function, which is the cumulative logistic distribution. Thus, it treats the same set of problems as prohibit regression using similar techniques, with the latter using a cumulative normal distribution curve instead.

**Classification problems** try to determine group membership by deriving probabilities. The first technique ever used was linear discriminant analysis (LDA), proposed by Sir R.A. Fisher in 1936—he used to classify irises. I do not understand it fully, but believe that it used linear regression to derive probabilities for each group, and then used a Mahalanobis distance measure to assign to the closest group.

**Classification Problems,** **Classification** is a central topic in machine learning that has to do with teaching machines how to group together data by criteria. Classification is the process where computers group data together based on predetermined characteristics — this is called supervised learning.

A linear classifier achieves this by making a classification decision based on the value of a linear combination of the characteristics. An object's characteristics are also known as feature values and are typically presented to the machine in a vector called a feature vector.