MCTA 3371 Computational Intelligence & MCTE 4322 Intelligent Control

Mini Project: Intelligent Heart Attack Risk Prediction using Computational Intelligence

SEMESTER 1, 23/24
SECTION 1

LECTURER:
DR. AZHAR BIN MOHD IBRAHIM
DR. HASAN FIRDAUS BIN MOHD ZAKI

No	GROUP MEMBERS	MATRIC NO.
1.	Muhammad Amru Bin Mohamad Sharis	2118833
2.	Tengku Muhammad Afnan Faliq bin Tuan Farezuddeen Ahmad	2119705
3.	Muhamad Nurhakimie Thaqif Bin Abdullah	2213217

TABLE OF CONTENTS

Concept Of Adaptive Neuro-Fuzzy Inference System (ANFIS) and Genetic Algorithm (GA) **1**
Design **3**
Design of ANFIS 4
Design of Genetic Algorithm 5
ANFIS and GA Training Code **7**
Results **9**
Using the given dataset 9
Using our dataset 10
Plots 12
Membership Function And Graphical User Interface **14**
Membership Function Code 14
GUI with Membership Functions Code 16
Discussions **24**
Further Improvements 24
Limitations of ANFIS 25
Conclusion **26**
Contribution **26**
Appendix **26**
ANFIS and GA Training Code 26
Fuzzy Membership Code 35
GUI Code 37
Reference **42**

Concept Of Adaptive Neuro-Fuzzy Inference System (ANFIS) and Genetic Algorithm (GA)

In the field of artificial intelligence and machine learning, two computational techniques are utilised: the Adaptive Neuro-Fuzzy Inference System (ANFIS) and the Genetic Algorithm (GA). Despite their disparate functions, they can be combined into a single framework to improve the processes of learning and optimization.

1. **Adaptive Neuro-Fuzzy Inference System (ANFIS):**
   ANFIS is a hybrid intelligent system that combines the capabilities of fuzzy logic and neural networks. It aims to leverage the strengths of both paradigms to create a more powerful system for modelling complex relationships and making decisions based on uncertain or imprecise data.

   The key components of ANFIS include:

   - Fuzzification: Converting crisp input data into fuzzy sets using membership functions.
   - Rule Base: Defining a set of fuzzy rules that capture the relationships between input and output variables.
   - Inference Engine: Combining fuzzy rules with input data to generate fuzzy outputs.
   - Defuzzification: Converting fuzzy outputs into crisp values.

   ANFIS is adaptive because it can adjust its parameters (such as membership function parameters and rule strengths) based on training data, allowing it to learn from examples and improve its performance over time.

2. **Genetic Algorithm (GA)**
   Genetic and evolutionary algorithms, which work based on natural selection (that is, survival of the fittest), have been used as optimization tools.

   The main components of a genetic algorithm include:

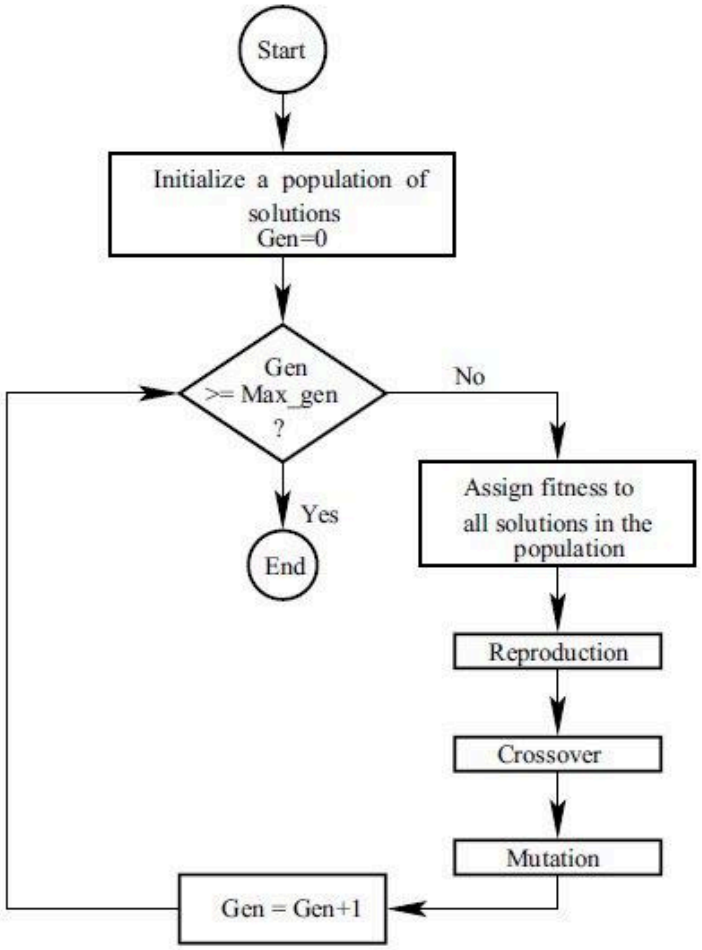


Fig 1) Flowchart of Genetic Algorithm

- Initialization: Creating an initial population of candidate solutions randomly or using heuristics.
- Selection: Choosing individuals from the population for reproduction based on their fitness (i.e., how well they solve the problem).
- Crossover/Reproduction: Creating new offspring by combining genetic material from selected individuals.
- Mutation: Introducing random changes to the genetic material of offspring to maintain diversity in the population.
- Evaluation: Assessing the fitness of each individual in the new population.
- Termination: Stopping the algorithm when a termination criterion is met (e.g., reaching a maximum number of generations or finding a satisfactory solution).

Genetic algorithms are particularly useful for solving optimization problems where traditional methods may be inefficient or impractical.

Design

Our system is designed based on the ANFIS model, with GA being used for training said model. We decided on ANFIS as it has a high degree of interpretability, as we can see how the membership functions process the data from the input and converts them into fuzzy values. It is also adaptable, as its parameters such as membership function parameters and the linear regression parameters can be updated as the system is trained over time. ANFIS is also a very robust system, as it can extract a lot of information from the input during both the fuzzification and defuzzification stage. To train the system, we had the option of either using back-propagation or genetic algorithm. We decided to use genetic algorithm as it provides us both with a large range of values to use, as well as a very complex way of finding the optimal solution. Both of these reasons put genetic algorithm as a much better choice for training our system instead of back-propagation.

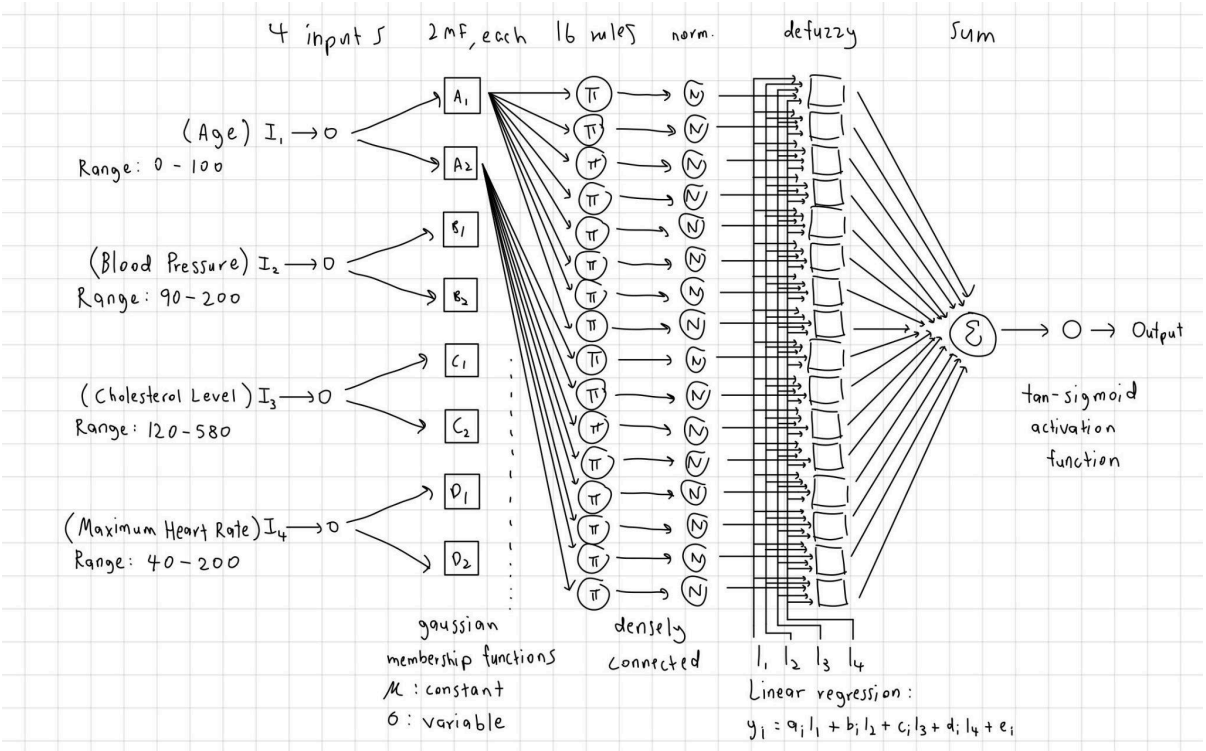


Fig 2) Design of ANFIS used in project

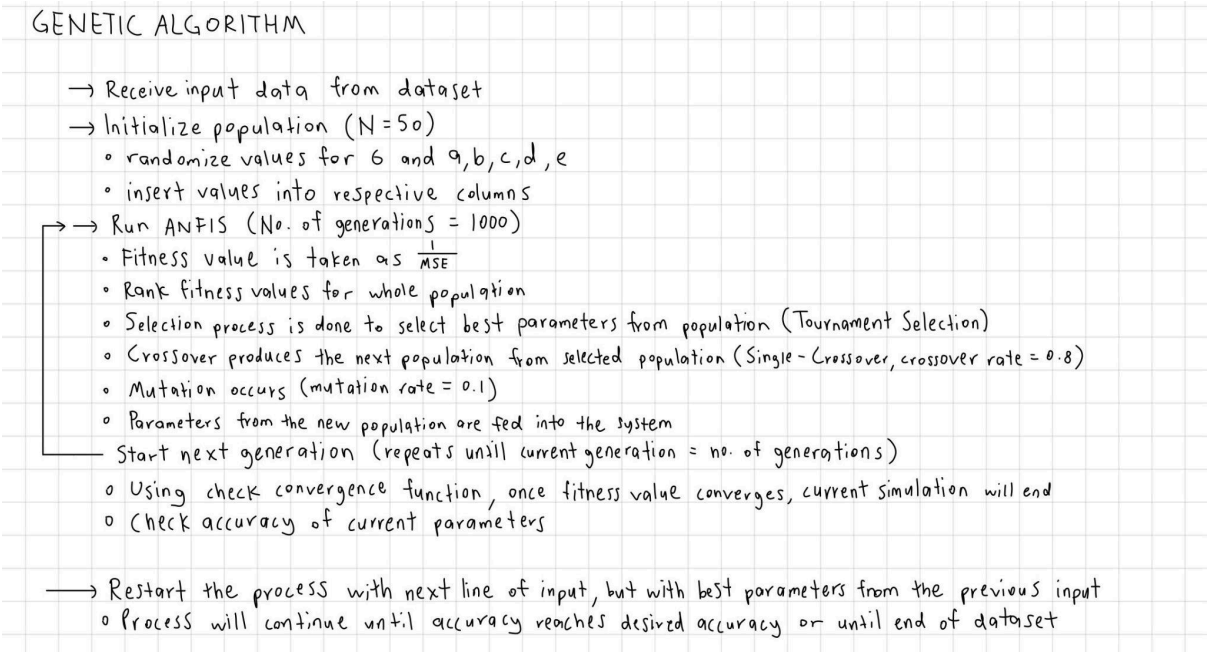


Fig 3) Design of Genetic Algorithm used in project

1. Design of ANFIS

Based on Figure 2, we developed an Adaptive Neuro-Fuzzy Inference System (ANFIS) model to assess the risk of heart attacks, incorporating four key inputs: Age, Blood Pressure, Cholesterol, and Maximum Heart Rate. These inputs were selected due to their significant correlation with heart attack risk, as identified through extensive research. We determined a range of possible values for each input based on the datasets utilized, setting the stage for a comprehensive analysis of how these factors contribute to heart attack risk.

Fuzzification Layer: The first step in our ANFIS model is fuzzification, where the real-world input values are converted into fuzzy values. Utilizing Gaussian membership functions, which are characterized by a constant mean and a variable standard deviation, each input is assigned Low and High membership values.

Rules Layer: After fuzzification, the membership values are processed through the rules layer. This layer applies a predefined set of 16 rules that combine the Low and High states of each input in various permutations. The output of this layer is a set of weights that signify the relevance of each rule based on the current inputs, effectively mapping input combinations to their potential outcomes.

Normalization Layer: Following the application of rules, the normalization layer adjusts the weights generated by the rules layer. It divides each weight by the sum of all weights to ensure that the output values are normalized.

Defuzzification Layer: The defuzzification stage is where the fuzzy logic outcomes are converted back into a crisp value. This is achieved by applying the normalized weights to a linear regression equation based on Takagi-Sugeno Fuzzy Model,
$y = aI1 + bI2 + cI3 + dI4 + e$, I represents input, (a b c d e) are the coefficients.

Summation Layer: The aggregated values from the defuzzification stage are then summed up in the sum layer. This consolidation ensures that the contributions of all inputs and their interactions are accounted for in a singular output, paving the way for the final risk assessment.

Tan-Sigmoid Activation Function: The final step involves passing the summed value through a tan-sigmoid activation function. This nonlinear function maps the preliminary risk score to a value between 0 and 1, providing a normalized and interpretable risk assessment. The output of this function represents the model's final estimation of heart attack risk.

2. Design of Genetic Algorithm

In Figure 3, we designed a genetic algorithm (GA) to optimize the parameters of our ANFIS model. This optimization process involves a series of steps designed to refine the model's accuracy by evolving the parameters over successive generations.

The genetic algorithm flow is as follows:

Population Initiation: The process starts with the initiation of the population, where 50 individuals are generated, each carrying a random set of values for the model's parameters, including the standard deviation and the coefficients (a, b, c, d, e) of the linear regression equation used in the ANFIS model.

Fitness Evaluation: Once the ANFIS model runs with the initial population, each individual's performance is assessed through a fitness evaluation. The fitness function is defined as the inverse of the Mean Squared Error (MSE), where MSE is the squared difference between the target and the actual outputs of the model. This step ranks individuals based on their effectiveness in predicting heart attack risk, with higher fitness scores indicating better performance.

Tournament Selection: Following fitness evaluation, the Tournament Selection method is employed to select superior individuals for reproduction. This process involves randomly selecting a subset of the population and choosing the best performer within this subset to proceed to the next phase. This selection strategy ensures that individuals with higher fitness have a greater chance of contributing their genes to the next generation.

Crossover and Mutation: The selected individuals then undergo crossover and mutation processes. Crossover is a genetic operation that combines the genetic information of two parents to produce one or more offspring, while mutation introduces random changes to an individual's genes.

Iteration and Convergence Check: The genetic algorithm iterates through cycles of selection, crossover, mutation, and fitness evaluation, generating new generations. This process continues until a stopping criterion is met, reaching a maximum number of generations (in this case, 1000), or achieving convergence of fitness values, or once it reaches the fitness limit.

Accuracy Evaluation/Testing: Upon completion of the genetic algorithm for that input, the best parameter's accuracy is evaluated by running it through the ANFIS with the dataset to get an overall accuracy. If the desired accuracy level is achieved, or the training process will stop and the best parameters will be shown. Training can also stop once it completes all lines in the dataset.

This structured approach allows the genetic algorithm to improve the ANFIS model's parameters, enhancing its predictive accuracy through a combination of genetic principles and computational intelligence.

ANFIS and GA Training Code

```
import math
import random
import matplotlib.pyplot as plt
import pandas as pd
import time
from numpy.polynomial import Polynomial

def load_data_from_spreadsheet(filename):...

def gaussian(x, mu, sigma):...

def get_input(prompt, min_val, max_val):...

def tanh(x):...

def calculate_mse(actual_output, calculated_output):...

def fitness(individual, age, blood_pressure, cholesterol, heart_rate, heart_attack_risk):...

def initialize_population(size):...

def tournament_selection(ranked_population, tournament_size=7):...

def crossover(parent1, parent2, crossover_rate=0.8):...

def mutate(population, mutation_rate=0.1):...

def check_convergence(ranked_population, threshold=0.005, generations_to_wait=15, fitness_limit=100):...

1 usage
def genetic_algorithm(row_index, age, blood_pressure, cholesterol, heart_rate, heart_attack_risk, population,
                      elitsm_percentage=0.05, generations=1000):...
```

```
def ANFIS(sd_values, a_values, b_values, c_values, d_values, e_values, age, blood_pressure, cholesterol, heart_rate,
          heart_attack_risk):...

def main(target=70):...

if __name__ == "__main__":
    main()
```

Libraries used

```
import math # for all mathematical operations
import random # used for genetic algorithm
import matplotlib.pyplot as plt # used for plotting graphs
import pandas as pd # used to extract data from excel sheets
import time # used to calculate the time taken for training
from numpy.polynomial import Polynomial # used for plotting best fit
line
```

Functions

1. Main():

Data Loading: It starts by loading the data from an Excel spreadsheet, which contains the input variables and the expected outcome.

Population Initialization: It initializes a population of solutions for the GA.

Chunk Processing: The data is processed in chunks. For each chunk, the GA trains the ANFIS model by optimizing its parameters to minimize the MSE between the predicted and actual outcomes.

Accuracy Evaluation: After processing each chunk, the accuracy of the model is evaluated on the entire dataset. If the set target accuracy is achieved or all inputs are processed, the training stops.

Result Presentation: Finally, it prints the best solution for the parameters and plots the evolution of accuracy and MSE over the generations.

2. Genetic_Algorithm():

Initialization_Population(): A population of individuals is initialized randomly. Each individual represents a set of parameters for the ANFIS model.

Fitness(): The fitness of each individual is evaluated by running the ANFIS() with its parameters and comparing the output to the expected outcome using Mean Squared Error (MSE). The fitness function handles this by calculating the inverse of MSE, where higher fitness values are better.

Tournament_Selection(): Individuals are selected for reproduction. The tournament selection method picks a subset of individuals randomly and then selects the best among them to be parents for the next generation.

Crossover() and Mutate(): The selected individuals undergo crossover and mutation operations to generate offspring that form the new population. These operations introduce variations in the offspring, allowing the GA to explore new areas of the solution space.

Check_Convergence(): The algorithm checks for convergence, meaning it looks for signs that there are no significant improvements on fitness over the generation. If the training has converged or a predefined fitness limit is reached, the training process stops.

Iteration: The previous steps are repeated for a specified number of generations or until convergence and returns the best solution.

3. ANFIS():

Input: The system receives the inputs (age, blood pressure, cholesterol, heart rate), the heart attack risk output, and the parameters all sent by the genetic algorithm function.
Fuzzification: These inputs are then fuzzified to low and high membership functions for each input using Gaussian functions.
Rule Evaluation: The fuzzified inputs are then passed through a set of fuzzy rules, calculating the weights for each rule based on the input membership.
Normalization: The weights are normalized so that their sum equals 1.
Defuzzification: The weighted average of each rule's consequence is defuzzified based on Takagi-Sugeno Fuzzy Model with the given parameters and inputs.
Summation: Sum all the values from the previous layer to get a single output.
Tan Sigmoid Activation Function: This final layer ensures that the final or predicted heart attack output will be between 0 and 1.

Results

For training, we decided to use two different datasets, one is the given dataset from kaggle, and the other is our own dataset which we found. Reason being the first dataset is actually generated by ChatGPT so we wanted to find real datasets from various hospitals to get more accurate results.

a) Using the given dataset

Achieved Accuracy: **76.639943039011%** after processing 164 chunk(s) over the entire dataset

Best Solution Parameters:
sd_values = [82.86044806149356, 94.59707297774989, 80.38621141496755, 65.35595633502379, 73.10170462426169, 31.503327442608057, 93.59235851715809, 35.650718763704816]

a_values = [0.4308396636920141, 0.9646026940026075, 0.8437856739039657, 0.9911780262146339, 0.9263603630513961, 0.7455788040360893, 0.16752449488145715, 0.8168902524298776, 0.5565786903149113, 0.006910675196607374, 0.30902782910568927, 0.8135564603411432, 0.5915682272111582, 0.8233940148344651, 0.5890465778733125, 0.6057335746874732]

b_values = [0.08998601146996998, 0.009912421340854016, 0.30047906687997616, 0.8241165269108038, 0.4681678889986304, 0.14475218188635797, 0.5353899527748768, 0.8613088000884596, 0.061098204730305805, 0.4212565802869014, 0.245870234545567, 0.04647028383772789, 0.8898775448398357, 0.7006334804967587, 0.08880990112378162, 0.6801792460741343]

c_values = [0.05002930958220342, 0.39195628516723524, 0.11644272608130446, 0.5586511459230679, 0.4237022808058043, 0.8287591056717787, 0.6093915875884952, 0.8586351322993112, 0.18053664775669342, 0.08506210056435215, 0.042935010543241225, 0.28438047596599036, 0.31571794577756374, 0.5496829225146349, 0.7515599698599947, 0.9286716052607082]

d_values = [0.3275983189742565, 0.9947559970317937, 0.6212725208917231, 0.005877177249024701, 0.1837032381676944, 0.024652064429272702, 0.9500420732526939, 0.4890131575968222, 0.08166370559924885, 0.6353962208061454, 0.04531131151390155, 0.37280295790904994, 0.5913268102255747, 0.047562361847201085, 0.9806267285859304, 0.4274951374847997]

e_values = [0.7827320811858516, 0.6856060614698414, 0.032202125971867, 0.721768574208083, 0.2662035345929349, 0.008874695401245747, 0.5444872381544111, 0.42564662978877976, 0.2421418995651332, 0.4050613346903509, 0.6882630596055908, 0.09044725974665879, 0.2932156444146502, 0.0006181245105866262, 0.6590179990685061, 0.9175091199629285]

b) Using our dataset

Achieved Accuracy: **79.58867955591279%** after processing 27 chunk(s) over the entire dataset

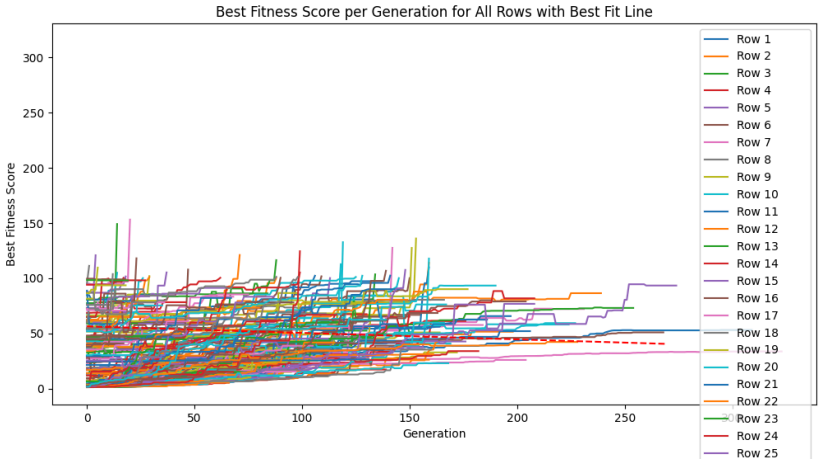
Best Solution Parameters:
sd_values = [30.37425780967358, 98.92323148607927, 31.30203993566636,
88.35776635756176, 71.2218471050781,
43.734075579319246, 67.63996847971873, 30.937387194856644]
a_values = [0.9732031435551686, 0.002688074077581315, 0.7943203599039085,
0.026442885173838748, 0.2738865067022688,
0.3233673717050167, 0.9236541443806265, 0.14600392757227287,
0.43551706252478106, 0.04351124072960244,
0.4098028348551067, 0.06671524745501212, 0.9460516326944615,
0.39275483342725104, 0.043528052946073426,
0.18453543291968943]
b_values = [0.4432199392040913, 0.037194388229018105, 0.2570273196928561,
0.09723134452831073, 0.16818769955303936,
0.25895733891737793, 0.1028899705491454, 0.7859214440275468,
0.04305857294284565, 0.09831302374188688,
0.7597817056067317, 0.4294200092367515, 0.4598972809917097, 0.8499582199532707,
0.8219892834888435,
0.2801227974078886]
c_values = [0.2365433657016054, 0.2001727628887343, 0.961140048391374,
0.10536567826209697, 0.5158219868486323,
0.33785117362711414, 0.6735582930337685, 0.21666303634202333,
0.05949757890955576, 0.14326558874800255,
0.9394102696667682, 0.012812112251490704, 0.7454729540633036,
0.6273417417146013, 0.41563968337136015,
0.8305842718545331]
d_values = [0.06136473359583394, 0.04385758888744107, 0.5255328522142582,
0.008550749064928032, 0.2074167659044439,
0.4497875055211833, 0.3425908291834572, 0.19384831046407147, 0.7434463052710825,
0.07912695990173457,
0.8512781317591239, 0.03912567423121471, 0.9369364752103462,
0.16299795840068443, 0.02071330803936222,

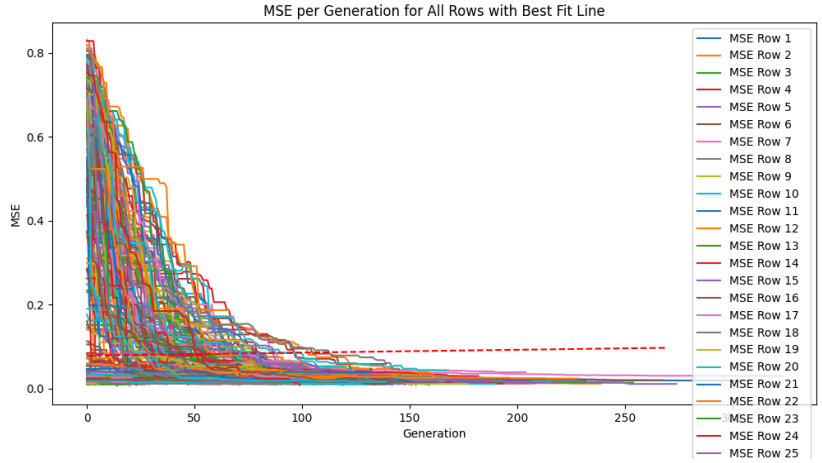
0.3527960988800706]

e_values = [0.23601646711979352, 0.12293345617954032, 0.8580113475436795,
0.01656766283823785, 0.4637885329033492,
0.7931768977953763, 0.15894282957550965, 0.9391566238537904,
0.21034956262735371, 0.027206347898655836, 0.1913517578065863,
0.016095303165399644, 0.8594526302435292, 0.9266191954170502,
0.7592609496608976,
0.08776750692808122]

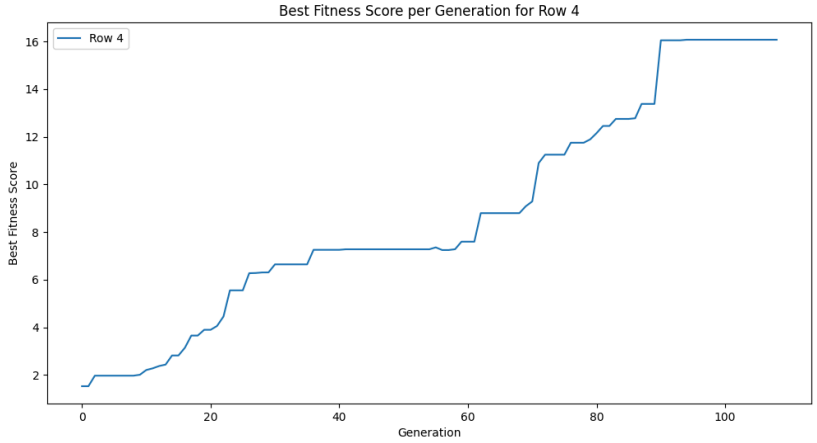
Based on these results, our final parameters chosen will be option b) as it produces the highest accuracy.

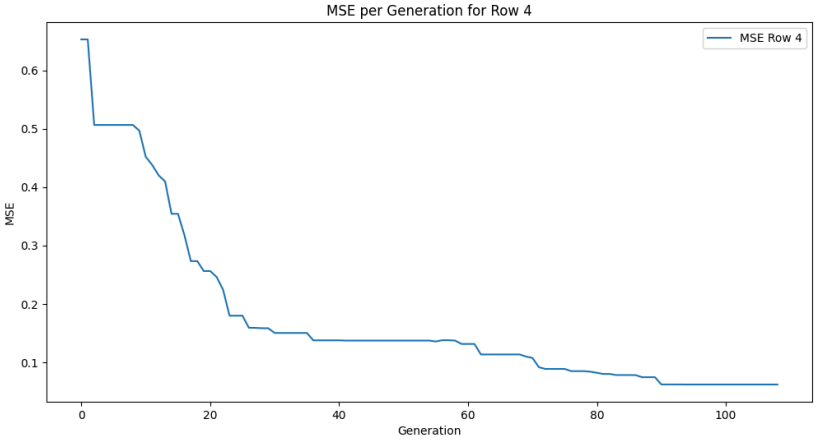
Plots



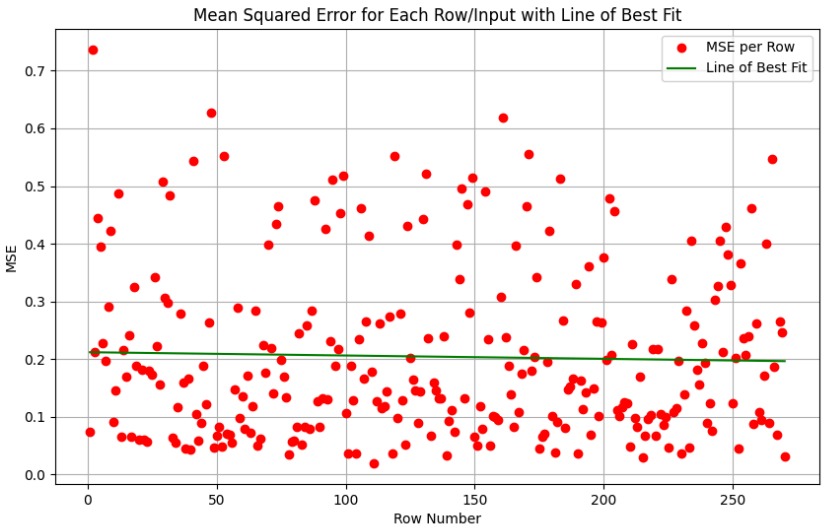


We can observe from the plots above that the fitness value increases and the MSE decreases over the generations, indicating that the genetic algorithm training works.





The plots above are taken from a random input to better visualize.



The mean square error for each input using the best parameters hover mostly around 0.2.

Membership Function And Graphical User Interface

Membership Function Code

```
# Define parameters for low and high regions for each variable
age_low = 0
age_high = 100

blood_pressure_low = 90
blood_pressure_high = 200

cholesterol_low = 120
cholesterol_high = 580

heart_rate_low = 40
heart_rate_high = 200

# Generate fuzzy membership functions using Gaussian membership functions
age_membership_low = fuzz.gaussmf(age, age_low, 30.37425780967358)
age_membership_high = fuzz.gaussmf(age, age_high, 98.92323148607927)

blood_pressure_membership_low = fuzz.gaussmf(blood_pressure, blood_pressure_low, 31.30203993566636)
blood_pressure_membership_high = fuzz.gaussmf(blood_pressure, blood_pressure_high, 88.35776635756176)

cholesterol_membership_low = fuzz.gaussmf(cholesterol, cholesterol_low, 71.2218471050781)
cholesterol_membership_high = fuzz.gaussmf(cholesterol, cholesterol_high, 43.734075579319246)

heart_rate_membership_low = fuzz.gaussmf(heart_rate, heart_rate_low, 67.63996847971873)
heart_rate_membership_high = fuzz.gaussmf(heart_rate, heart_rate_high, 30.937387194856644)
```

Importing Libraries:

- numpy as np: NumPy is used for numerical computations in Python.
- skfuzzy as fuzz: scikit-fuzzy is a fuzzy logic toolkit for Python.
- matplotlib.pyplot as plt : used to plot the fuzzy output

Defining Universe Variables:

- Four arrays are defined to represent the universe of discourse for each variable (age, blood_pressure, cholesterol, heart_rate). These arrays represent the possible range of values for each variable.

```
import numpy as np
import skfuzzy as fuzz
import matplotlib.pyplot as plt

# Define universe variables
age = np.arange(0, 100, 1)
blood_pressure = np.arange(90, 200, 1)
cholesterol = np.arange(120, 580, 1)
heart_rate = np.arange(40, 200, 1)

# Define parameters for low and high regions for each variable
age_low = 0
age_high = 100

blood_pressure_low = 90
blood_pressure_high = 200

cholesterol_low = 120
cholesterol_high = 580

heart_rate_low = 40
heart_rate_high = 200
```

Defining Parameters for Low and High Regions:

- For each variable, low and high values are defined to represent the lower and upper bounds of the membership functions.

Generating Fuzzy Membership Functions:

- Gaussian membership functions (fuzz.gaussmf) are generated for each variable, representing the degree of membership of input values in the fuzzy sets. The parameters passed to ‘gaussmf ’ include the universe variable, mean, and standard deviation

```
# Visualize membership functions
plt.figure(figsize=(12, 10))

plt.subplot(421)
plt.plot(age, age_membership_low, 'b', linewidth=1.5, label='Low')
plt.plot(age, age_membership_high, 'r', linewidth=1.5, label='High')
plt.title('Age')
plt.xlabel('Age')
plt.ylabel('Membership')
plt.legend()

plt.subplot(422)
plt.plot(blood_pressure, blood_pressure_membership_low, 'b', linewidth=1.5, label='Low')
plt.plot(blood_pressure, blood_pressure_membership_high, 'r', linewidth=1.5, label='High')
plt.title('Blood Pressure')
plt.xlabel('Blood Pressure')
plt.ylabel('Membership')
plt.legend()

```

```
plt.subplot(423)
plt.plot(cholesterol, cholesterol_membership_low, 'b', linewidth=1.5, label='Low')
plt.plot(cholesterol, cholesterol_membership_high, 'r', linewidth=1.5, label='High')
plt.title('Cholesterol')
plt.xlabel('Cholesterol')
plt.ylabel('Membership')
plt.legend()

plt.subplot(424)
plt.plot(heart_rate, heart_rate_membership_low, 'b', linewidth=1.5, label='Low')
plt.plot(heart_rate, heart_rate_membership_high, 'r', linewidth=1.5, label='High')
plt.title('Heart Rate')
plt.xlabel('Heart Rate')
plt.ylabel('Membership')
plt.legend()

plt.tight_layout()
plt.show()
```

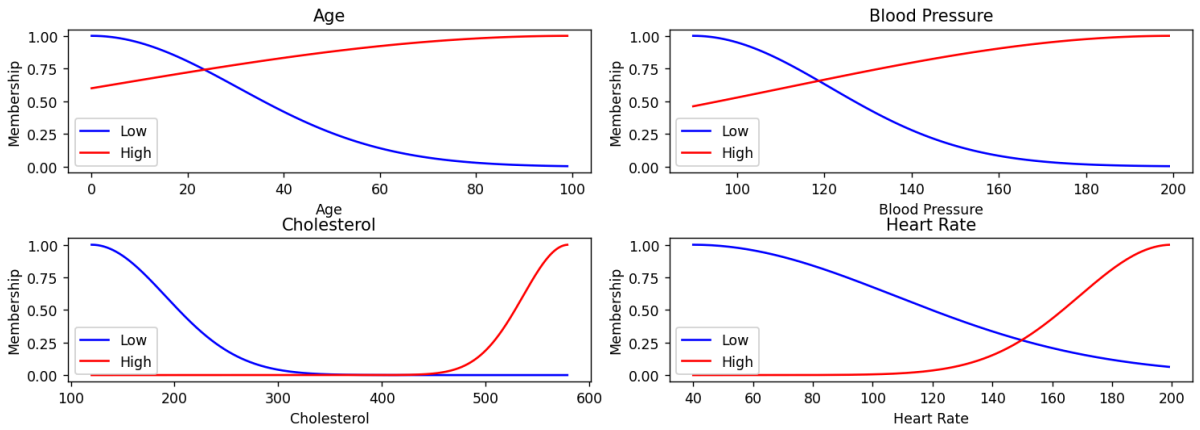
Visualising Membership Functions:

- Using Matplotlib, the code plots the membership functions for each variable.
- Subplots are created for each variable.
- For each subplot, the low and high membership functions are plotted with different colours ('b' for blue representing low, and 'r' for red representing high).
- Titles, labels, and legends are added to the plots for clarity.

Displaying the Plots:

- plt.tight_layout() adjusts the spacing between subplots to prevent overlap.
- plt.show() displays the plots.

This code demonstrates how to define and visualise fuzzy membership functions, which are essential components of fuzzy logic systems. These membership functions represent how each input variable contributes to the fuzzy inference process by assigning degrees of membership to different linguistic terms (e.g., "low," "medium," "high"). Visualising these membership functions helps in understanding how input variables are interpreted in the fuzzy logic system and how they influence the system's output.



Membership Function

GUI with Membership Functions Code

Using this code we implement it onto the Graphical User Input (GUI) for the tester to use.

```python
import numpy as np
import skfuzzy as fuzz
import matplotlib.pyplot as plt
import PySimpleGUI as sg
import math

def gaussian(x, mu, sigma):···

def calculate_mse(actual_output, calculated_output):···

def tanh(x):···

def get_input(prompt, min_val, max_val):···

def user_input():···

def calculate_mse(actual_output, calculated_output):···

def ANFIS(sd_values, a_values, b_values, c_values, d_values, e_values, age, blood_pressure, cholesterol, heart_rate):···
```

We decided to use the library PySimpleGUI a simple Graphical User Interface (GUI) to make a simple GUI for user to input.

```python
def gaussian(x, mu, sigma):
    """
    Calculate the Gaussian function value for a given x, mean (mu), and standard deviation (sigma).
    """
    return math.exp(-0.5 * ((x - mu) / sigma) ** 2)
```

Function gaussian is using the library maths to calculate the gaussian function value with the given mean (mu) and standard deviation (Sigma)

```python
def calculate_mse(actual_output, calculated_output):
    """
    Calculate the Mean Squared Error (MSE) between the actual and calculated outputs.
    """
    return 1 * (actual_output - calculated_output) ** 2\
```

Function Mean Squared Error (MSE) is used to calculate the mean squared error between actual and calculated outputs.

```python
def tanh(x):
    return math.tanh(1/400 * x)

```

Function tanh is used as a hyperbolic tangent activation function to the input x after scaling it by a factor of 1/400.

```python
def get_input(prompt, min_val, max_val):
    """
    Function to get and validate user input.
    """
    while True:
        try:
            value = int(input(prompt))
            if min_val <= value <= max_val:
                return value
            else:
                print(f"Please enter a value between {min_val} and {max_val}.")
        except ValueError:
            print("Invalid input. Please enter a valid number.")

```

get_Input Function is used to get and validate user input.

```python
def user_input():
    age = get_input("Enter your age (0-100): ", 0, 100)
    blood_pressure = get_input("Enter your blood pressure (90-200): ", 90, 200)
    cholesterol = get_input("Enter your cholesterol level (120-580): ", 120, 580)
    heart_rate = get_input("Enter your heart rate (40-200): ", 40, 200)
    heart_attack_risk = get_input("Enter your heart attack risk (0 for no risk, 1 for certain risk): ", 0, 1)

    # Output the gathered information
    print("\nYour inputs:")
    print(
        f"Age: {age}, Blood Pressure: {blood_pressure}, Cholesterol: {cholesterol}, Heart Rate: {heart_rate}, Heart Attack Risk: {heart

    return age, blood_pressure, cholesterol, heart_rate, heart_attack_risk

```

Using the get_Input Function, User_Input function will assign each variable like age, blood pressure, cholesterol, heart rate and heart attack risk after it being validate by get_Input Function.

```python
def calculate_mse(actual_output, calculated_output):
    """
    Calculate the Mean Squared Error (MSE) between the actual and calculated outputs.
    """
    return 0.5 * (actual_output - calculated_output) ** 2\
```

Calculate_mse function is used to calculate the mean error squared error (MSE) between actual and calculated outputs.

```python
def ANFIS(sd_values, a_values, b_values, c_values, d_values, e_values, age, blood_pressure, cholesterol, heart_rate):

    # Specified Mean Values
    mean_values = [0, 100, 90, 200, 120, 580, 40, 200]

    gaussian_parameters = {
        'age': {
            'low': {'mu': mean_values[0], 'sigma': sd_values[0]},
            'high': {'mu': mean_values[1], 'sigma': sd_values[1]}
        },
        'blood_pressure': {
            'low': {'mu': mean_values[2], 'sigma': sd_values[2]},
            'high': {'mu': mean_values[3], 'sigma': sd_values[3]}
        },
        'cholesterol': {
            'low': {'mu': mean_values[4], 'sigma': sd_values[4]},
            'high': {'mu': mean_values[5], 'sigma': sd_values[5]}
        },
        'heart_rate': {
            'low': {'mu': mean_values[6], 'sigma': sd_values[6]},
            'high': {'mu': mean_values[7], 'sigma': sd_values[7]}
        }
    }
```

```python
    # Calculate the membership values
    age_low = gaussian(age, **gaussian_parameters['age']['low'])
    age_high = gaussian(age, **gaussian_parameters['age']['high'])

    bp_low = gaussian(blood_pressure, **gaussian_parameters['blood_pressure']['low'])
    bp_high = gaussian(blood_pressure, **gaussian_parameters['blood_pressure']['high'])

    cholesterol_low = gaussian(cholesterol, **gaussian_parameters['cholesterol']['low'])
    cholesterol_high = gaussian(cholesterol, **gaussian_parameters['cholesterol']['high'])

    heart_rate_low = gaussian(heart_rate, **gaussian_parameters['heart_rate']['low'])
    heart_rate_high = gaussian(heart_rate, **gaussian_parameters['heart_rate']['high'])

```

```python
    # Calculate weights
    weights = {
        'w1': age_low * cholesterol_low * bp_low * heart_rate_low,
        'w2': age_low * cholesterol_low * bp_low * heart_rate_high,
        'w3': age_low * cholesterol_low * bp_high * heart_rate_low,
        'w4': age_low * cholesterol_low * bp_high * heart_rate_high,
        'w5': age_low * cholesterol_high * bp_low * heart_rate_low,
        'w6': age_low * cholesterol_high * bp_low * heart_rate_high,
        'w7': age_low * cholesterol_high * bp_high * heart_rate_low,
        'w8': age_low * cholesterol_high * bp_high * heart_rate_high,
        'w9': age_high * cholesterol_low * bp_low * heart_rate_low,
        'w10': age_high * cholesterol_low * bp_low * heart_rate_high,
        'w11': age_high * cholesterol_low * bp_high * heart_rate_low,
        'w12': age_high * cholesterol_low * bp_high * heart_rate_high,
        'w13': age_high * cholesterol_high * bp_low * heart_rate_low,
        'w14': age_high * cholesterol_high * bp_low * heart_rate_high,
        'w15': age_high * cholesterol_high * bp_high * heart_rate_low,
        'w16': age_high * cholesterol_high * bp_high * heart_rate_high,
    }
```

```
    # Output the calculated weights
    print("\nCalculated weights:")
    for key, value in weights.items():
        print(f"{key}: {value}")

    # Calculate the sum of all weights
    total_weight = sum(weights.values())

    # Normalize the weights
    normalized_weights = {key: value / total_weight for key, value in weights.items()}

    # Output the normalized weights
    print("\nNormalized weights:")
    for key, value in normalized_weights.items():
        print(f"{key}: {value}")

    # Calculate O1 to O16 using the provided equations
    O_values = [normalized_weights[f'w{i}'] * (
                a_values[i - 1] * age + b_values[i - 1] * blood_pressure + c_values[i - 1] * cholesterol + d_values[
            i - 1] * heart_rate + e_values[i - 1]) for i in range(1, 17)]

    # Output O1 to O16
    print("\nOutputs:")
    for i, O in enumerate(O_values, start=1):
        print(f"O{i}: {O}")
```

```
    # Sum up all the output values
    total_output = sum(O_values)

    # Output the sum
    print(f"Total Output: {total_output}")

    tanh_output = tanh(total_output)

    # Sigmoid Output
    print(f"\nOutput: {tanh_output}")

    return tanh_output
```

```
sd_values = [30.37425780967358, 98.92323148607927, 31.30203993566636, 88.35776635756176, 71.2218471050781,
             43.734075579319246, 67.63996847971873, 30.937387194856644]
a_values = [0.9732031435551686, 0.002688074077581315, 0.7943203599039085, 0.026442885173838748, 0.2738865067022688,
            0.3233673717050167, 0.9236541443806265, 0.14600392757227287, 0.43551706252478106, 0.04351124072960244,
            0.4098028348551067, 0.06671524745501212, 0.9460516326944615, 0.39275483342725104, 0.043528052946073426,
            0.18453543291968943]
b_values = [0.4432199392040913, 0.037194388229018105, 0.2570273196928561, 0.09723134452831073, 0.16818769955303936,
            0.25895733891737793, 0.1028899705491454, 0.7859214440275468, 0.04305857294284565, 0.09831302374188688,
            0.7597817056067317, 0.4294200092367515, 0.4598972809917097, 0.8499582199532707, 0.8219892834888435,
            0.2801227974078886]
c_values = [0.2365433657016054, 0.2001727628887343, 0.961140048391374, 0.10536567826209697, 0.5158219868486323,
            0.33785117362711414, 0.6735582930337685, 0.21666303634202333, 0.05949757890955576, 0.14326558874800255,
            0.9394102696667682, 0.012812112251490704, 0.7454729540633036, 0.6273417417146013, 0.41563968337136015,
            0.8305842718545331]
d_values = [0.06136473359583394, 0.043857588887441107, 0.5255328522142582, 0.008550749064928032, 0.2074167659044439,
            0.4497875055211833, 0.3425908291834572, 0.19384831046407147, 0.7434463052710825, 0.07912695990173457,
            0.8512781317591239, 0.03912567423121471, 0.9369364752103462, 0.16299795840068443, 0.02071330803936222,
            0.3527960988800706]
e_values = [0.23601646711979352, 0.12293345617954032, 0.8580113475436795, 0.01656766283823785, 0.4637885329033492,
            0.7931768977953763, 0.15894282957550965, 0.9391566238537904, 0.21034956262735371, 0.027206347898655836,
            0.1913517578065863, 0.016095303165399644, 0.8594526302435292, 0.9266191954170502, 0.7592609496608976,
            0.08776750692808122]
```

```python
# Define universe variables
age = np.arange(0, 100, 1)
blood_pressure = np.arange(90, 200, 1)
cholesterol = np.arange(120, 580, 1)
heart_rate = np.arange(40, 200, 1)

# Define the layout of the GUI
layout = [
    [sg.Text('Enter your age (0-100):'), sg.InputText(key='age')],
    [sg.Text('Enter your blood pressure (90-200):'), sg.InputText(key='bp')],
    [sg.Text('Enter your cholesterol level (120-580):'), sg.InputText(key='ch')],
    [sg.Text('Enter your max heart rate (40-200):'), sg.InputText(key='hr')],
    [sg.Button('Check Heart Attack Risk'), sg.Button('Exit')],
    [sg.Text('Result:'), sg.Text('', size=(10, 1), key='result')],
    [sg.Text('Notes:'), sg.Text('', size=(30, 1), key='notes')]
]

# Create the GUI window
window = sg.Window('Heart Attack Risk Detector - Powered by ANFIS-GA', layout)
```

The ANFIS function takes several parameters including sd_values, a_values, b_values, c_values, d_values, e_values, age, blood_pressure, cholesterol, and heart_rate. This is using the best value that we had after the training of the ANFIS.

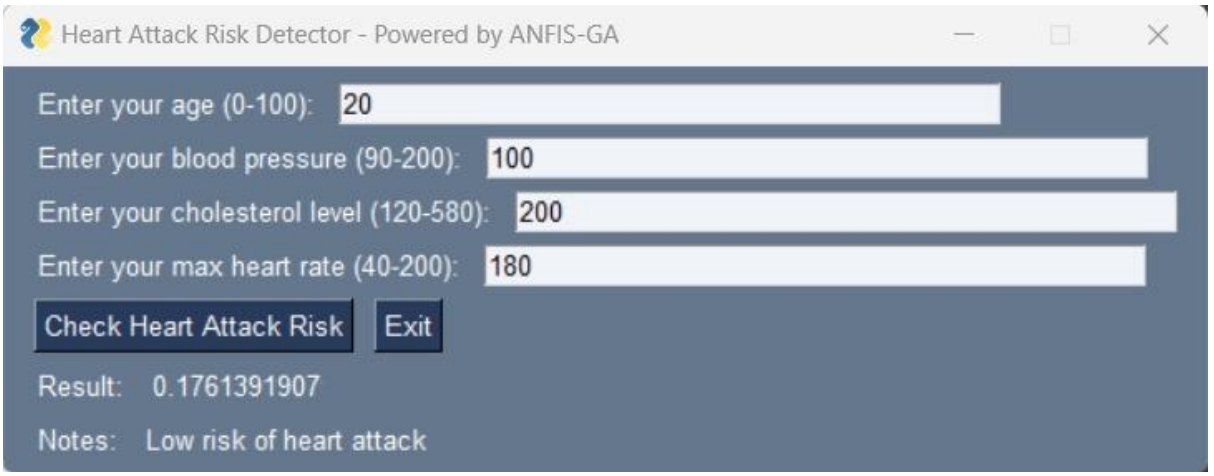
It calculates membership values and weights based on the input parameters. It then prints out the calculated weights, normalises them, computes outputs using provided equations, sums up the output values, applies the hyperbolic tangent function (tanh), and finally returns the tanh output.

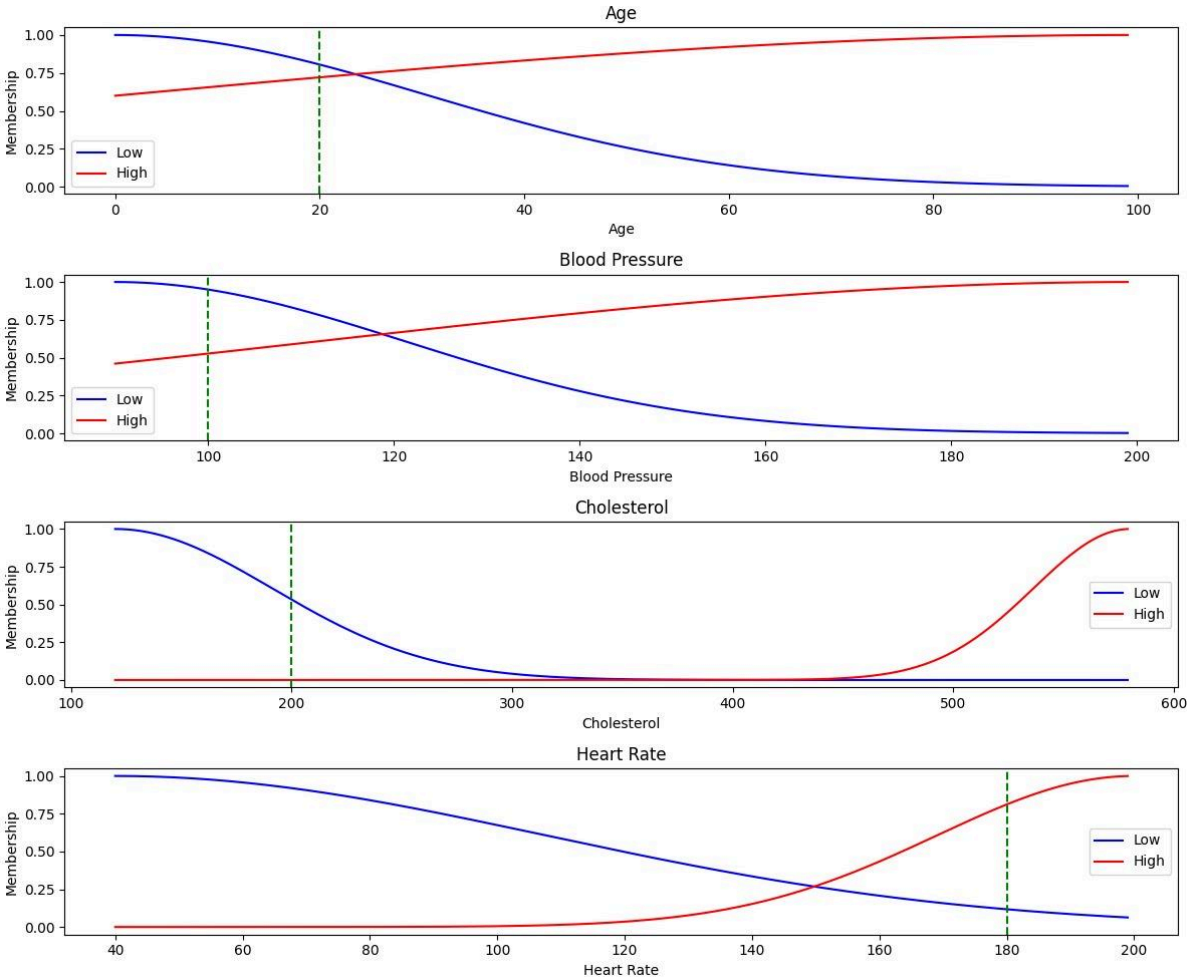
And then it is going to loop where it waits for user interactions with the GUI window. It listens for button clicks and responds accordingly. When the "Check Heart Attack Risk" button is clicked, the input values are collected, and the ANFIS function is called to compute the heart attack risk. The result is then displayed in the GUI window.

```
234 > def visualize_fuzzy(age_val, bp_val, ch_val, hr_val): ···
270
271   # Event loop to process events and get values from the GUI
272   while True:
273       event, values = window.read()
274       if event == sg.WINDOW_CLOSED or event == 'Exit':
275           break
276       if event == 'Check Heart Attack Risk':
277           try:
278               age_val = float(values['age'])
279               bp_val = float(values['bp'])
280               ch_val = float(values['ch'])
281               hr_val = float(values['hr'])
282               output = ANFIS(sd_values, a_values, b_values, c_values, d_values, e_values
283               if output<0.3:
284                   notes = "Low risk of heart attack"
285               elif output>0.3 and output<0.7:
286                   notes = "Moderate risk of heart attack"
287               else:
288                   notes = "High risk of heart attack"
289
290               visualize_fuzzy(age_val, bp_val, ch_val, hr_val)
291
292               window['result'].update(output)
293               window['notes'].update(notes)
294           except ValueError:
295               sg.popup_error('Please enter valid numbers.')
296
297   # Close the GUI window
298   window.close()
299
```

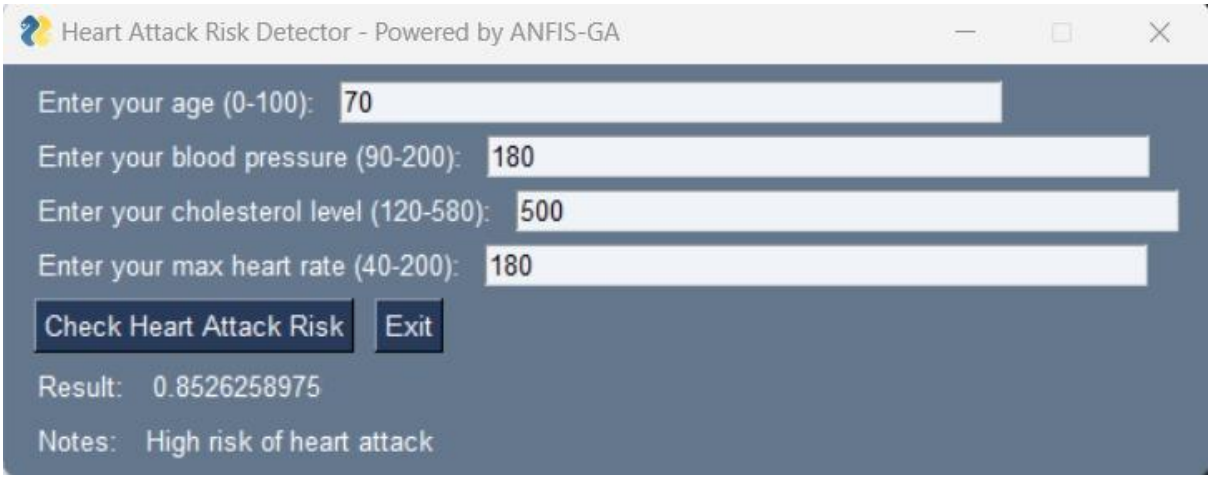
Visualize_fuzzy is a function to show the user input to visualise it into the membership function

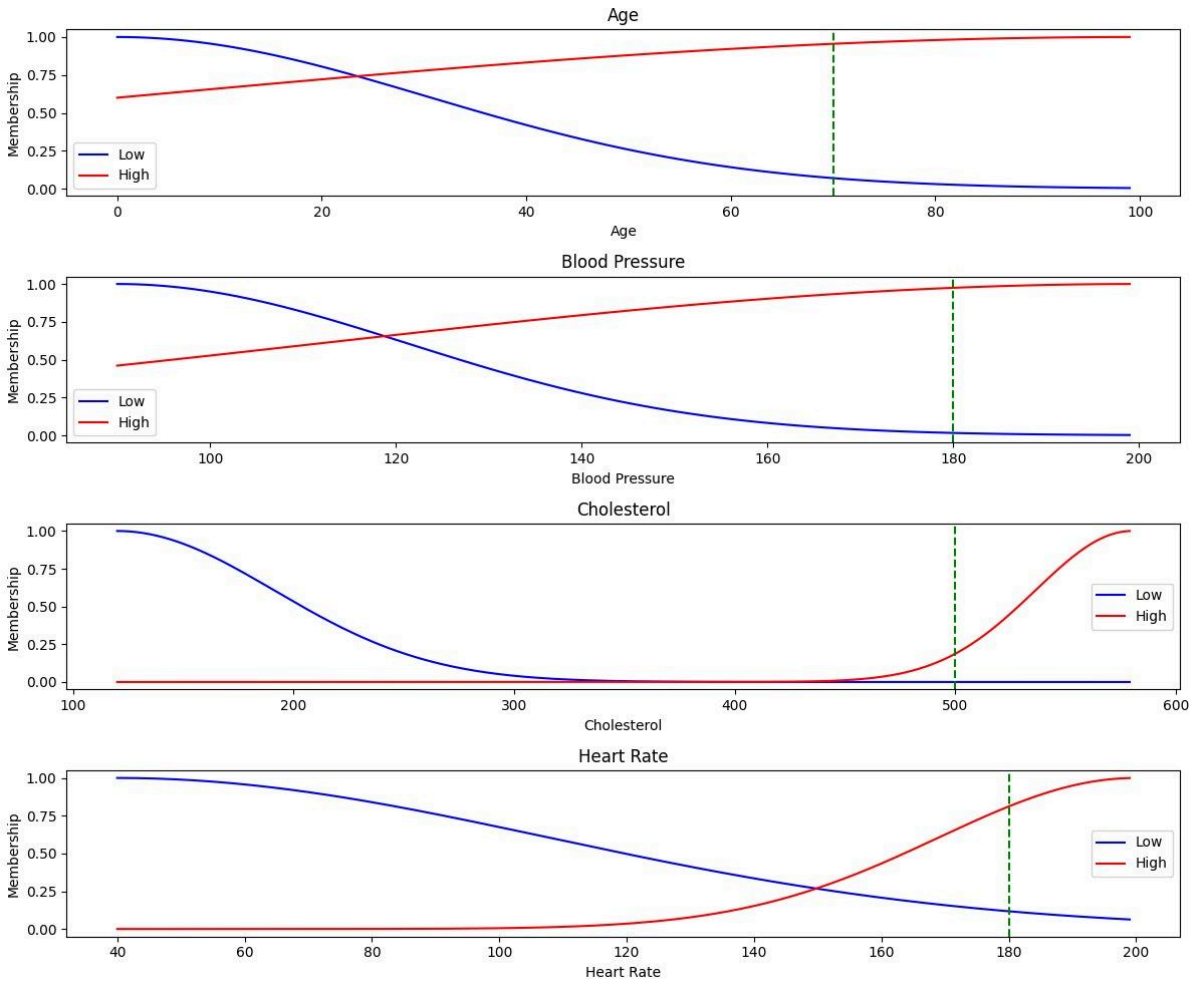
As an example,





Low heart Risk Example





High heart risk example

Discussions

Further Improvements

From the results of our project, we found some possible changes that could be made for future improvements.

1. Increase the number of inputs

In our project, we focused on the 4 most important factors in determining the risk of heart attack. However, we could further increase the accuracy of the system by adding more input variables, such as gender, medical conditions such as diabetes or obesity, as well as risk factors such as smoking. By adding these additional inputs, our system could become even more powerful in predicting the risk of heart attack.

2. Using more datasets

We were only able to use 2 datasets in our project. If we used even more datasets, it could drastically improve our accuracy as we will have more data to train the system.

Limitations of ANFIS

ANFIS, while a powerful approach in certain contexts, comes with limitations that hinder its widespread adoption in various machine learning and deep learning applications.

1. Computational Intensity
One of the primary limitations of ANFIS lies in its computational demands. ANFIS requires significant computational resources due to the intricate processes of both fuzzification and defuzzification of input data. This necessity for extensive computation can become a bottleneck, especially when dealing with large datasets or complex systems.

2. Exponential Complexity with Additional Inputs
The complexity of ANFIS increases exponentially with the addition of each input variable. When incorporating additional inputs, the need to design and adjust membership functions for fuzzification and defuzzification amplifies significantly. For each input, multiple membership functions need to be defined, leading to a rapid growth in computational requirements and model complexity.

3. Escalating Rules Layer Complexity
The rules layer in ANFIS adds another layer of complexity, particularly as the number of inputs increases. With each input variable, the number of rules grows exponentially. This exponential growth in rules can quickly overwhelm the system, making it challenging to manage and optimise effectively.

4. Increased Complexity in Linear Regression Equations
Expanding the number of inputs in ANFIS also escalates the complexity of the linear regression equations. Each additional input necessitates the inclusion of more variables in the equations, leading to longer and more intricate formulations. As the number of inputs grows, the complexity of the regression equations becomes a significant impediment to scalability and efficiency.

Comparison with Neural Networks
In comparison to neural network models, ANFIS exhibits distinct limitations regarding scalability and computational efficiency. While ANFIS struggles with exponential increases in complexity with additional inputs, neural networks offer a more linear scalability in terms of the number of inputs and associated weights. This characteristic makes neural networks more adaptable to diverse datasets and applications, often outperforming ANFIS in terms of accuracy and computational efficiency.

While ANFIS remains a valuable tool in certain domains, its limitations have contributed to its relatively limited adoption compared to other machine learning models, such as neural networks. Understanding these limitations is crucial for effectively leveraging ANFIS and exploring alternative approaches in machine learning and artificial intelligence research and applications.

Conclusion

In conclusion, we successfully designed an ANFIS-GA model to predict heart attack risk based on age, blood pressure, cholesterol level and maximum heart rate. From our results, our model is quite accurate with the maximum accuracy found to be 79.58%. We have learnt a lot throughout this process and hope to apply our knowledge gained in the future.

Contribution

As a team, we developed the ANFIS-GA together and divided individual tasks for parts of the coding.

AMRU (2118833)
Research on NN, Training and Optimization of ANFIS

AFNAN (2119705)
Research on GA, Developed code for the GUI

KIMI (2213217)
Research on FUZZY, Developed code for the graphs of membership function

Appendix

ANFIS and GA Training Code

```
import math
import random
import matplotlib.pyplot as plt
import pandas as pd
import time

def load_data_from_spreadsheet(filename):
    """
    Load data from Excel file using Pandas.
    """
    data = pd.read_excel(filename)
    return data

def gaussian(x, mu, sigma):
    """
    Calculate the Gaussian function value for a given x, mean, and standard
deviation.
    """
    return math.exp(-0.5 * ((x - mu) / sigma) ** 2)

def get_input(prompt, min_val, max_val):
    """
    Function to get and validate user input.
```

```
    """
    while True:
        try:
            value = int(input(prompt))
            if min_val <= value <= max_val:
                return value
            else:
                print(f"Please enter a value between {min_val} and
{max_val}.")
        except ValueError:
            print("Invalid input. Please enter a valid number.")

def tanh(x):
    """
    Tan Sigmoid function used as activation function to get output between
0-1.
    """
    return math.tanh(1 / 400 * x)

def calculate_mse(actual_output, calculated_output):
    """
    Calculate the Mean Squared Error (MSE) between the actual and predicted
output.
    """
    return 1 * (actual_output - calculated_output) ** 2\

def fitness(individual, age, blood_pressure, cholesterol, heart_rate,
heart_attack_risk):
    """
    Evaluate fitness of population by running through the ANFIS to get MSE.
    """

    sd_values = individual['sd_values']
    a_values = individual['a_values']
    b_values = individual['b_values']
    c_values = individual['c_values']
    d_values = individual['d_values']
    e_values = individual['e_values']

    mse = ANFIS(sd_values, a_values, b_values, c_values, d_values, e_values,
age, blood_pressure, cholesterol, heart_rate, heart_attack_risk)
    if mse == 0:
        return 99999
    else:
        return abs(1 / mse)

def initialize_population(size):
    """
    Randomly initialize population within the set range.
    """
    population = []
    for _ in range(size):
        individual = {
```

```
            'sd_values': [random.uniform(30, 100) for _ in range(8)],
            'a_values': [random.uniform(0, 1) for _ in range(16)],
            'b_values': [random.uniform(0, 1) for _ in range(16)],
            'c_values': [random.uniform(0, 1) for _ in range(16)],
            'd_values': [random.uniform(0, 1) for _ in range(16)],
            'e_values': [random.uniform(0, 1) for _ in range(16)]
        }
        population.append(individual)
    return population

def tournament_selection(ranked_population, tournament_size=7):
    """
    Selection method used is Tournament Selection.
    """
    new_population = []
    while len(new_population) < len(ranked_population):
        tournament = random.sample(ranked_population, tournament_size)
        winner = sorted(tournament, key=lambda x: x[0], reverse=True)[0][1]
# Select the best individual
        new_population.append(winner)
    return new_population

def crossover(parent1, parent2, crossover_rate=0.8):
    """
    Perform single crossover.
    """
    # Create deep copies of the parents to avoid modifying the original
parents
    child1, child2 = parent1.copy(), parent2.copy()

    # Check if crossover should occur
    if random.random() <= crossover_rate:
        # Perform crossover for each list of parameters
        for key in ['sd_values', 'a_values', 'b_values', 'c_values',
'd_values', 'e_values']:
            if len(parent1[key]) > 1:  # Ensure there's something to
crossover
                crossover_point = random.randint(1, len(parent1[key]) - 1)
                child1[key] = parent1[key][:crossover_point] +
parent2[key][crossover_point:]
                child2[key] = parent2[key][:crossover_point] +
parent1[key][crossover_point:]

    return child1, child2

def mutate(population, mutation_rate=0.1):
    """
    Perform Mutation.
    """
    for individual in population:
        if random.random() < mutation_rate:
            param_to_mutate = random.choice(
```

```
                ['sd_values', 'a_values', 'b_values', 'c_values', 'd_values',
'e_values'])
            num_mutations = random.randint(1, 3)

            for _ in range(num_mutations):
                # Choose a random index to mutate
                index_to_mutate = random.randint(0,
len(individual[param_to_mutate]) - 1)
                # Apply mutation based on the parameter's specific range
                if param_to_mutate == 'sd_values':
                    mutation_value = random.uniform(30, 100)
                else:  # a_values, b_values, c_values, d_values, e_values
                    mutation_value = random.uniform(0, 1)

                individual[param_to_mutate][index_to_mutate] = mutation_value

    return population

def check_convergence(ranked_population, threshold=0.005,
generations_to_wait=15, fitness_limit=100):
    """
    If the fitness is converged or reached fitness limit, genetic algorithm
loop breaks.
    """
    # `convergence_history` is a list storing the best fitness of the last N
generations
    global convergence_history
    current_best_fitness = ranked_population[0][0]
    convergence_history.append(current_best_fitness)

    if len(convergence_history) > generations_to_wait:
        convergence_history.pop(0)
        # Check if the improvement over the last N generations is less than
the threshold
        if max(convergence_history) - min(convergence_history) < threshold:
            return True  # Converged

    # Check if fitness limit is reached
    if fitness_limit is not None and current_best_fitness >= fitness_limit:
        return True  # Converged

    return False  # Not converged yet

def genetic_algorithm(row_index, age, blood_pressure, cholesterol,
heart_rate, heart_attack_risk, population, elitsm_percentage=0.05,
generations=1000):
    """
    Genetic algorithm used to train the ANFIS.
    """

    global convergence_history
    convergence_history = []
    elitism_size = max(1, int(len(population) * elitsm_percentage))
```

```
    # Setup plotting
    fitness_values = []
    mse_values = []
    print("\n\n")

    print(f"Row: {row_index + 1}, Inputs: Age: {age}, BP: {blood_pressure},
Chol: {cholesterol}, HR: {heart_rate}, Risk: {heart_attack_risk}")

    for generation in range(generations):
        # Evaluate fitness
        ranked_population = [
            (fitness(individual, age, blood_pressure, cholesterol,
heart_rate, heart_attack_risk), individual) for
            individual in population]
        ranked_population.sort(key=lambda x: x[0], reverse=True)  # Assuming
higher fitness is better

        # Extract the elite individuals
        elites = [individual for _, individual in
ranked_population[:elitism_size]]

        # Extract the best MSE from the current generation
        best_fitness = ranked_population[0][0]
        fitness_values.append(best_fitness)
        mse_for_generation = 1 / best_fitness
        mse_values.append(mse_for_generation)
        print(f"Best fitness at generation {generation}: {best_fitness}")

        # Selection
        selected_population =
tournament_selection(ranked_population[elitism_size:])

        # Crossover
        new_population = elites[:]
        while len(new_population) < len(population):
            parent1, parent2 = random.sample(selected_population, 2)
            child1, child2 = crossover(parent1, parent2)
            new_population.extend([child1, child2])

        # Ensure the population does not exceed the intended size due to
addition of elite individuals
        new_population = new_population[:len(population)]

        # Mutation
        population = mutate(new_population)

        # convergence check to break the loop early
        if check_convergence(ranked_population):
            break

    # Return the best solution found
    best_solution = ranked_population[0][1]
```

```
    return best_solution, population, fitness_values, mse_values

def ANFIS(sd_values, a_values, b_values, c_values, d_values, e_values, age,
blood_pressure, cholesterol, heart_rate, heart_attack_risk):
    """
    ANFIS
    """

    # Specified Mean Values
    mean_values = [0, 100, 90, 200, 120, 580, 40, 200]

    gaussian_parameters = {
        'age': {
            'low': {'mu': mean_values[0], 'sigma': sd_values[0]},
            'high': {'mu': mean_values[1], 'sigma': sd_values[1]}
        },
        'blood_pressure': {
            'low': {'mu': mean_values[2], 'sigma': sd_values[2]},
            'high': {'mu': mean_values[3], 'sigma': sd_values[3]}
        },
        'cholesterol': {
            'low': {'mu': mean_values[4], 'sigma': sd_values[4]},
            'high': {'mu': mean_values[5], 'sigma': sd_values[5]}
        },
        'heart_rate': {
            'low': {'mu': mean_values[6], 'sigma': sd_values[6]},
            'high': {'mu': mean_values[7], 'sigma': sd_values[7]}
        }
    }

    # Calculate the membership values
    age_low = gaussian(age, **gaussian_parameters['age']['low'])
    age_high = gaussian(age, **gaussian_parameters['age']['high'])

    bp_low = gaussian(blood_pressure,
**gaussian_parameters['blood_pressure']['low'])
    bp_high = gaussian(blood_pressure,
**gaussian_parameters['blood_pressure']['high'])

    cholesterol_low = gaussian(cholesterol,
**gaussian_parameters['cholesterol']['low'])
    cholesterol_high = gaussian(cholesterol,
**gaussian_parameters['cholesterol']['high'])

    heart_rate_low = gaussian(heart_rate,
**gaussian_parameters['heart_rate']['low'])
    heart_rate_high = gaussian(heart_rate,
**gaussian_parameters['heart_rate']['high'])

    # Calculate weights
    weights = {
        'w1': age_low * cholesterol_low * bp_low * heart_rate_low,
        'w2': age_low * cholesterol_low * bp_low * heart_rate_high,
```

```
        'w3': age_low * cholesterol_low * bp_high * heart_rate_low,
        'w4': age_low * cholesterol_low * bp_high * heart_rate_high,
        'w5': age_low * cholesterol_high * bp_low * heart_rate_low,
        'w6': age_low * cholesterol_high * bp_low * heart_rate_high,
        'w7': age_low * cholesterol_high * bp_high * heart_rate_low,
        'w8': age_low * cholesterol_high * bp_high * heart_rate_high,
        'w9': age_high * cholesterol_low * bp_low * heart_rate_low,
        'w10': age_high * cholesterol_low * bp_low * heart_rate_high,
        'w11': age_high * cholesterol_low * bp_high * heart_rate_low,
        'w12': age_high * cholesterol_low * bp_high * heart_rate_high,
        'w13': age_high * cholesterol_high * bp_low * heart_rate_low,
        'w14': age_high * cholesterol_high * bp_low * heart_rate_high,
        'w15': age_high * cholesterol_high * bp_high * heart_rate_low,
        'w16': age_high * cholesterol_high * bp_high * heart_rate_high,
    }

    # Calculate the sum of all weights
    total_weight = sum(weights.values())

    # Return infinity if weight is 0
    if total_weight == 0:
        return float('inf')

    # Normalize the weights
    normalized_weights = {key: value / total_weight for key, value in
weights.items()}

    # Calculate O1 to O16 using Sugeno Fuzzy
    O_values = [normalized_weights[f'w{i}'] * (
            a_values[i - 1] * age + b_values[i - 1] * blood_pressure +
c_values[i - 1] * cholesterol + d_values[
        i - 1] * heart_rate + e_values[i - 1]) for i in range(1, 17)]

    # Sum up all the output values
    total_output = sum(O_values)

    # Calculate the tanh/tan sigmoid of total_output
    tanh_output = tanh(total_output)

    # Calculate the MSE using the actual and predicted output
    mse = calculate_mse(heart_attack_risk, tanh_output)

    return mse

def main(target=70):
    """
    Main function to glue everything
    Reads data from given Excel Dataset
    Training will stop until it reaches accuracy target per chunk or no
inputs left
    Once finished, will return the best parameters for the ANFIS
    """
    # Load Excel file start time
```

```
    start_time = time.time()
    population = initialize_population(50)
    filename = r"FILE PATH"
    data = load_data_from_spreadsheet(filename)

    chunk_size = 10
    total_rows = len(data)
    best_solution = None
    best_accuracy = 0
    chunks_processed = 0
    fitness_scores_all_rows = []
    mse_scores_all_rows = []
    accuracies_all_chunks = []

    for start_row in range(0, total_rows, chunk_size):
        end_row = min(start_row + chunk_size, total_rows)
        print(f"\nProcessing rows {start_row + 1} to {end_row}...")
        chunks_processed += 1

        for index, row in data.iloc[start_row:end_row].iterrows():
            # Extract data from the row
            age, blood_pressure, cholesterol, heart_rate, heart_attack_risk =
row["Age"], row["Blood Pressure"], row[
                "Cholesterol"], row["Max HR"], row["Heart Disease"]

            # Calling Genetic Algorithm function
            best_solution, population, fitness_value, mse_value =
genetic_algorithm(index, age, blood_pressure,

cholesterol, heart_rate,

heart_attack_risk, population)
            fitness_scores_all_rows.append(fitness_value)
            mse_scores_all_rows.append(mse_value)

        # After updating the model with the current chunk, evaluate it on the
entire dataset
        mse_values = []  # Reinitialize mse_values for the entire dataset
evaluation
        for index, row in data.iterrows():
            # Calculate MSE for each row in the entire dataset
            mse = ANFIS(best_solution['sd_values'],
best_solution['a_values'], best_solution['b_values'],
                        best_solution['c_values'], best_solution['d_values'],
best_solution['e_values'], row["Age"],
                        row["Blood Pressure"], row["Cholesterol"], row["Max
HR"], row["Heart Disease"])
            mse_values.append(mse)

        # Calculate accuracy for the entire dataset up to this point
        average_mse = sum(mse_values) / len(mse_values)
        current_accuracy = 100 * (1 - average_mse)
```

```
        print(f"Current Accuracy after {chunks_processed} chunk(s) over the
entire dataset: {current_accuracy}%")

        accuracies_all_chunks.append(current_accuracy)

        if current_accuracy > best_accuracy:
            best_accuracy = current_accuracy

        # Stop if accuracy target reached
        if current_accuracy > target:
            print(
                f"Achieved desired accuracy after processing
{chunks_processed} chunk(s) over the entire dataset. Stopping the
training.")
            break

    # Print best solution if it exists and if any accuracy improvement was
noted
    if best_solution and best_accuracy > 0:
        print(
            f"\nAchieved Accuracy: {best_accuracy}% after processing
{chunks_processed} chunk(s) over the entire dataset")
        print("Best Solution Parameters:")
        for key, value in best_solution.items():
            print(f"{key}:")
            if isinstance(value, list):
                for i, val in enumerate(value):
                    print(f"    {i + 1}: {val}")
            else:
                print(f"    {value}")

    # Print total time taken
    end_time = time.time()
    total_time = end_time - start_time
    print(f"\nTotal runtime: {total_time} seconds")

    # Plot Accuracy After Each Chunk
    plt.figure(figsize=(12, 6))
    plt.plot(accuracies_all_chunks, marker='o', linestyle='-', color='b')
    plt.xlabel('Chunk Number')
    plt.ylabel('Accuracy')
    plt.title('Accuracy After Each Chunk')
    plt.xticks(range(len(accuracies_all_chunks)), [f'Chunk {i + 1}' for i in
range(len(accuracies_all_chunks))])
    plt.show()

    # Plotting MSE over generations for each row
    for i, mse_scores in enumerate(mse_scores_all_rows):
        plt.figure(figsize=(12, 6))
        plt.plot(mse_scores, label=f'MSE Row {i + 1}')
        plt.xlabel('Generation')
        plt.ylabel('MSE')
        plt.title(f'MSE per Generation for Row {i + 1}')
```

```
        plt.legend()
        plt.show()

    # Plotting Fitness Score over generations for each row
    for i, fitness_scores in enumerate(fitness_scores_all_rows):
        plt.figure(figsize=(12, 6))
        plt.plot(fitness_scores, label=f'Row {i + 1}')
        plt.xlabel('Generation')
        plt.ylabel('Best Fitness Score')
        plt.title(f'Best Fitness Score per Generation for Row {i + 1}')
        plt.legend()
        plt.show()

if __name__ == "__main__":
    main()
```

Fuzzy Membership Code

```
import numpy as np
import skfuzzy as fuzz
import matplotlib.pyplot as plt

# Define universe variables
age = np.arange(0, 100, 1)
blood_pressure = np.arange(90, 200, 1)
cholesterol = np.arange(120, 580, 1)
heart_rate = np.arange(40, 200, 1)

# Define parameters for low and high regions for each variable
age_low = 0
age_high = 100

blood_pressure_low = 90
blood_pressure_high = 200

cholesterol_low = 120
cholesterol_high = 580

heart_rate_low = 40
heart_rate_high = 200

# Generate fuzzy membership functions using Gaussian membership functions
age_membership_low = fuzz.gaussmf(age, age_low, 30.37425780967358)
age_membership_high = fuzz.gaussmf(age, age_high, 98.92323148607927)

blood_pressure_membership_low = fuzz.gaussmf(blood_pressure,
blood_pressure_low, 31.302039935666636)
blood_pressure_membership_high = fuzz.gaussmf(blood_pressure,
blood_pressure_high, 88.35776635756176)

cholesterol_membership_low = fuzz.gaussmf(cholesterol, cholesterol_low,
71.2218471050781)
cholesterol_membership_high = fuzz.gaussmf(cholesterol, cholesterol_high,
43.734075579319246)
```

```
heart_rate_membership_low = fuzz.gaussmf(heart_rate, heart_rate_low,
67.63996847971873)
heart_rate_membership_high = fuzz.gaussmf(heart_rate, heart_rate_high,
30.937387194856644)

# Visualize membership functions
plt.figure(figsize=(12, 10))

plt.subplot(421)
plt.plot(age, age_membership_low, 'b', linewidth=1.5, label='Low')
plt.plot(age, age_membership_high, 'r', linewidth=1.5, label='High')
plt.title('Age')
plt.xlabel('Age')
plt.ylabel('Membership')
plt.legend()

plt.subplot(422)
plt.plot(blood_pressure, blood_pressure_membership_low, 'b', linewidth=1.5,
label='Low')
plt.plot(blood_pressure, blood_pressure_membership_high, 'r', linewidth=1.5,
label='High')
plt.title('Blood Pressure')
plt.xlabel('Blood Pressure')
plt.ylabel('Membership')
plt.legend()

plt.subplot(423)
plt.plot(cholesterol, cholesterol_membership_low, 'b', linewidth=1.5,
label='Low')
plt.plot(cholesterol, cholesterol_membership_high, 'r', linewidth=1.5,
label='High')
plt.title('Cholesterol')
plt.xlabel('Cholesterol')
plt.ylabel('Membership')
plt.legend()

plt.subplot(424)
plt.plot(heart_rate, heart_rate_membership_low, 'b', linewidth=1.5,
label='Low')
plt.plot(heart_rate, heart_rate_membership_high, 'r', linewidth=1.5,
label='High')
plt.title('Heart Rate')
plt.xlabel('Heart Rate')
plt.ylabel('Membership')
plt.legend()

plt.tight_layout()
plt.show()
```

GUI Code

```
import numpy as np
import skfuzzy as fuzz
```

```
import matplotlib.pyplot as plt
import PySimpleGUI as sg
import math

def gaussian(x, mu, sigma):
    """
    Calculate the Gaussian function value for a given x, mean, and standard
deviation.
    """
    return math.exp(-0.5 * ((x - mu) / sigma) ** 2)

def calculate_mse(actual_output, calculated_output):
    """
    Calculate the Mean Squared Error (MSE) between the actual and predicted
output.
    """
    return 1 * (actual_output - calculated_output) ** 2\

def tanh(x):
    """
    Tan Sigmoid function used as activation function to get output between
0-1.
    """
    return math.tanh(1 / 400 * x)

def calculate_mse(actual_output, calculated_output):
    """
    Calculate the Mean Squared Error (MSE) between the actual and predicted
output.
    """
    return 1 * (actual_output - calculated_output) ** 2\

def ANFIS(sd_values, a_values, b_values, c_values, d_values, e_values, age,
blood_pressure, cholesterol, heart_rate):
    """
    ANFIS
    """

    # Specified Mean Values
    mean_values = [0, 100, 90, 200, 120, 580, 40, 200]

    gaussian_parameters = {
        'age': {
            'low': {'mu': mean_values[0], 'sigma': sd_values[0]},
            'high': {'mu': mean_values[1], 'sigma': sd_values[1]}
        },
        'blood_pressure': {
            'low': {'mu': mean_values[2], 'sigma': sd_values[2]},
            'high': {'mu': mean_values[3], 'sigma': sd_values[3]}
        },
        'cholesterol': {
            'low': {'mu': mean_values[4], 'sigma': sd_values[4]},
            'high': {'mu': mean_values[5], 'sigma': sd_values[5]}
```

```
        },
        'heart_rate': {
            'low': {'mu': mean_values[6], 'sigma': sd_values[6]},
            'high': {'mu': mean_values[7], 'sigma': sd_values[7]}
        }
    }

    # Calculate the membership values
    age_low = gaussian(age, **gaussian_parameters['age']['low'])
    age_high = gaussian(age, **gaussian_parameters['age']['high'])

    bp_low = gaussian(blood_pressure,
**gaussian_parameters['blood_pressure']['low'])
    bp_high = gaussian(blood_pressure,
**gaussian_parameters['blood_pressure']['high'])

    cholesterol_low = gaussian(cholesterol,
**gaussian_parameters['cholesterol']['low'])
    cholesterol_high = gaussian(cholesterol,
**gaussian_parameters['cholesterol']['high'])

    heart_rate_low = gaussian(heart_rate,
**gaussian_parameters['heart_rate']['low'])
    heart_rate_high = gaussian(heart_rate,
**gaussian_parameters['heart_rate']['high'])

    # Calculate weights
    weights = {
        'w1': age_low * cholesterol_low * bp_low * heart_rate_low,
        'w2': age_low * cholesterol_low * bp_low * heart_rate_high,
        'w3': age_low * cholesterol_low * bp_high * heart_rate_low,
        'w4': age_low * cholesterol_low * bp_high * heart_rate_high,
        'w5': age_low * cholesterol_high * bp_low * heart_rate_low,
        'w6': age_low * cholesterol_high * bp_low * heart_rate_high,
        'w7': age_low * cholesterol_high * bp_high * heart_rate_low,
        'w8': age_low * cholesterol_high * bp_high * heart_rate_high,
        'w9': age_high * cholesterol_low * bp_low * heart_rate_low,
        'w10': age_high * cholesterol_low * bp_low * heart_rate_high,
        'w11': age_high * cholesterol_low * bp_high * heart_rate_low,
        'w12': age_high * cholesterol_low * bp_high * heart_rate_high,
        'w13': age_high * cholesterol_high * bp_low * heart_rate_low,
        'w14': age_high * cholesterol_high * bp_low * heart_rate_high,
        'w15': age_high * cholesterol_high * bp_high * heart_rate_low,
        'w16': age_high * cholesterol_high * bp_high * heart_rate_high,
    }

    # Calculate the sum of all weights
    total_weight = sum(weights.values())

    # Return infinity if weight is 0
    if total_weight == 0:
        return float('inf')
```

```
    # Normalize the weights
    normalized_weights = {key: value / total_weight for key, value in
weights.items()}

    # Calculate O1 to O16 using Sugeno Fuzzy
    O_values = [normalized_weights[f'w{i}'] * (
            a_values[i - 1] * age + b_values[i - 1] * blood_pressure +
c_values[i - 1] * cholesterol + d_values[
        i - 1] * heart_rate + e_values[i - 1]) for i in range(1, 17)]

    # Sum up all the output values
    total_output = sum(O_values)

    # Calculate the tanh/tan sigmoid of total_output
    tanh_output = tanh(total_output)

    return tanh_output

# Optimized Parameters taken from ANFIS/GA Training

sd_values = [30.37425780967358, 98.92323148607927, 31.30203993566636,
88.35776635756176, 71.2218471050781,
                 43.734075579319246, 67.63996847971873, 30.9373871948566441]
a_values = [0.9732031435551686, 0.002688074077581315, 0.7943203599039085,
0.026442885173838748, 0.2738865067022688,
                0.3233673717050167, 0.9236541443806265, 0.14600392757227287,
0.43551706252478106, 0.04351124072960244,
                0.4098028348551067, 0.06671524745501212, 0.9460516326944615,
0.39275483342725104, 0.043528052946073426,
                0.18453543291968943]
b_values = [0.4432199392040913, 0.037194388229018105, 0.2570273196928561,
0.09723134452831073, 0.16818769955303936,
                0.25895733891737793, 0.1028899705491454, 0.7859214440275468,
0.04305857294284565, 0.09831302374188688,
                0.7597817056067317, 0.4294200092367515, 0.4598972809917097,
0.8499582199532707, 0.8219892834888435,
                0.28012279740788861]
c_values = [0.2365433657016054, 0.2001727628887343, 0.961140048391374,
0.10536567826209697, 0.5158219868486323,
                0.33785117362711414, 0.6735582930337685, 0.21666303634202333,
0.05949757890955576, 0.14326558874800255,
                0.9394102696667682, 0.012812112251490704, 0.7454729540633036,
0.6273417417146013, 0.41563968337136015,
                0.83058427185453311]
d_values = [0.06136473359583394, 0.04385758888744107, 0.5255328522142582,
0.008550749064928032, 0.2074167659044439,
                0.4497875055211833, 0.3425908291834572, 0.19384831046407147,
0.7434463052710825, 0.07912695990173457,
                0.8512781317591239, 0.03912567423121471, 0.9369364752103462,
0.16299795840068443, 0.02071330803936222,
                0.35279609888007061]
e_values = [0.23601646711979352, 0.12293345617954032, 0.8580113475436795,
0.01656766283823785, 0.4637885329033492,
```

```
                0.7931768977953763, 0.15894282957550965, 0.9391566238537904,
0.21034956262735371, 0.027206347898655836,
                0.1913517578065863, 0.016095303165399644, 0.8594526302435292,
0.9266191954170502, 0.7592609496608976,
                0.08776750692808122]

# Define universe variables
age = np.arange(0, 100, 1)
blood_pressure = np.arange(90, 200, 1)
cholesterol = np.arange(120, 580, 1)
heart_rate = np.arange(40, 200, 1)

# Define the layout of the GUI
layout = [
    [sg.Text('Enter your age (0-100):'), sg.InputText(key='age')],
    [sg.Text('Enter your blood pressure (90-200):'), sg.InputText(key='bp')],
    [sg.Text('Enter your cholesterol level (120-580):'),
sg.InputText(key='ch')],
    [sg.Text('Enter your max heart rate (40-200):'), sg.InputText(key='hr')],
    [sg.Button('Check Heart Attack Risk'), sg.Button('Exit')],
    [sg.Text('Result:'), sg.Text('', size=(10, 1), key='result')],
    [sg.Text('Notes:'), sg.Text('', size=(30, 1), key='notes')]
]

# Create the GUI window
window = sg.Window('Heart Attack Risk Detector - Powered by ANFIS-GA',
layout)

def visualize_fuzzy(age_val, bp_val, ch_val, hr_val):
    # Generate fuzzy membership functions
    age_membership_low = fuzz.gaussmf(age, 0, 30.37425780967358)
    age_membership_high = fuzz.gaussmf(age, 100, 98.92323148607927)

    blood_pressure_membership_low = fuzz.gaussmf(blood_pressure, 90,
31.302039935666636)
    blood_pressure_membership_high = fuzz.gaussmf(blood_pressure, 200,
88.35776635756176)

    cholesterol_membership_low = fuzz.gaussmf(cholesterol, 120,
71.2218471050781)
    cholesterol_membership_high = fuzz.gaussmf(cholesterol, 580,
43.734075579319246)

    heart_rate_membership_low = fuzz.gaussmf(heart_rate, 40,
67.63996847971873)
    heart_rate_membership_high = fuzz.gaussmf(heart_rate, 200,
30.937387194856644)

    # Visualize membership functions with user input highlighted
    plt.figure(figsize=(12, 10))

    plots = [
        (age, age_membership_low, age_membership_high, age_val, 'Age'),
```

```
        (blood_pressure, blood_pressure_membership_low,
blood_pressure_membership_high, bp_val, 'Blood Pressure'),
        (cholesterol, cholesterol_membership_low,
cholesterol_membership_high, ch_val, 'Cholesterol'),
        (heart_rate, heart_rate_membership_low, heart_rate_membership_high,
hr_val, 'Heart Rate')
    ]

    for i, (var, mem_low, mem_high, val, title) in enumerate(plots, start=1):
        plt.subplot(4, 1, i)
        plt.plot(var, mem_low, 'b', linewidth=1.5, label='Low')
        plt.plot(var, mem_high, 'r', linewidth=1.5, label='High')
        plt.axvline(x=val, color='g', linestyle='--')
        plt.title(title)
        plt.xlabel(title)
        plt.ylabel('Membership')
        plt.legend()

    plt.tight_layout()
    plt.show()

# Event loop to process events and get values from the GUI
while True:
    event, values = window.read()
    if event == sg.WINDOW_CLOSED or event == 'Exit':
        break
    if event == 'Check Heart Attack Risk':
        try:
            age_val = float(values['age'])
            bp_val = float(values['bp'])
            ch_val = float(values['ch'])
            hr_val = float(values['hr'])
            output = ANFIS(sd_values, a_values, b_values, c_values, d_values,
e_values, age_val, bp_val, ch_val, hr_val)
            if output<0.3:
                notes = "Low risk of heart attack"
            elif output>0.3 and output<0.7:
                notes = "Moderate risk of heart attack"
            else:
                notes = "High risk of heart attack"

            visualize_fuzzy(age_val, bp_val, ch_val, hr_val)

            window['result'].update(output)
            window['notes'].update(notes)
        except ValueError:
            sg.popup_error('Please enter valid numbers.')

# Close the GUI window
window.close()
```

Reference

https://youtu.be/w8yWXqWQYmU?si=feoU5gKE_YeTEKSp
https://youtu.be/Ug5Ec6Ym7f4?si=1lOGJGxkbYX6wTK2
https://youtu.be/fcLmRJY9GHQ?si=K711SycU4Qcz8rD9
Heart Attack Risk Prediction Dataset (kaggle.com)
Heart Failure Prediction Dataset (kaggle.com)
Heart attack - Symptoms & causes - Mayo Clinic

Haznedar, B., & Kalinli, A. (2016). Training ANFIS Using Genetic Algorithm for Dynamic Systems Identification. *International Journal of Intelligent Systems and Applications in Engineering*, *4*, 44–44. https://doi.org/10.18201/ijisae.266053