CS6301 MACHINE LEARNING LAB WEEK – 4 MLP

SRIHARI. S – 2018103601

Date: 15-03-2021 Monday

Aim: To experiment on Multilayer Perceptron by varying different hyperparameters and observing which results in better accuracy using two labelled datasets from UCI repository.

Dataset 1: **Acute Inflammations Dataset**

Url: https://archive.ics.uci.edu/ml/datasets/Acute+Inflammations

Description:

The main idea of this data set is to prepare the algorithm of the expert system, which will perform the presumptive diagnosis of two diseases of urinary system. It will be the example of diagnosing of the acute inflammations of urinary bladder and acute nephritises. Acute inflammation of urinary bladder is characterised by sudden occurrence of pains in the abdomen region and the urination in form of constant urine pushing, micturition pains and sometimes lack of urine keeping. At proper treatment, symptoms decay usually within several days. However, there is inclination to returns. At persons with acute inflammation of urinary bladder, we should expect that the illness will turn into protracted form. It begins fever and is accompanied by shivers and one- or both-side lumbar pains, which are sometimes very strong. Symptoms of acute inflammation of urinary bladder appear very often. Quite not infrequently there are nausea and vomiting and spread pains of whole abdomen. The data was created by a medical expert as a data set to test the expert system, which will perform the presumptive diagnosis of two diseases of urinary system. The basis for rules detection was Rough Sets Theory. Each instance represents a potential patient. The data is in an ASCII file. Attributes are separated by TAB.

Input: The following 6 attributes

- Occurrence of nausea { 0, 1 },
- Lumbar pain { 0, 1},
- Urine pushing { 0, 1},
- Micturition pains { 0, 1},
- Burning of urethra { 0, 1},
- Inflammation of urinary bladder { 0, 1},

jupyter diagnosis-data.csv ✔ a day ago

File Edit View Language

```
temperature	nausea	lumbarpain	urinepushing	pains	burning	inflamation	decision
"35,5"	0	1	0	0	0	0	0
"35,9"	0	0	1	1	1	1	0
"35,9"	0	1	0	0	0	0	0
"36,0"	0	0	1	1	1	1	0
"36,0"	0	1	0	0	0	0	0
"36,0"	0	1	0	0	0	0	0
"36,2"	0	0	1	1	1	1	0
"36,2"	0	1	0	0	0	0	0
"36,3"	0	0	1	1	1	1	0
"36,6"	0	0	1	1	1	1	0
```

Output: decision: Nephritis of renal origin { 0, 1}

TABULATION

Dataset	No of neurons in the hidden layers	Activation Function	Learning Rate (eta)	Epochs	Accuracy (%)	Inference
Acute Inflammations	6	Sigmoid	0.1	100	52.09	Initially accuracy is 52.09%
	4	Sigmoid	0.1	100	43.05	When the number of hidden layer nodes is decreased the accuracy drops
	8	Sigmoid	0.0025	50	53.8	When no of hidden layer neurons is increased and learning rate and epochs are reduced **Accuracy is increased**
	8	Softmax	0.1	50	57.08	When Softmax activation is used accuracy is maximised.

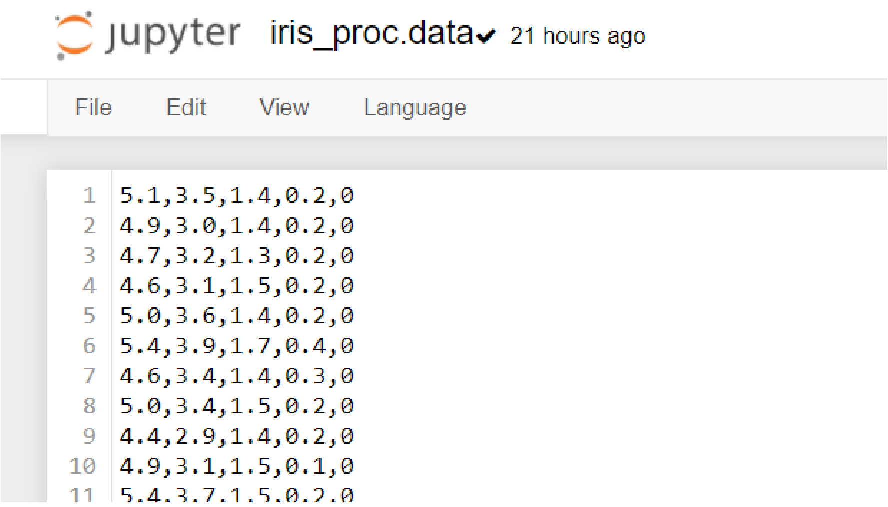
Dataset 2: **Iris Dataset**

Url: https://archive.ics.uci.edu/ml/datasets/iris

Description: The **Iris Dataset** contains four features (length and width of sepals and petals) of 50 samples of three species of **Iris** (**Iris** setosa, **Iris** virginica and **Iris** versicolor).

Input: The following 4 attributes

- sepal length in cm,
- sepal width in cm,
- petal length in cm,
- petal width in cm,




```
5.1,3.5,1.4,0.2,0
4.9,3.0,1.4,0.2,0
4.7,3.2,1.3,0.2,0
4.6,3.1,1.5,0.2,0
5.0,3.6,1.4,0.2,0
5.4,3.9,1.7,0.4,0
4.6,3.4,1.4,0.3,0
5.0,3.4,1.5,0.2,0
4.4,2.9,1.4,0.2,0
4.9,3.1,1.5,0.1,0
5.4,3.7,1.5,0.2,0
```

Output: decision: Multiclass classification among 3 classes of flowers: Iris Setosa, Iris Versicolour, Iris Virginica.

Dataset	No of neurons in the hidden layers	Activation Function	Learning Rate (eta)	Epochs	Accuracy (%)	Inference
Iris	6	Sigmoid	0.1	100	97.29	Initially accuracy is 97.29%
	4	Sigmoid	0.4	100	40.95	When the number of hidden layer nodes is decreased and the learning rate is increased the accuracy drops
	8	Softmax	0.025	5000	91.8	When no of hidden layer neurons and epochs are increased along with softmax activation **Accuracy is improves**
	8	Softmax	0.1	5000	94.54	When learning rate is increased along with softmax activation accuracy is maximised.

VIVA QUESTIONS:

1. **What is the effect of the hyperparameter eta (learning rate) on the model?**
   While experimenting the effect of learning rate, it was observed that neither a learning rate which is high nor a lower one produced an impressive result. The model was trained using three different learning rates 0.1, 0.4 and 0.025. It is observed that the accuracy was at the maximum when eta = 0.1. While it was considerably lower at 0.4 and 0.025. Hence setting a learning rate between 0.1 and 0.4 is appreciated.

2. **What is the effect of increasing the number of hidden layer nodes?**
   With respect to the two datasets taken, it was duly observed that as we increase the number of hidden layer nodes the accuracy increases. The accuracy reached the maximum of 57% when no. of hidden layer nodes is 12. On further increase the accuracy is observed to be stagnant at 58%
   Hence concluding that more the number of hidden layer neurons the better the accuracy (in this case). But after some point it tends to saturate at the maximum value.
3. **Which activation function worked better on the model?**

   The model was trained on 3 different activation functions – Sigmoid, SoftMax and Linear. For the given datasets, its observed that the Sigmoid activation function produces more accurate results than SoftMax or linear activations.

4. **Which approach is better – Increasing the number of hidden layers or the number of nodes in a hidden layer?**
   In a Multilayer perceptron, increasing the number of hidden layers seem to be the wiser choice. With backpropagation the error is minimized if the no of hidden layers is more. The minimization of error isn't found to be of a similar extent if we just increase the number of nodes in the hidden layer.

EXPERIMENTATION:

A. Effect of no. of hidden layer nodes

Illustrated from Dataset - 1

When no. of hidden layer nodes = 6

```
Initial:
HiddenW: [0.46737053 0.32610539 0.64210147 0.5228506  0.2780411  0.64322595] [0.08146968 0.98614474 0.15313826 0.80348666 0.272
87831 0.06384024] [0.33377983 0.8161339  0.37509577 0.10450444 0.585775   0.12962218] [0.84653276 0.04444411 0.88551256 0.12684
624 0.69759123 0.94007891] [0.15038777 0.05313697 0.41074944 0.26362925 0.86717925 0.38886812] [0.44087887 0.02456626 0.4136133
7 0.99212135 0.20557317 0.37361811]
OutputW: [0.67366477] [0.11379732] [0.68214481] [0.44020528] [0.64735236] [0.44840657]
HiddenB: [0.22189343 0.79200523 0.1870442  0.43378445 0.16520778 0.07448311]
OutputB: [0.58080474]

Final:
HiddenW: [0.46737052720918937 0.3261053928690044 0.6421014733167366
 0.5228506014349665 0.27804109640665153 0.6432259500452587] [0.08146967510382008 0.9861447390304077 0.15313825597293673
 0.8034866566923147 0.2728783079747623 0.06384023858813692] [0.3337798308122041 0.8161338961527351 0.375095768531604
 0.10450444448724383 0.5857750033418904 0.12962217804693832] [0.8465327575883126 0.04444410597751147 0.8855125590480672
 0.12684624143408674 0.6975912349968553 0.940078906892575] [0.15038777283954008 0.053136971906522 0.41074944375709466
 0.2636292504457275 0.8671792527212371 0.3888681225486855] [0.44087887253601465 0.024566261696141845 0.4136133746068472
 0.992121345759018 0.2055731745919367 0.3736181098258472]
OutputW: [0.67366477] [0.11379732] [0.68214481] [0.44020528] [0.64735236] [0.44840657]
HiddenB: [0.22189343 0.79200523 0.1870442  0.43378445 0.16520778 0.07448311]
OutputB: [0.58080474]


Predicted Value of Y: [0.] [1.] [0.] [1.] [0.] [0.] [1.] [0.] [1.] [1.] [1.] [0.] [0.] [1.] [0.] [0.] [1.] [1.] [1.] [0.] [0.]
[0.] [0.] [1.] [1.] [1.] [1.] [0.] [0.] [1.] [0.] [0.] [0.] [0.] [0.] [1.] [0.] [0.] [0.] [0.] [0.] [0.] [1.] [0.] [0.] [0.]
[0.] [1.] [0.] [0.] [0.] [0.] [0.] [1.] [0.] [0.] [0.] [0.] [1.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [1.]
[1.] [1.] [0.] [0.] [0.] [0.] [0.] [1.] [1.] [0.] [0.] [0.] [1.] [1.] [1.] [0.] [0.] [1.] [1.] [0.] [0.] [1.] [1.] [0.] [0.]
[0.] [0.] [1.] [1.] [0.] [1.] [0.] [0.] [0.] [1.] [1.] [0.] [0.] [0.] [1.] [0.] [0.] [0.] [1.] [0.] [0.] [0.] [0.] [0.]
Expected Value of Y:  [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0]
[0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0]
[0] [0] [1] [1] [1] [1] [1] [1] [1] [1] [1] [1] [1] [1] [1] [0] [0] [1] [1] [1] [1] [1] [0] [1] [1] [1] [1] [1] [0] [1] [1] [1]
[0] [1] [1] [1] [0] [1] [1] [1] [1] [1] [1] [1] [0] [1] [1] [1] [1] [0] [1] [1] [1] [0] [1] [1] [1] [1] [0] [1] [1] [1]
Accuracy =  52.916666666666664 %
Recall =  0.325
Precision =  0.4166666666666667
```

When no of hidden layer nodes = 4

```
Initial:
HiddenW: [0.10486589 0.00475398 0.49845734 0.97078858] [0.96353931 0.26720761 0.14463688 0.72478569] [0.03302638 0.51421961 0.5
9553732 0.94689796] [0.47740976 0.25652889 0.1447705  0.12224321] [0.33699347 0.00244102 0.74301772 0.44327337] [0.29512366 0.7
8434709 0.01456419 0.42343647]
OutputW: [0.23603216] [0.3686501] [0.98155441] [0.11455933]
HiddenB: [0.05336223 0.52752456 0.26043619 0.70096266]
OutputB: [0.30746993]

Final:
HiddenW: [0.10486588878747438 0.004753982831310455 0.49845734231203964
 0.9707885762468413] [0.9635393132426979 0.26720760591725523 0.14463688024489263
 0.7247856871461777] [0.033026383847891916 0.5142196081700298 0.5955373199953556
 0.9468979599210711] [0.47740975523415463 0.25652888713403477 0.14477050130925895
 0.12224320762022156] [0.3369934670438518 0.0024410215498328336 0.7430177186228925
 0.4432733711252238] [0.29512366216552455 0.7843470917137284 0.014564192591496772
 0.423436473289017]
OutputW: [0.23603216] [0.3686501] [0.98155441] [0.11455933]
HiddenB: [0.05336223 0.52752456 0.26043619 0.70096266]
OutputB: [0.30746993]


Predicted Value of Y: [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.]
[1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.]
[1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.]
[1.] [1.] [0.] [0.] [1.] [1.] [1.] [1.] [1.] [0.] [1.] [1.] [1.] [1.] [1.] [0.] [1.] [1.] [1.] [0.] [1.] [1.] [1.] [0.] [1.]
[1.] [1.] [1.] [1.] [1.] [1.] [0.] [1.] [1.] [1.] [1.] [0.] [1.] [1.] [1.] [0.] [1.] [1.] [1.] [1.] [0.] [1.] [1.] [1.]
Expected Value of Y:  [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0]
[0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0]
[0] [0] [1] [1] [1] [1] [1] [1] [1] [1] [1] [1] [1] [1] [1] [0] [0] [1] [1] [1] [1] [1] [0] [1] [1] [1] [1] [1] [0] [1] [1] [1]
[0] [1] [1] [1] [0] [1] [1] [1] [1] [1] [1] [1] [0] [1] [1] [1] [1] [0] [1] [1] [1] [0] [1] [1] [1] [1] [0] [1] [1] [1]
Accuracy =  43.05555555555556 %
Recall =  0.9166666666666666
Precision =  0.4166666666666667
```

When no. of hidden layer nodes = 8

```
Initial:
HiddenW: [0.52113332 0.69672239 0.0481251  0.36627131 0.04919404 0.95640331
 0.24374593 0.61485313] [0.74847152 0.97846479 0.66685576 0.74361684 0.65795176 0.77969315
 0.99724443 0.09359119] [0.32297078 0.7219364  0.28765601 0.31494942 0.97149863 0.13768391
 0.05146214 0.95213362] [0.79434009 0.89079386 0.17006882 0.08243361 0.5379732  0.90570966
 0.90705052 0.10653489] [0.95881731 0.32438906 0.57974572 0.56224188 0.97707219 0.75961122
 0.40804178 0.65215237] [0.56941702 0.19856479 0.80770734 0.99720451 0.7861559  0.35336204
 0.04599711 0.62373123]
OutputW: [0.59002125] [0.15536256] [0.71605378] [0.17068063] [0.83662483] [0.29081319] [0.96852818] [0.79560823]
HiddenB: [0.39220734 0.67326825 0.2227368  0.18915635 0.53809742 0.58543378
 0.42597209 0.45989145]
OutputB: [0.83586633]

Final:
HiddenW: [0.5211333218304788 0.6967223908750331 0.04812510480638821
 0.36627130591260326 0.04919403842770653 0.9564033130075997
 0.2437459276108106 0.6148531321582815] [0.7484715185383662 0.978464790665175 0.666855757223601
 0.7436168412537234 0.6579517609472271 0.7796931532647176
 0.9972444281029884 0.09359119214036893] [0.32297078119823197 0.7219364044704759 0.28765600849943895
 0.3149494239731654 0.9714986317422118 0.13768391000354185
 0.05146214373228619 0.952133616526254] [0.7943400854778193 0.8907938567305937 0.1700688221447968
 0.08243360556066337 0.5379731951613618 0.9057096572844452
 0.907050522809205 0.10653489450637987] [0.9588173083967793 0.3243890630522033 0.5797457165416643
 0.5622418849602633 0.9770721868543508 0.7596112239796414
 0.40804178147464554 0.6521523668320899] [0.5694170246992304 0.19856479458821885 0.8077073371304012
 0.9972045123679127 0.7861559035677493 0.3533620433768193
 0.04599711187640709 0.6237312300370585]
OutputW: [0.59002125] [0.15536256] [0.71605378] [0.17068063] [0.83662483] [0.29081319] [0.96852818] [0.79560823]
HiddenB: [0.39220734 0.67326825 0.2227368  0.18915635 0.53809742 0.58543378
 0.42597209 0.45989145]
OutputB: [0.83586633]


Predicted Value of Y: [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.]
[0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.]
[0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [1.]
[1.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [1.] [0.] [0.] [0.] [1.] [0.] [0.] [0.] [0.] [0.] [1.] [0.] [0.] [1.] [0.] [0.] [0.]
[0.] [0.] [0.] [0.] [0.] [1.] [0.] [0.] [0.] [1.] [0.] [0.] [0.] [1.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.]
Expected Value of Y:  [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0]
[0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0]
[0] [0] [1] [1] [1] [1] [1] [1] [1] [1] [1] [1] [1] [1] [1] [0] [0] [1] [1] [1] [1] [1] [0] [1] [1] [1] [1] [1] [0] [1] [1] [1]
[0] [1] [1] [1] [0] [1] [1] [1] [1] [1] [1] [1] [0] [1] [1] [1] [1] [0] [1] [1] [1] [0] [1] [1] [1] [1] [0] [1] [1] [1]
Accuracy =  57.08333333333333 %
Recall =  0.075
Precision =  0.4166666666666667
```

We see that as the number of hidden layer nodes increases the accuracy improves.

B. Effect of Activation Function

Illustrated using Dataset – 2

When **Sigmoid** Activation Function is applied

```
3
Iteration:  0  Error:  0.014361966164698895
Confusion matrix is:
[[15.  0.  0.]
 [ 0. 11.  0.]
 [ 0.  0. 11.]]
Percentage Correct:  100.0
```

When **Softmax** Activation Function is applied

```
97
Iteration:  0  Error:  0.07875221023048719
Confusion matrix is:
[[14.  0.  0.]
 [ 0.  8.  1.]
 [ 0.  2. 12.]]
Percentage Correct:  91.8918918918919
```

When **linear** Activation Function is applied

```
74
Iteration:  0  Error:  0.500438720386741
Confusion matrix is:
[[13.  0.  0.]
 [ 0. 14.  0.]
 [ 0.  2.  8.]]
Percentage Correct:  94.5945945945946
```

We can see that the sigmoid activation function converges quickly producing better accuracy, while the softmax and linear activations aren't as accurate and take more time to converge.

C. Effect of the number of epochs

Illustrated using Dataset – 1

When no of epochs = 100

```
Predicted Value of Y: [0.] [1.] [0.] [1.] [0.] [0.] [1.] [0.] [1.] [1.] [1.] [0.] [0.] [1.] [0.] [0.] [1.] [1.] [1.] [0.] [0.]
[0.] [0.] [1.] [1.] [1.] [1.] [0.] [0.] [1.] [0.] [0.] [0.] [0.] [0.] [1.] [0.] [0.] [0.] [0.] [0.] [0.] [1.] [0.] [0.] [0.]
[0.] [1.] [0.] [0.] [0.] [0.] [0.] [1.] [0.] [0.] [0.] [0.] [1.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [0.] [1.]
[1.] [1.] [0.] [0.] [0.] [0.] [0.] [1.] [1.] [0.] [0.] [0.] [1.] [1.] [1.] [0.] [0.] [1.] [1.] [0.] [0.] [1.] [1.] [0.] [0.]
[0.] [0.] [1.] [1.] [0.] [1.] [0.] [0.] [0.] [1.] [1.] [0.] [0.] [0.] [1.] [0.] [0.] [0.] [1.] [0.] [0.] [0.] [0.] [0.]
Expected Value of Y:  [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0]
[0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0]
[0] [0] [1] [1] [1] [1] [1] [1] [1] [1] [1] [1] [1] [1] [1] [0] [0] [1] [1] [1] [1] [1] [0] [1] [1] [1] [1] [1] [0] [1] [1] [1]
[0] [1] [1] [1] [0] [1] [1] [1] [1] [1] [1] [1] [0] [1] [1] [1] [1] [0] [1] [1] [1] [0] [1] [1] [1] [1] [0] [1] [1] [1]
Accuracy =  52.916666666666664 %
Recall =  0.325
Precision =  0.4166666666666667
```

When no of epochs = 50

```
Predicted Value of Y: [0.] [1.] [0.] [1.] [0.] [0.] [1.] [0.] [1.] [1.] [1.] [0.] [0.] [1.] [0.] [0.] [1.] [1.] [1.] [0.] [1.]
[1.] [0.] [1.] [1.] [1.] [1.] [0.] [0.] [1.] [0.] [1.] [0.] [0.] [0.] [1.] [0.] [0.] [0.] [1.] [0.] [0.] [1.] [0.] [0.] [1.]
[1.] [1.] [1.] [1.] [0.] [0.] [0.] [1.] [0.] [1.] [1.] [0.] [1.] [0.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.]
[1.] [1.] [0.] [0.] [0.] [0.] [1.] [1.] [1.] [0.] [0.] [1.] [1.] [1.] [1.] [0.] [0.] [1.] [1.] [0.] [0.] [1.] [1.] [0.] [0.]
[1.] [1.] [1.] [1.] [1.] [1.] [0.] [0.] [1.] [1.] [1.] [0.] [0.] [1.] [1.] [0.] [0.] [1.] [1.] [1.] [0.] [0.] [1.] [1.]
Expected Value of Y:  [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0]
[0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0]
[0] [0] [1] [1] [1] [1] [1] [1] [1] [1] [1] [1] [1] [1] [1] [0] [0] [1] [1] [1] [1] [1] [0] [1] [1] [1] [1] [1] [0] [1] [1] [1]
[0] [1] [1] [1] [0] [1] [1] [1] [1] [1] [1] [1] [0] [1] [1] [1] [1] [0] [1] [1] [1] [0] [1] [1] [1] [1] [0] [1] [1] [1]
Accuracy =  48.61111111111111 %
Recall =  0.5833333333333334
Precision =  0.4166666666666667
```

When no of epochs = 5000

```
Predicted Value of Y: [0.] [1.] [0.] [1.] [0.] [0.] [1.] [0.] [1.] [1.] [1.] [0.] [0.] [1.] [0.] [0.] [1.] [1.] [1.] [0.] [1.]
[1.] [0.] [1.] [1.] [1.] [1.] [0.] [0.] [1.] [0.] [1.] [0.] [0.] [0.] [1.] [0.] [0.] [0.] [1.] [0.] [0.] [1.] [0.] [0.] [1.]
[1.] [1.] [1.] [1.] [0.] [0.] [0.] [1.] [0.] [1.] [1.] [0.] [1.] [0.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.]
[1.] [1.] [0.] [0.] [1.] [1.] [1.] [1.] [1.] [0.] [1.] [1.] [1.] [1.] [1.] [0.] [1.] [1.] [1.] [0.] [1.] [1.] [1.] [0.] [1.]
[1.] [1.] [1.] [1.] [1.] [1.] [0.] [1.] [1.] [1.] [1.] [0.] [1.] [1.] [1.] [0.] [1.] [1.] [1.] [1.] [0.] [1.] [1.] [1.]
Expected Value of Y:  [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0]
[0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0]
[0] [0] [1] [1] [1] [1] [1] [1] [1] [1] [1] [1] [1] [1] [1] [0] [0] [1] [1] [1] [1] [1] [0] [1] [1] [1] [1] [1] [0] [1] [1] [1]
[0] [1] [1] [1] [0] [1] [1] [1] [1] [1] [1] [1] [0] [1] [1] [1] [1] [0] [1] [1] [1] [0] [1] [1] [1] [1] [0] [1] [1] [1]
Accuracy =  47.22222222222222 %
Recall =  0.6666666666666666
Precision =  0.4166666666666667
```

Though we can see that the accuracy improves as the number of epochs are increased from 50 to 100. But it is also necessary to see that increasing the no. of epochs doesn't necessarily mean better accuracy as we see a dip in accuracy when we trained for 5000 epochs.

D. Effect of Learning rate

Illustrated using dataset-2

When eta = 0.1

```
Confusion matrix is:
[[12.  0.  0.]
 [ 0. 13.  1.]
 [ 0.  0. 11.]]
Percentage Correct:  97.2972972972973
```

When eta = 0.4

```
Confusion matrix is:
[[15. 14.  8.]
 [ 0.  0.  0.]
 [ 0.  0.  0.]]
Percentage Correct:  40.54054054054054
```

When eta = 0.025

```
Confusion matrix is:
[[10.  0.  0.]
 [ 0. 15.  0.]
 [ 0.  2. 10.]]
Percentage Correct:  94.5945945945946
```

Neither a learning rate which is high nor a lower one produced an impressive result. The model was trained using three different learning rates 0.1, 0.4 and 0.025. It is observed that the accuracy was at the maximum when eta = 0.1. While it was considerably lower at 0.4 and 0.025. Hence setting a learning rate between 0.1 and 0.4 is appreciated.

CODE FOR DATASET – 1

```
import numpy as np      # for math functions
import pandas as pd     # for importing and using datasets
import random           # for generating random weights

def load_data(filename, target):
    dataset = pd.read_csv("dataset/"+filename, sep="\t")  # read .csv file
    print(dataset,"\n")
    x = np.array(dataset.drop([target],1))
    y = np.array(dataset[target]) # y contains the target class
    print("\nX = \n",x)
    print("\nY = \n",y)
    print("\n\n")
    return (dataset,x,y,target)

inp = load_data("diagnosis-data.csv","decision")
# x_data = add_initial_column(inp[1],-1)
x_data = np.array(inp[1])
y_data = np.array([inp[2]])
y_data = np.transpose(y_data)
x_data = x_data[:,1:]
print(x_data,"\n")
print(y_data)
def activation_func(x):
```

```
    return sigmoid(x)

def sigmoid(x):
    # print("Exp: ", np.exp(-x))
    x = np.array(x, dtype=int)
    return 1/(1 + np.exp(-x))

def sign(x):
    res = 0 if x <= 0 else 1
    # print(res)
    return res
def sigmoid_deriv(x):
    x = np.array(x, dtype=int)
    return x * ( 1 - x)
```

```
class MLP_single_hidden():

    def __init__(self, numI, numH, numO, x_data, y_data):
        self.numI = numI
        self.numH = numH
        self.numO = numO
        self.x = x_data
        self.y = y_data
        self.epochs = 5000
        self.lr = 0.1
        self.tp = 0
        self.tn = 0
        self.fp = 0
        self.fn = 0
        self.cost_array = []
        self.initialize_weights()
        self.initialize_bias()

    def initialize_weights(self):
        self.hiddenW = np.random.uniform(size=(self.numI, self.numH))
        self.outputW = np.random.uniform(size=(self.numH, self.numO))

    def initialize_bias(self):
        self.hiddenB = np.random.uniform(size=(1,self.numH))
        self.outputB = np.random.uniform(size=(1,self.numO))
```

```
    def forward_propagation(self):
        self.hidden_layer_activation = np.dot(self.x, self.hiddenW)
        self.hidden_layer_activation += self.hiddenB
        self.hidden_layer_output = sigmoid(self.hidden_layer_activation)

        self.output_layer_activation = np.dot(self.hidden_layer_output,
self.outputW)
        self.output_layer_activation += self.outputB
        self.y_predict = sigmoid(self.output_layer_activation)

    def backward_propagation(self):
        self.error = ((self.y - self.y_predict)) #**2).mean()
        self.d_y_predict = self.error * sigmoid_deriv(self.y_predict)

        self.error_hidden_layer = self.d_y_predict.dot(self.outputW.T)
        self.d_hidden_layer =
self.error_hidden_layer*sigmoid_deriv(self.hidden_layer_output)

    def update_weights(self):
        self.outputW =
(self.outputW)+((self.hidden_layer_output.T.dot(self.d_y_predict))* self.lr)
        self.hiddenW = (self.hiddenW + (self.x.T.dot(self.d_hidden_layer)) * self.lr
)

    def update_bias(self):
        self.outputB = self.outputB + (np.sum(self.d_y_predict, axis=0,
keepdims=True) * self.lr)
        self.hiddenB = self.hiddenB + (np.sum(self.d_hidden_layer, axis=0,
keepdims=True) * self.lr)

    def train(self):
        for i in range(self.epochs):
            self.forward_propagation()
            self.backward_propagation()
            self.update_weights()
            self.update_bias()
            self.cost_array.append(self.error)
```

```
    def print_weights(self):
        print("HiddenW: ",end="")
        print(*self.hiddenW)
        print("OutputW: ",end="")
        print(*self.outputW)

    def print_bias(self):
        print("HiddenB: ",end="")
        print(*self.hiddenB)
        print("OutputB: ",end="")
        print(*self.outputB)

    def print_y_predict(self):
        print("\n\nPredicted Value of Y: ", end="")
        print(*self.y_predict)
        print("Expected Value of Y:  ", end="")
        print(*self.y)

        for i in self.y_predict:
            for j in self.y:
                if i==1 and j==1:
                    self.tp += 1
                elif i == 0 and j == 0:
                    self.tn += 1
                elif i == 1 and  j == 0:
                    self.fp += 1
                else:
                    self.fn += 1

        print("Accuracy =
",((self.tp+self.tn)/(self.tp+self.tn+self.fp+self.fn))*100,"%")
        print("Recall = ",(self.tp)/(self.tp+self.fn))
        print("Precision = ",(self.tp)/(self.tp+self.fp))

    def apply_threshold(self):
        for i in range(self.y_predict.size):
            if self.y_predict[i][0]  < 0.5:
                self.y_predict[i][0] = 0
            else:
```

```
                self.y_predict[i][0] = 1

XOR = MLP_single_hidden(6,8,1,x_data,y_data)

print("Initial: ")
XOR.print_weights()
XOR.print_bias()

XOR.train()

print("\nFinal: ")
XOR.print_weights()
XOR.print_bias()
XOR.apply_threshold()
XOR.print_y_predict()
```

OUTPUT

```
Predicted Value of Y: [0.] [1.] [0.] [1.] [0.] [0.] [1.] [0.] [1.] [1.] [1.] [0.] [0.] [1.] [0.] [0.] [1.] [1.] [1.] [0.] [0.]
[0.] [0.] [1.] [1.] [1.] [1.] [0.] [0.] [1.] [0.] [0.] [0.] [0.] [0.] [1.] [0.] [0.] [0.] [0.] [0.] [0.] [1.] [0.] [0.] [0.]
[0.] [1.] [0.] [0.] [0.] [0.] [0.] [1.] [0.] [0.] [0.] [0.] [1.] [0.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.]
[1.] [1.] [0.] [0.] [0.] [0.] [1.] [1.] [1.] [0.] [0.] [1.] [1.] [1.] [1.] [0.] [0.] [1.] [1.] [0.] [0.] [1.] [1.] [0.] [0.]
[1.] [1.] [1.] [1.] [1.] [1.] [0.] [0.] [1.] [1.] [1.] [0.] [0.] [1.] [1.] [0.] [0.] [1.] [1.] [1.] [0.] [0.] [1.] [1.]
Expected Value of Y:  [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0]
[0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0]
[0] [0] [1] [1] [1] [1] [1] [1] [1] [1] [1] [1] [1] [1] [1] [0] [0] [1] [1] [1] [1] [1] [0] [1] [1] [1] [1] [1] [0] [1] [1] [1]
[0] [1] [1] [1] [0] [1] [1] [1] [1] [1] [1] [1] [0] [1] [1] [1] [1] [0] [1] [1] [1] [0] [1] [1] [1] [1] [0] [1] [1] [1]
Accuracy =  50.0 %
Recall =  0.5
Precision =  0.4166666666666667
```

CODE FOR DATASET – 2

```
import numpy as np
class mlp:
def__init__(self,inputs,targets,nhidden,beta=1,momentum=0.9,outtype='logist
ic'):
      """ Constructor """
      # Set up network size
      self.nin = np.shape(inputs)[1]
      self.nout = np.shape(targets)[1]
      self.ndata = np.shape(inputs)[0]
```

```
        self.nhidden = nhidden

        self.beta = beta
        self.momentum = momentum
        self.outtype = outtype

        # Initialise network
        self.weights1 = (np.random.rand(self.nin+1,self.nhidden)-
0.5)*2/np.sqrt(self.nin)
        self.weights2 = (np.random.rand(self.nhidden+1,self.nout)-
0.5)*2/np.sqrt(self.nhidden)

    def earlystopping(self,inputs,targets,valid,validtargets,eta,niterations=100):

        valid = np.concatenate((valid,-np.ones((np.shape(valid)[0],1))),axis=1)

        old_val_error1 = 100002
        old_val_error2 = 100001
        new_val_error = 100000

        count = 0
        while (((old_val_error1 - new_val_error) > 0.001) or ((old_val_error2 -
old_val_error1)>0.001)):
            count+=1
            print(count)
            self.mlptrain(inputs,targets,eta,niterations)
            old_val_error2 = old_val_error1
            old_val_error1 = new_val_error
            validout = self.mlpfwd(valid)
            new_val_error = 0.5*np.sum((validtargets-validout)**2)

        #print("Stopped", new_val_error,old_val_error1, old_val_error2)
        return new_val_error

    def mlptrain(self,inputs,targets,eta,niterations):
        """ Train the thing """
        # Add the inputs that match the bias node
        inputs = np.concatenate((inputs,-np.ones((self.ndata,1))),axis=1)
        change = range(self.ndata)
```

```
        updatew1 = np.zeros((np.shape(self.weights1)))
        updatew2 = np.zeros((np.shape(self.weights2)))

        for n in range(niterations):

            self.outputs = self.mlpfwd(inputs)

            error = 0.5*np.sum((self.outputs-targets)**2)
            if (np.mod(n,100)==0):
                print("Iteration: ",n, " Error: ",error)

            # Different types of output neurons
            if self.outtype == 'linear':
                    deltao = (self.outputs-targets)/self.ndata
            elif self.outtype == 'logistic':
                    deltao = self.beta*(self.outputs-targets)*self.outputs*(1.0-
self.outputs)
            elif self.outtype == 'softmax':
                deltao = (self.outputs-targets)*(self.outputs*(-
self.outputs)+self.outputs)/self.ndata
            else:
                    print("error")

            deltah = self.hidden*self.beta*(1.0-
self.hidden)*(np.dot(deltao,np.transpose(self.weights2)))

            updatew1 = eta*(np.dot(np.transpose(inputs),deltah[:,:-1])) +
self.momentum*updatew1
            updatew2 = eta*(np.dot(np.transpose(self.hidden),deltao)) +
self.momentum*updatew2
            self.weights1 -= updatew1
            self.weights2 -= updatew2

    def mlpfwd(self,inputs):
        """ Run the network forward """

        self.hidden = np.dot(inputs,self.weights1);
        self.hidden = 1.0/(1.0+np.exp(-self.beta*self.hidden))
        self.hidden = np.concatenate((self.hidden,-
np.ones((np.shape(inputs)[0],1))),axis=1)
```

```
        outputs = np.dot(self.hidden,self.weights2);

        # Different types of output neurons
        if self.outtype == 'linear':
            return outputs
        elif self.outtype == 'logistic':
            return 1.0/(1.0+np.exp(-self.beta*outputs))
        elif self.outtype == 'softmax':
            normalisers =
np.sum(np.exp(outputs),axis=1)*np.ones((1,np.shape(outputs)[0]))
            return np.transpose(np.transpose(np.exp(outputs))/normalisers)
        else:
            print("error")

    def confmat(self,inputs,targets):
        """Confusion matrix"""

        # Add the inputs that match the bias node
        inputs = np.concatenate((inputs,-np.ones((np.shape(inputs)[0],1))),axis=1)
        outputs = self.mlpfwd(inputs)

        nclasses = np.shape(targets)[1]

        if nclasses==1:
            nclasses = 2
            outputs = np.where(outputs>0.5,1,0)
        else:
            # 1-of-N encoding
            outputs = np.argmax(outputs,1)
            targets = np.argmax(targets,1)

        cm = np.zeros((nclasses,nclasses))
        for i in range(nclasses):
            for j in range(nclasses):
                cm[i,j] = np.sum(np.where(outputs==i,1,0)*np.where(targets==j,1,0))

        print("Confusion matrix is:")
        print(cm)
        print("Percentage Correct: ",np.trace(cm)/np.sum(cm)*100)
```

```
def preprocessIris(infile,outfile):

    stext1 = 'Iris-setosa'
    stext2 = 'Iris-versicolor'
    stext3 = 'Iris-virginica'
    rtext1 = '0'
    rtext2 = '1'
    rtext3 = '2'

    fid = open(infile,"r")
    oid = open(outfile,"w")

    for s in fid:
        if s.find(stext1)>-1:
            oid.write(s.replace(stext1, rtext1))
        elif s.find(stext2)>-1:
            oid.write(s.replace(stext2, rtext2))
        elif s.find(stext3)>-1:
            oid.write(s.replace(stext3, rtext3))
    fid.close()
    oid.close()

import numpy as np

iris = np.loadtxt('dataset/iris_proc.data',delimiter=',')
iris[:,:4] = iris[:,:4]-iris[:,:4].mean(axis=0)
imax =
np.concatenate((iris.max(axis=0)*np.ones((1,5)),np.abs(iris.min(axis=0)*np.one
s((1,5)))),axis=0).max(axis=0)
iris[:,:4] = iris[:,:4]/imax[:4]
print(iris[0:5,:])

# Split into training, validation, and test sets
target = np.zeros((np.shape(iris)[0],3));
indices = np.where(iris[:,4]==0)
target[indices,0] = 1
indices = np.where(iris[:,4]==1)
target[indices,1] = 1
indices = np.where(iris[:,4]==2)
```

```
target[indices,2] = 1

order = list(range(np.shape(iris)[0]))
np.random.shuffle(order)
iris = iris[order,:]
target = target[order,:]

train = iris[::2,0:4]
traint = target[::2]
valid = iris[1::4,0:4]
validt = target[1::4]
test = iris[3::4,0:4]
testt = target[3::4]

net = mlp(train,traint,4,outtype='logistic')
net.earlystopping(train,traint,valid,validt,0.025)
net.confmat(test,testt)
```

OUTPUT

```
[[-0.36142626  0.33135215 -0.7508489  -0.76741803  0.        ]
 [-0.45867099 -0.04011887 -0.7508489  -0.76741803  0.        ]
 [-0.55591572  0.10846954 -0.78268251 -0.76741803  0.        ]
 [-0.60453809  0.03417533 -0.71901528 -0.76741803  0.        ]
 [-0.41004862  0.40564636 -0.7508489  -0.76741803  0.        ]]
1
Iteration:  0  Error:  26.18596967951238
2
Iteration:  0  Error:  0.29392039840562867
3
Iteration:  0  Error:  0.19687826637173314
4
Iteration:  0  Error:  0.13901872119146352
5
Iteration:  0  Error:  0.1023685860773854
Confusion matrix is:
[[11.  0.  0.]
 [ 0. 11.  0.]
 [ 0.  1. 14.]]
Percentage Correct:  97.2972972972973
```