

**The Experiment Report of**

***Deep Learning***

**College: Software Engineering**

**Subject: Deep Learning**

**Name : Jamsheed Nazir**

**Student ID:**201722800098

**E-mail: hmj.nazir@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:** H.M Jamsheed Nazir

**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**

* windows
* python 3.6.0
* *import* numpy *as* np  
  *import* matplotlib.pyplot *as* plt  
  *from* sklearn.datasets *import* load\_svmlight\_file  
  *from* sklearn.model\_selection *import* train\_test\_split  
  *from* sklearn.metrics *import* accuracy\_score  
    
  iteration\_times = 500 # iteration times  
  train\_valid\_ratio = 0.8 # determine the ratio of train to validation data  
  prob\_threshold = 0.5
* a9a\_train = load\_svmlight\_file('E:/Master/machinelearning/机器学习/data/a9a\_train.svm')  
  a9a\_test = load\_svmlight\_file('E:/Master/machinelearning/机器学习/data/a9a\_test.t', n\_features=a9a\_train[0].shape[1])  
  *print*('train data shape:', a9a\_train[0].shape, 'test data shape:', a9a\_test[0].shape)

*def* prepare\_data(*train*, *test*, *shuffle* = *False*):  
 x\_train, y\_train, x\_valid, y\_valid = *train*[0], *train*[1], *test*[0], *test*[1]  
 # print('train data shape:', x\_train.shape, '---validation data shape:', x\_valid.shape)  
 y\_train[y\_train == -1] = 0  
 y\_valid[y\_valid == -1] = 0  
 *return* x\_train, x\_valid, y\_train, y\_valid

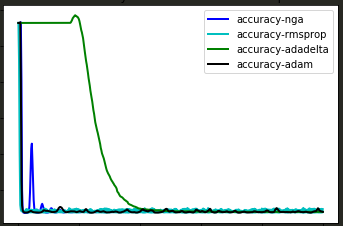
*def* init\_param(*nin*, *method*='randn', *scale*=0.01):  
 """  
 initilize params, defalt scale 0.01  
 """  
 *if nin* < 0:  
 *return  
 if method* == 'randn':  
 *if nin* > 1:  
 *return scale* \* np.random.random(size=*nin*)  
 *else*:  
 *return scale* \* np.random.random()  
 *if method* == 'zero':  
 *if nin* > 1:  
 *return* np.zeros(*nin*)  
 *else*:  
 *return* 0

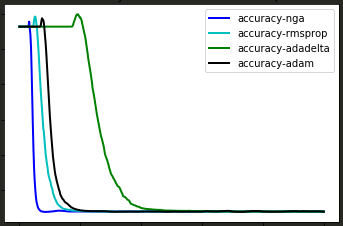
*def* linear\_model(*x*, *w*):  
 *try*:  
 y\_prob = *x*\**w* # judge the class of linear model  
 y = np.array([0]\*y\_prob.shape[0])  
 y[y\_prob < prob\_threshold] = 0  
 y[y\_prob > prob\_threshold] = 1  
 *return* y\_prob, y  
 *except* Exception *as* e:  
 *print*("x don't match w...")  
  
*def* loss\_function(*y*, *y\_pred*, *w*):  
 hinge = np.max([[0]\**y*.shape[0],1 - *y*\**y\_pred*], axis=0)  
 *return* np.mean(hinge) + 0.5 \* np.linalg.norm(*w*) # use hinge loss function  
  
*def* compute\_gradient(*x*, *y*, *y\_pred*, *w*):  
 # compute gradient, very important, we should compute the average, and add the regulation term  
 *return* -1 \* *y*[*y* \* *y\_pred* < 1]\**x*[*y* \* *y\_pred* < 1]/(*y*[*y* \* *y\_pred* < 1]).shape[0] + *w  
  
  
def* get\_batch(*x*, *batch\_size*=20):  
 *if batch\_size* < 0:  
 *return* # default batch\_size 20  
 *return* np.random.choice(range(*x*.shape[0]) , size=*batch\_size*)  
  
  
*def* update\_params\_nga(*x*, *y*, *y\_pred*, *w*, *v*, *mu*, *alpha*):  
 """  
 update weight with momentum. w, v are the iteratively params  
 x : input, y : true label, y\_pred: predict label, w: update params,  
 v: velocity, mu: momentum factor, alpha: learning rate  
 """  
 d\_w = compute\_gradient(*x*, *y*, *y\_pred*, *w*) # evaluate dx\_head  
 v\_prev = *v* v = *mu* \* *v* - *alpha* \* d\_w # alpha is learning rate  
 w = *w* + *mu* \* v\_prev + (1 + *mu*) \* *v  
 return w*, *v  
  
  
def* update\_params\_rmsprop(*x*, *y*, *y\_pred*, *w*, *cache*, *decay\_rate*, *eps*, *alpha*):  
 """  
 adaptive learning rate alpha, consider gradient of w. w, cache are the iteratively params  
 x : input, y : true label, y\_pred: predict label, w: update params, cache: cumulative parameter  
 decay\_rate, eps: hyperparameter, alpha: learning rate  
 """  
 d\_w = compute\_gradient(*x*, *y*, *y\_pred*, *w*) # alpha is learning rate, compute the gradient of w  
 cache = *decay\_rate* \* *cache* + (1 - *decay\_rate*) \* (d\_w \*\* 2) # compute cache, larger cache, smaller learning rate  
 w = *w* - *alpha* \* d\_w / (np.sqrt(*cache*) + *eps*) # update weight  
 *return w*, *cache  
  
  
def* update\_params\_adadelta(*x*, *y*, *y\_pred*, *w*, *decay\_rate*, *eps*, *cum\_grad*, *cum\_u\_w*, *u\_w\_list*):  
 """  
 adaptive learning rate alpha, consider gradient and update of w. w, cum\_grad and cum\_u\_w are the iteratively params  
 x : input, y : true label, y\_pred: predict label, w: update params, decay\_rate, eps: hyperparameter  
 cum\_u\_w: cumulative update value of w, u\_w\_list: each time update of parameter w, cum\_grad: cumulative gradient  
 """  
 # comput root mean square of cumulative gradient  
 d\_w = compute\_gradient(*x*, *y*, *y\_pred*, *w*) # alpha is learning rate, compute the gradient of w  
 cum\_grad = *decay\_rate* \* *cum\_grad* + (1 - *decay\_rate*) \* (d\_w \*\* 2)  
 rms\_grad = np.sqrt(*cum\_grad* + *eps*)  
  
 # comput root mean square of cumulative update value of w  
 cum\_u\_w = *decay\_rate* \* *cum\_u\_w* + (1 - *decay\_rate*) \* (*u\_w\_list*[-1] \*\* 2)  
 rms\_u\_w = np.sqrt(*cum\_u\_w* + *eps*)  
  
 # update weight  
 u\_w = rms\_u\_w \* d\_w / rms\_grad  
 w = *w* - u\_w # alpha is learning rate  
  
 # store params  
 *u\_w\_list*.append(u\_w)  
  
 *return w*, *cum\_grad*, *cum\_u\_w  
  
  
def* update\_params\_adam(*x*, *y*, *y\_pred*, *w*, *m*, *v*, *betal\_one*, *belta\_two*, *eps*, *alpha*, *iter\_step*):  
 """  
 adaptive learning rate alpha and momentum. w, m and v are the iteratively params  
 x : input, y : true label, y\_pred: predict label, w: update params, m: momentum factor, v:velocity, betal\_one and betal\_two:decay\_rate  
 eps: hyperparameter, alpha:learning rate, iter\_step: current step  
 """  
  
 d\_w = compute\_gradient(*x*, *y*, *y\_pred*, *w*) # alpha is learning rate, compute the gradient of w  
 m = *betal\_one* \* *m* + (1 - *betal\_one*) \* d\_w  
 mt = *m* / (1 - np.power(*betal\_one*, *iter\_step* + 1))  
 v = *belta\_two* \* *v* + (1 - *belta\_two*) \* (d\_w \*\* 2)  
 vt = *v* / (1 - np.power(*belta\_two*, *iter\_step* + 1))  
 w = *w* - *alpha* \* mt / (np.sqrt(vt) + *eps*)  
 *return w*, *m*, *v*

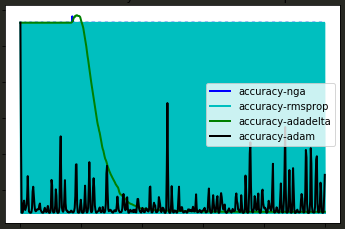
*def* nga\_model(*x\_train*, *x\_valid*, *y\_train*, *y\_valid*, *iteration\_times*, *mu*, *alpha*):  
 # mu: momentum, alpha: learning rate  
 train\_loss = []  
 valid\_loss = []  
 valid\_accu = []  
 # initial params  
 w = init\_param(*x\_train*.shape[1]) # initialize weight and bias  
 v = init\_param(*x\_train*.shape[1], method='zero') # initial with zeros  
 *for* i *in* range(*iteration\_times*):  
 # get batch train and test data  
 index = get\_batch(*x\_train*, batch\_size=int(*x\_train*.shape[0]))  
 x\_train\_batch = *x\_train*[index, :]  
 y\_train\_batch = *y\_train*[index]  
  
 # logistic model, and train loss  
 y\_pred, y\_pred\_label = linear\_model(x\_train\_batch, w)  
 loss\_t = loss\_function(y\_train\_batch, y\_pred, w) # square loss function of train data  
 train\_loss.append(loss\_t)  
  
 # logistic model, and validation loss  
 y\_valid\_pred, y\_valid\_pred\_label = linear\_model(*x\_valid*, w)  
 loss\_v = loss\_function(*y\_valid*, y\_valid\_pred, w) # square loss function of validation data  
 valid\_loss.append(loss\_v)  
 valid\_accu.append(accuracy\_score(*y\_valid*, y\_valid\_pred\_label))  
  
 # if i % (iteration\_times/2) == 0: # print info  
 # print('alpha:', a, 'iteration:', i)  
  
 w, v = update\_params\_nga(x\_train\_batch, y\_train\_batch, y\_pred, w, v, *mu*, *alpha*)  
 # print('learning rate:', alpha, "---min train loss:", min(train\_loss), '--min valid loss:', min(valid\_loss))  
 *return* train\_loss, valid\_loss, valid\_accu  
  
  
*def* rmsprop\_model(*x\_train*, *x\_valid*, *y\_train*, *y\_valid*, *iteration\_times*, *decay\_rate*, *eps*, *alpha*):  
 # decay\_rate: decay rate, eps: hyperparameter,, alpha: learning rate  
 train\_loss = []  
 valid\_loss = []  
 valid\_accu = []  
 # initial params  
 w = init\_param(*x\_train*.shape[1]) # initialize weight and bias  
 cache = init\_param(1, method='zero') # initial with zeros  
 *for* i *in* range(*iteration\_times*):  
 # get batch train and test data  
 index = get\_batch(*x\_train*, batch\_size=int(*x\_train*.shape[0]))  
 x\_train\_batch = *x\_train*[index, :]  
 y\_train\_batch = *y\_train*[index]  
  
 # logistic model, and train loss  
 y\_pred, y\_pred\_label = linear\_model(x\_train\_batch, w)  
 loss\_t = loss\_function(y\_train\_batch, y\_pred, w) # square loss function of train data  
 train\_loss.append(loss\_t)  
  
 # logistic model, and validation loss  
 y\_valid\_pred, y\_valid\_pred\_label = linear\_model(*x\_valid*, w)  
 loss\_v = loss\_function(*y\_valid*, y\_valid\_pred, w) # square loss function of validation data  
 valid\_loss.append(loss\_v)  
 valid\_accu.append(accuracy\_score(*y\_valid*, y\_valid\_pred\_label))  
  
 # if i % (iteration\_times/2) == 0: # print info  
 # print('alpha:', a, 'iteration:', i)  
  
 w, cache = update\_params\_rmsprop(x\_train\_batch, y\_train\_batch, y\_pred, w, cache, *decay\_rate*, *eps*, *alpha*)  
 # print('learning rate:', alpha, "---min train loss:", min(train\_loss), '--min valid loss:', min(valid\_loss))  
 *return* train\_loss, valid\_loss, valid\_accu  
  
  
*def* adadelta\_model(*x\_train*, *x\_valid*, *y\_train*, *y\_valid*, *iteration\_times*, *decay\_rate*, *eps*):  
 # decay\_rate: decay rate, eps: hyperparameter,, alpha: learning rate  
 train\_loss = []  
 valid\_loss = []  
 valid\_accu = []  
 # initial params  
 w = init\_param(*x\_train*.shape[1]) # initialize weight and bias  
 cum\_grad = init\_param(1, method='zero') # initial with zero  
 cum\_u\_w = init\_param(1, method='zero') # initial with zero  
 u\_w\_list = [init\_param(*x\_train*.shape[1], method='zero')]  
  
 *for* i *in* range(*iteration\_times*):  
 # get batch train and test data  
 index = get\_batch(*x\_train*, batch\_size=int(*x\_train*.shape[0]))  
 x\_train\_batch = *x\_train*[index, :]  
 y\_train\_batch = *y\_train*[index]  
  
 # logistic model, and train loss  
 y\_pred, y\_pred\_label = linear\_model(x\_train\_batch, w)  
 loss\_t = loss\_function(y\_train\_batch, y\_pred, w) # square loss function of train data  
 train\_loss.append(loss\_t)  
  
 # logistic model, and validation loss  
 y\_valid\_pred, y\_valid\_pred\_label = linear\_model(*x\_valid*, w)  
 loss\_v = loss\_function(*y\_valid*, y\_valid\_pred, w) # square loss function of validation data  
 valid\_loss.append(loss\_v)  
 valid\_accu.append(accuracy\_score(*y\_valid*, y\_valid\_pred\_label))  
  
 # if i % (iteration\_times/2) == 0: # print info  
 # print('alpha:', a, 'iteration:', i)  
  
 w, cum\_grad, cum\_u\_w = update\_params\_adadelta(x\_train\_batch, y\_train\_batch, y\_pred, w, *decay\_rate*, *eps*, cum\_grad, cum\_u\_w,  
 u\_w\_list)  
 # print('learning rate:', alpha, "---min train loss:", min(train\_loss), '--min valid loss:', min(valid\_loss))  
 *return* train\_loss, valid\_loss, valid\_accu  
  
  
*def* adam\_model(*x\_train*, *x\_valid*, *y\_train*, *y\_valid*, *iteration\_times*, *betal\_one*, *belta\_two*, *eps*, *alpha*):  
 # decay\_rate: decay rate, eps: hyperparameter,, alpha: learning rate  
 train\_loss = []  
 valid\_loss = []  
 valid\_accu = []  
  
 # initial params  
 w = init\_param(*x\_train*.shape[1]) # initialize weight and bias  
 m = init\_param(1, method='zero') # initial with zero  
 v = init\_param(1, method='zero') # initial with zero  
  
 *for* i *in* range(*iteration\_times*):  
 # get batch train and test data  
 index = get\_batch(*x\_train*, batch\_size=int(*x\_train*.shape[0]/*iteration\_times*))  
 x\_train\_batch = *x\_train*[index, :]  
 y\_train\_batch = *y\_train*[index]  
  
 # logistic model, and train loss  
 y\_pred, y\_pred\_label = linear\_model(x\_train\_batch, w)  
 loss\_t = loss\_function(y\_train\_batch, y\_pred, w) # square loss function of train data  
 train\_loss.append(loss\_t)  
  
 # logistic model, and validation loss  
  
 y\_valid\_pred, y\_valid\_pred\_label = linear\_model(*x\_valid*, w)  
 loss\_v = loss\_function(*y\_valid*, y\_valid\_pred, w) # square loss function of validation data  
 valid\_loss.append(loss\_v)  
 valid\_accu.append(accuracy\_score(*y\_valid*, y\_valid\_pred\_label))  
  
 # if i % (iteration\_times/2) == 0: # print info  
 # print('alpha:', a, 'iteration:', i)  
  
 w, m, v = update\_params\_adam(x\_train\_batch, y\_train\_batch, y\_pred, w, m, v, *betal\_one*, *belta\_two*, *eps*, *alpha*, i)  
 *return* train\_loss, valid\_loss, valid\_accu

x\_train, x\_valid, y\_train, y\_valid = prepare\_data(a9a\_train, a9a\_test) # prepare train and test data  
train\_total\_loss = [] # train loss value  
valid\_total\_loss = [] # test loss value  
  
mu = 0.9 # momentum  
alpha = [0.1, 0.01, 0.001] # learning rate  
decay\_rate = 0.9 # decay rate  
eps = 1e-8 # hyprparameter  
betal\_one = 0.9  
betal\_two = 0.99  
  
nga\_total\_train\_loss = []  
nga\_total\_valid\_loss = []  
nga\_total\_valid\_accu = []  
*for* a *in* alpha:  
 train\_loss, valid\_loss, valid\_accu = nga\_model(x\_train, x\_valid, y\_train, y\_valid, iteration\_times, mu, a)  
 nga\_total\_train\_loss.append(train\_loss)  
 nga\_total\_valid\_loss.append(valid\_loss)  
 nga\_total\_valid\_accu.append(valid\_accu)  
  
rmsprop\_total\_train\_loss = []  
rmsprop\_total\_valid\_loss = []  
rmsprop\_total\_valid\_accu = []  
*for* a *in* alpha:  
 train\_loss, valid\_loss, valid\_accu = rmsprop\_model(x\_train, x\_valid, y\_train, y\_valid, iteration\_times, decay\_rate, eps, a)  
 rmsprop\_total\_train\_loss.append(train\_loss)  
 rmsprop\_total\_valid\_loss.append(valid\_loss)  
 rmsprop\_total\_valid\_accu.append(valid\_accu)  
  
  
adadelta\_total\_train\_loss = []  
adadelta\_total\_valid\_loss = []  
adadelta\_total\_valid\_accu = []  
*for* a *in* alpha:  
 train\_loss, valid\_loss, valid\_accu = adadelta\_model(x\_train, x\_valid, y\_train, y\_valid, iteration\_times, decay\_rate, eps)  
 adadelta\_total\_train\_loss.append(train\_loss)  
 adadelta\_total\_valid\_loss.append(valid\_loss)  
 adadelta\_total\_valid\_accu.append(valid\_accu)  
  
  
adam\_total\_train\_loss = []  
adam\_total\_valid\_loss = []  
adam\_total\_valid\_accu = []  
*for* a *in* alpha:  
 train\_loss, valid\_loss, valid\_accu = adam\_model(x\_train, x\_valid, y\_train, y\_valid, iteration\_times, betal\_one, betal\_two, eps, a)  
 adam\_total\_train\_loss.append(train\_loss)  
 adam\_total\_valid\_loss.append(valid\_loss)  
 adam\_total\_valid\_accu.append(valid\_accu)  
*print*("done")

*def* plot\_result(*train\_loss*, *valid\_loss*, *fig\_config*):  
 # train loss plt with different alpha  
 *for* i *in* range(len(alpha)):  
 plt.figure()  
 plt.title('train and validation loss with learning rate '+ str(alpha[i]))  
 plt.plot(range(len(*train\_loss*[i])), *train\_loss*[i], linewidth=2.0,  
 color=*fig\_config*['color'][0], label=*fig\_config*['label'][0])  
 plt.plot(range(len(*valid\_loss*[i])), *valid\_loss*[i], linewidth=2.0,  
 color=*fig\_config*['color'][1], label=*fig\_config*['label'][1])  
 plt.xlabel('iteration times')  
 plt.ylabel('loss value')  
 plt.legend(*fig\_config*['label'])  
 plt.show()  
  
  
# plot params  
fig\_config = { # train line  
 'label':['train loss-nga', 'valid loss-nga'], # validation line  
 'color': ['r', 'g'],  
}  
plot\_result(nga\_total\_train\_loss, nga\_total\_valid\_loss, fig\_config)  
  
# plot params  
fig\_config = { # train line  
 'label':['train loss-rmsprop', 'valid loss-rmsprop'], # validation line  
 'color': ['r', 'g'],  
}  
plot\_result(rmsprop\_total\_train\_loss, rmsprop\_total\_valid\_loss, fig\_config)  
  
  
fig\_config = { # train line  
 'label':['train loss-adadelta', 'valid loss-adadelta'], # validation line  
 'color': ['r', 'g'],  
}  
plot\_result(adadelta\_total\_train\_loss, adadelta\_total\_valid\_loss, fig\_config)  
  
  
fig\_config = { # train line  
 'label':['train loss-adam', 'valid loss-adam'], # validation line  
 'color': ['r', 'g'],  
}  
plot\_result(adam\_total\_train\_loss, adam\_total\_valid\_loss, fig\_config)  
  
  
  
*def* plot\_result\_accu(*accu\_list*, *fig\_config*, *alpha*):  
 # train loss plt with different alpha  
 plt.figure()  
 plt.title('validation accuracy with different method alpha-'+str(*alpha*))  
  
 *for* i *in* range(len(*accu\_list*)):  
 plt.plot(range(len(*accu\_list*[i])), *accu\_list*[i], linewidth=2.0,  
 color=*fig\_config*['color'][i], label=*fig\_config*['label'][i])  
 plt.xlabel('iteration times')  
 plt.ylabel('accuracy value')  
 plt.legend(*fig\_config*['label'])  
 plt.show()  
  
# plot params  
fig\_config = { # train line  
 'label':['accuracy-nga', 'accuracy-rmsprop', 'accuracy-adadelta', 'accuracy-adam'], # validation line  
 'color': ['b', 'c', 'g', 'k'],  
}  
*for* i *in* range(len(nga\_total\_valid\_accu)):  
 plot\_result\_accu([nga\_total\_valid\_accu[i], rmsprop\_total\_valid\_accu[i], adadelta\_total\_valid\_accu[i], adam\_total\_valid\_accu[i]], fig\_config, alpha[i])







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.

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.

**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.