Prediction of Nephro Syndrome

Deepika

ACKNOWLEDGEMENTS

We would like to thank our professor Dr. Md Farhadur Reza for guiding us and encouraging us with valuable feedback.

In addition, we also offer our sincere thanks to UCI for documenting data and metadata on this dataset.

Chronic_Kidney_Disease Data Set

https://archive.ics.uci.edu/ml/datasets/Chronic_Kidney_Disease

INTRODUCTION

Nephro Syndrome is one of the most important issues of public health with a growing need for early detection for successful and sustainable care. Many factors contribute to the gradual loss of kidney function over time, such as blood pressure, diabetes, and other disorders.

Machine learning and data exploration can be used for identifying early-stage CKD with the help of medical history and diagnostic information.

Significance of Problem

Nephro Syndrome refers to a gradual loss of kidney function which helps in filtering wastes from our blood. It should be noted that any problem identified during the early stages can be treated efficiently. It is said that unless that kidney function is prominently disrupted, we may not be able to diagnose the disease.

Treatment involves slowing the damage progression probably by keeping in control the main cause. Besides, end-stage renal failure could be averted using dialysis or a transplant.

Objective

The main objective of our project is to predict and classify whether the patient has Nephro syndrome or not using Artificial Neural Networks.

We will make use of dataset to train our ANN model to achieve better accuracy in prediction.

Dataset Overview

The dataset used in this report is taken from the UCI- Machine Learning Repository. It consists of 400 instances and 25 attributes out of which 24 are independent attributes and 1 dependent attribute(class) which will predict whether the patient has CKD or not. We have noted that attributes are either nominal or numerical.

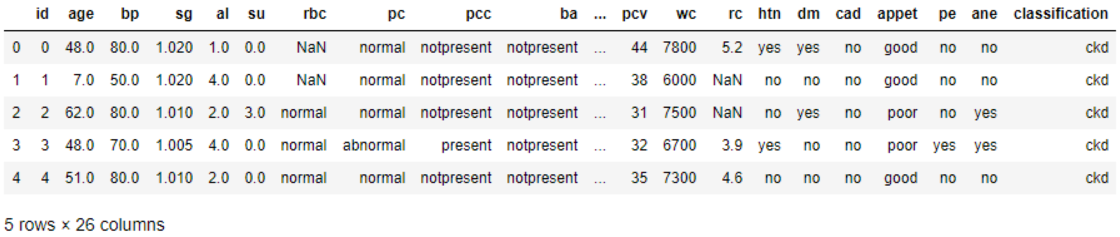
Attribute Information

age	Age
bp	Blood pressure
Sg	Specific gravity
Al	Albumin
Su	Sugar
rbc	Red Blood Cells
pc	Pus cell
pcc	Pus cell clumps
Ba	Bacteria
bgr	blood glucose random
bu	blood urea
Sc	Serum Creatinine
sod	Sodium

pot	Potassium
hemo	Hemoglobin
pcv	packed cell volume
wc	white blood cell count
Rc	red blood cell count
htn	Hypertension
dm	diabetes mellitus
cad	coronary artery disease
appet	Appetite
pe	pedal edema
ane	Anemia
class	Classification

Data Load

For our classification neural network model, we will first load data using Pandas where 'df' stands for data frame. Pandas reads in the csv file as a data frame. The 'head ()' function will show the first 5 rows of the data frame so we can check that the data has been read in properly and can take an initial look at how the data is structured.



	id	age	bp	sg	al	su	rbc	pc	pcc	ba	...	pcv	wc	rc	htn	dm	cad	appet	pe	ane	classification
0	0	48.0	80.0	1.020	1.0	0.0	NaN	normal	notpresent	notpresent	...	44	7800	5.2	yes	yes	no	good	no	no	ckd
1	1	7.0	50.0	1.020	4.0	0.0	NaN	normal	notpresent	notpresent	...	38	6000	NaN	no	no	no	good	no	no	ckd
2	2	62.0	80.0	1.010	2.0	3.0	normal	normal	notpresent	notpresent	...	31	7500	NaN	no	yes	no	poor	no	yes	ckd
3	3	48.0	70.0	1.005	4.0	0.0	normal	abnormal	present	notpresent	...	32	6700	3.9	yes	no	no	poor	yes	yes	ckd
4	4	51.0	80.0	1.010	2.0	0.0	normal	normal	notpresent	notpresent	...	35	7300	4.6	no	no	no	good	no	no	ckd

5 rows × 26 columns

Dataset Split -Input Vs Target

Next, we need to split up our dataset into inputs and our target. Our input will be every column except 'class' because we are trying to predict based on our attributes. We will use pandas ‘drop’ function to drop the column from our data frame and store it in the variable.

```
df.drop(["classification"], axis=1).plot.line(title='Nephro Syndrome Dataset')
```

Data Visualization

Visualization of data is the discipline of trying to understand data by placing it in a visual context in order to expose patterns, trends and correlations that might otherwise not be detected.

Python offers several great libraries of graphing that come packed with many different features. Few of the popular plotting libraries are as follows:

Matplotlib: low level, provides lots of freedom

Pandas Visualization: easy to use interface, built on Matplotlib

Seaborn: high-level interface, great default styles

Plotly: can create interactive plots

Data Transformation

We identified problem on df['dm'] with string having extra space. This had to be fixed as it will result in mismatch issue between X_train and X_test.

```
df['dm']. unique ()
```

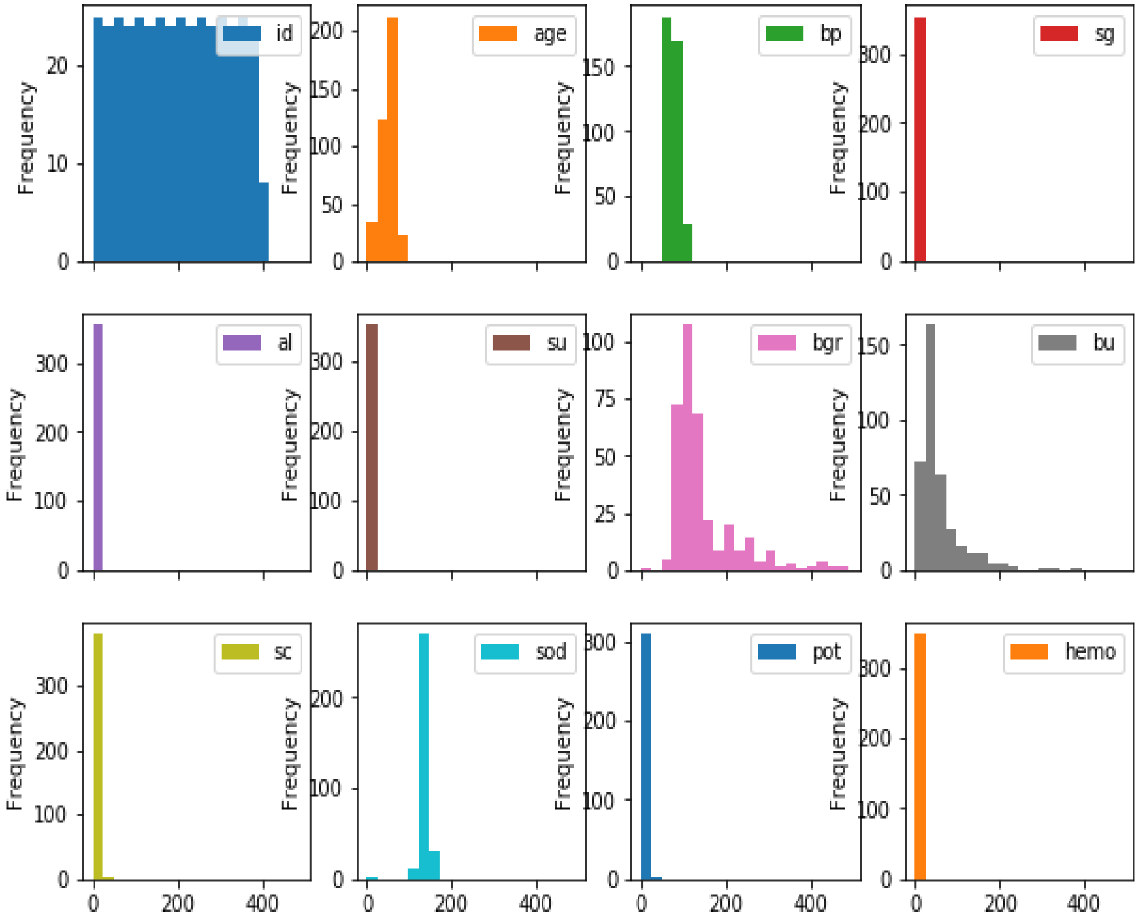
```
array(['yes', 'no', ' yes', '\tno', '\tyes', nan], dtype=object)
```

In addition, we also transform non-numeric columns into numerical columns using fit_transform function

Data Analysis

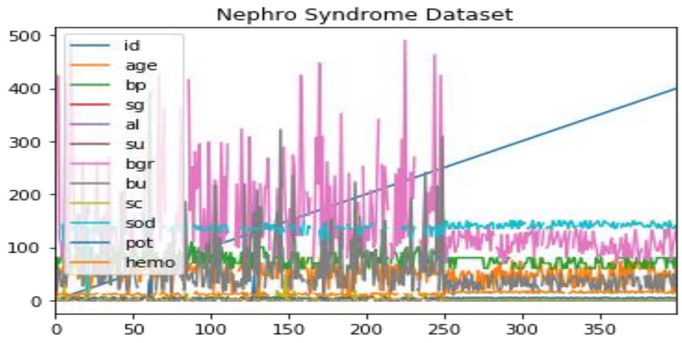
Histogram

In Pandas, we can use the plot.hist method to create a Histogram. There are no arguments required, but we can pass some such as bin size as an option.



Line Chart

We can call < dataframe>.plot.line to create a line-chart in Pandas.



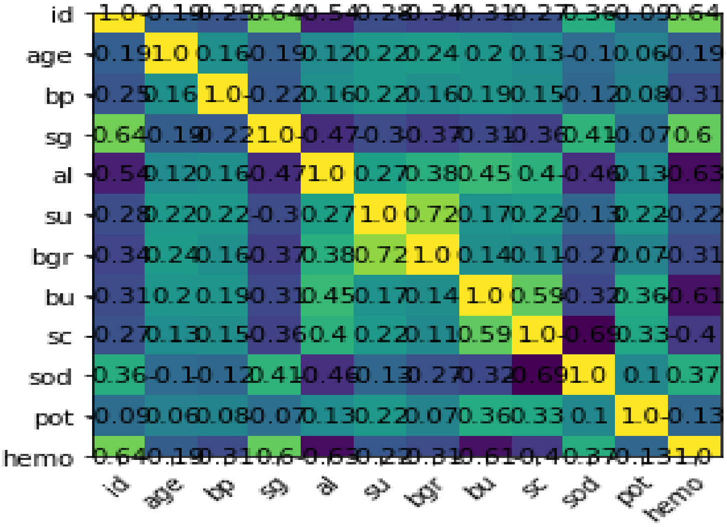
Categorical Attributes along with unique values present in the dataset

rbc	2
pc	2
pcc	2
ba	2
htn	2
dm	5
cad	3
appet	2
pe	2
ane	2

Correlation Matrix

A correlation matrix is a table showing coefficients of correlation between variables. The relationship between two variables is shown by each cell in the table.

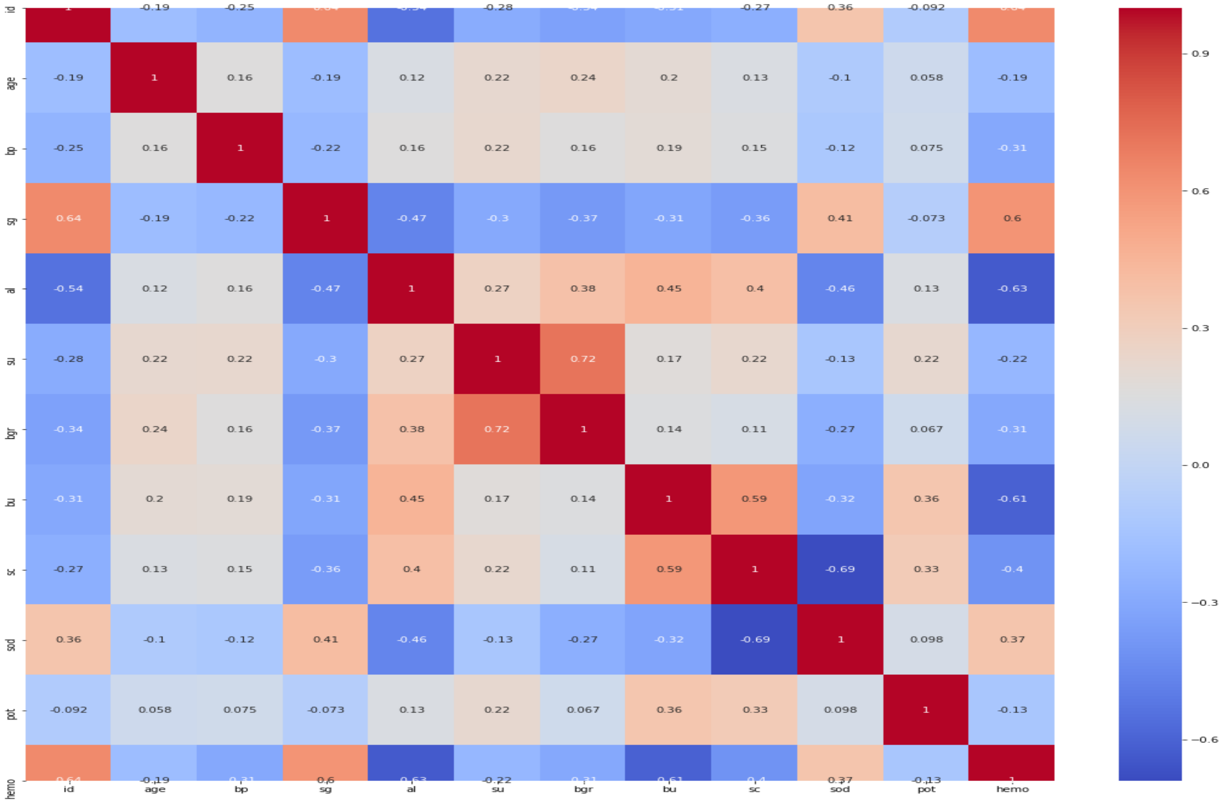
A matrix of correlation is used to analyze results, as an input for a more detailed analysis and as a diagnostic for advanced analysis.



Heat Map

A Heatmap is a graphical representation of data in which the individual values in a matrix are depicted as colors. Heatmaps are perfect to explore a data set's correlation of features.

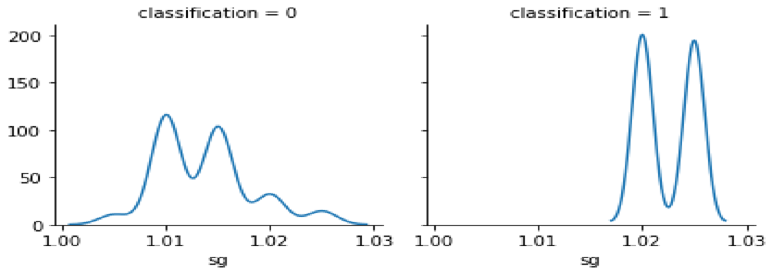
We may call < dataset>.corr, which is a Pandas dataframe tool, to get the comparison of the features within a dataset. This will show us the matrix of correlation.

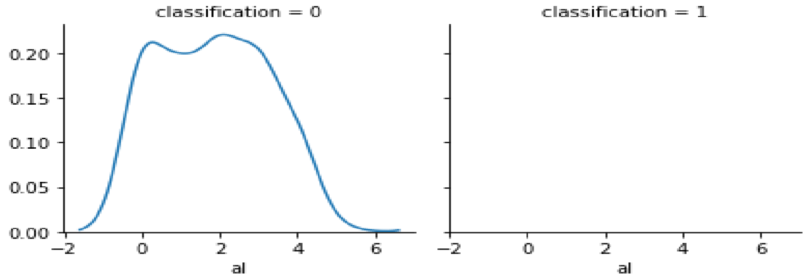


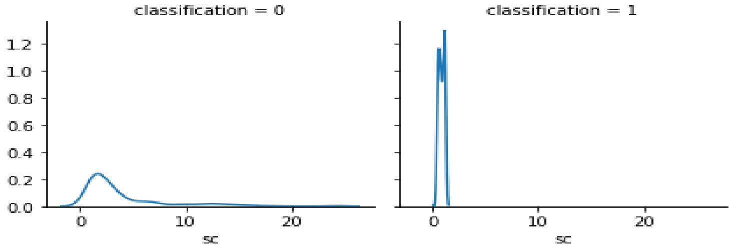
Faceting

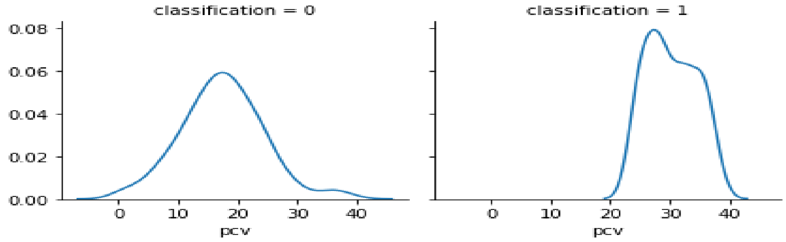
Faceting is the act of breaking up data variables across multiple subplots and combining these subplots into a single figure. Faceting is very helpful if you want to explore your dataset quickly.

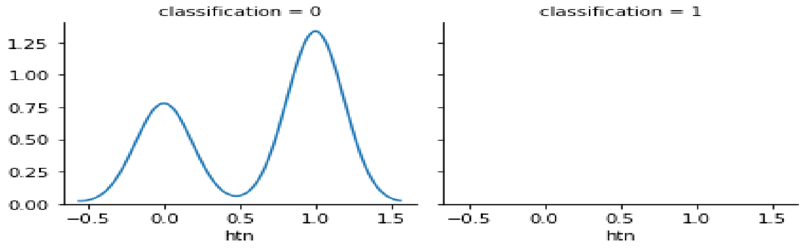
We can use the Facet Grid to use one form of faceting in Seaborn. First, we need to identify the Facet Grid and move it on to our data as well as a row or column to divide the data. Then on our Facet Grid object we need to call the map function and specify the form of plot we want to use and the column we want to chart.


classification = 0
classification = 1
200
150
100
50
0
1.00
1.01
1.02
1.03
1.00
1.01
1.02
1.03
sg
sg

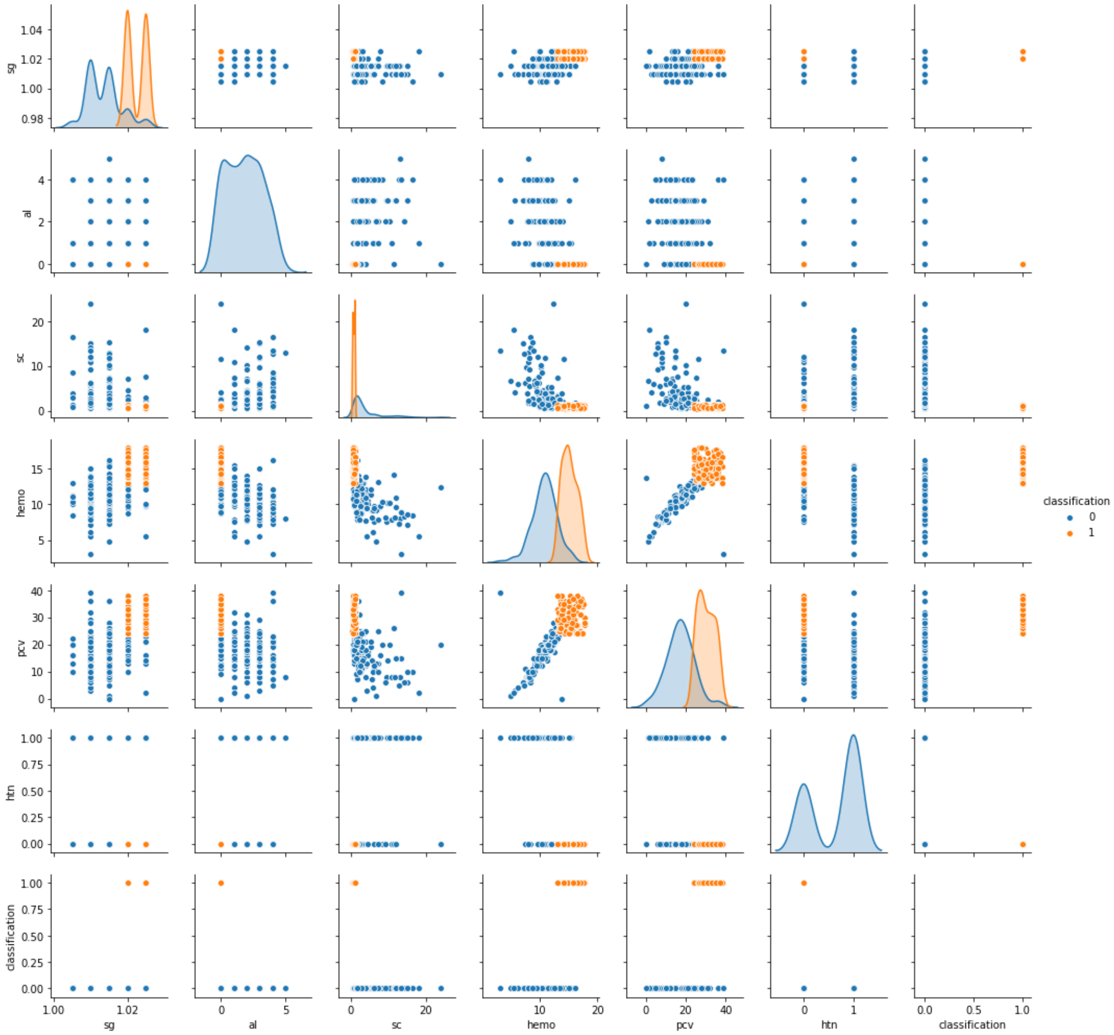

classification = 0
classification = 1
0.20
0.15
0.10
0.05
0.00
−2
0
2
4
6
−2
0
2
4
6
al
al


classification = 0
classification = 1
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0
10
20
0
10
20
sc
sc


classification = 0
classification = 1
0.08
0.06
0.04
0.02
0.00
0
10
20
30
40
0
10
20
30
40
pcv
pcv


classification = 0
classification = 1
1.25
1.00
0.75
0.50
0.25
0.00
−0.5
0.0
0.5
1.0
1.5
−0.5
0.0
0.5
1.0
1.5
htn
htn

The next step is to capture the relationship between the features and the classes. In our data, we have more than two characteristics (multivariate data) involving relationships between them. For each of two elements, we will use bivariate visualizations to capture all their relationships.



Analyzing Individual Attributes

Lets explore how the values are distributed in one of the attributes say ‘pcv’

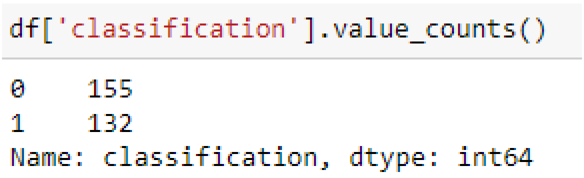
```
df['pcv'].value_counts()
```

```
36    19
```

25	19
28	19
32	17
24	16
29	11
18	11
20	11
26	11
17	11
27	10
34	10
16	10
21	10
30	8
13	8
12	8
15	8
14	7
23	7
10	6
19	6
8	5
38	4
31	4
33	4
35	4
22	3
11	3
37	3

```
6    3
9    2
7    2
3    1
1    1
2    1
39    1
4    1
5    1
0    1
Name: pcv, dtype: int64
```

Classification of Dataset-Overview



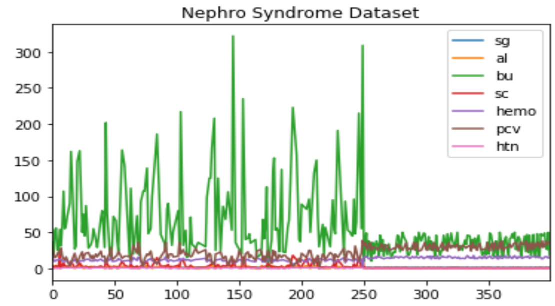
```
df['classification'].value_counts()
```

```
0    155
1    132
Name: classification, dtype: int64
```

Feature Selection

Based on heat map and correlation matrix we decided that below attributes have high predictive power for classification.

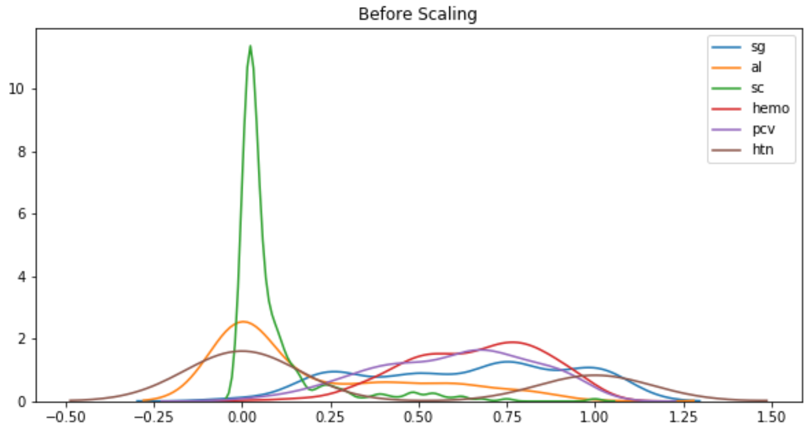
"sg", "al", "sc", "hemo","bu", "pcv", "wbcc", "rbcc", "htn", "classification"



Feature Scaling

We can use a plot to see what scaling of features did to our values of data. This rescaled the features in such a way that they have the properties of a normal standard distribution with a zero mean and one standard deviation.

min-max scaler method scales the dataset so that all the input features lie between 0 and 1 inclusive



Frameworks and Libraries

TensorFlow: End to end open source platform for machine learning

Keras: High-level neural networks library which is running on the top of TensorFlow

Scikit-Learn: Efficient for data mining and data analysis

Matplotlib: Plotting library for graphical representations

Numpy: General purpose array processing package

Pandas: Data manipulation and analysis package

Keras: Deep Learning library for Theano and TensorFlow

Keras is a Python-written high-level neural network API that can run on top of either TensorFlow or Theano. It was designed to allow rapid experimentation with a focus. It is necessary for good research to be able to go from idea to outcome with the least possible delay.

Advantages of Keras

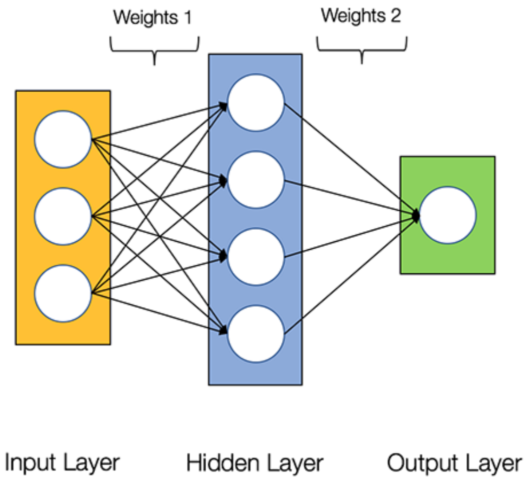
- Allows easy and quick prototyping (user-friendly, modular and extensible).
- Supports both convolutionary and recurrent networks and combinations of both.
- Easy operation on the CPU and GPU.

Algorithm - Artificial Neural Network

Artificial Neural Network is an information processing technique. It functions like the way information is processed by the human brain. ANN includes a large number of related processing units working together and produce meaningful outcomes from it.
It can be used for both for classification and regression of continuous numeric attribute.
Mathematical function that maps a given input to a desired output. It consists of the following components:

- Input layer, x - Raw information that can feed into the network.
- Hidden layers - To determine each hidden unit's activity. The input unit activities and the weights on the input connections to the hidden units. One or more secret layers can exist.

- Output layer, ŷ - The output unit behavior depends on the hidden unit activity and the weights between the hidden units and the output units.
- Weights and Biases between each layer
- Activation function(σ) for each hidden layer



The diagram above shows the architecture of a 2-layer Neural Network

Training Neural Network Model

The output $\hat{y}$ of a simple 2-layer Neural Network is:

$$\hat{y} = \sigma(W_2\sigma(W_1x + b_1) + b_2)$$

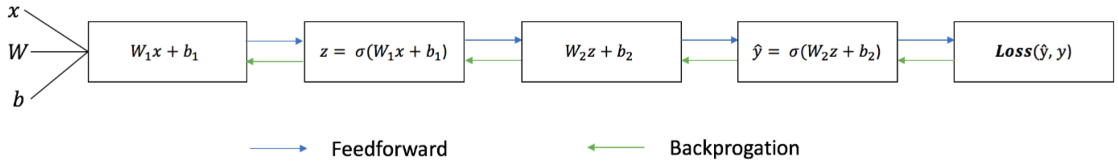
We can notice that the weights W and biases b are the only variables affecting the output in the above equation.

The correct weight and bias values, of course, determine the strength of the predictions. The process of fine-tuning the input data weights and biases is known as the Neural Network training.

Each training process iteration consists of the following steps:

- Calculation of the predicted output, known as feedforward
- Updating weights and biases, known as backpropagation

The sequential graph below illustrates the process.



Feedforward

As we saw in the sequential graph above, feedforward is a simple calculation and the performance of the Neural Network is for a typical2-layer neural network:

$$\hat{y} = \sigma(W_2\sigma(W_1x + b_1) + b_2)$$

Nonetheless, we still need a way to evaluate our predictions which can be obtained using the Loss function.

Loss Function

The function used to evaluate a candidate solution (i.e. a set of weights) is called the objective function in the context of an optimization algorithm.

We may either try to maximize or minimize the objective function, which means we are looking for a candidate solution with the highest or lowest score respectively.

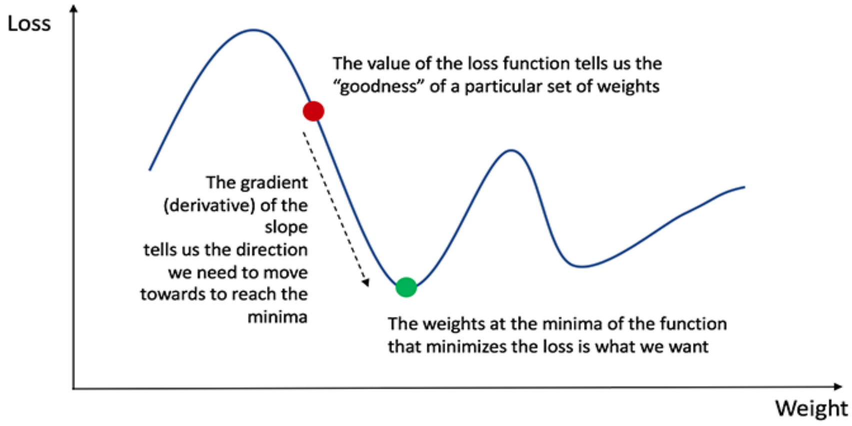
Typically, we try to minimize the error with neural networks.

We need to find the optimal set of weights and biases which will minimize the loss function.

Backpropagation

Now that we have calculated our prediction error (loss), we need to find a way to propagate the error and update our weights and biases.

We need to know the derivative of the loss function in terms of weights and biases in order to know the appropriate amount to change the weights and biases by.


Loss
The value of the loss function tells us the "goodness" of a particular set of weights
The gradient (derivative) of the slope tells us the direction we need to move towards to reach the minima
The weights at the minima of the function that minimizes the loss is what we want
Weight

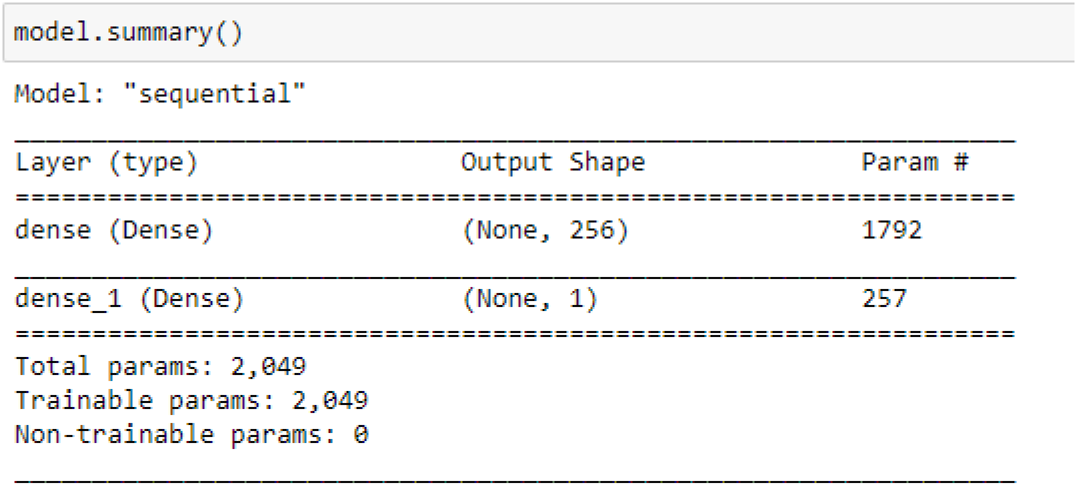
Sequential Model

Keras ' core data structure is a model, a way for layers to be organized. The simplest type of model is a linear stack of layers, the Sequential model. We can use Keras functional API for more complex architectures, which enables arbitrary layer graphs to be built.

The Sequential model is a linear stack of layers.

2 layers:

- The first with 256 neurons and the function of ReLu activation & an initializer that determines the way to set the Keras layers ' initial random weights. We are going to use an initializer that generates a normal distribution of tensors.
- The other layer will have 1 neuron with the' hard sigmoid' activation function.



```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 256)               1792
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 257
=================================================================
Total params: 2,049
Trainable params: 2,049
Non-trainable params: 0
_________________________________________________________________
```

We use the ‘add()’ function to add layers to our model. We will add two layers and an output layer. The last layer is the output layer. It only has one node, which is for our prediction.

Dense Neural Network

A dense layer in a neural network is just a regular layer of neurons. That neuron in the previous layer receives input from all the neurons, thus being densely connected. The layer has a matrix of weight, a bias vector, and previous activations layer.

Specifying the input shape¶

The model needs to know what the form of the input should be. For this reason, the first layer in a sequential model needs to receive information about the shape. Subsequent layers can infer automatic shape from previous layers.

Some 2D layers, such as Dense, support the specification of their input shape via the argument input_dim

input_dim=len(X.columns)

Above argument will ensure the input dimensions are derived for the first layer

Kernel Initializer- he_normal Initializer

The neural network needs to start with some weights and then iteratively update them to better values. The term kernel_initializer will specify which distribution to use for initializing the weights.

In our project, we have used he_normal Initializer, which draws samples from a truncated normal distribution centered on 0 with std_dev = sqrt(2 / fan_in) where fan_in is the number of input units in the weight tensor.

Argument seed is given for the initializer which is a python integer used for random number generation.

Activation

The activation function in a neural network is responsible for transforming the summed weighted input from the node into the node or output activation for that input.

A neural network consists of node layers and learns how to map input examples to outputs. Activations can either be used through an Activation layer, or through the activation argument supported by all forward layers.

For a given node, the inputs are multiplied and added together by the weights in a node. This value is called the node's aggregate activation. Then the summed activation is transformed through an activation function and defines the node's specific output or "activation."

Types of Activation function

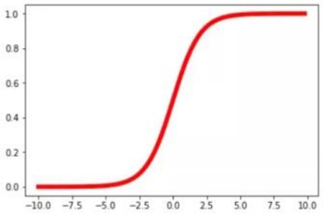
The simplest activation function is called linear activation, where there is no transform at all. A network consisting of only linear activation functions is very easy to train but is unable to learn complex mapping functions. Linear activation functions for networks predicting a quantity (e.g. regression problems) are still used in the output layer.

Nonlinear activation functions are preferred as they allow the nodes to learn more complex data structures. Traditionally, the functions of sigmoid and hyperbolic tangent activation are two commonly used nonlinear activation functions.

Sigmoid

Traditionally, the sigmoid activation function, also known as the logistic function, is a very common neural network activation function and was the default for long time. The function input is converted into a value from 0.0 to 1.0. Similarly, values much smaller than 0.0 are snapped to 0.0 outputs that are much greater than 1.0 are converted to the value 1.0. The function shape for all possible inputs.

$$S(x) = \frac{1}{1+e^{-x}}$$



Limitation

A general problem with Sigmoid is that they are saturating. It means that large values range from 1.0 to-1 or 0 to small values. In addition, the functions are only sensitive to changes in their input midpoint, such as 0.5 for sigmoid.

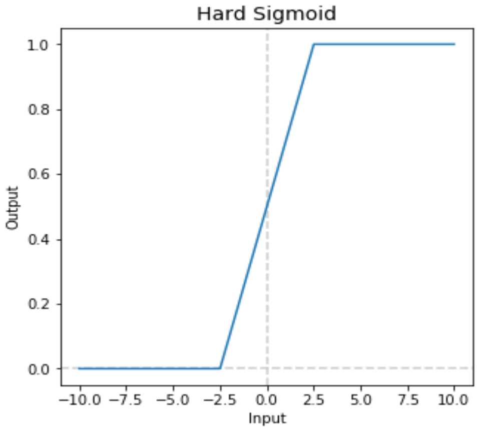
The function's minimal sensitivity and saturation occurs irrespective of whether the node's pooled activation given as input includes useful information. When saturated, continuing to change the weights to boost the model's performance is difficult for the learning algorithm.

Error is backpropagated over the network and used for weight changes. According to the derivative of the chosen activation feature, the amount of error decreases significantly with each additional layer through which it is propagated. This is called the problem of the vanishing gradient and prevents active learning in deep (multi-layered) networks.

Lower gradients make it difficult to know which way the parameters will travel in order to improve the value function

Hard Sigmoid

This function is a piece wise linear approximation of the sigmoid equation piece. It is equal to 0 in the[-Inf; -2.5] range, then increases linearly from 0 to 1 in the[-2.5; + 2.5] range and stays equal to 1 in the range (+ 2.5; + Inf]. Hard Sigmoid computation is known to be faster than normal Sigmoid computation, because the exponent will not have to be measured and it offers reasonable results on classification tasks.Since our project is a binary classification task we have used Hard Sigmoid instead of Sigmoid activation function for the above reason.

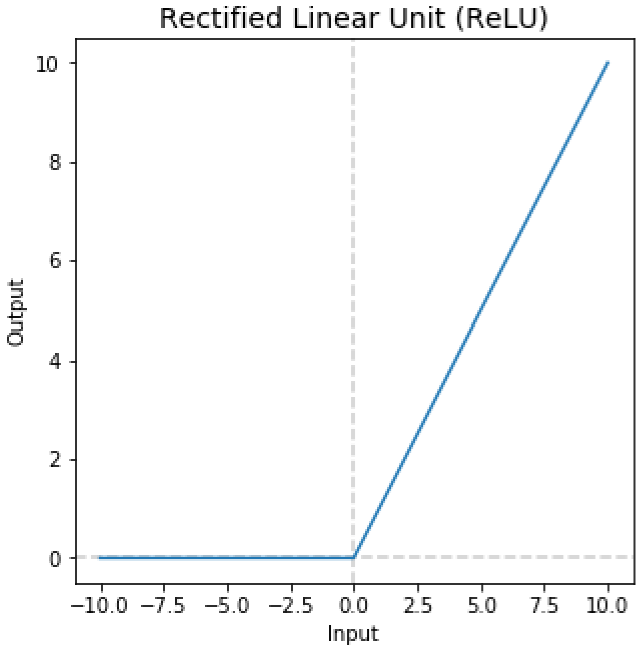


Limitation

Hard Sigmoid should not be used for regression tasks simply because it is an approximation, as the error is going to be much higher than that for normal sigmoid.

Relu- rectified linear unit

Relu is a simple yet powerful activation function which will output will be same as input if the input is positive and 0 otherwise. It is claimed to be the most common activation function for training neural networks, to produce better results than Sigmoid and TanH.



$$y = max(0, x)$$

Advantages

- Due to lack of complex computation involved its cheaper to compute. In addition, it takes less time to train the model.
- Faster convergence is achieved. There is no problem of saturation due to linearity and thus doesn’t suffer from vanishing gradient problem.
- Relu is sparsely activated. Because Relu is zero for all negative inputs, no activation at all is likely to occur for any given unit. Sparsity leads to concise models, often with better predictive power and less overfitting / noise. In a sparse network, neurons are more likely to process meaningful aspects of the problem.

Limitation

For essentially all inputs, Relu neurons can sometimes be pushed into states where they become inactive. No gradients flow back through the neuron in this state, and thus the neuron becomes stuck in a perpetually inactive state and "dies." In some cases, large numbers of neurons in a network may be stuck in dead states, effectively decreasing model capacity.

A neuron of the Relu is "dead" when it is stuck in the negative side and always produces 0. Because Relu’s slope in the negative range is also 0, it is unlikely to recover once a neuron becomes negative.

Model Compilation

We need to configure the learning process, which is done using the compile method, before training a model. Below three arguments are given to it:

- An optimizer: Existing optimizer's string identifier or an Optimizer class instance.
- A loss function: Goal of the model is to minimize loss function. It may be an existing loss function's string identifier or it may be an objective function.
- A list of metrics: A metric may be an existing metric string identifier or a custom metric function.

Optimizer

Optimizers helps to minimize (or maximize) an Objective function. For example— we call the neural network's Weights(W) and Bias(b) values its internal learning parameters that are used in the measurement of output values and are trained and modified in the direction of optimal solution, i.e. minimizing the loss through the training process of the network and also playing a major role in the Neural Network Model training process. Optimizers controls the learning rate. The learning speed determines how easily the correct weights are determined for the model. A lower learning level can result in more accurate weights (up to a certain point), but it will take longer to calculate the weights.

- **Adam- Adaptive Moment Estimation**

Adam is a method for calculating adaptive learning rates for each parameter. As Adam converges quickly when compared to other optimizers it works well. Also learning speed of the model is high. It also corrects every problem faced by other optimization techniques such as fading learning rate, slow convergence or high variance in parameter updates which leads to fluctuating functional losses

In our project, the question was to select the best optimizer for our Neural Network Model to converge quickly and to properly learn and adjust the internal parameters to minimize the loss function. When experimenting with other optimizers, Adam works well and exceeds other adaptive techniques.

- **Adagrad**

Adagrad updates learning rate based on parameter. So for uncommon parameters it makes big updates and for regular parameters it makes small updates. That's why it's well-suited to handle sparse data.

Limitation

learning rate-η is always decreasing and decaying. The model simply stops training entirely and starts gaining new information. Because we know that as the learning rate decreases and decreases, the Model's ability to learn decreases rapidly and this gives very slow convergence and takes a long time to train and learn, i.e. learning speed suffers and decreases.

- **Adadelta**

It is an AdaGrad extension that tends to remove the problem of decaying learning rate. Adadelta restricts the amount of cumulative past gradients to some fixed size w instead of accumulating all previous squared gradients.

Loss function

Machines learn through loss functions. It is a way of determining how well the data are modelled by different algorithms. Gradually, the loss function learns to reduce the error in estimation with the aid of some optimization function.

Generally, loss functions can be divided into two main categories depending on the type of learning activity we deal with — regression and classification loss function.

Classification Loss Types

- **Hinge Loss**

Simply put, by some safety margin (usually one) the score of the appropriate category should be greater than the sum of scores of all the wrong categories. And thus, the loss of the hinge is used for classification of the maximum margin. It is a convex function, although not differentiable, which makes it easy to work with the usual convex optimizers used in machine learning.

$$SVMLoss = \sum_{j \neq y_i} max(0, s_j - s_{y_i} + 1)$$

- **Cross Entropy Loss**

This is the most common setting for issues with classification. Loss of cross-entropy decreases as the expected probability varies from the tag itself. An important aspect of this is that cross-entropy loss strongly penalizes positive yet incorrect predictions.

Cross-entropy will calculate a score that summarizes the average difference between the actual and predicted probability distributions for predicting class 1. The score is minimized and a perfect cross-entropy value is 0.

$$CrossEntropyLoss = -(y_i log(\hat{y}_i) + (1 - y_i) log(1 - \hat{y}_i))$$

- **Squared Hinge Loss**

A popular extension is called the loss of the squared hinge, which calculates the loss of the score hinge square. It has the effect of smoothing and numerically making it easier to work with the surface of the error function.

If using a hinge loss results in improved performance on a given issue of binary classification, a squared hinge loss is likely to be enough. As with the hinge loss function, it is necessary to change the target variable to have values in {-1, 1} range.

- **Mean Squared Error**

It is the sum, over all the data points, of the square of the difference between the predic ted and actual target variables, divided by the number of data points.It is usually used wi th Regression based problems.

$$\sum_{i=1}^{n} \frac{\left(w^T x(i) - y(i)\right)^2}{n}$$

Metric

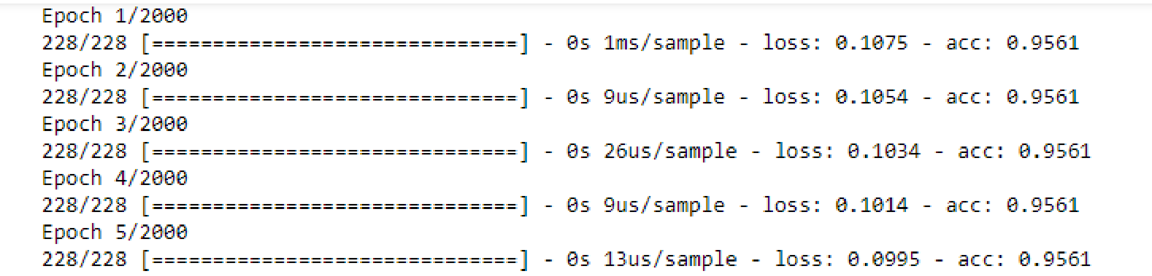
Since our project deals with classification problem we have used accuracy as our metric.

Model Training

We will use the function fit() on our model to train with the following five parameters: training data (X_train), target data (y_train) and number of epochs.

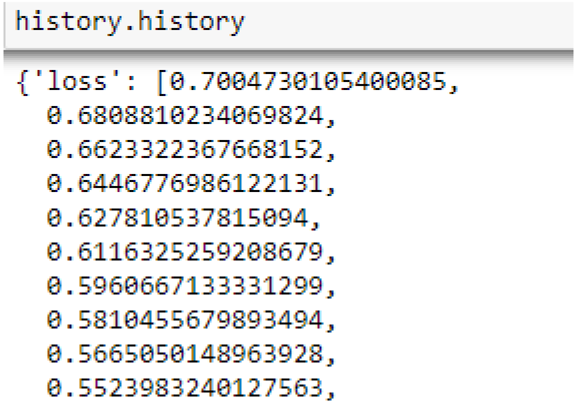
The number of epochs that the model will cycle through the data is the number of times. The more epochs we run, the stronger the model, to some degree. The model should stop improving after that point during each epoch. However, the longer the system would take to run, the more epochs.

history = model.fit(X_train, y_train, epochs=2000,batch_size=X_train.shape[0])



```
Epoch 1/2000
228/228 [==============================] - 0s 1ms/sample - loss: 0.1075 - acc: 0.9561
Epoch 2/2000
228/228 [==============================] - 0s 9us/sample - loss: 0.1054 - acc: 0.9561
Epoch 3/2000
228/228 [==============================] - 0s 26us/sample - loss: 0.1034 - acc: 0.9561
Epoch 4/2000
228/228 [==============================] - 0s 9us/sample - loss: 0.1014 - acc: 0.9561
Epoch 5/2000
228/228 [==============================] - 0s 13us/sample - loss: 0.0995 - acc: 0.9561
```

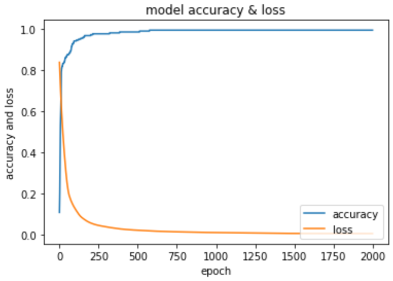
We can see that loss keep decreasing with the iteration



```
history.history
```

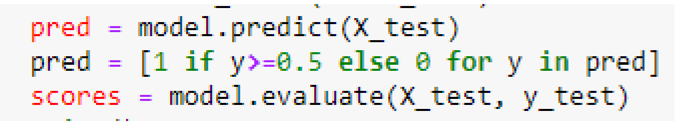
```
{'loss': [0.7004730105400085,
  0.6808810234069824,
  0.662332236768152,
  0.644677698612213,
  0.627810537815094,
  0.6116325259208679,
  0.5960667133331299,
  0.5810455679893494,
  0.5665050148963928,
  0.5523983240127563,
```

Comparison on Model accuracy and Loss with respect to number of iteration



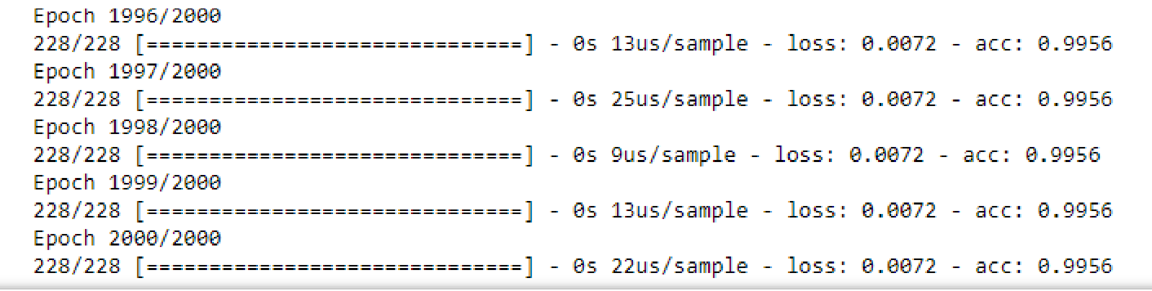
Predictions

We would use ‘predict()’ function, to test our model built with test data(X_test)

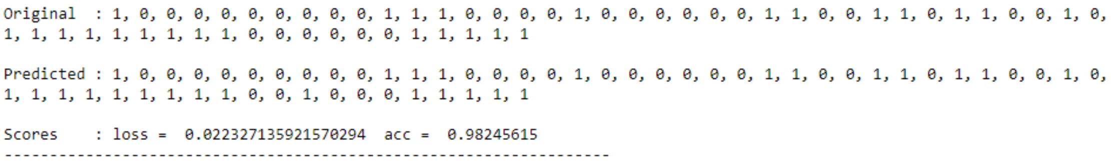


```
pred = model.predict(X_test)
pred = [1 if y>=0.5 else 0 for y in pred]
scores = model.evaluate(X_test, y_test)
```

Final Model Results



```
Epoch 1996/2000
228/228 [==============================] - 0s 13us/sample - loss: 0.0072 - acc: 0.9956
Epoch 1997/2000
228/228 [==============================] - 0s 25us/sample - loss: 0.0072 - acc: 0.9956
Epoch 1998/2000
228/228 [==============================] - 0s 9us/sample - loss: 0.0072 - acc: 0.9956
Epoch 1999/2000
228/228 [==============================] - 0s 13us/sample - loss: 0.0072 - acc: 0.9956
Epoch 2000/2000
228/228 [==============================] - 0s 22us/sample - loss: 0.0072 - acc: 0.9956
```



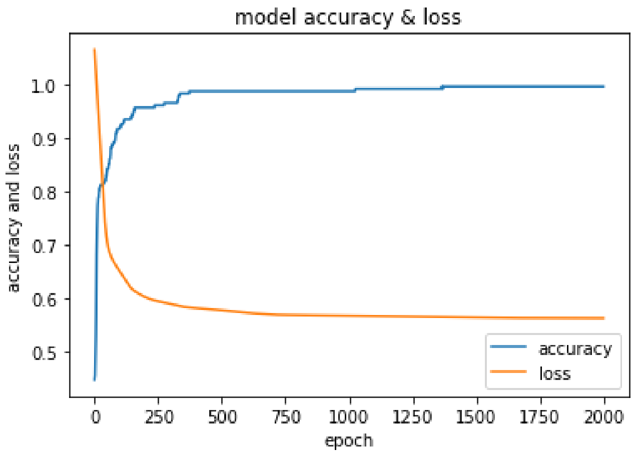
```
Original  : 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0,
1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1

Predicted : 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0,
1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1

Scores    : loss =  0.022327135921570294  acc =  0.98245615
-------------------------------------------------------------------
```

Comparison of Model

- **Significance of Loss**

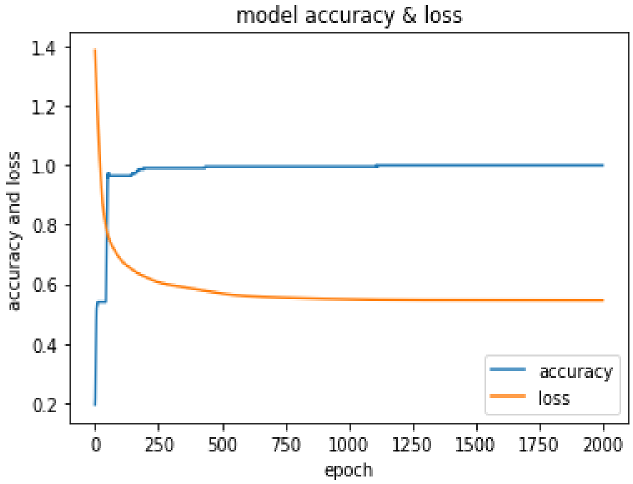
Hinge Loss

```
Scores     : loss =  0.4891763517731114  acc =  0.98245615
```



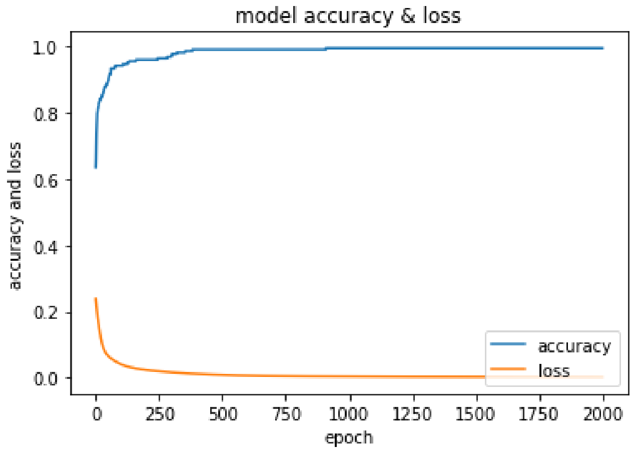
Squared Hinge Loss

```
Scores     : loss =  0.5610291602318747  acc =  0.98245615
```



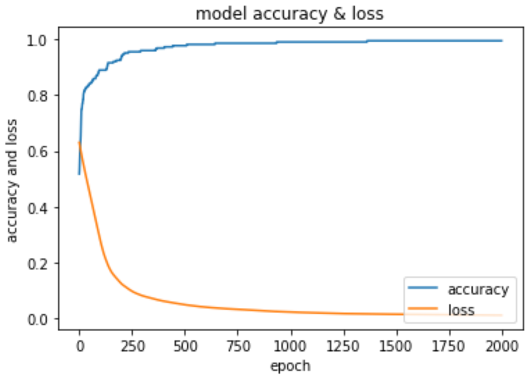
Mean squared Loss

Scores : loss = 3.420444932488496e-05 acc = 1.0



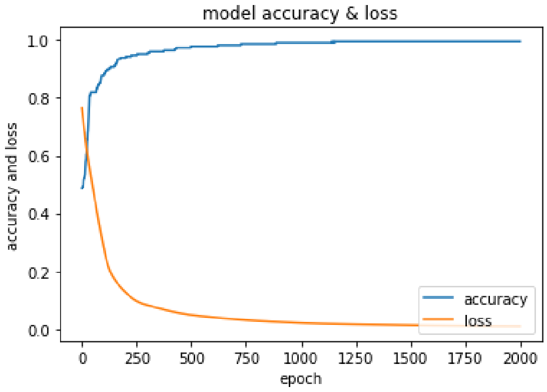
- **Significance of Neurons**

Dimension 50



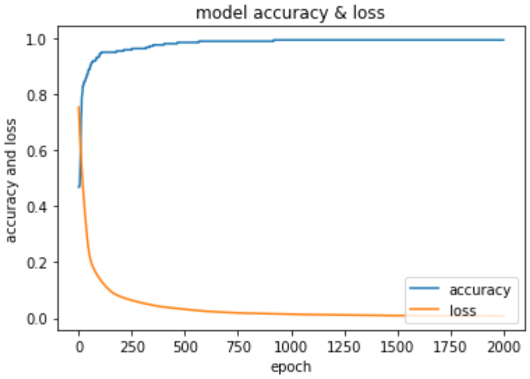
loss = 0.04795878393608227 acc = 0.98245615

Dimension 100



loss = 0.001329377685722552 acc = 1.0

Dimension 250

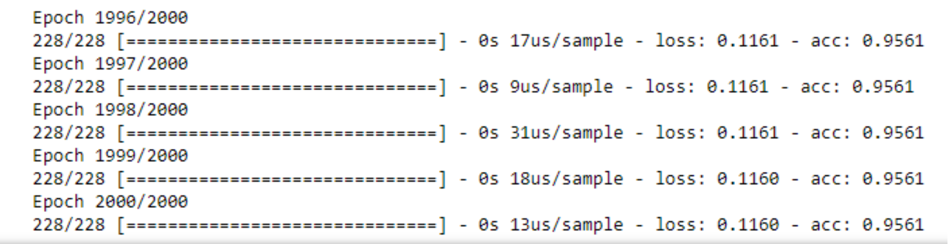


loss = 0.0 acc = 1.0

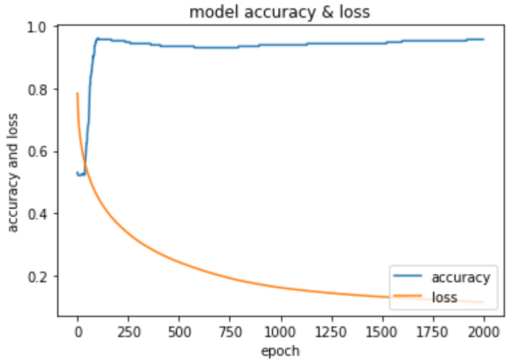
Observation: From the above graphs we find that as the number of dimension increases we achieve a better model in terms of loss and accuracy faster.

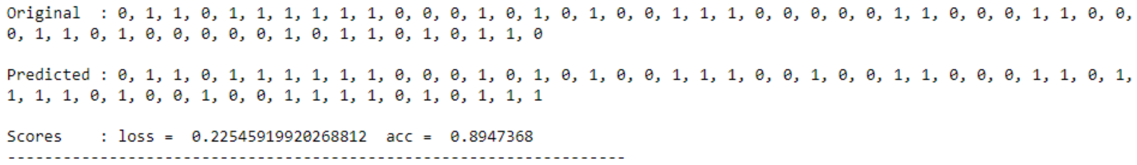
- **Significance of Optimizers**

Adagrad



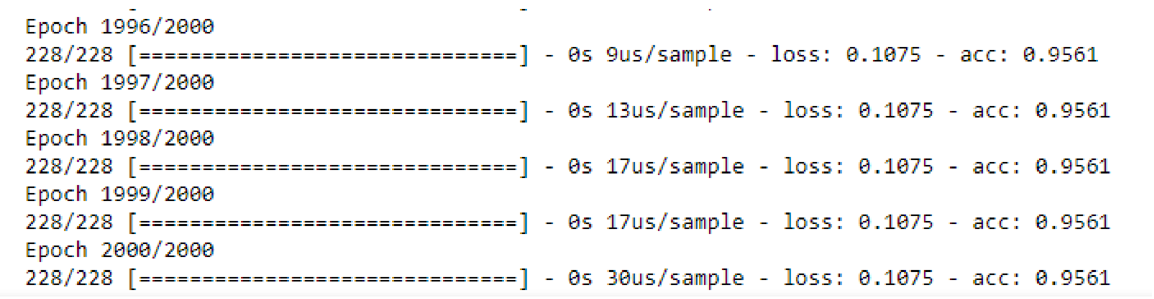
```
Epoch 1996/2000
228/228 [==============================] - 0s 17us/sample - loss: 0.1161 - acc: 0.9561
Epoch 1997/2000
228/228 [==============================] - 0s 9us/sample - loss: 0.1161 - acc: 0.9561
Epoch 1998/2000
228/228 [==============================] - 0s 31us/sample - loss: 0.1161 - acc: 0.9561
Epoch 1999/2000
228/228 [==============================] - 0s 18us/sample - loss: 0.1160 - acc: 0.9561
Epoch 2000/2000
228/228 [==============================] - 0s 13us/sample - loss: 0.1160 - acc: 0.9561
```



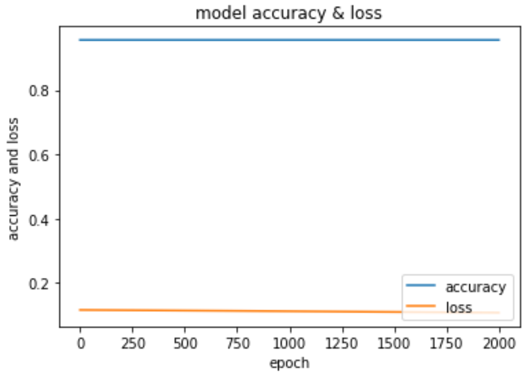


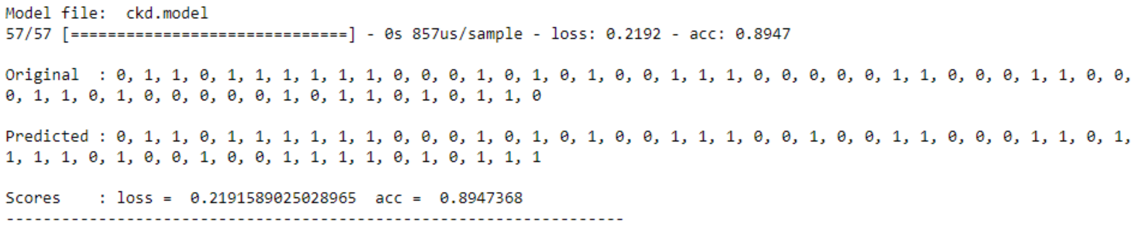
```
Original  : 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0,
0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0

Predicted : 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1,
1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1

Scores    : loss =  0.22545919920268812  acc =  0.8947368
-------------------------------------------------------------------
```

Adadelta



```
Epoch 1996/2000
228/228 [==============================] - 0s 9us/sample - loss: 0.1075 - acc: 0.9561
Epoch 1997/2000
228/228 [==============================] - 0s 13us/sample - loss: 0.1075 - acc: 0.9561
Epoch 1998/2000
228/228 [==============================] - 0s 17us/sample - loss: 0.1075 - acc: 0.9561
Epoch 1999/2000
228/228 [==============================] - 0s 17us/sample - loss: 0.1075 - acc: 0.9561
Epoch 2000/2000
228/228 [==============================] - 0s 30us/sample - loss: 0.1075 - acc: 0.9561
```



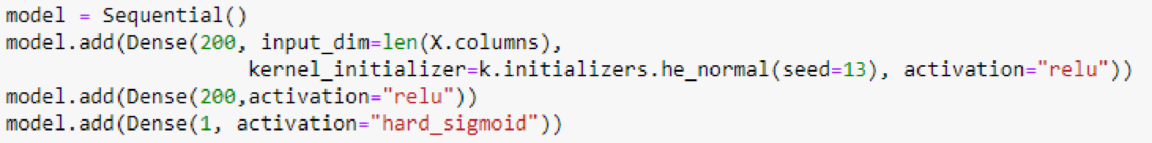


```
Model file:  ckd.model
57/57 [==============================] - 0s 857us/sample - loss: 0.2192 - acc: 0.8947

Original  : 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0,
0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0

Predicted : 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1,
1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1

Scores    : loss =  0.2191589025028965  acc =  0.8947368
------------------------------------------------------------------
```

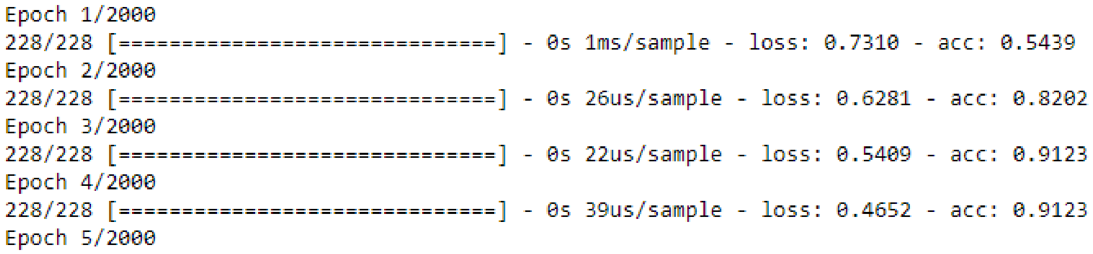
- **Significance of Model Capacity**

As the number of nodes and layers in a model increases, the efficiency of the model increases. Increasing model ability will result in a more accurate model, up to a point where the model can stop improving. Usually the more information you provide about training, the larger the model should be. We only use a small amount of data, so our model is quite small. The larger the model, the greater the computing power it will require and the longer it will take to practice.

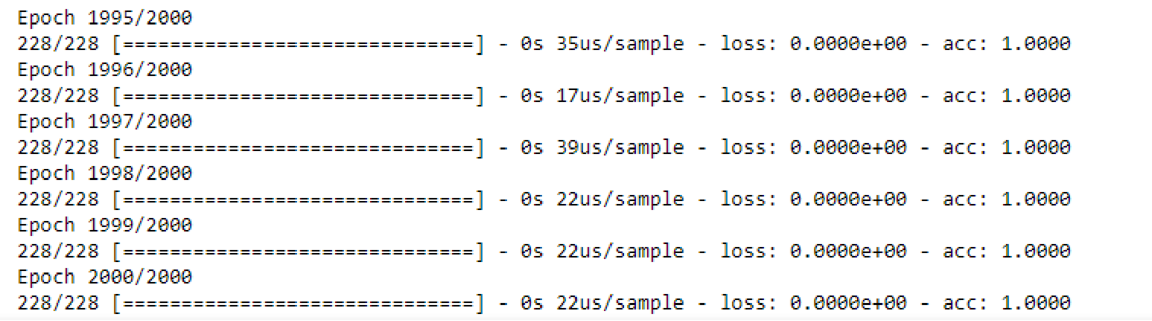
Let's use the same training data as our previous model to create a new model. This time we're going to add a layer to each layer and growing the nodes to 200. We will train the model to see if our validation score will be improved by increasing the model's capacity.



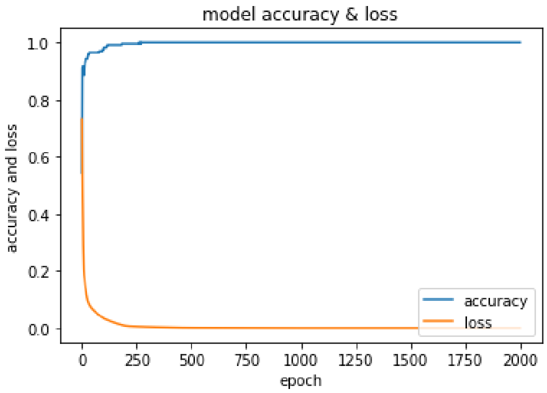
```
model = Sequential()
model.add(Dense(200, input_dim=len(X.columns),
                kernel_initializer=k.initializers.he_normal(seed=13), activation="relu"))
model.add(Dense(200,activation="relu"))
model.add(Dense(1, activation="hard_sigmoid"))
```

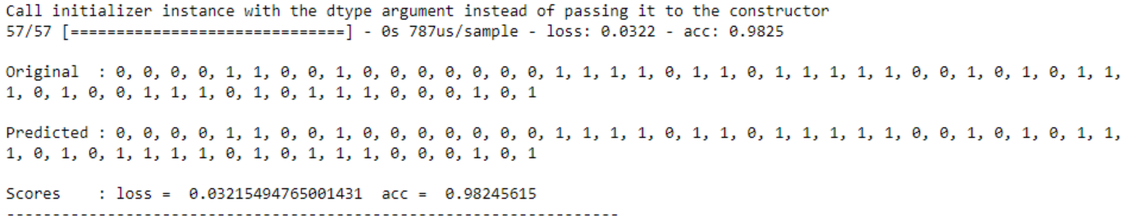


```
Epoch 1/2000
228/228 [==============================] - 0s 1ms/sample - loss: 0.7310 - acc: 0.5439
Epoch 2/2000
228/228 [==============================] - 0s 26us/sample - loss: 0.6281 - acc: 0.8202
Epoch 3/2000
228/228 [==============================] - 0s 22us/sample - loss: 0.5409 - acc: 0.9123
Epoch 4/2000
228/228 [==============================] - 0s 39us/sample - loss: 0.4652 - acc: 0.9123
Epoch 5/2000
```



```
Epoch 1995/2000
228/228 [==============================] - 0s 35us/sample - loss: 0.0000e+00 - acc: 1.0000
Epoch 1996/2000
228/228 [==============================] - 0s 17us/sample - loss: 0.0000e+00 - acc: 1.0000
Epoch 1997/2000
228/228 [==============================] - 0s 39us/sample - loss: 0.0000e+00 - acc: 1.0000
Epoch 1998/2000
228/228 [==============================] - 0s 22us/sample - loss: 0.0000e+00 - acc: 1.0000
Epoch 1999/2000
228/228 [==============================] - 0s 22us/sample - loss: 0.0000e+00 - acc: 1.0000
Epoch 2000/2000
228/228 [==============================] - 0s 22us/sample - loss: 0.0000e+00 - acc: 1.0000
```





```
Call initializer instance with the dtype argument instead of passing it to the constructor
57/57 [==============================] - 0s 787us/sample - loss: 0.0322 - acc: 0.9825

Original  : 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1,
1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1

Predicted : 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1,
1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1

Scores    : loss =  0.03215494765001431  acc =  0.98245615
--------------------------------------------------------------------
```

Pros & Cons:

Artificial neural networks have the ability to learn and model non-linear and complex relationships. However, Multi-layer neural networks are usually hard to train, and require tuning lots of parameters.

Workflow

The implementation phase is initiated by performing data cleansing activities followed by feature set selection. There are over 25 features in the data set, however only selected features are used by the ANN algorithm to predict the classification problem. Subsequently the feature scaling is performed so that all input features lie between 0 and 1. After this process, the input data set is divided into training and test data for building the model. The model is then trained and tested to evaluate the prediction accuracy for the problem statement.

Conclusion:

The above Artificial Neural Network model is built by analyzing various facets of model such as:

- Loss Function
- Number of Neurons (Dimensions)
- Optimizers
- And Model Capacity

The final accuracy of **98.2 %** is obtained after carefully selecting the best performing model based on the various facets.

References

IEEE Conference Papers:

1. Manish Mishra and Monika Srivastava, “A view of Artificial Neural Network”, Jan-2015

2. L.Ozyilmaz and T.Yildirim ,”Artificial Neural Networks for diagnosis of hepatitis disease”,2013

3. T. Jayalskshmi and A. Santhakumaran,”Impact of Preprocessing for Diagnosis of Diabetes Mellitus using Artificial Neural Network”,Dec-2011