BITS F464

Machine Learning

Assignment - 2 Final Report

Submitted by:

Pranjal Gupta (2017A7PS0124H)
Ujjwal Raizada (2017A7PS1398H)
Simran Sandhu (2017A7PS1454H)

Logistic Regression:

Preprocessing

The dataset used for this assignment titled "Data Banknote Authentication" consists of data extracted from images that were taken from genuine and forged banknote-like specimens. For digitization, an industrial camera usually used for print inspection was used. The final images have 400x 400 pixels. Due to the object lens and distance to the investigated object gray-scale pictures with a resolution of about 660 dpi were gained. Wavelet Transform tool were used to extract features from images.

Attribute Information:

1. Variance of Wavelet Transformed image
2. Skewness of Wavelet Transformed image
3. Curtosis of Wavelet Transformed image
4. Entropy of image
5. Class (integer)

The target variable in the dataset is the last column (Class) which is indicating whether the note is forged or not.

he following steps have been used for preprocessing the dataset:

1. We have standardized the features by removing the mean and scaling to unit variance. The standard score of a sample x is calculated as:

   $$x = (x - u) / s$$

   where u is the mean of the training samples or zero and is the standard deviation of the training samples or one
2. The training and test split of the dataset is 80% and 20% respectively.

3. Various hyperparameters have been experimented with and tuned accordingly.

Loss Function

The loss function used in logistic regression is called **Logistic Loss.**

$$Cost(h_\theta(x), y) = \begin{cases} -log(h_\theta(x)) & \text{if } y = 1 \\ -log(1 - h_\theta(x)) & \text{if } y = 0 \end{cases}$$

If y = 1, when prediction = 1, the cost = 0, when prediction = 0, the learning algorithm gives a large cost. Similarly, if y = 0, predicting 0 gives cost = 0 while predicting 1 has a large cost.

One advantage of this loss function is that it can be written in a single formula which is very convenient for our calculation:

$$Cost(h_\theta(x), y) = -ylog(h_\theta(x)) - (1 - y)log(h_\theta(x))$$

So, the cost function of the model is the summation from all training data samples:

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} Cost(h_\theta(x^{(i)}), y^{(i)})$$

$$J(\theta) = \frac{1}{m}[\sum_{i=1}^{m} -y^{(i)}log(h_\theta(x^{(i)})) + (1 - y^{(i)})log(1 - h_\theta(x^{(i)}))]$$

m = number of samples

Regularisation

There are two commonly used regularization types, **L1(Lasso)** and **L2(Ridge).** Instead of optimizing above cost function directly, with regularization, we add a constraint on how big the coefficients can get in order to prevent overfitting.

$$L1: \quad J(\theta) = \frac{1}{m}\sum_{i=1}^{m} Cost(h_\theta(x^{(i)}), y^{(i)}) \quad s.t. \quad \|\theta\|_1 <= C$$

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} Cost(h_\theta(x^{(i)}), y^{(i)}) + \frac{\lambda}{m}\sum_{j=1}^{n} |\theta_j|$$

$$L2: \quad J(\theta) = \frac{1}{m}\sum_{i=1}^{m} Cost(h_\theta(x^{(i)}), y^{(i)}) \quad s.t. \quad \|\theta\|_2^2 <= C^2$$

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} Cost(h_\theta(x^{(i)}), y^{(i)}) + \frac{\lambda}{2m}\sum_{j=1}^{n} \theta_j^2$$

$$m = number\ of\ samples, \quad n = number\ of\ features$$

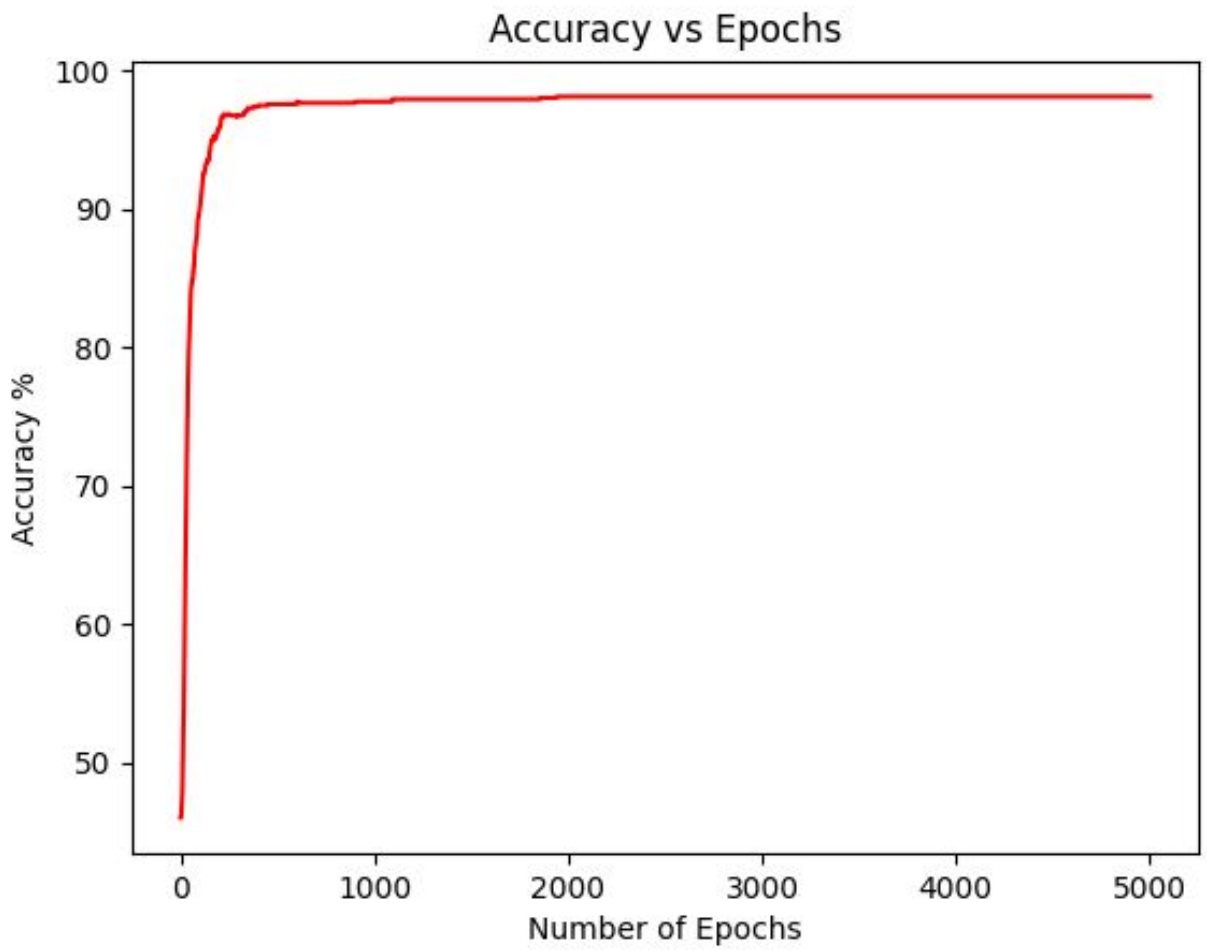
Optimization

With the right learning algorithm, we can start to fit by minimizing J(θ) as a function of θ to find optimal parameters. We have used **Gradient Descent** as the optimization algorithm.

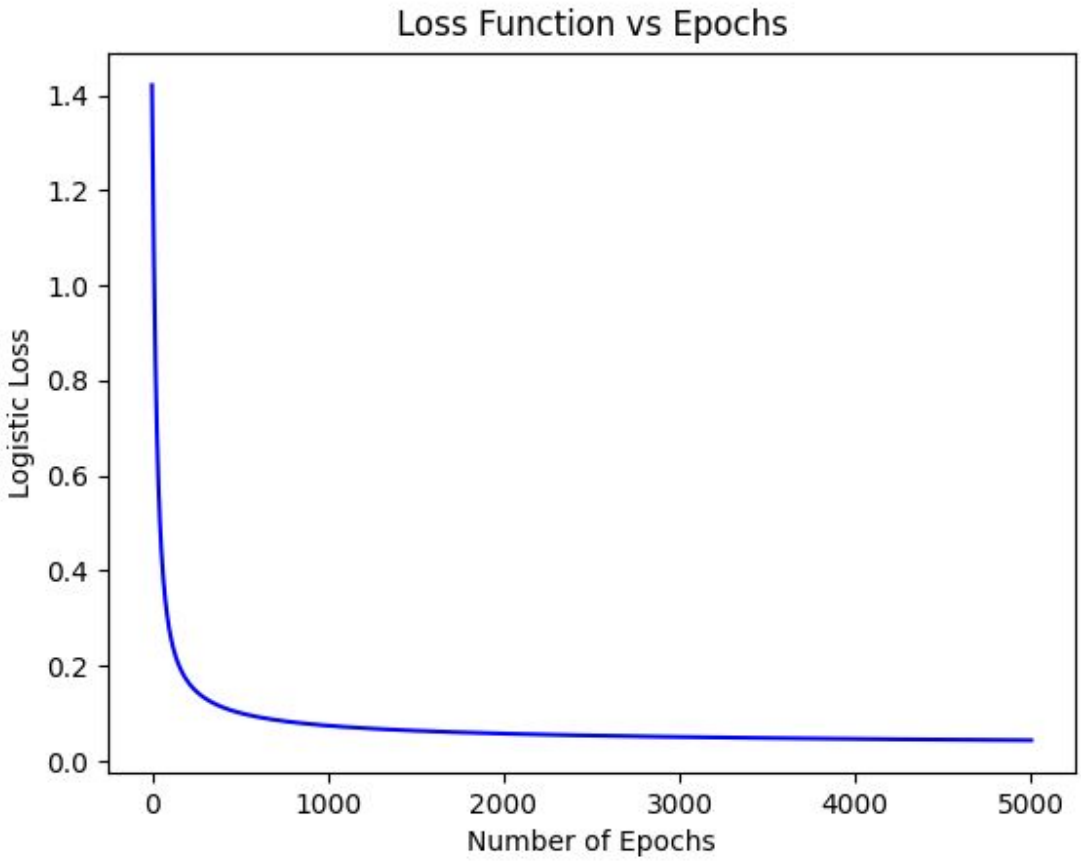
PLOTS

I) Without Regularisation

1. Model representing accuracy against number of iterations:



2. **Model representing error against iterations**



Configuration of the model:

Testing Accuracy: 98.54545454545455

F-Score: 98.34710743801654

Model Parameters :

Initialization Technique : 'gaussian'

Regularisation Used : None

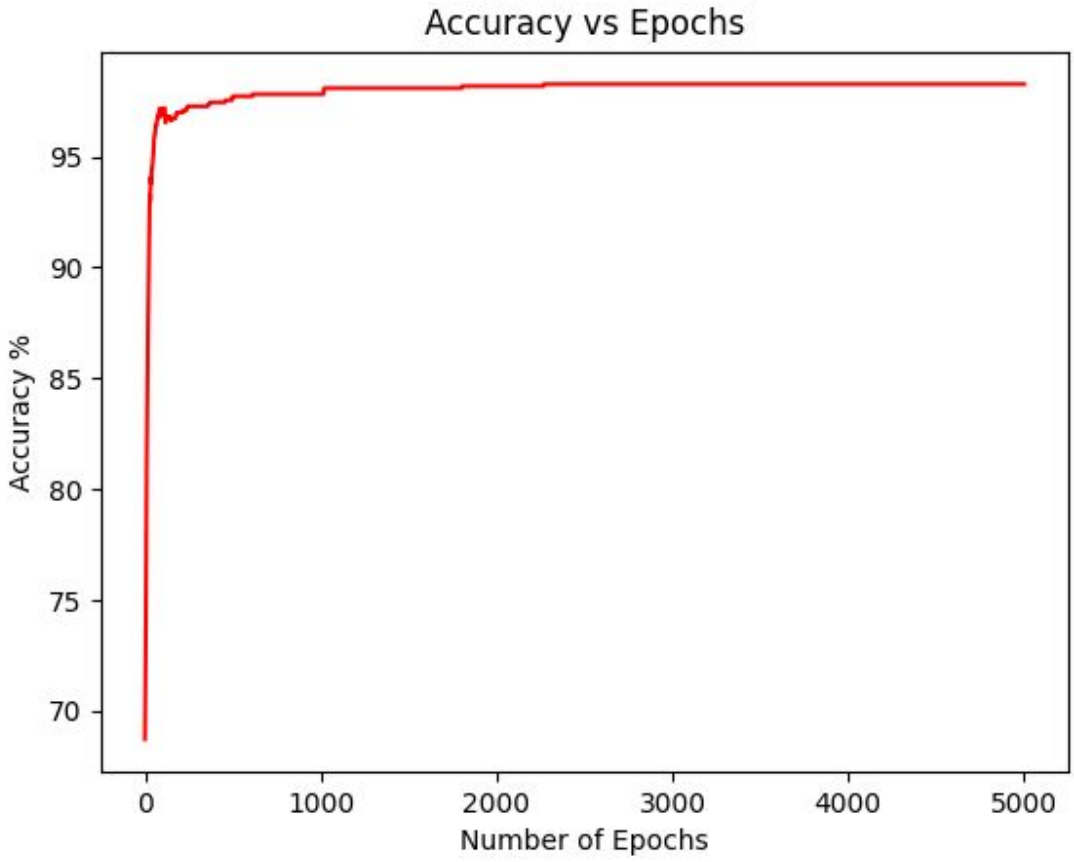
Weight (W): [[-5.22659875 -5.23590744 -4.77510834 0.25482909]]

Bias (b): -1.5526604982766123

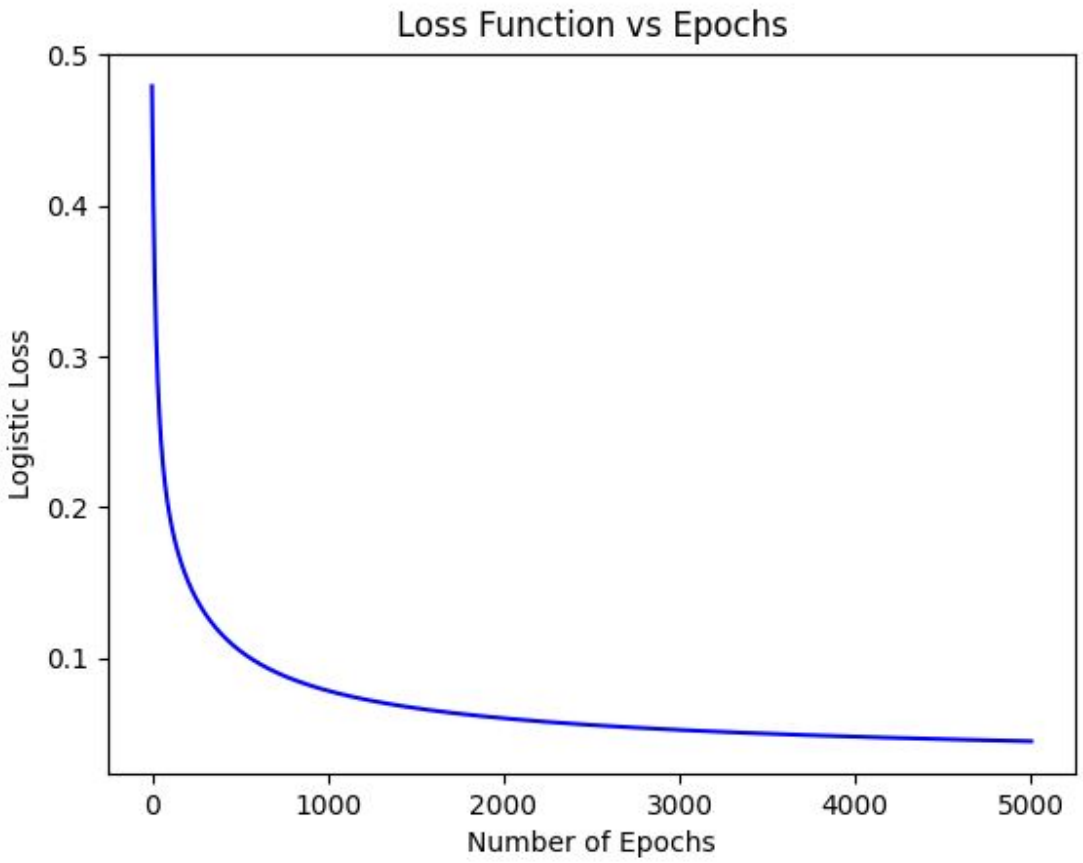
Learning rates : Alpha : 0.1, Beta : 0.1

II) **With L1 Regularisation**

1. **Model representing accuracy against number of iterations:**



2. **Model representing error against iterations**



Configuration of the model:

Testing Accuracy: 98.9090909090909

F-Score: 98.8235294117647

Model Parameters :

Initialization Technique : 'gaussian'

Regularisation Used : L1

Weight (W): [[-5.12205343 -5.3302224 -4.88148698 0.27838914]]

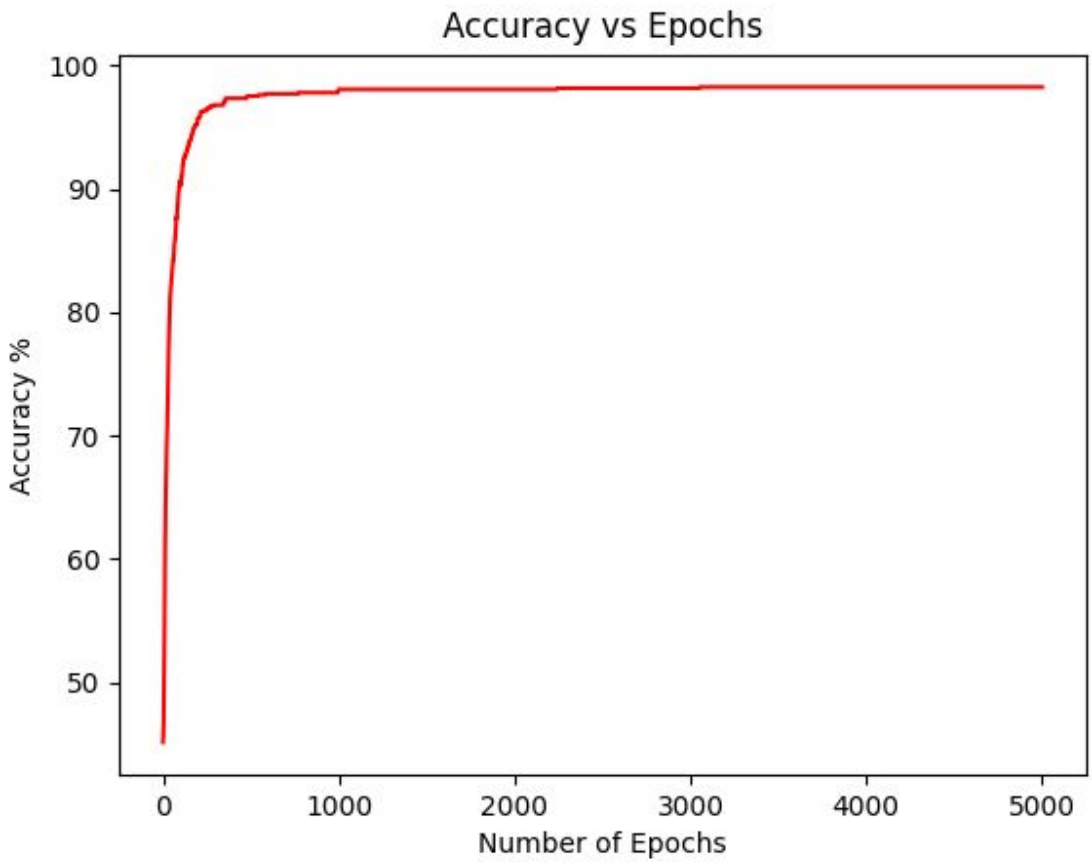
Bias (b): -1.6966924054365378

Learning rates :

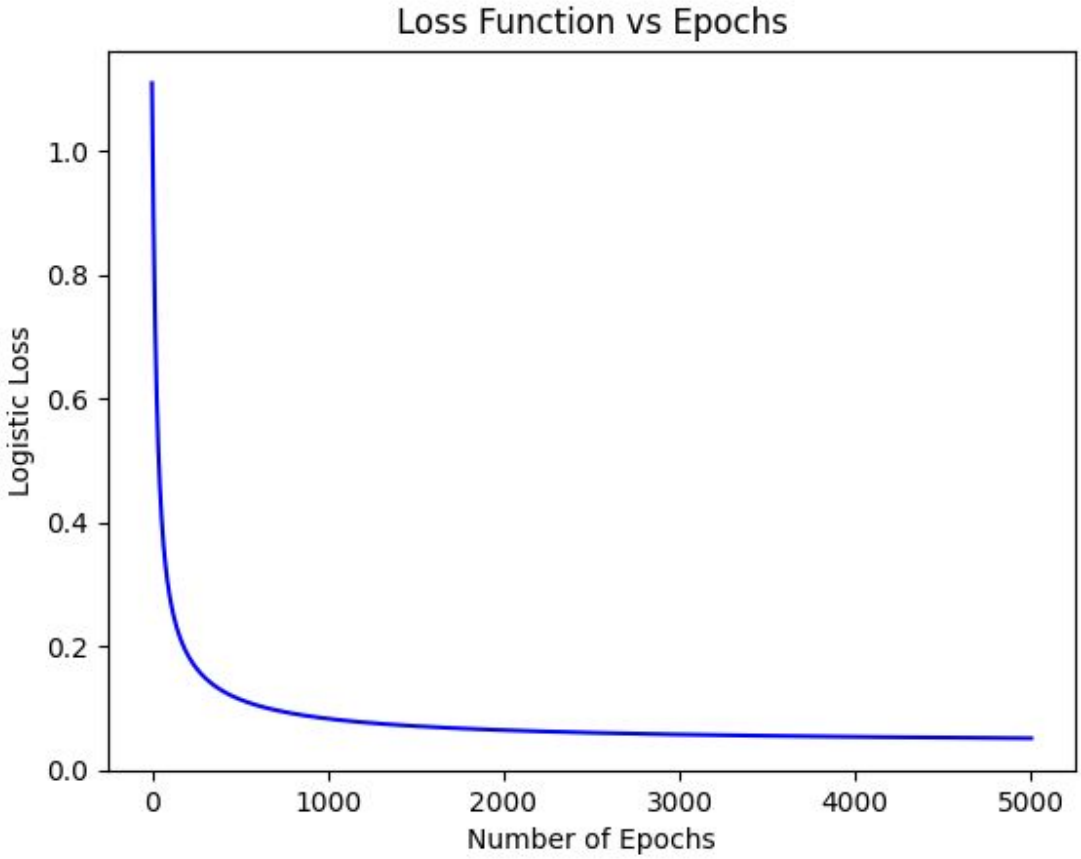
Alpha : 0.1, Beta : 0.1

III) With L2 Regularisation

1. Model representing accuracy against number of iterations:



2. **Model representing error against iterations**



Configuration of Model:

Validation Accuracy: 98.54545454545455

F-Score: 98.48484848484848

Model Parameters :

Initialization Technique : 'gaussian'

Regularisation Used : L2

Weight (W): [[-5.04923131 -5.23876169 -4.7609556 0.30740413]]

Bias (b): -1.6254509271254844

Learning rates :

Alpha : 0.1

Beta : 0.1

Initialization Techniques

We have used multiple initialisation techniques for our weights and tested on the dataset.

Configuration of Model:

Iterations = 5000

Learning Rate:

Alpha = 0.1

Beta = 0.1

Initialization Techniques	Training Accuracy	Training Error	F1-score	Testing Accuracy
Gaussian	97.9945	0.0450	0.98785	98.909
Random (0, 1)	98.0857	0.0448	0.973451	97.818
Uniform (0,1)	98.3592	0.0421	0.96969	97.454
Ones	98.1768	0.0438	0.9766	97.818

Feature Importance

From the results we can conclude that if the weights are negative, the feature of that respective weight will favour the negative class of data if increased and vice versa.
And similarly, if the weight is positive, the feature of that respective weight when increased will favour the positive class of data.

So, In conclusion,

1) The note is more likely to be real if the variance is higher. -> strong
2) The note is more likely to be fake if skewness is higher. -> strong
3) The note is more likely to be fake if the kurtosis is higher. -> weak
4) The note is more likely to be fake if the entropy is higher. -> medium

The absolute value of weights directly corresponds to how much contribution that feature has towards classification.

From our results above, we can say for sure that the last feature is of least importance while the first and the second feature of most importance followed by the third feature.

Neural Network:

Preprocessing

The dataset used for this assignment titled “House Price Data” consists of data about different types of houses and our objective is to predict whether the house price is above the median price or not.

Attribute Information:

1. LotArea
2. OverallQual
3. OverallCond
4. TotalBsmtSF
5. FullBath
6. HalfBath
7. BedroomAbvGr
8. TotRmsAbvGrd
9. Fireplaces
10. GarageArea
11. AboveMedianPrice (Target)

The target variable in the dataset is the last column (AboveMedianPrice) which indicates whether the price of the house is above the median or not.

The following steps have been used for preprocessing the dataset:

4. We have standardized the features by removing the mean and scaling to unit variance. The standard score of a sample x is calculated as:

   $$x = (x - u) / s$$

   where u is the mean of the training samples or zero and is the standard deviation of the training samples or one
5. The training and test split of the dataset is 70% and 30% respectively.

6. Various hyperparameters have been experimented with and tuned accordingly.

Loss Function

Binary classification is a common machine learning task. It involves predicting whether a given example is part of one class or the other. Here **we interpret the output of the neural network as the probability** instead of raw value.
Note: We have used the classification threshold for the model as **0.5**.

$$Loss = L(y, \hat{p}) = -y \log(\hat{p}) - (1 - y) \log(1 - \hat{p})$$

where $y \Rightarrow$ label,
$\hat{p} \Rightarrow$ estimated probability of belonging to the positive class

The derivative of the Loss function wrt to predicted value *p* is :

$$\begin{aligned}
\frac{\partial L(y, \hat{p})}{\partial \hat{p}} &= \frac{\partial}{\partial \hat{p}}(-y \log(\hat{p}) - (1 - y) \log(1 - \hat{p})) \\
&= \frac{\partial}{\partial \hat{p}}(-y \log(\hat{p})) - \frac{\partial}{\partial \hat{p}}((1 - y) \log(1 - \hat{p})) \\
&= \frac{-y}{\hat{p}} - \left(\frac{1 - y}{1 - \hat{p}} * -1\right) \\
&= \frac{-y}{\hat{p}} + \frac{1 - y}{1 - \hat{p}}
\end{aligned}$$

The derivative of the Binary Cross Entropy Cost Function

$$Cost(\mathbf{Y}, \hat{\mathbf{P}}) = \frac{1}{m} \sum -\mathbf{Y} \odot \log(\hat{\mathbf{P}}) - (1 - \mathbf{Y}) \odot \log(1 - \hat{\mathbf{P}})$$

Since the derivative above involves division of a vector (consisting of 0s and 1s), it may happen that the division occurs between two 0s that results in **inf** in numpy and python interprets it as **nan(Not a number)** which will give erroneous results.

After doing some manipulations, our loss function can be adjusted as :

*In piecewise form our **stable Binary Cross-Entropy Loss function** look as follows:*

$$Loss(y, z) = \begin{cases} z - zy + \log(1 + e^{-z}) & \text{if } z \geqslant 0 \\ -zy + \log(e^{z} + 1) & \text{if } z < 0 \end{cases}$$

We can elegantly combine this into one equation as follows:

$$Loss(y, z) = max(z, 0) - zy + \log(1 + e^{-|z|})$$

Where $|z|$ means absolute of z also called mod in mathematics.

*Similarly, the Binary Cross-Entropy **Cost** function becomes:*

$$Cost(Y, Z) = \frac{1}{m} \sum max(Z, 0) - ZY + \log(1 + e^{-|Z|})$$

***Note**: max here is element-wise maximum *compared to zero*(in NumPy this is denoted as $np.maximum$)*

And the updated for the nth layer derivative as :

$$
\begin{aligned}
\frac{\partial Loss}{\partial z} &= UpstreamGradient * LocalGradient \\
&= \frac{\partial Loss}{\partial \hat{p}} * \frac{\partial \hat{p}}{\partial z} \\
&= \left(\frac{-y}{\hat{p}} + \frac{1-y}{1-\hat{p}}\right) * (\hat{p}(1-\hat{p})) \\
&= \left(\frac{-y}{\hat{p}} * \hat{p}(1-\hat{p})\right) + \left(\frac{1-y}{1-\hat{p}} * \hat{p}(1-\hat{p})\right) \\
&= -y(1-\hat{p}) + (1-y)\hat{p} \\
&= -y + y\hat{p} + \hat{p} - y\hat{p} \\
&= -y + \hat{p} \\
&= \mathbf{\hat{p} - y}
\end{aligned}
$$

We can use this ***stable*** *binary cross entropy loss* function with the above derivative directly passed to the (n-1)th layer to avoid getting **NANs** while making our implementation vectorised.
Now our Neural Network implementation is stable.

PLOTS

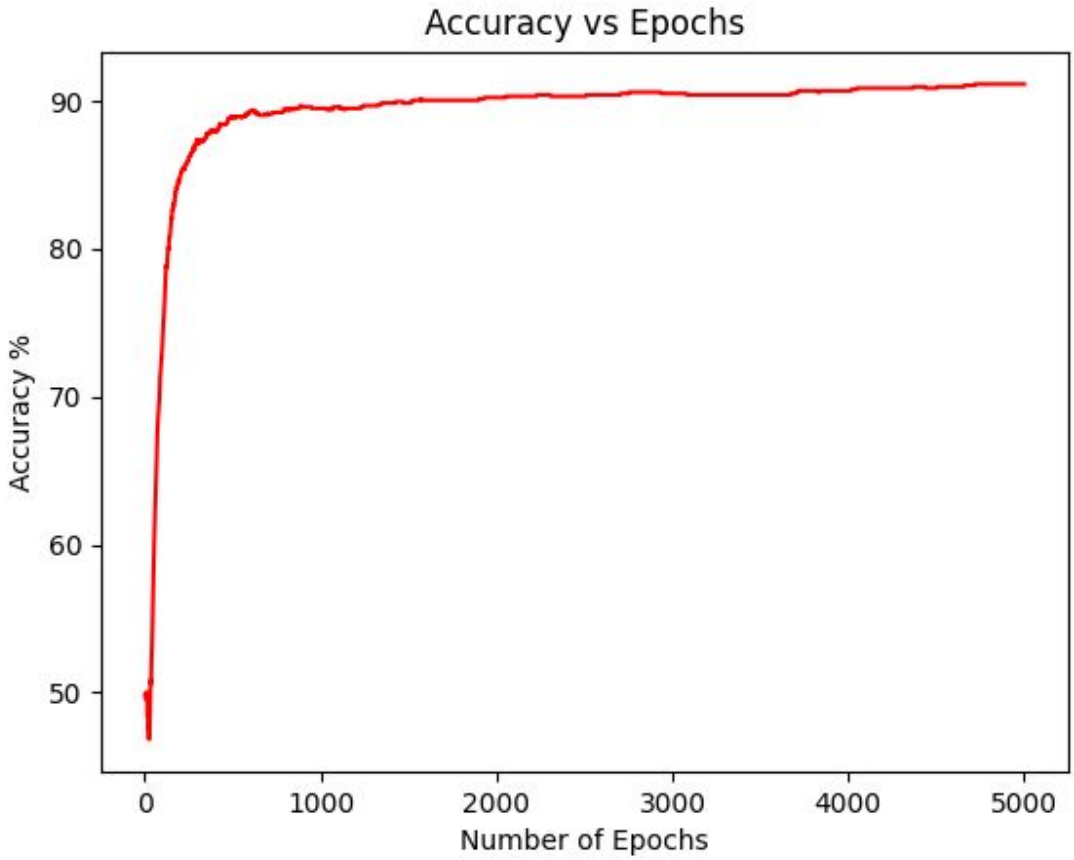
1. Neural Net Architecture:

   Layer dims = [10 ---> 5 ---> 5 ---> 1]

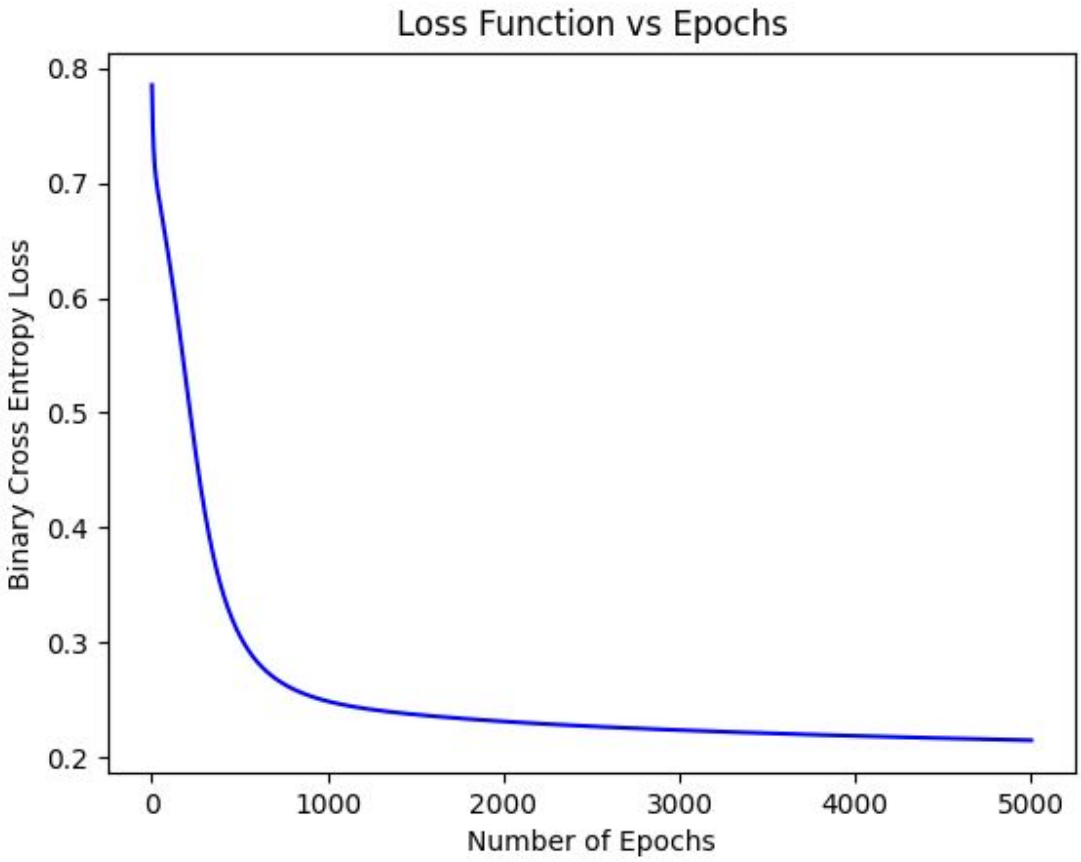
   Activation = Sigmoid (in all layers)

   Initialization = Uniform

i. Model representing accuracy against number of iterations:



ii. Model representing error against iterations



Results:

Accuracy : 90.95890410958904

F1-Score : 0.907563025210084

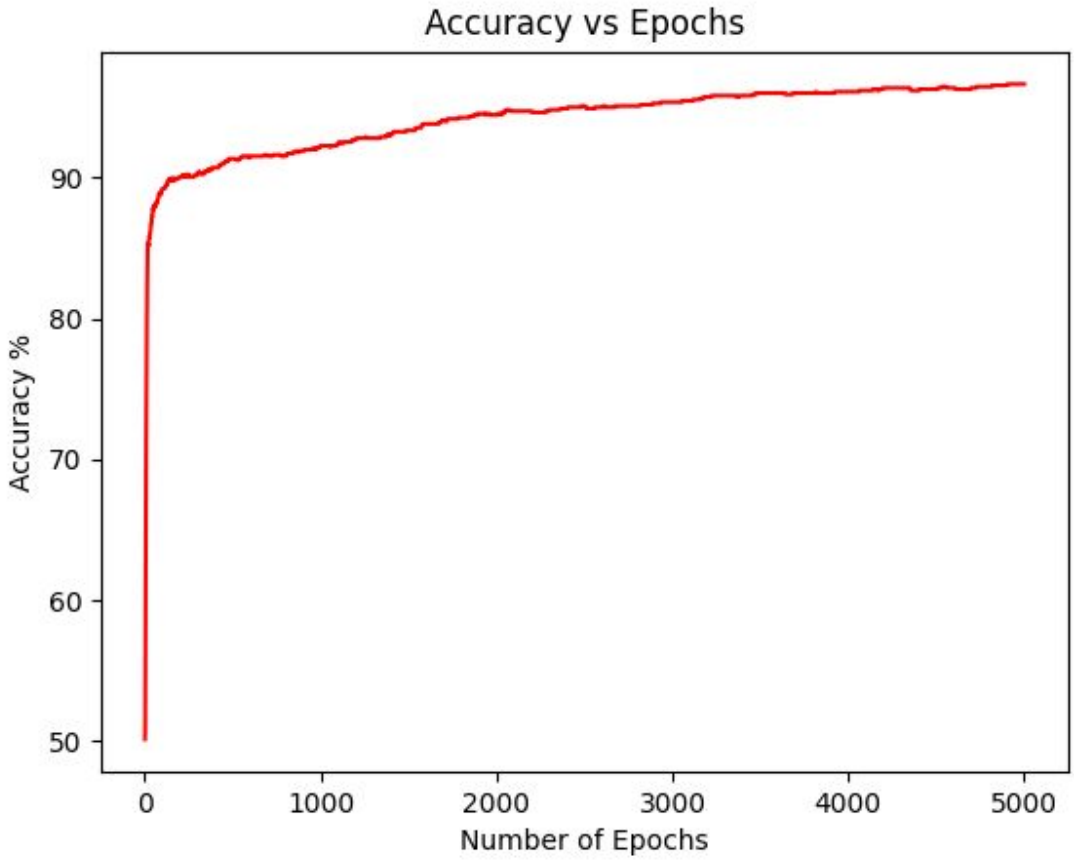
2. Neural Net Architecture:

Layer dims = [10 ---> 25 ---> 15 ---> 5 ---> 1]

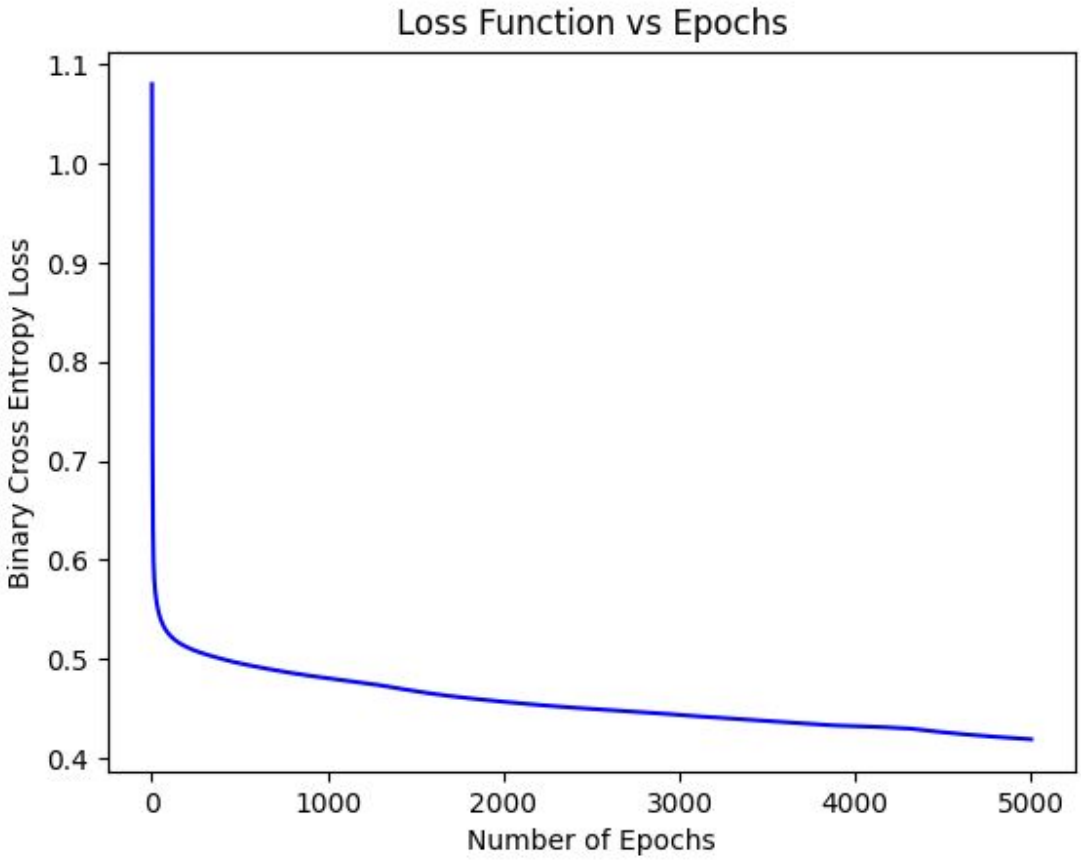
Activation = Tanh (in all layers)

Initialization = Uniform

i. Model representing accuracy against number of iterations:



ii. Model representing error against iterations



Results:

Accuracy : 89.04109589041096

F1-Score : 0.8901098901098902

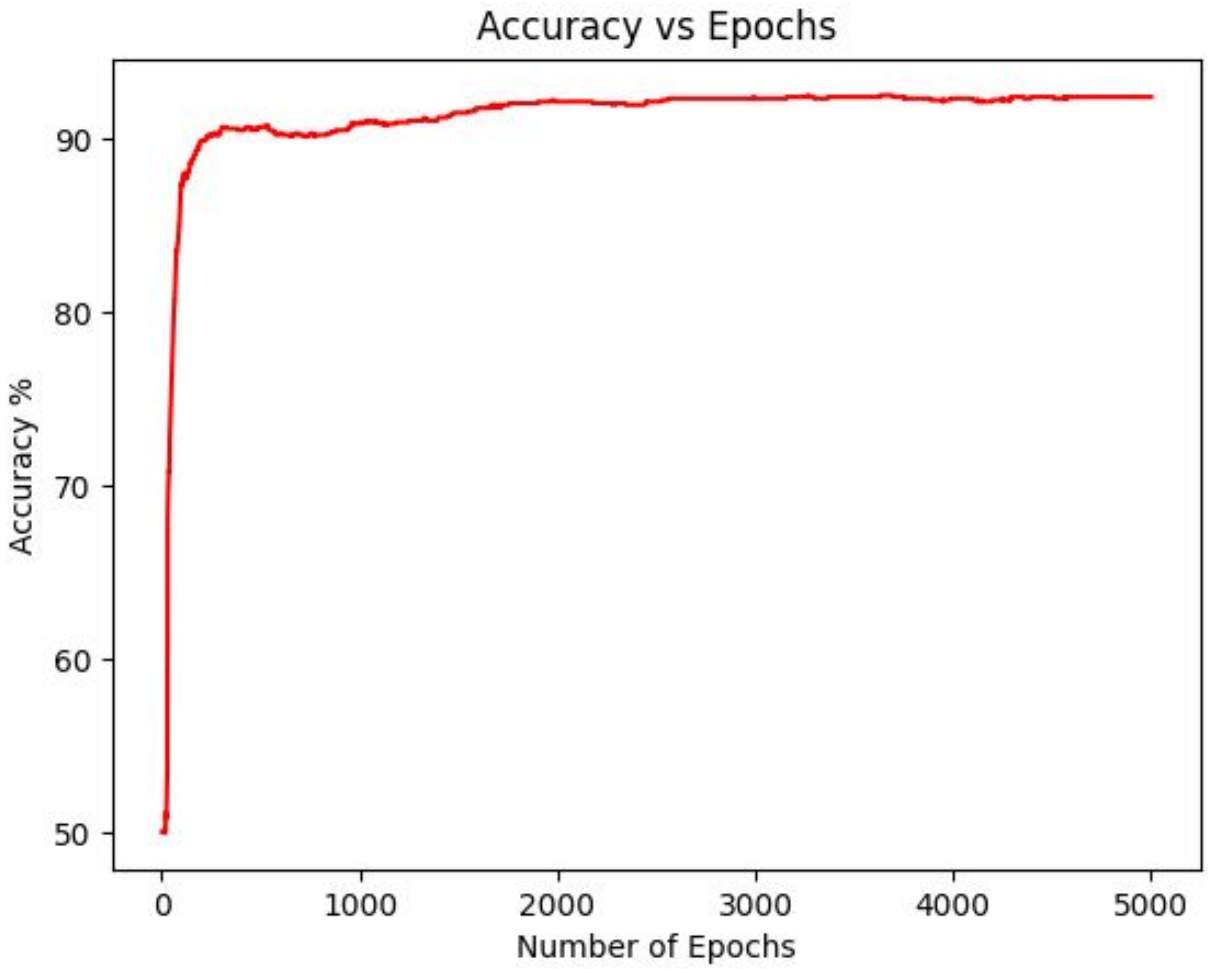
3. Neural Net Architecture:

Layer dims = [10 ---> 5 ---> 5 ---> 1]

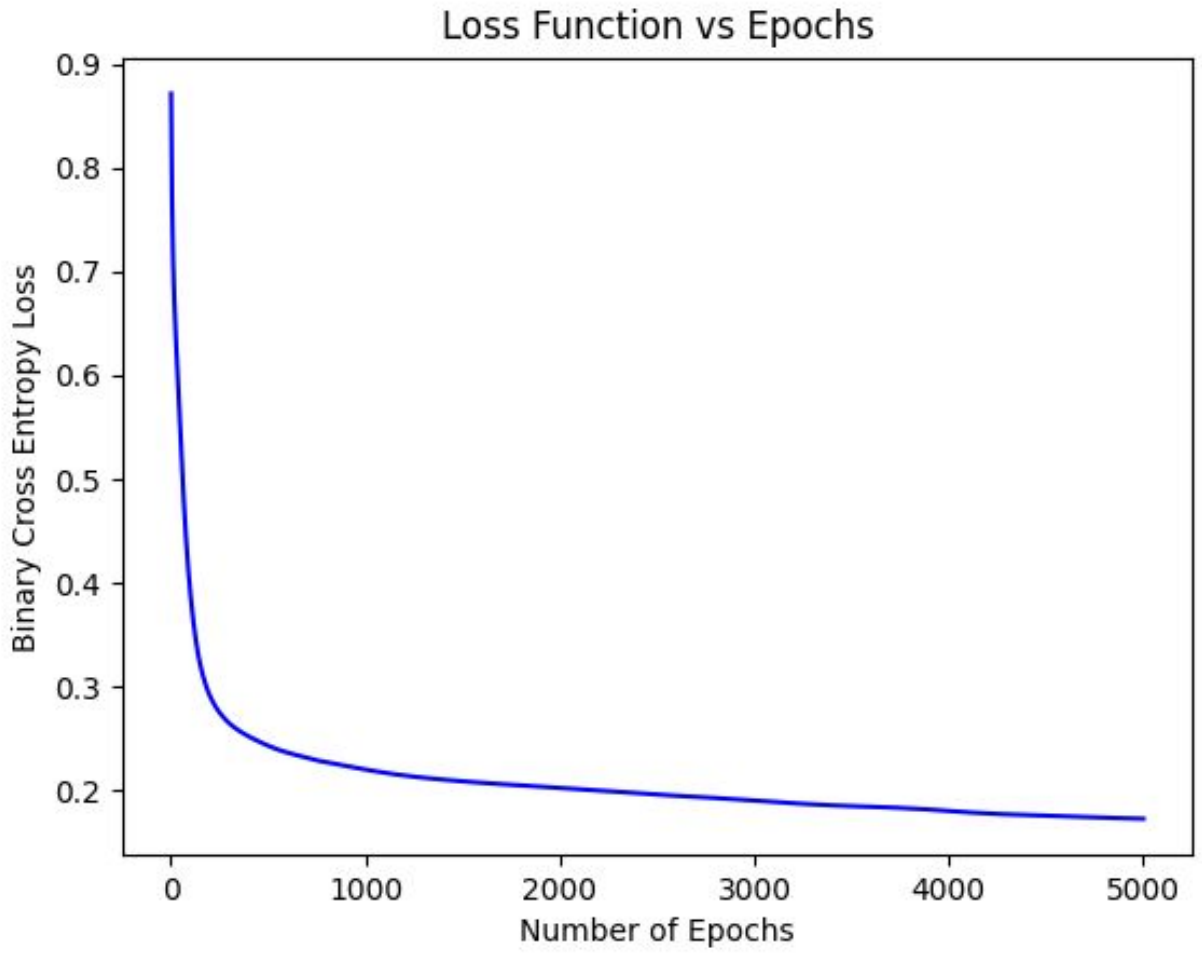
Activation = [Tanh ---> Relu ----> Sigmoid]

Initialization = Uniform

i. Model representing accuracy against number of iterations:



ii. Model representing error against iterations



Results:

Accuracy : 89.86301369863014

F1-Score : 0.8963585434173669

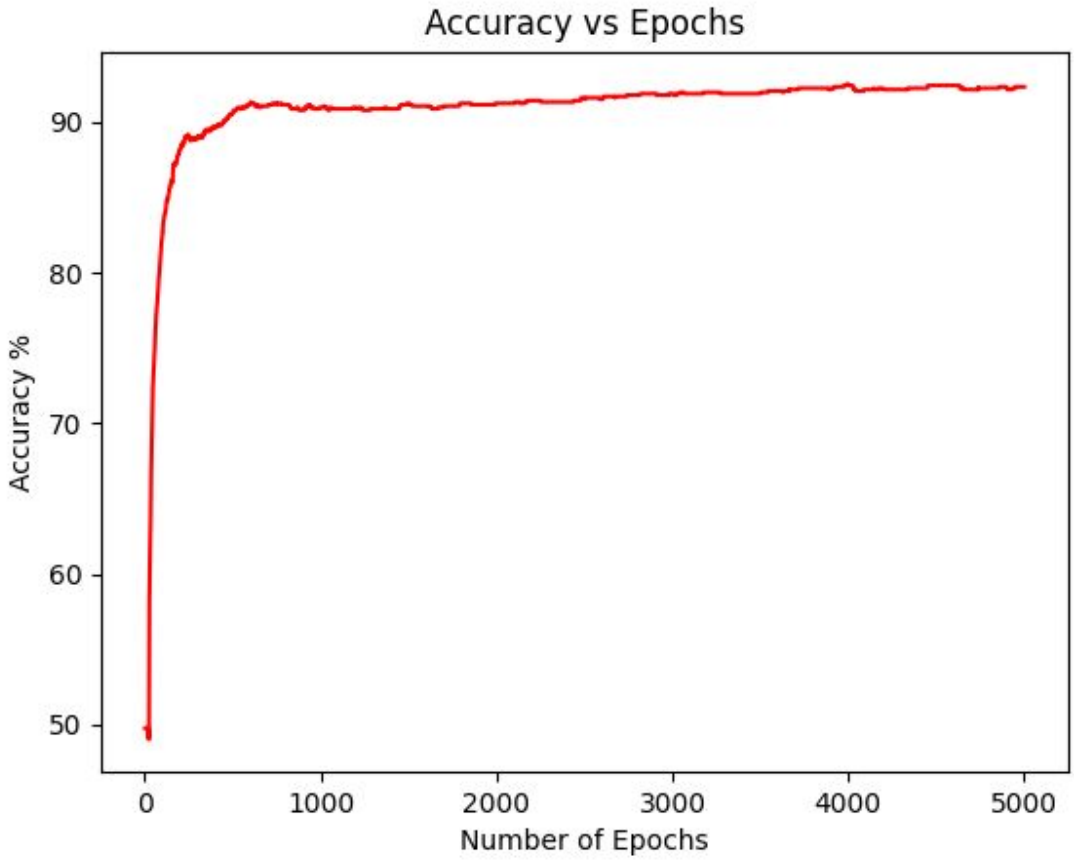
4. Neural Net Architecture:

Layer dims = [10 ---> 5 ---> 5 ---> 1]

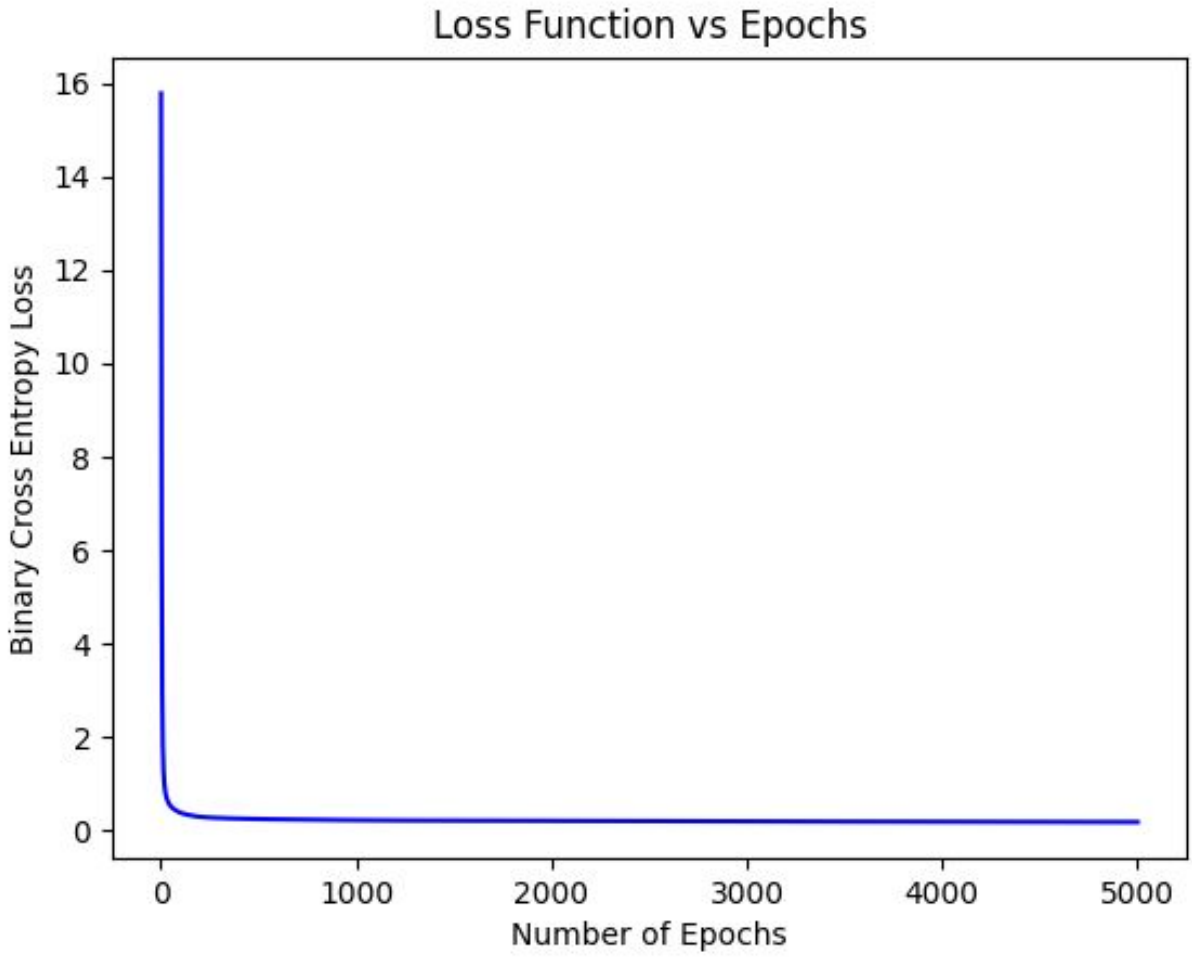
Activation = [Tanh ---> Relu ----> Sigmoid]

Initialization = Gaussian

i. Model representing accuracy against number of iterations:



ii. Model representing error against iterations



Results:

Accuracy : 89.04109589041096

F1-Score : 0.8876404494382022

Note: With *gaussian* initialization, we reach the accuracy faster than *uniform* initialization.