INTERNATIONAL MULTI-AWARD WINNING INSTITUTION FOR SUSTAINABILITY

KULLIYYAH OF ENGINEERING

DEPARTMENT OF MECHATRONICS ENGINEERING

MCTE 4322 INTELLIGENT CONTROL

SECTION 1

SEMESTER 1 SESSION 23/24

MINI PROJECT

TITLE: CARDIAC DISEASE DETECTION USING NEURAL NETWORK AND ANFIS-GA

NO.	NAME	MATRIC NO.
1	Nur Bidayatul Hidayah Binti Abdul Kader	2013180
2	Siti Salwa binti Muhammad Rais	2015454
3	Nur Azshimadathul Asyqin binti Darwis	1910828

DATE OF SUBMISSION	**TOTAL MARKS**
16th FEBRUARY 2023	

TABLE OF CONTENT

1. Introduction

Cardiac arrhythmias, characterized by irregular heart rhythms, pose a significant threat to global health, leading to complications such as strokes, heart failure, and sudden cardiac death. Early detection and accurate classification of arrhythmias are crucial for timely intervention and effective management. In recent years, computational intelligence techniques, including Adaptive Neuro-Fuzzy Inference System (ANFIS) combined with Genetic Algorithm System (GA), have emerged as promising tools for detecting and classifying cardiac arrhythmias.

ANFIS is a hybrid intelligence system that combines the adaptive capabilities of neural networks with the reasoning and interpretability of fuzzy logic, and the GA system is a computational model inspired by the process of natural selection and genetics. This study explores the application of ANFIS in the detection of cardiac arrhythmias, aiming to enhance diagnostic accuracy and facilitate timely clinical decision-making.

The successful implementation of an ANFIS-GA-based cardiac arrhythmia detection model could contribute to the improvement of diagnostic accuracy, enabling healthcare professionals to make informed decisions regarding patient care. This study not only explores the technical aspects of ANFIS but also aims to bridge the gap between computational intelligence and clinical practice, fostering advancements in the field of cardiac health monitoring and diagnosis. Ultimately, the findings may pave the way for more efficient and reliable tools in the early detection and management of cardiac arrhythmias, thereby improving patient outcomes and reducing the associated healthcare burden.

2. Problem statement

Cardiovascular diseases stand as a prominent contributor to global mortality rates. Anticipating the potential risk of heart issues holds paramount importance for initiating early interventions and preventive strategies. This mini-project aimed to create a predictive model utilizing computational intelligence methods to evaluate the probability of an individual encountering heart problems.

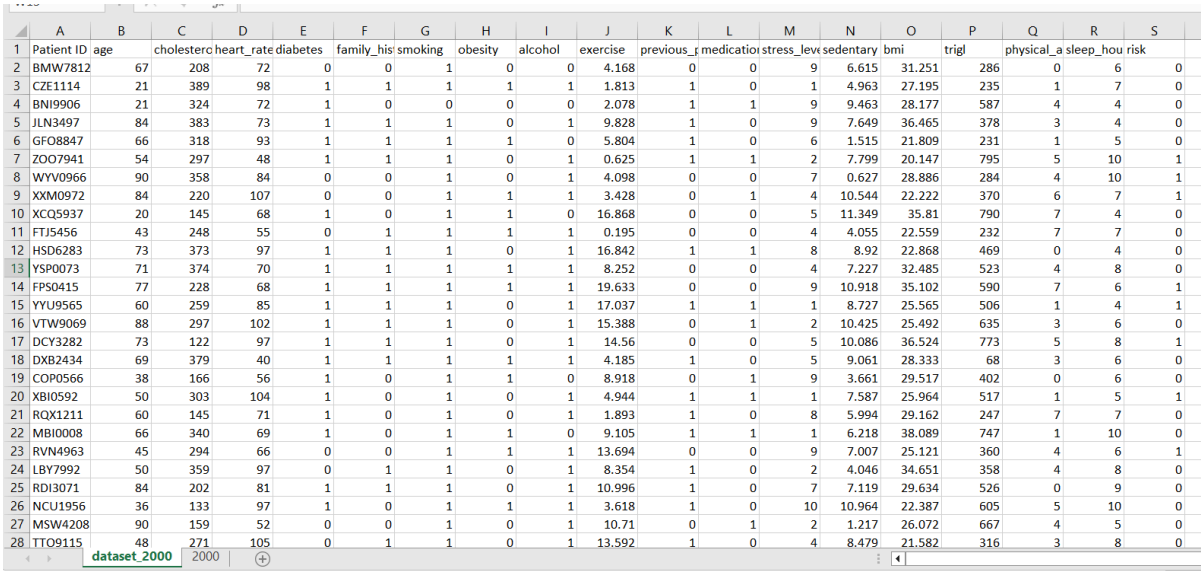
3. Objective

I. Development of Neural Network and ANFIS-GA-based classification model to predict the risk of heart problems in individuals, leveraging relevant health data.

II. Design and train Neural Network and ANFIS-GA to discern intricate patterns and nonlinear associations, thereby enhancing the model's ability to predict the likelihood of heart problems.

4. Dataset

The dataset is taken from, ‘Kaggle Heart Attack Prediction Dataset’.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
1	Patient ID	age	cholesterc	heart_rate	diabetes	family_his	smoking	obesity	alcohol	exercise	previous_p	medicatio	stress_leve	sedentary	bmi	trigl	physical_a	sleep_hou	risk
2	BMW7812	67	208	72	0	0	1	0	0	4.168	0	0	9	6.615	31.251	286	0	6	0
3	CZE1114	21	389	98	1	1	1	1	1	1.813	1	0	1	4.963	27.195	235	1	7	0
4	BNI9906	21	324	72	1	0	0	0	0	2.078	1	1	9	9.463	28.177	587	4	4	0
5	JLN3497	84	383	73	1	1	1	0	1	9.828	1	0	9	7.649	36.465	378	3	4	0
6	GFO8847	66	318	93	1	1	1	1	0	5.804	1	0	6	1.515	21.809	231	1	5	0
7	ZOO7941	54	297	48	1	1	1	0	1	0.625	1	1	2	7.799	20.147	795	5	10	1
8	WYV0966	90	358	84	0	0	1	0	1	4.098	0	0	7	0.627	28.886	284	4	10	1
9	XXM0972	84	220	107	0	0	1	1	1	3.428	0	1	4	10.544	22.222	370	6	7	1
10	XCQ5937	20	145	68	1	0	1	1	0	16.868	0	0	5	11.349	35.81	790	7	4	0
11	FTJ5456	43	248	55	0	1	1	1	1	0.195	0	0	4	4.055	22.559	232	7	7	0
12	HSD6283	73	373	97	1	1	1	0	1	16.842	1	1	8	8.92	22.868	469	0	4	0
13	YSP0073	71	374	70	1	1	1	1	1	8.252	0	0	4	7.227	32.485	523	4	8	0
14	FPS0415	77	228	68	1	1	1	1	1	19.633	0	0	9	10.918	35.102	590	7	6	1
15	YYU9565	60	259	85	1	1	1	0	1	17.037	1	1	1	8.727	25.565	506	1	4	1
16	VTW9069	88	297	102	1	1	1	0	1	15.388	0	1	2	10.425	25.492	635	3	6	0
17	DCY3282	73	122	97	1	1	1	0	1	14.56	0	0	5	10.086	36.524	773	5	8	1
18	DXB2434	69	379	40	1	1	1	1	1	4.185	1	0	5	9.061	28.333	68	3	6	0
19	COP0566	38	166	56	1	0	1	1	0	8.918	0	1	9	3.661	29.517	402	0	6	0
20	XBI0592	50	303	104	1	0	1	0	1	4.944	1	1	1	7.587	25.964	517	1	5	1
21	RQX1211	60	145	71	1	0	1	0	1	1.893	1	0	8	5.994	29.162	247	7	7	0
22	MBI0008	66	340	69	1	0	1	1	0	9.105	1	1	1	6.218	38.089	747	1	10	0
23	RVN4963	45	294	66	0	0	1	1	1	13.694	0	0	9	7.007	25.121	360	4	6	1
24	LBY7992	50	359	97	0	1	1	0	1	8.354	1	0	2	4.046	34.651	358	4	8	0
25	RDI3071	84	202	81	1	1	1	0	1	10.996	1	0	7	7.119	29.634	526	0	9	0
26	NCU1956	36	133	97	1	0	1	1	1	3.618	1	0	10	10.964	22.387	605	5	10	0
27	MSW4208	90	159	52	0	0	1	0	1	10.71	0	1	2	1.217	26.072	667	4	5	0
28	TTO9115	48	271	105	0	1	1	0	1	13.592	1	0	4	8.479	21.582	316	3	8	0

dataset_2000 | 2000

Before implementing intelligent analysis, we convert the linguistic data into values so that the system can do calculations. We also do normalization to ensure fair comparison and to help in handling outliers, which can significantly impact the mean and standard deviation, leading to skewed normalization results.

5. Soft Computing Modelling

I. Data Collection And Processing

With the presented dataset, a concentrated approach is taken by picking 17 input features from 24 features given from Kaggle dataset and highlighting their relevance to heart risk prediction. These features include age, cholesterol, heart rate, diabetes, family history, smoking, obesity, alcohol, exercise, previous problems, medication, stress level, sedentary, BMI, trigl, physical activity, and sleep hour. The dataset is carefully preprocessed after this feature selection procedure, which includes resolving missing values by eliminating them from the dataset, normalizing numerical features, and encoding categorical variables. We only choose 1000 datasets out of around 8000 from the provided data.

II. Neural Network Model Development

The neural network architecture that is used to forecast cardiac risk is constructed with two hidden layers, each containing 16 neurons, one output layer, and 17 input features. For the hidden layers in this design, the Rectified Linear Unit (ReLU) activation function was selected because it makes it easier to introduce non-linearity and helps to capture intricate patterns in the data. A sigmoid activation function is another feature of the output layer that makes it ideal for binary classification jobs like heart risk prediction.

Stochastic Gradient Descent (SGD) is the backpropagation optimization technique used for the training phase. SGD effectively modifies the model's parameters in order to reduce error and improve prediction accuracy. The neural network can learn and generalize patterns in the dataset because of the strong foundation created by this mix of optimization techniques and design, which eventually helps to produce accurate cardiac risk predictions. We also experiment with various architectures, such as changing the number of layers and neurons, to determine the best configuration. Use regularization techniques, such as dropout, to avoid overfitting.

III. Training the Neural Network:

The training phase of the neural network for heart risk prediction commences by utilizing the training set to adjust the weights and biases through the backpropagation process iteratively. To enhance the model's performance, a meticulous optimization of hyperparameters is undertaken. This involves a strategic modification of the architecture by increasing the hidden layers from 2 to 3 and the number of neurons from 16 to 32. The learning rate, a critical parameter influencing the convergence of the model, is also adjusted to 0.01. In a systematic experimentation approach, eight different models are constructed, each with unique configurations, to identify the most effective combination of hyperparameters. This iterative process aims to fine-tune the neural network, achieving optimal predictive accuracy for heart risk. Throughout the training, the model's generalization capabilities are continuously monitored using validation data, helping to detect and mitigate potential overfitting issues. This comprehensive strategy ensures the neural network adapts to the intricacies of the dataset and maximizes its capacity to make accurate predictions.

IV. Evaluation of Neural Network:

The trained neural network undergoes a rigorous evaluation process using the test set to gauge its generalization performance. By applying metrics such as accuracy and precision, a comprehensive assessment of the model's effectiveness in predicting heart risk is conducted. Accuracy provides an overall measure of correct predictions while precision evaluates the accuracy of positive predictions.

V. ANFIS-GA Model Development:

As for ANFIS-GA model, the initialization of ANFIS parameters involves setting the membership function parameters within a random range of 0-1, with the inclusion of three membership functions. Additionally, rule weights and consequent parameters are initialized with small random values. This approach aims to introduce diversity in the initial ANFIS structure to facilitate effective learning and adaptation to the intricacies of the dataset. On the Genetic Algorithm (GA) side, the crossover and mutation rates are randomly initialized, and the number of generations is set to 300. With 10 offsprings generated per iteration, a learning rate of 0.01, and a simple rule combination strategy, the GA strives to optimize the ANFIS parameters for enhanced predictive accuracy. This comprehensive parameter initialization strategy seeks to strike a balance between exploration and exploitation, enabling both components to collaboratively contribute to the model's robust performance in predicting heart risk levels.

VI. Training the ANFIS-GA Model:

The Genetic Algorithm (GA) is instrumental in optimizing the parameters of our heart risk prediction model, the Adaptive Neuro-Fuzzy Inference System (ANFIS).

Using the training data, the GA systematically refines the ANFIS parameters, including membership functions and rule weights. This optimization process aims to improve the model's ability to discern complex relationships within the dataset and fine-tune its fuzzy rule-based decision-making. Following this, a meticulous fine-tuning stage takes place using the validation set, a crucial measure to prevent overfitting. During the fine-tuning phase of our heart risk prediction model, we carefully consider the adjustments made to key parameters in the genetic algorithm. Specifically, we fine-tune the number of generations, varying it across different values: 300, 50, and 500. This deliberate exploration aims to identify the optimal number of iterations that contribute to the model's convergence and effectiveness in capturing the nuances of heart risk factors. Additionally, we fine-tune the mutation rate, experimenting with values of 0.1 and 0.002. The mutation rate is a crucial parameter influencing the genetic algorithm's exploration-exploitation trade-off, and variations in this parameter are systematically explored to strike the right balance. This fine-tuning process, guided by an iterative and adaptive approach, seeks to maximize the model's predictive accuracy and generalization capabilities, ensuring its robust performance across diverse scenarios. Regular validation against performance metrics provides valuable insights into the impact of these fine-tuned parameters on the overall effectiveness of the heart risk prediction model.

VII. Evaluation of ANFIS-GA Model:

The evaluation of our trained neural network for heart risk prediction involves a comprehensive analysis using the test set, aiming to assess its generalization performance beyond the training data. Through the utilization of diverse metrics,

including accuracy, precision, and Root Mean Squared Error (RMSE), we gain a nuanced understanding of the model's predictive capabilities. Accuracy serves as a fundamental measure of correct predictions, providing an overall assessment of the model's correctness. Precision evaluates the accuracy of positive predictions, crucial in scenarios where the consequences of false positives are significant, such as in healthcare. Additionally, RMSE offers insights into the magnitude of prediction errors, providing a quantitative measure of how well the model's predictions align with the actual heart risk levels.

6. Coding

Neural Network Coding (**Appendix A**)

In github

ANFIS-GA Coding (**Appendix B**)

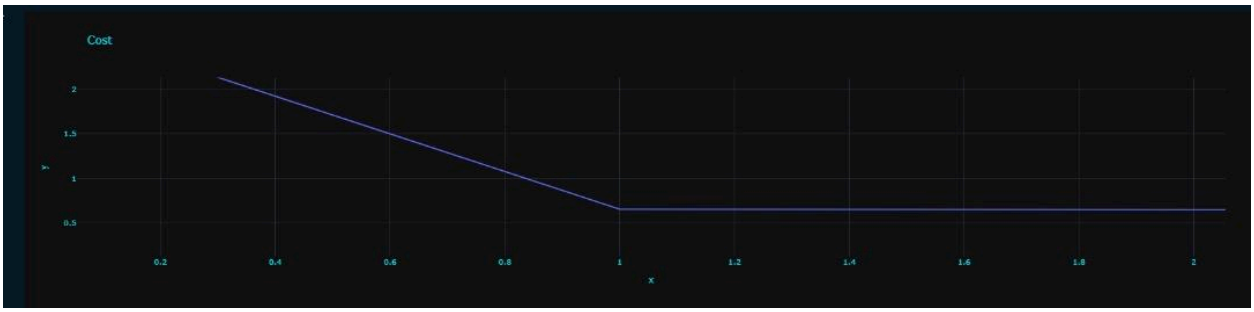
In github

7. Results and Analysis

7.1 Neural Network

	Learning_Rate	Layers	Epochs	Accuracy
Model_1	0.100000	[16, 100, 100, 100, 1]	2000	0.315000
Model_2	0.001000	[16, 16, 1]	3000	0.315000
Model_3	0.000100	[16, 16, 1]	3000	0.685000
Model_4	0.000100	[16, 16, 1]	30000	0.685000
Model_5	0.000100	[16, 16, 16, 1]	30000	0.685000
Model_6	0.000100	[16, 16, 16, 16, 1]	30000	0.315000
Model_7	0.000100	[16, 32, 32, 1]	30000	0.685000
Model_8	0.000100	[16, 128, 128, 1]	30000	0.685000

This figure shows the results of fine-tuning the hidden layer, and also the number of hidden layers, and the number of neurons in each hidden layer.



The result is in converged form. This typically means that the algorithm has reached a stable or optimal solution. Convergence occurs when the algorithm iteratively refines its parameters or weights, and these adjustments progressively lead to diminishing changes in the objective function or loss.

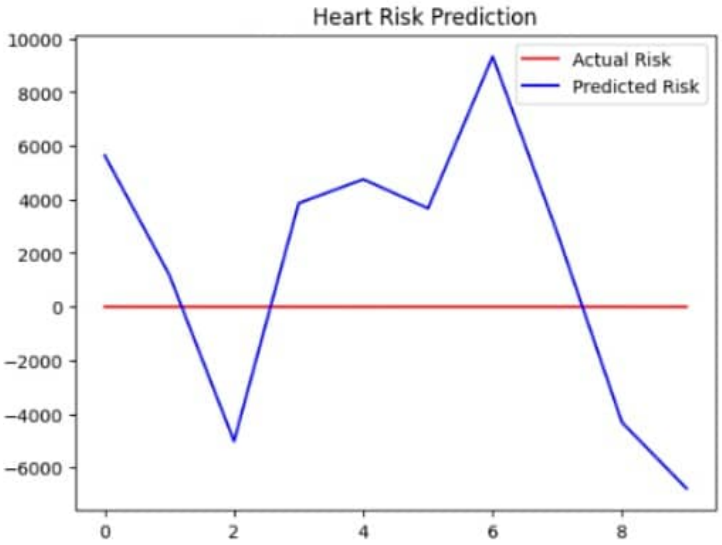
```
Cost after iteration 0: 2.761261976546644
Cost after iteration 5000: 0.6556041911703235
Cost after iteration 10000: 0.6507210899167413
Cost after iteration 15000: 0.6474279800161966
Cost after iteration 20000: 0.6451902173713114
Cost after iteration 25000: 0.6437173549032525
```

When the cost (or loss) after each iteration in a machine learning training process is decreasing, it signifies that the model is making progress in learning from the data. A decreasing cost is a positive sign.

```
Test_risk = [0 0 0 0 0 0 0 0 1 1 0 0 1 0 1 0 0 0 0 1 0 1 0 1 1 0 1 1 0 0 0 1 1 0 0 1 0
 0 0 1 0 0 0 1 0 0 0 1 0 1 0 1 0 0 0 1 0 0 0 0 1 0 0 0 0 0 1 0 1 0 0 0 1 0
 0 1 0 0 0 1 0 1 0 0 0 0 0 1 1 0 1 0 1 0 1 0 1 1 1 0 1 0 0 1 0 1 1 1 0 1 0
 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 1 0 0 1 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0
 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 1 1 1 0 1 0 0 0 0 1 0 0 0 1 1 0 1 0 1
 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0]
Accuracy =  0.6849999999999999
```

Our Neural Network system accuracy, 0.84999.

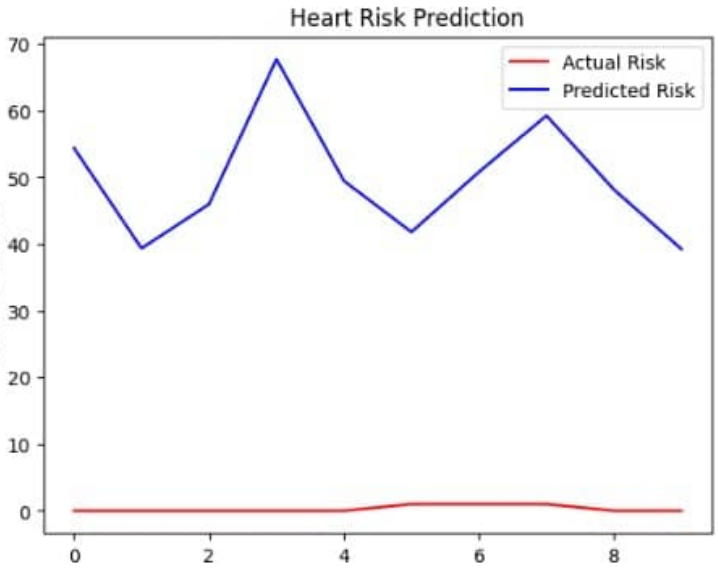
7.3 ANFIS-GA



Mutation rate = 0.1

Learning rate = 0.01

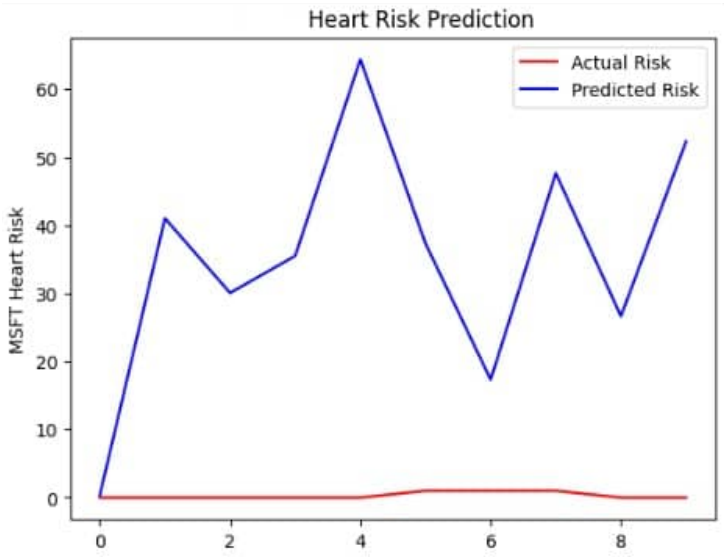
Generation = 300



Mutation rate = 0.002

Learning rate = 0.001

Generation = 50



Mutation rate = 0.0002

Learning rate = 0.001

Generation = 200

The analysis of the results from fine-tuning three ANFIS-GA models, each with varying mutation rates (0.1, 0.002, 0.0002), learning rates (0.01, 0.001), and generations (300, 50, 200), suggests a complex optimization landscape for heart risk prediction. None of the resulting graphs exhibit a close alignment with the actual target, indicating challenges in achieving optimal predictive performance. The influence of mutation rates, learning rates, and the number of generations on the convergence behavior of the genetic algorithm is evident. Finding the right balance between exploration and exploitation remains a considerable challenge. Experimentation with additional hyperparameter tuning, ensemble models, and diagnostic tools may provide valuable insights into refining the ANFIS-GA models for improved heart risk prediction accuracy.

8. Conclusion

In conclusion, our efforts in developing a Neural Network and Adaptive Neuro-Fuzzy Inference System (ANFIS)-GA for heart risk prediction have successfully achieved the defined objectives, yet the current results indicate the necessity for further fine-tuning in the ANFIS-GA model. Despite fulfilling the primary goals of leveraging relevant health data to design and train the ANFIS-GA model, we recognize the importance of refinement to enhance its predictive accuracy and robustness. The iterative nature of fine-tuning, involving a comprehensive exploration of parameters and optimization strategies, underscores our commitment to delivering a high-performing and reliable tool for classifying the risk of heart problems in individuals.

9. Reference

1. https://www.kaggle.com/datasets/iamsouravbanerjee/heart-attack-prediction-dataset
2. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1380898
3. https://www.kaggle.com/code/fareselmenshawii/neural-network-from-scratch#notebook-container
4. https://www.kaggle.com/code/salwarais/genetic-algorithm-basics/edit

10. Appendix

Appendix A

```python
import seaborn as sns
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import plotly.express as px

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score

q = 16
```

✓ 2.0s Python

```python
#from google.colab import drive
#drive.mount('/content/drive')

df = pd.read_csv("dataset_1000.csv")
df.head()
```

✓ 0.0s Python

```python
#df.drop('id', axis=1, inplace=True) #drop redundant columns
df.describe().T
```

✓ 0.0s Python

```python
corr = df.corr()
plt.figure(figsize=(20,20))
sns.heatmap(corr, cmap='mako_r',annot=True)
plt.show()
```

Python

```python
# Get the absolute value of the correlation
cor_target = abs(corr["risk"])

# Select highly correlated features (thresold = 0.2)
relevant_features = cor_target[cor_target>0.002]

# Collect the names of the features
names = [index for index, value in relevant_features.iteritems()]

# Drop the target variable from the results
names.remove('risk')

# Display the results
print(names)
```

Python

```python
X = df[names].values
y = df['risk'].values

```

Python

```python
def train_test_split(X, y, random_state=41, test_size=0.2):

    # Get number of samples
    n_samples = X.shape[0]

    # Set the seed for the random number generator
    np.random.seed(random_state)

    # Shuffle the indices
    shuffled_indices = np.random.permutation(np.arange(n_samples))

    # Determine the size of the test set
    test_size = int(n_samples * test_size)

    # Split the indices into test and train
    test_indices = shuffled_indices[:test_size]
    train_indices = shuffled_indices[test_size:]

    # Split the features and target arrays into test and train
    X_train, X_test = X[train_indices], X[test_indices]
    y_train, y_test = y[train_indices], y[test_indices]

    return X_train, X_test, y_train, y_test
```

Python

```python
X = scale(X)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, ran
```

Python

```python
def relu(Z):

    A = np.maximum(0,Z)
    cache = Z
    return A, cache
```

Python

```python
z = np.linspace(-12, 12, 200)
fig = px.line(x=z, y=relu(z)[0],title='ReLU Function',template="plotly_dark")
fig.update_layout(
    title_font_color="#00F1FF",
    xaxis=dict(color="#00F1FF"),
    yaxis=dict(color="#00F1FF")
)
fig.show()
```

Python

```python
def relu_backward(dA, cache):

    Z = cache
    dZ = np.array(dA, copy=True)
    # When z <= 0, dz is equal to 0 as well.
    dZ[Z <= 0] = 0

    return dZ
```

Python

```python
def sigmoid(Z):

    A = 1/(1+np.exp(-Z))
    cache = Z
    return A, cache
```

Python

```python
z = np.linspace(-12, 12, 200)
fig = px.line(x=z, y=sigmoid(z)[0],title='Sigmoid Function',template="plotly_d
fig.update_layout(
    title_font_color="#00F1FF",
    xaxis=dict(color="#00F1FF"),
    yaxis=dict(color="#00F1FF")
)
fig.show()

```

Python

```python
def sigmoid_backward(dA, cache):

    Z = cache
    s = 1/(1+np.exp(-Z))
    dZ = dA * s * (1-s)
    return dZ
```

Python

```python
class NeuralNetwork:
    def __init__(self, layer_dimensions=[q,16,16,1],learning_rate=0.00001):
        """
        Parameters
        ----------

        layer_dimensions : list
            python array (list) containing the dimensions of each layer in our

        learning_rate :  float
            learning rate of the network.

        """
        self.layer_dimensions = layer_dimensions
        self.learning_rate = learning_rate


    def initialize_parameters(self):
        """initializes the parameters"""
        np.random.seed(3)
        self.n_layers =  len(self.layer_dimensions)
        for l in range(1, self.n_layers):
            vars(self)[f'W{l}'] = np.random.randn(self.layer_dimensions[l], se
            vars(self)[f'b{l}'] = np.zeros((self.layer_dimensions[l], 1))

```

```python
    def _linear_forward(self, A, W, b):

        # Compute Z
        Z = np.dot(W,A) + b
        # Cache  A, W , b for backpropagation
        cache = (A, W, b)
        return Z, cache

    def _forward_propagation(self,A_prev ,W ,b , activation):


        # Compute Z using the function defined above, compute A using the acti
        if activation == "sigmoid":
            Z, linear_cache = self._linear_forward(A_prev, W, b)
            A, activation_cache = sigmoid(Z)
        elif activation == "relu":
            Z, linear_cache = self._linear_forward(A_prev, W, b)
            A, activation_cache = relu(Z)
            #Store the cache for backpropagation
        cache = (linear_cache, activation_cache)
        return A, cache

```

```python
    def forward_propagation(self, X):

        # Initialize empty list to store caches
        caches = []
        # Set initial A to X
        A = X
        L =  self.n_layers -1
        for l in range(1, L):
            A_prev = A
            # Forward propagate through the network except the last layer
            A, cache = self._forward_propagation(A_prev, vars(self)['W' + str(
            caches.append(cache)
        # Forward propagate through the output layer and get the predictions
        predictions, cache = self._forward_propagation(A, vars(self)['W' + str
        # Append the cache to caches list recall that cache will be (linear_ca
        caches.append(cache)

        return predictions, caches

```

```
68
69      def compute_cost(self, predictions, y):
70
71          # Get number of training examples
72          m = y.shape[0]
73          # Compute cost we're adding small epsilon for numeric stability
74          cost = (-1/m) * (np.dot(y, np.log(predictions+1e-9).T) + np.dot((1-y),
75          # squeeze the cost to set it into the correct shape
76          cost = np.squeeze(cost)
77          return cost
78
79      def _linear_backward(self, dZ, cache):
80
81          # Get the cache from forward propagation
82          A_prev, W, b = cache
83          # Get number of training examples
84          m = A_prev.shape[1]
85          # Compute gradients for W, b and A
86          dW = (1/m) * np.dot(dZ, A_prev.T)
87          db = (1/m) * np.sum(dZ, axis=1, keepdims=True)
88          dA_prev = np.dot(W.T,dZ)
89          return dA_prev, dW, db
90
```

```
91
92      def _back_propagation(self, dA, cache, activation):
93
94          # get the cache from forward propagation and activation derivates func
95          linear_cache, activation_cache = cache
96          # compute gradients for Z depending on the activation function
97          if activation == "relu":
98              dZ = relu_backward(dA, activation_cache)
99
100         elif activation == "sigmoid":
101             dZ = sigmoid_backward(dA, activation_cache)
102         # Compute gradients for W, b and A
103         dA_prev, dW, db = self._linear_backward(dZ, linear_cache)
104         return dA_prev, dW, db
105
106     def back_propagation(self, predictions, Y, caches):
107
108         L =  self.n_layers - 1
109         # Get number of examples
110         m = predictions.shape[1]
111         Y = Y.reshape(predictions.shape)
112         # Initializing the backpropagation we're adding a small epsilon for nu
113         dAL = - (np.divide(Y, predictions+1e-9) - np.divide(1 - Y, 1 - predict
114         current_cache = caches[L-1] # Last Layer
115         # Compute gradients of the predictions
116         vars(self)[f'dA{L-1}'], vars(self)[f'dW{L}'], vars(self)[f'db{L}'] = s
117         for l in reversed(range(L-1)):
118             # update the cache
119             current_cache = caches[l]
120             # compute gradients of the network layers
121             vars(self)[f'dA{l}'] , vars(self)[f'dW{l+1}'], vars(self)[f'db{l+1
122
```

```
    def update_parameters(self):
            """
            Updates parameters using gradient descent
            """
            L = self.n_layers - 1
            # Loop over parameters and update them using computed gradients
            for l in range(L):
                vars(self)[f'W{l+1}'] = vars(self)[f'W{l+1}'] - self.learning_
                vars(self)[f'b{l+1}']  = vars(self)[f'b{l+1}'] - self.learning


    def fit(self,X, Y, epochs=2000, print_cost=True):

            # Transpose X to get the correct shape
            X = X.T
            np.random.seed(1)
            #create empty array to store the costs
            costs = []
            # Get number of training examples
            m = X.shape[1]
            # Initialize parameters
            self.initialize_parameters()
            # loop for stated number of epochs
            for i in range(0, epochs):
                # Forward propagate and get the predictions and caches
                predictions, caches = self.forward_propagation(X)
                #compute the cost function
                cost = self.compute_cost(predictions, Y)
                # Calculate the gradient and update the parameters
                self.back_propagation(predictions, Y, caches)

                self.update_parameters()
```

```python
                if print_cost and i % 5000 == 0:
                    print("Cost after iteration {}: {}".format(i, np.squeeze(c
                if print_cost and i % 5000 == 0:
                    costs.append(cost)
            if print_cost:
            # Plot the cost over training
                fig = px.line(y=np.squeeze(costs),title='Cost',template="plotl
                fig.update_layout(
                    title_font_color="#00F1FF",
                    xaxis=dict(color="#00F1FF"),
                    yaxis=dict(color="#00F1FF")
                )
                fig.show()


    def predict(self,X,y):

        X = X.T
        # Get predictions from forward propagation
        predictions, _ = self.forward_propagation(X)
        # Predictions Above 0.5 are True otherwise they are False
        predictions = (predictions > 0.5)
        # Squeeze the predictions into the correct shape and cast true/false v
        predictions = np.squeeze(predictions.astype(int))
        #Print the accuracy
        return np.sum((predictions == y)/X.shape[1]), predictions.T
```

Python

```python
def train_evaluate_model(X_train, y_train, X_test, y_test, learning_rate, laye

    # create model instance with the given hyperparameters
    model = NeuralNetwork(learning_rate=learning_rate,layer_dimensions=layers)
    # fit the model
    model.fit(X_train, y_train,epochs=epochs,print_cost=False)
    accuracy, predictions = model.predict(X_test, y_test) # calculate accuracy

    #create a dataframe to visualize the results
    eval_df = pd.DataFrame([[learning_rate, layer_dimensions, epochs, accuracy
    return eval_df
```

Python

```python
learning_rate = 0.1
layers = [q,100, 100, 100,1]
epochs = 2000
results = train_evaluate_model(X_train, y_train, X_test, y_test, learning_rate

results.index = ['Model_1']

results.style.background_gradient(cmap =sns.cubehelix_palette(start=.5, rot=-.
```

Python

```python
learning_rate = 0.001
layers = [q,16,1]
epochs = 3000
temp_df = train_evaluate_model(X_train, y_train, X_test, y_test, learning_rate
temp_df.index = ['Model_2']
results = results.append(temp_df)


```

Python

```python
learning_rate = 0.0001
layers = [q,16,1]
epochs = 3000
temp_df = train_evaluate_model(X_train, y_train, X_test, y_test, learning_rate
temp_df.index = ['Model_3']
results = results.append(temp_df)
```

Python

```python
learning_rate = 0.0001
layers = [q,16,1]
epochs = 30000
temp_df = train_evaluate_model(X_train, y_train, X_test, y_test, learning_rate
temp_df.index = ['Model_4']
results = results.append(temp_df)
```

Python

```python
learning_rate = 0.0001
layers = [q, 16,16,1]
epochs = 30000
temp_df = train_evaluate_model(X_train, y_train, X_test, y_test, learning_rate
temp_df.index = ['Model_5']
results = results.append(temp_df)

```

Python

```python
learning_rate = 0.0001
layers = [q,16,16,16,1]
epochs = 30000
temp_df = train_evaluate_model(X_train, y_train, X_test, y_test, learning_rate
temp_df.index = ['Model_6']
results = results.append(temp_df)
```

Python

```python
learning_rate = 0.0001
layers = [q,32,32,1]
epochs = 30000
temp_df = train_evaluate_model(X_train, y_train, X_test, y_test, learning_rate
temp_df.index = ['Model_7']
results = results.append(temp_df)
```

Python

```python
learning_rate = 0.0001
layers = [q,128,128,1]
epochs = 30000
temp_df = train_evaluate_model(X_train, y_train, X_test, y_test, learning_rate
temp_df.index = ['Model_8']
results = results.append(temp_df)
```

Python

```python
results.style.background_gradient(cmap =sns.cubehelix_palette(start=.5, rot=-.
```

Python

```python
model = NeuralNetwork(learning_rate=0.00001)

model.fit(X_train, y_train,epochs=30000,print_cost=True)
```

Python

```python
accuracy,predictions= model.predict(X_test, y_test)
print("Test_risk =" ,y_test)
print("Accuracy = ",accuracy)
print(predictions)
```

Python

Appendix B

```
import random
import numpy as np
from copy import deepcopy
from sklearn.linear_model import LinearRegression

class EVOLUTIONARY_ANFIS:
    def
__init__(self,functions,generations,offsprings,mutationRate,learnin
gRate,chance,ruleComb):
        self.functions = functions
        self.generations = generations
        self.offsprings = offsprings
        self.mutationRate = mutationRate
        self.learningRate = learningRate
```

```
        self.chance = chance #50 percent chance of changing std.
        self.ruleComb = ruleComb
        self._noParam = 17

    def gaussian(self,x, mu, sig):
        return np.exp((-np.power(x - mu, 2.) / (2 * np.power(sig,
2.))))

    def initialize(self,X):
        functions = self.functions
        noParam = self._noParam
        ruleComb = self.ruleComb
        inputs = np.zeros((X.shape[1],X.shape[0],functions))
        Ant = np.zeros((noParam,X.shape[1],X.shape[0],functions))
        L1 = np.zeros((X.shape[1],X.shape[0],functions))
        if ruleComb == "simple":
            L2 = np.zeros((X.shape[0],functions))
        elif ruleComb == "complete":
            rules = X.shape[1]**functions
            L2 = np.zeros((X.shape[0],rules))
        return inputs, Ant, L1, L2

    def mutation(self,arr):
        mutationRate = self.mutationRate
        learningRate = self.learningRate
        chance = self.chance
        temp = np.asarray(arr)   # Cast to numpy array
        mean = temp[0]
        meanShape = mean.shape
        std = temp[1]
        stdShape = std.shape
        mean = mean.flatten()    # Flatten to 1D
        std = std.flatten()    # Flatten to 1D
        num = int(mutationRate*mean.size) # number of elements to
get
        if random.uniform(0,1)>chance:
            inds = np.random.choice(mean.size, size=num)   # Get
random indices
```

```
            mean[inds] -=
np.random.uniform(0,1,size=num)*learningRate        # Fill with
something
            mean = mean.reshape(meanShape)                      #
Restore original shape
            std = std.reshape(stdShape)
        else:
            inds = np.random.choice(std.size, size=num)   # Get
random indices
            std[inds] -=
np.random.uniform(0,1,size=num)*learningRate        # Fill with
something
            std = std.reshape(stdShape)                      #
Restore original shape
            std = np.where(std==0, 0.0001, std) #standard deviation
cannot be zero
            #temp = np.where(temp<=0, 0.0001, temp)
            #temp = np.where(temp>=1, 0.9999, temp)

            mean = mean.reshape(meanShape)
        temp[0] = mean
        temp[1] = std
        return temp

    def init_population(self,X):
        noParam = self._noParam
        functions = self.functions
        offsprings = self.offsprings
        bestParam = np.random.rand(noParam,X.shape[1],functions)
        parentParam = deepcopy(bestParam)
        popParam = []
        for i in range(offsprings):
            popParam.append(self.mutation(parentParam))
        return popParam

    def init_model(self,model=LinearRegression()):
        models = []
        for i in range(self.functions):
                models.append(model)
        return models
```

```
    def forwardPass(self,param,X,inputs,Ant,L1,L2,functions):
        noParam = self._noParam

        for i in range(X.shape[1]):   #input variables
            inputs[i] =
np.repeat(X[:,i].reshape(-1,1),functions,axis=1)

        for ii in range(noParam):   #Anticedent parameters
            for i in range(X.shape[1]):
                Ant[ii] =
np.repeat(param[ii][i,:].reshape(1,-1),X.shape[0],axis=0)

        for i in range(X.shape[1]):  #Membership values using
Gaussian membership function
            L1[i,:,:] =
self.gaussian(x=inputs[i],mu=Ant[0][i],sig=Ant[1][i])

        for j in range(functions):      #rule
            for i in range(1,X.shape[1]):
                L2[:,j] =
(L1[i-1,:,j]*L1[i,:,j])#+(L1[i-1,:,j]+L1[i,:,j])

        summ = np.sum(L2,axis=1).reshape(-1,1) #Weights
normalization
        summation = np.repeat(summ,functions,axis=1)
        L3 = L2/summation
        L3 = np.round(L3,5)
        #Errorcheck = np.sum(L3,axis=1)

        consequent = X
        L4 = np.zeros((functions,X.shape[0],X.shape[1]))
        for i in range (functions):
            L4[i] = consequent
            L4[i] = L4[i]*L3[:,i].reshape(-1,1)
        return L1,L2,L3,L4

    def linear_fit(self,L3,L4,X,y,functions,models):
        pred_train = np.zeros((X.shape[0],functions))
        for i in range(functions):
```

```
            models[i].fit(L4[i],y)
            predTemp = models[i].predict(L4[i])
            pred_train[:,i] = predTemp[:,0]
        pred_train = pred_train*L3 #consequent function output *
normalized weights
        pred_train = np.sum(pred_train,axis=1)
        return pred_train, models

    def linear_predict(self,L3,L4,X,functions,Trained_models):
        pred_test = np.zeros((X.shape[0],functions))
        for i in range(functions):
            predTemp =
Trained_models[i].predict(L4[i]).reshape(-1,1)
            pred_test[:,i] = predTemp[:,0]
        pred_test = pred_test*L3 #consequent function output *
normalized weights
        pred_test = np.sum(pred_test,axis=1)
        return pred_test

    @staticmethod
    def rmse(true, pred):
      loss = np.sqrt(np.mean((true - pred)**2))
      return loss

    def
fit(self,X_train,y_train,X_test=None,y_test=None,optimize_test_data
=False):
        generations = self.generations
        offsprings = self.offsprings
        functions = self.functions
        popParam = self.init_population(X_train)
        inputsTrain,AntTrain,L1Train,L2Train =
self.initialize(X_train)
        if optimize_test_data:
            inputsTest,AntTest,L1Test,L2Test =
self.initialize(X_test)
        models = self.init_model()
        bestParam = popParam[0]
        for gen in range(generations):
            parentParam = deepcopy(bestParam)
```

```
            popParam[0] = deepcopy(bestParam)
            for ii in range(1,offsprings):
                mut = self.mutation(parentParam)
                popParam[ii] = deepcopy(mut)

            PopulationError = []
            bestModelLst = []
            for i in range(len(popParam)):
                L1,L2,L3,L4 =
self.forwardPass(popParam[i],X_train,inputsTrain,AntTrain,L1Train,L
2Train,functions)
                pred_train, Trained_models =
self.linear_fit(L3,L4,X_train,y_train,functions,models)
                mse_train = self.rmse(y_train,pred_train)

                if optimize_test_data:
                    L1,L2,L3,L4 =
self.forwardPass(popParam[i],X_test,inputsTest,AntTest,L1Test,L2Tes
t,functions)
                    pred_test =
self.linear_predict(L3,L4,X_test,functions,Trained_models)
                    mse_test = self.rmse(y_test,pred_test)

                    PopulationError.append((mse_train+mse_test)/2)
                    bestModelLst.append(Trained_models)
                else:
                    PopulationError.append(mse_train)
                    bestModelLst.append(Trained_models)

            bestParamIndex = np.argmin(PopulationError)
            bestParam = deepcopy(popParam[bestParamIndex])
            bestModel = bestModelLst[bestParamIndex]
            print(gen,"RMSE is: ",PopulationError[bestParamIndex])
        return bestParam, bestModel

    def predict(self,X,bestParam,bestModel):
        functions = self.functions
        inputs,Ant,L1,L2 = self.initialize(X)
        L1,L2,L3,L4 =
self.forwardPass(bestParam,X,inputs,Ant,L1,L2,functions)
```

```
        pred = self.linear_predict(L3,L4,X,functions,bestModel)
        return pred
from sklearn.model_selection import train_test_split
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from sklearn.preprocessing import MinMaxScaler

from google.colab import drive
drive.mount('/content/drive')

data =
pd.read_csv('/content/drive/MyDrive/Project_IC_Heart/dataset_500.cs
v')
test =
pd.read_csv('/content/drive/MyDrive/Project_IC_Heart/dataset_500.cs
v')
data.head()
Data = data.loc[:,['age','cholesterol','heart_rate', 'diabetes',
'family_history', 'smoking', 'obesity', 'alcohol', 'exercise',
'previous_problems', 'medication', 'stress_level', 'sedentary',
'bmi', 'trigl', 'physical_activity', 'sleep_hour', 'risk']]
test = test.loc[:,['age','cholesterol','heart_rate', 'diabetes',
'family_history', 'smoking', 'obesity', 'alcohol', 'exercise',
'previous_problems', 'medication', 'stress_level', 'sedentary',
'bmi', 'trigl', 'physical_activity', 'sleep_hour', 'risk']]

#Data = data.loc[:,['FC','VM','Ash','Moisture','Cal V']]

# digitizing continuous variable
aa = Data['risk']
minima = aa.min()
maxima = aa.max()
bins = np.linspace(minima-1,maxima+1, 18)
binned = np.digitize(aa, bins)
plt.hist(binned, bins=50)
data_train, data_test = train_test_split(Data, test_size=0.2,

random_state=101,stratify=binned)
```

```
X_train = data_train.drop("risk",axis=1).values
y_train = data_train["risk"].copy().values
X_test = data_test.drop("risk",axis=1).values
y_test = data_test["risk"].copy().values
X_val = test.drop("risk",axis=1).values
y_val = test["risk"].copy().values

scaler_x = MinMaxScaler()
scaler_y = MinMaxScaler()
scaler_x.fit(X_train)
X_train = scaler_x.transform(X_train)
X_test = scaler_x.transform(X_test)
#X_val = scaler_x.transform(X_val)
scaler_y.fit(y_train.reshape(-1,1))
y_train = scaler_y.transform(y_train.reshape(-1,1))
y_test = scaler_y.transform(y_test.reshape(-1,1))
#y_val = scaler_y.transform(y_val.reshape(-1,1))

#from ANFIS import EVOLUTIONARY_ANFIS

E_Anfis =
EVOLUTIONARY_ANFIS(functions=3,generations=500,offsprings=10,

mutationRate=0.2,learningRate=0.2,chance=0.7,ruleComb="simple")

bestParam, bestModel =
E_Anfis.fit(X_train,y_train,optimize_test_data=False)

bestParam, bestModel =
E_Anfis.fit(X_train,y_train,X_test,y_test,optimize_test_data=True)
from scipy.stats import pearsonr
pred_train = E_Anfis.predict(X_train,bestParam,bestModel)
pred_train_flat = pred_train.flatten()

pearsonr(y_train,pred_train_flat.reshape(-1,1))

pred_test = E_Anfis.predict(X_test,bestParam,bestModel)
pearsonr(y_test,pred_test.reshape(-1,1))
```

```
pred_test = E_Anfis.predict(X_test,bestParam,bestModel)
#print(pred_test)
pred_test = sc.inverse_transform(pred_test.reshape(-1,1))
print(pred_test)
# plot loss during training
#pyplot.subplot(211)
#pyplot.title('PRED')
#pyplot.plot(pred_test, label='ANFIS')
#pyplot.legend()


plt.plot(dataset_test['risk'].values[:10], color = 'red', label = 'Actual Risk')
plt.plot(pred_test[0:10], color = 'blue', label = 'Predicted Risk')
plt.title('MSFT Heart Risk Prediction ')
#plt.xlabel('Time')
plt.ylabel('MSFT Heart Risk')
plt.legend()
plt.show()
```