



**UNIVERSITI
MALAYA**

*Faculty of Computer Science
and Information Technology*

WIX3001 SOFT COMPUTING

GROUP ASSIGNMENT 3: EVOLUTIONARY COMPUTING

SEMESTER 2 SESSION 2023/2024

LECTURER: DR LIEW WEI SHIUNG

NO	NAME	MATRIC NUMBER
1.	QUAH JUN CHUAN	22004851
2.	TER ZHEN HUANG	22004736
3.	CHENG YEE ERN	22004791
4.	RACHEL LIM	22052702

Table of Contents

Table of Contents	2
1. Dataset	3
1.1 Introduction	3
1.2 About Dataset	3
1.3 Solution Approach	3
1.4 Data Cleaning	4
1.5 Feature Selection	5
2. Fuzzy Control System	7
2.1 Input Fuzzification	7
2.2 Fuzzy Variable	8
2.3 Membership Function	8
3. Evolutionary Algorithms	10
3.1 Population Generation	10
3.2 Evolutionary Strategy 1	11
3.3 Evolutionary Strategy 2	15
4. Chromosome Analysis	19
4.1 Best Chromosomes Extraction	19
4.2 Unsupervised Clustering Feature Importance	19
5. Result and Discussions	21
5.1 Compare EA Result	21
5.2 Compare Feature Importance in EA	22
5.3 Compare Methodology EA1 and EA2	23
6. Conclusion	26
References	27

1. Dataset

1.1 Introduction

Parkinson's disease (PD) is a neurodegenerative progressive disorder that affects movement and is characterized by symptoms like tremors, stiffness, slowness of movement, and difficulty with balance and coordination. Tracking the progression of Parkinson's disease symptoms often involves the time-consuming and expensive Unified Parkinson's Disease Rating Scale (UPDRS) assessments conducted by medical professionals.

Therefore, this research aims to replicate UPDRS assessments remotely using simple, self-administered, and noninvasive speech tests. By doing so, we seek to facilitate the telemonitoring frameworks for utilizing large-scale clinical trials for new treatments for PD.

1.2 About Dataset

The dataset chosen for this project, "Parkinson's Telemonitoring," is obtained from the UCI Machine Learning Repository. This dataset comprises a range of biomedical voice measurements from 42 people with early-stage Parkinson's disease who were recruited to a six-month trial of a telemonitoring device for remote symptom progression monitoring. The recordings were automatically captured in the patients' homes.

Columns in the table contain a subject number, subject age, subject gender, the time interval from baseline recruitment date, motor UPDRS, total UPDRS, and 16 biomedical voice measures. Each row corresponds to one of 5,875 voice recordings from these individuals. The main aim of the data is to predict the total UPDRS scores ('total_UPDRS') from the 16 voice measures.

1.3 Solution Approach

To address the problem of predicting the total UPDRS scores from the biomedical voice measurements in the "Parkinson's Telemonitoring" dataset, we utilized an evolutionary algorithm with fuzzy rule bases. We aim to develop a robust fuzzy rule base that can accurately predict the total UPDRS scores from the biomedical voice measurements, ensuring the models are both accurate and interpretable. By developing and comparing different fitness functions and evolutionary strategies, we will identify the most effective method for this specific problem.

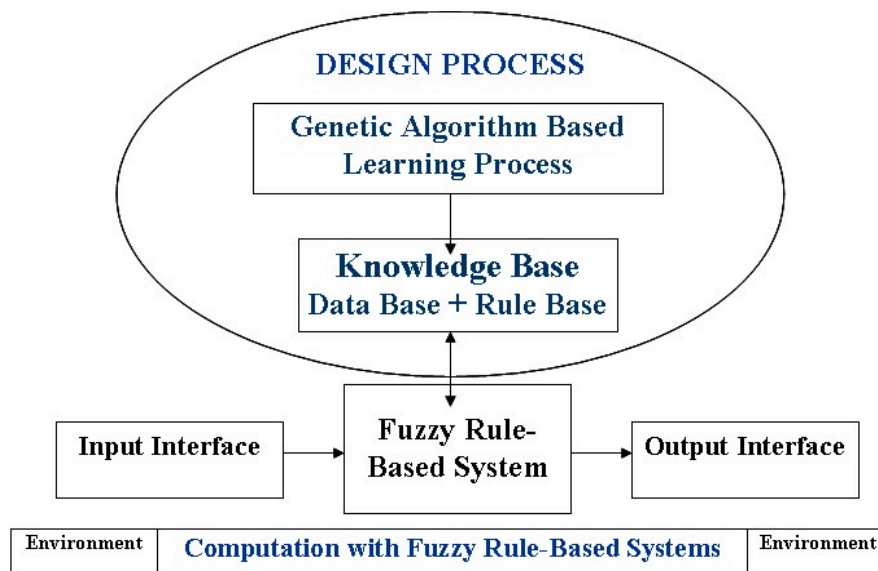


Diagram 1.3.1

1.4 Data Cleaning

We imported our dataset named 'parkinsons_updrs.data' then rename some columns to remove unique characters like %, : and (). We also checked and ensured the dataset was free from duplicated rows and null or missing values.

```
df = pd.read_csv('parkinsons_updrs.data')
```

	subject#	age	sex	test_time	motor_UPDRS	total_UPDRS	Jitter(%)	Jitter(Abs)	Jitter:RAP	Jitter:PPQ5	...	Shimmer(dB)
0	1	72	0	5.6431	28.199	34.398	0.00662	0.000034	0.00401	0.00317	...	0.230
1	1	72	0	12.6660	28.447	34.894	0.00300	0.000017	0.00132	0.00150	...	0.179
2	1	72	0	19.6810	28.695	35.389	0.00481	0.000025	0.00205	0.00208	...	0.181
3	1	72	0	25.6470	28.905	35.810	0.00528	0.000027	0.00191	0.00264	...	0.327
4	1	72	0	33.6420	29.187	36.375	0.00335	0.000020	0.00093	0.00130	...	0.176
...
5870	42	61	0	142.7900	22.485	33.485	0.00406	0.000031	0.00167	0.00168	...	0.160
5871	42	61	0	149.8400	21.988	32.988	0.00297	0.000025	0.00119	0.00147	...	0.215
5872	42	61	0	156.8200	21.495	32.495	0.00349	0.000025	0.00152	0.00187	...	0.244
5873	42	61	0	163.7300	21.007	32.007	0.00281	0.000020	0.00128	0.00151	...	0.131
5874	42	61	0	170.7300	20.513	31.513	0.00282	0.000021	0.00135	0.00166	...	0.171

Diagram 1.4.1

```
# Check for duplicated rows
duplicated_rows = df.duplicated().any()
print("Duplicated rows?", duplicated_rows)

# Check for the number of rows with any missing/null values
rows_with_missing_values = df.isna().any(axis=1).sum()
print(f"Number of rows with missing values: {rows_with_missing_values}")

Duplicated rows? False
Number of rows with missing values: 0
```

Diagram 1.4.2

To ensure accurate interpretation of our dataset, we identified if there is any test times whose values are negative, which we then removed them from our dataset, while dropping 'subject#' and 'motor_UPDRS' columns as they do not contribute to solving our problem.

```
negative_test_times = df['test_time'].lt(0).sum()
print(f"Number of negative test times: {negative_test_times}")

Number of negative test times: 12

df=df[df['test_time']>0]
df=df.drop('subject#',axis=1)
df=df.drop('motor_UPDRS',axis=1)
```

Diagram 1.4.3

1.5 Feature Selection

The feature selection process involves training a linear regression model to predict the target variable 'total_UPDRS' using all available features except for the target itself. After extracting the coefficients of the features from the model, a DataFrame is created to store the feature names and their corresponding coefficients, sorted by their absolute values in descending order. A threshold value of 20 is set to determine which features are significant based on their coefficient magnitudes, with features having absolute coefficients above the threshold selected. The original DataFrame is filtered to retain only these selected features alongside the target variable. This process effectively identifies and retains the most influential features for predicting 'total_UPDRS' while discarding less impactful ones, facilitating more focused analysis or model development. At the end of this step, 11 features were retained for the model development.

```

X = df.drop(['total_UPDRS'], axis=1)
y = df['total_UPDRS']

# Train a linear regression model
lr = LinearRegression()
lr.fit(X, y)

# Get coefficients of the features
feature_coefficients = lr.coef_

# Create a DataFrame to store feature names and coefficients
feature_coefficients_df = pd.DataFrame({'Feature': X.columns, 'Coefficient': feature_coefficients})

# Sort features by coefficient magnitude (absolute value)
feature_coefficients_df['Abs_Coefficient'] = np.abs(feature_coefficients_df['Coefficient'])
feature_coefficients_df = feature_coefficients_df.sort_values(by='Abs_Coefficient', ascending=False)

# Set a threshold for coefficient magnitude
threshold = 20

# Select features with coefficient magnitude above the threshold
selected_features = feature_coefficients_df[feature_coefficients_df['Abs_Coefficient'] > threshold]['Feature']

# Filter the DataFrame to keep only selected features
df = df[selected_features]

# Add the target variable 'total_UPDRS' back into the DataFrame
df['total_UPDRS'] = y

```

Diagram 1.5.1

The purpose of this code segment is to build and plot boxplots for each feature. To do so, we first calculate the number of rows and columns for subplots while we set the figure size and create the subplot figure. We then calculate the whiskers and quartiles by iterating each feature with their axes being flattened if there is only one row of values. If there are any empty subplots, it is removed and the boxplots of each feature are shown.

```

# Calculate the number of rows and columns for subplots
num_features = len(df.columns)
num_cols = 5
num_rows = (num_features + 1) // num_cols

# Set the figure size based on the number of features
fig_width = 8
fig_height = 4 * num_rows

# Create the figure and subplots
fig, axes = plt.subplots(num_rows, num_cols, figsize=(fig_width, fig_height))

# Flatten the axes if there's only one row
if num_rows == 1:
    axes = [axes]

# Iterate through each feature and generate a box plot
for i, (col, ax) in enumerate(zip(df.columns, axes.flatten())):
    # Generate box plot for the current feature
    bp = ax.boxplot(df[col])

    # Print box plot values
    print(f"Feature: {col}")
    print("Whiskers:")
    for whisker in bp['whiskers']:
        print(f" - {whisker.get_data()[0]}")
    print("Quartiles:")
    print(f" - Q1: {bp['whiskers'][0].get_data()[1]}")
    print(f" - Q2 (Median): {np.median(df[col])}")
    print(f" - Q3: {bp['whiskers'][1].get_data()[1]}")

    ax.set_title(col) # Set title as feature name
    ax.set_ylabel('Values') # Set y-axis label

# Remove any empty subplots
for i in range(num_features, num_rows * num_cols):
    fig.delaxes(axes.flatten()[i])

# Adjust layout to prevent overlapping titles
plt.tight_layout()

# Show the plots
plt.show()

```

Diagram 1.5.2

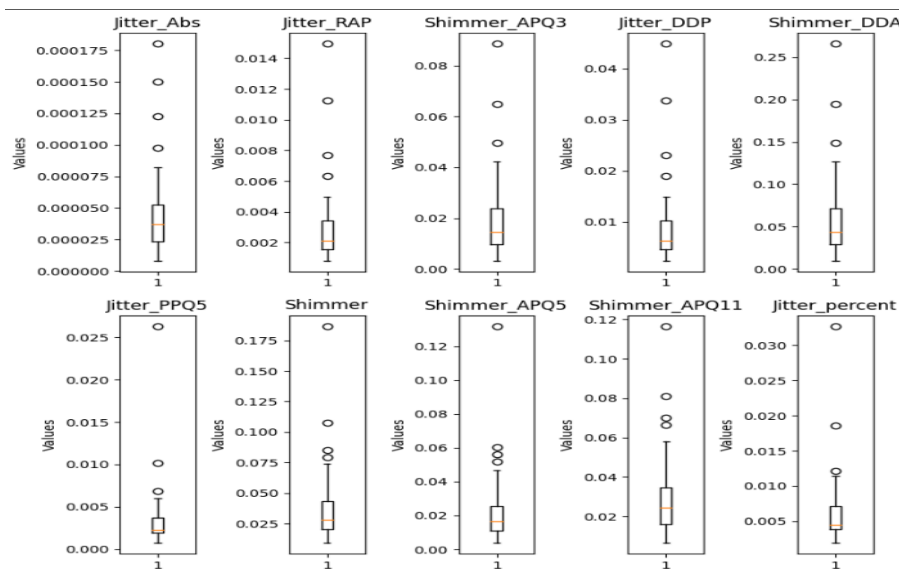


Diagram 1.5.3

2. Fuzzy Control System

2.1 Input Fuzzification

We start the fuzzy system by creating the universe for each features (Jitter_percent , Jitter_Abs etc.) The universe represents the range of all possible value that the feature can take, in this case, we achieve it by using the `np.linspace()` function, which helps generates 100 equally spaced value between the minimum and maximum value of each feature in the dataframe where we define it using the `min()` and `max()` function. Each line defines a universe of a specific feature.

```
def create_fuzzy_logic_system():
    # Define universe variables for each feature
    Jitter_percent_universe = np.linspace(df['Jitter_percent'].min(), df['Jitter_percent'].max(), 100)
    Jitter_Abs_universe = np.linspace(df['Jitter_Abs'].min(), df['Jitter_Abs'].max(), 100)
    Jitter_RAP_universe = np.linspace(df['Jitter_RAP'].min(), df['Jitter_RAP'].max(), 100)
    Jitter_PPQ5_universe = np.linspace(df['Jitter_PPQ5'].min(), df['Jitter_PPQ5'].max(), 100)
    Jitter_DDP_universe = np.linspace(df['Jitter_DDP'].min(), df['Jitter_DDP'].max(), 100)
    Shimmer_universe = np.linspace(df['Shimmer'].min(), df['Shimmer'].max(), 100)
    Shimmer_APQ3_universe = np.linspace(df['Shimmer_APQ3'].min(), df['Shimmer_APQ3'].max(), 100)
    Shimmer_APQ5_universe = np.linspace(df['Shimmer_APQ5'].min(), df['Shimmer_APQ5'].max(), 100)
    Shimmer_APQ11_universe = np.linspace(df['Shimmer_APQ11'].min(), df['Shimmer_APQ11'].max(), 100)
    Shimmer_DDA_universe = np.linspace(df['Shimmer_DDA'].min(), df['Shimmer_DDA'].max(), 100)
    DFA_universe = np.linspace(df['DFA'].min(), df['DFA'].max(), 100)
    total_UPDRS_universe = np.linspace(df['total_UPDRS'].min(), df['total_UPDRS'].max(), 100)
```

Diagram 2.1.1

2.2 Fuzzy Variable

Next, we create fuzzy variables for each feature using the `skfuzzy.control.Antecedent` and `skfuzzy.control.Consequent` classes. The antecedents represent the input variable for the fuzzy logic system while the consequent represents the output variable for the fuzzy logic system.

```
# Create fuzzy variables for each feature
Jitter_percent = ctrl.Antecedent(Jitter_percent_universe, 'Jitter_percent')
Jitter_Abs = ctrl.Antecedent(Jitter_Abs_universe, 'Jitter_Abs')
Jitter_RAP = ctrl.Antecedent(Jitter_RAP_universe, 'Jitter_RAP')
Jitter_PPQ5 = ctrl.Antecedent(Jitter_PPQ5_universe, 'Jitter_PPQ5')
Jitter_DDP = ctrl.Antecedent(Jitter_DDP_universe, 'Jitter_DDP')
Shimmer = ctrl.Antecedent(Shimmer_universe, 'Shimmer')
Shimmer_APQ3 = ctrl.Antecedent(Shimmer_APQ3_universe, 'Shimmer_APQ3')
Shimmer_APQ5 = ctrl.Antecedent(Shimmer_APQ5_universe, 'Shimmer_APQ5')
Shimmer_APQ11 = ctrl.Antecedent(Shimmer_APQ11_universe, 'Shimmer_APQ11')
Shimmer_DDA = ctrl.Antecedent(Shimmer_DDA_universe, 'Shimmer_DDA')
DFA = ctrl.Antecedent(DFA_universe, 'DFA')
total_UPDRS = ctrl.Consequent(total_UPDRS_universe, 'total_UPDRS')
```

Diagram 2.2.1

2.3 Membership Function

We allocate membership functions to both input and output fuzzification variables by utilizing manually specified values stored in arrays. Each feature is divided into three membership functions: 'low', 'medium', and 'high', each utilizing different types of membership functions. Specifically, we employ Z-membership functions for 'low', triangle membership functions for 'medium', and S-membership functions for 'high'.

```
mf_val = np.array([

    #JITTER_PERCENT
    0, 0.005,
    0.00358, 0.0049, 0.0068,
    0.004, 0.1,

    #JITTER_ABS
    0, 3.46e-05,
    2.2425e-05, 3.453e-05, 5.334e-05,
    3.44e-05, 4.5e-04,

    #JITTER_RAP
    0, 0.0023,
    0.00158, 0.00225, 0.00329,
    0.0022, 0.06,

    #JITTER_PPQ5
    0, 0.0025,
    0.001825, 0.00249, 0.00347,
    0.0024, 0.07,

    #JITTER_DDP
    0, 0.0068,
    0.00473, 0.00675, 0.00988,
    0.0067, 0.18,

    #SHIMMER
    0, 0.0276,
    0.01911, 0.02753, 0.039775,
    0.0275, 0.27,

    #SHIMMER_APQ3
    0, 0.0138,
    0.00928, 0.0137, 0.020605,
    0.0136, 0.017,

    #SHIMMER_APQ5
    0, 0.016,
    0.01079, 0.01594, 0.023775,
    0.015, 0.17,

    #SHIMMER_APQ11
    0, 0.0228,
    0.01566, 0.02271, 0.032725,
    0.0226, 0.28,

    #SHIMMER_DDA
    0, 0.042,
    0.02783, 0.04111, 0.061805,
    0.041, 0.49,

    #DFA
    0, 0.644,
    0.5961, 0.64355, 0.71148,
    0.643, 0.87,

    #TOTAL_UPDRS
    0, 30,
    7.0, 27.522, 36.4005,
    20, 55

])
```

Diagram 2.3.1


```

def create_fuzzy_logic_system():
    # Define universe variables for each feature
    Jitter_percent_universe = np.linspace(df['Jitter_percent'].min(), df['Jitter_percent'].max(), 100)
    Jitter_Abs_universe = np.linspace(df['Jitter_Abs'].min(), df['Jitter_Abs'].max(), 100)
    Jitter_RAP_universe = np.linspace(df['Jitter_RAP'].min(), df['Jitter_RAP'].max(), 100)
    Jitter_PPQ5_universe = np.linspace(df['Jitter_PPQ5'].min(), df['Jitter_PPQ5'].max(), 100)
    Jitter_DDP_universe = np.linspace(df['Jitter_DDP'].min(), df['Jitter_DDP'].max(), 100)
    Shimmer_universe = np.linspace(df['Shimmer'].min(), df['Shimmer'].max(), 100)
    Shimmer_APQ3_universe = np.linspace(df['Shimmer_APQ3'].min(), df['Shimmer_APQ3'].max(), 100)
    Shimmer_APQ5_universe = np.linspace(df['Shimmer_APQ5'].min(), df['Shimmer_APQ5'].max(), 100)
    Shimmer_APQ11_universe = np.linspace(df['Shimmer_APQ11'].min(), df['Shimmer_APQ11'].max(), 100)
    Shimmer_DDA_universe = np.linspace(df['Shimmer_DDA'].min(), df['Shimmer_DDA'].max(), 100)
    DFA_universe = np.linspace(df['DFA'].min(), df['DFA'].max(), 100)
    total_UPDRS_universe = np.linspace(df['total_UPDRS'].min(), df['total_UPDRS'].max(), 100)

    # Create fuzzy variables for each feature
    Jitter_percent = ctrl.Antecedent(Jitter_percent_universe, 'Jitter_percent')
    Jitter_Abs = ctrl.Antecedent(Jitter_Abs_universe, 'Jitter_Abs')
    Jitter_RAP = ctrl.Antecedent(Jitter_RAP_universe, 'Jitter_RAP')
    Jitter_PPQ5 = ctrl.Antecedent(Jitter_PPQ5_universe, 'Jitter_PPQ5')
    Jitter_DDP = ctrl.Antecedent(Jitter_DDP_universe, 'Jitter_DDP')
    Shimmer = ctrl.Antecedent(Shimmer_universe, 'Shimmer')
    Shimmer_APQ3 = ctrl.Antecedent(Shimmer_APQ3_universe, 'Shimmer_APQ3')
    Shimmer_APQ5 = ctrl.Antecedent(Shimmer_APQ5_universe, 'Shimmer_APQ5')
    Shimmer_APQ11 = ctrl.Antecedent(Shimmer_APQ11_universe, 'Shimmer_APQ11')
    Shimmer_DDA = ctrl.Antecedent(Shimmer_DDA_universe, 'Shimmer_DDA')
    DFA = ctrl.Antecedent(DFA_universe, 'DFA')
    total_UPDRS = ctrl.Consequent(total_UPDRS_universe, 'total_UPDRS')

    # Define membership functions for each variable
    Jitter_percent['low'] = fuzz.zmf(Jitter_percent.universe, mf_val[0], mf_val[1])
    Jitter_percent['medium'] = fuzz.trimf(Jitter_percent.universe, [mf_val[2], mf_val[3], mf_val[4]])
    Jitter_percent['high'] = fuzz.smf(Jitter_percent.universe, mf_val[5], mf_val[6])

    # Reassign the values for Jitter_Abs
    Jitter_Abs['low'] = fuzz.zmf(Jitter_Abs.universe, mf_val[7], mf_val[8])
    Jitter_Abs['medium'] = fuzz.trimf(Jitter_Abs.universe, [mf_val[9], mf_val[10], mf_val[11]])
    Jitter_Abs['high'] = fuzz.smf(Jitter_Abs.universe, mf_val[12], mf_val[13])

    # Reassign the values for Jitter_RAP
    Jitter_RAP['low'] = fuzz.zmf(Jitter_RAP.universe, mf_val[14], mf_val[15])
    Jitter_RAP['medium'] = fuzz.trimf(Jitter_RAP.universe, [mf_val[16], mf_val[17], mf_val[18]])
    Jitter_RAP['high'] = fuzz.smf(Jitter_RAP.universe, mf_val[19], mf_val[20])

    # Reassign the values for Jitter_PPQ5
    Jitter_PPQ5['low'] = fuzz.zmf(Jitter_PPQ5.universe, mf_val[21], mf_val[22])
    Jitter_PPQ5['medium'] = fuzz.trimf(Jitter_PPQ5.universe, [mf_val[23], mf_val[24], mf_val[25]])
    Jitter_PPQ5['high'] = fuzz.smf(Jitter_PPQ5.universe, mf_val[26], mf_val[27])

    # Reassign the values for Jitter_DDP
    Jitter_DDP['low'] = fuzz.zmf(Jitter_DDP.universe, mf_val[28], mf_val[29])
    Jitter_DDP['medium'] = fuzz.trimf(Jitter_DDP.universe, [mf_val[30], mf_val[31], mf_val[32]])
    Jitter_DDP['high'] = fuzz.smf(Jitter_DDP.universe, mf_val[33], mf_val[34])

    # Reassign the values for Shimmer
    Shimmer['low'] = fuzz.zmf(Shimmer.universe, mf_val[35], mf_val[36])
    Shimmer['medium'] = fuzz.trimf(Shimmer.universe, [mf_val[37], mf_val[38], mf_val[39]])
    Shimmer['high'] = fuzz.smf(Shimmer.universe, mf_val[40], mf_val[41])

    # Reassign the values for Shimmer_APQ3
    Shimmer_APQ3['low'] = fuzz.zmf(Shimmer_APQ3.universe, mf_val[42], mf_val[43])
    Shimmer_APQ3['medium'] = fuzz.trimf(Shimmer_APQ3.universe, [mf_val[44], mf_val[45], mf_val[46]])
    Shimmer_APQ3['high'] = fuzz.smf(Shimmer_APQ3.universe, mf_val[47], mf_val[48])

    # Reassign the values for Shimmer_APQ5
    Shimmer_APQ5['low'] = fuzz.zmf(Shimmer_APQ5.universe, mf_val[49], mf_val[50])
    Shimmer_APQ5['medium'] = fuzz.trimf(Shimmer_APQ5.universe, [mf_val[51], mf_val[52], mf_val[53]])
    Shimmer_APQ5['high'] = fuzz.smf(Shimmer_APQ5.universe, mf_val[54], mf_val[55])

    # Reassign the values for Shimmer_APQ11
    Shimmer_APQ11['low'] = fuzz.zmf(Shimmer_APQ11.universe, mf_val[56], mf_val[57])
    Shimmer_APQ11['medium'] = fuzz.trimf(Shimmer_APQ11.universe, [mf_val[58], mf_val[59], mf_val[60]])
    Shimmer_APQ11['high'] = fuzz.smf(Shimmer_APQ11.universe, mf_val[61], mf_val[62])

    # Reassign the values for Shimmer_DDA
    Shimmer_DDA['low'] = fuzz.zmf(Shimmer_DDA.universe, mf_val[63], mf_val[64])
    Shimmer_DDA['medium'] = fuzz.trimf(Shimmer_DDA.universe, [mf_val[65], mf_val[66], mf_val[67]])
    Shimmer_DDA['high'] = fuzz.smf(Shimmer_DDA.universe, mf_val[68], mf_val[69])

    DFA['low'] = fuzz.zmf(DFA.universe, mf_val[70], mf_val[71])
    DFA['medium'] = fuzz.trimf(DFA.universe, [mf_val[72], mf_val[73], mf_val[74]])
    DFA['high'] = fuzz.smf(DFA.universe, mf_val[75], mf_val[76])

    total_UPDRS['low'] = fuzz.zmf(total_UPDRS.universe, mf_val[77], mf_val[78])
    total_UPDRS['medium'] = fuzz.trimf(total_UPDRS.universe, [mf_val[79], mf_val[80], mf_val[81]])
    total_UPDRS['high'] = fuzz.smf(total_UPDRS.universe, mf_val[82], mf_val[83])

    fuzzy_vars = [Jitter_percent, Jitter_Abs, Jitter_RAP, Jitter_PPQ5, Jitter_DDP, Shimmer, Shimmer_APQ3,
                  Shimmer_APQ5, Shimmer_APQ11, Shimmer_DDA, DFA, total_UPDRS]

    return fuzzy_vars

```

Diagram 2.3.2

3. Evolutionary Algorithms

3.1 Population Generation

For the evolutionary algorithm, we implemented two different algorithms, each using different fitness functions, selection methods, crossover, and mutation techniques. Algorithm A is used to maximize the fuzzy rule base accuracy and minimize the number of fuzzy rules, while Algorithm B is used to maximize the fuzzy rule base coverage with the minimum number of features. Both algorithms will be run for 50 evolutions.

First, we need to initialize a random population of chromosomes, where each chromosome represents one fuzzy rule base. We created a `generate_rule` function that takes in three parameters: `num_features`, representing the total number of features to be included in a single rule; `num_states_per_feature`, which represents the number of states per feature and is a list of states for each feature, with each feature having three states ('low', 'medium', and 'high'); and `min_num_features`, set to 8 to prevent the generation of crisp output values due to insufficient feature inclusion. This function randomly selects features and their corresponding states to form the rule's antecedents, assigns a consequent from the three possible classes, combines these antecedents using logical operations, and returns the constructed fuzzy rule.

```
def generate_rule(num_features, num_states_per_feature, min_num_features=8, fuzzy_vars=None):
    while True:
        rule = []
        # Randomly select features for the rule
        selected_features = np.random.choice(num_features, size=np.random.randint(min_num_features, num_features + 1), replace=False)
        for feature in selected_features:
            state = np.random.randint(num_states_per_feature[feature])
            rule.append((feature, state))
        # Consequent: Assuming 3 classes: low, medium, high
        consequent = np.random.choice(['low', 'medium', 'high'])
        rule.append(consequent)

        antecedents = []
        for feature_index, state in rule[:-1]:
            term_name = 'low' if state == 0 else 'medium' if state == 1 else 'high'
            antecedents.append(fuzzy_vars[feature_index][term_name])

        if antecedents:
            combined_antecedent = antecedents[0]
            for antecedent in antecedents[1:]:
                if np.random.rand() > 0.5: # Randomly choose AND or OR
                    combined_antecedent = combined_antecedent & antecedent
                else:
                    combined_antecedent = combined_antecedent | antecedent
            consequent_state = rule[-1]
            term_name = 'low' if consequent_state == 'low' else 'medium' if consequent_state == 'medium' else 'high'
            fuzzy_rule = ctrl.Rule(antecedent=combined_antecedent, consequent=fuzzy_vars[-1][term_name])
            return fuzzy_rule
```

Diagram 3.1.1

The `initialize_population` function generates a random population of chromosomes, where each chromosome represents a fuzzy rule base. It takes four parameters: `population_size`, `num_rules`, `num_features`, and `num_states_per_feature`. Within the function, a fuzzy logic system is created, and for each individual in the population, a random number of rules is generated using the `generate_rule` function. Each generated rule is added to the individual's

chromosome, provided it is valid. The function returns the populated list of chromosomes and prints the population for verification.

```
def initialize_population(population_size, num_rules, num_features, num_states_per_feature):
    population = []
    fuzzy_vars = create_fuzzy_logic_system()

    for _ in range(population_size):
        chromosome = []
        # Randomly determine the number of rules to generate for each individual in the population
        num_rules_for_individual = np.random.randint(1, num_rules + 1)
        for _ in range(num_rules_for_individual):
            rule = generate_rule(num_features, num_states_per_feature, fuzzy_vars=fuzzy_vars)
            if rule: # Check if a valid rule is generated
                chromosome.append(rule)
        population.append(chromosome)
    print(str(population))
    return population
```

Diagram 3.1.2

3.2 Evolutionary Strategy 1

We developed a fitness function (Fitness Function A) to assess the quality of candidate solutions (chromosomes) which takes in three parameters: 'chromosome' (candidate solution), 'data' (input data) and 'labels' corresponding to the input data. The function utilizes a 'predict' function to generate predictions using the parameters. Later, it calculates the prediction accuracy using the R-squared score ('r2_score') and evaluates the complexity of the chromosome based on its length. Lastly, we determine the fitness value using a weighted combination of accuracy and complexity with more weight given to accuracy ('w1' = 0.9) and less weight given to complexity ('w2' = 0.1). Finally, the function prints and returns the fitness value.

```
# Fitness function A
def fitness_function_A(chromosome, data, labels):
    predictions = predict(chromosome, data)
    accuracy = r2_score(labels, predictions)
    complexity = len(chromosome)
    w1 = 0.9
    w2 = 0.1
    fitness_value = w1 * accuracy - w2 * complexity

    print("Fitness A fitness value : " + str(fitness_value))

    return fitness_value

def predict(chromosome, data):
    ctrl_sys = ctrl.ControlSystem(chromosome)
    inference = ctrl.ControlSystemSimulation(ctrl_sys)
    predictions = []
    for index, row in data.iterrows():
        for key, value in row.items():
            inference.input[key] = value
        inference.compute()
        prediction = inference.output['total_UPDRS']
        predictions.append(prediction)
    return predictions
```

Diagram 3.2.1

A 'tournament_selection' function is developed by taking three parameters: 'population' (a list of chromosomes), 'fitness_scores', and a 'tournament size' set to 3, which facilitates the selection of chromosomes from a population based on their fitness scores for the next generation. It initiates by creating an empty 'selected' list and determining the 'population_size' (population size). The function proceeds by iteratively selecting individual chromosomes until the 'selected' list reaches the 'population_size'. In each iteration, it randomly picks 'tournament_size' individuals using the 'random.sample' function. The tournament winner, determined by the highest 'fitness_score' via a 'max' function with a lambda key, contributes their chromosome to

the 'selected' list. Finally, upon completion of the selection process, the function returns the list of chosen chromosomes.

```
# Selection algorithm
def tournament_selection(population, fitness_scores, tournament_size):
    selected = []
    population_size = len(population)

    while len(selected) < population_size:
        # Select 'tournament_size' random individuals from the population
        tournament = random.sample(list(enumerate(population)), tournament_size)

        # Determine the winner based on the highest fitness score
        winner = max(tournament, key=lambda x: fitness_scores[x[0]])

        # Append the winner's chromosome to the selected list
        selected.append(winner[1])

    return selected
```

Diagram 3.2.2

We then implemented one one-point cross-over between two parent chromosomes to produce two offspring. The function first ensures the shorter chromosome is parent1 and the longer one is parent2. If parent1 has one or zero elements, it returns copies of the parents as offspring. Otherwise, it selects a random crossover point within parent1 and swaps segments of the parents' chromosomes at this point to create the offspring. The resulting offspring's chromosomes are then returned.

```
# Cross-Over Algorithm
def one_point_crossover(parent1, parent2):
    if len(parent1) > len(parent2):
        parent1, parent2 = parent2, parent1

    if len(parent1) <= 1:
        return parent1[:], parent2[:]

    crossover_point = random.randint(1, len(parent1) - 1)

    offspring1 = parent1[:crossover_point] + parent2[crossover_point:]
    offspring2 = parent2[:crossover_point] + parent1[crossover_point:]

    return offspring1, offspring2
```

Diagram 3.2.3

The mutate_consequent function mutates a randomly selected rule in a chromosome by changing its consequent. It selects a rule from the chromosome, chooses a new consequent ('low', 'medium', or 'high'), and updates the rule's consequent if the new one differs from the current one. The modified chromosome is then returned.

```

# Mutation Algorithm
def mutate_consequent(chromosome):
    rule = random.choice(chromosome)
    new_output_choice = random.choice(['low', 'medium', 'high',])
    new_consequent = fuzzy_vars[11][new_output_choice]

    if rule.consequent != new_consequent:
        rule.consequent = new_consequent

    return chromosome

```

Diagram 3.2.4

The `genetic_algorithm_A` function runs an evolutionary algorithm to optimize a fuzzy rule base over a specified number of generations. It starts by initializing a population of fuzzy rule bases and tracks the best score and chromosome. In each generation, it evaluates fitness scores, updates the best score and chromosome, adjusts the mutation rate based on stagnation, and applies elitism by keeping the best chromosome in the next generation. It uses tournament selection for parent selection and performs crossover and mutation to generate the next population using the function defined above. The process continues until 50 generations are reached, with the best scores and chromosomes recorded and returned.

```

# Define Evolutionary Algorithm
def genetic_algorithm_A(population_size, data, labels, num_generations, num_rules, num_features, mutation_rate, num_states_per_feature, tournament_size, crossover_rate):
    fuzzy_population = initialize_population(population_size, num_rules, num_features, num_states_per_feature)
    best_score = float('-inf')
    best_chromosome = None
    stagnation_counter = 0
    best_scores = []
    best_chromosomes = []

    for generation in range(num_generations):
        fitness_scores = [fitness_function_A(chromosome, data, labels) for chromosome in fuzzy_population]

        max_fitness = max(fitness_scores)

        if max_fitness > best_score:
            best_score = max_fitness
            best_chromosome = fuzzy_population[fitness_scores.index(max_fitness)]
            best_chromosomes.append(best_chromosome)
            stagnation_counter = 0
        else:
            stagnation_counter += 1

        if stagnation_counter > 5:
            mutation_rate = min(1.0, mutation_rate + 0.05)
        else:
            mutation_rate = max(0.1, mutation_rate - 0.01)

        # Elitism: Keep the best chromosome in the next generation
        next_generation = [best_chromosome]

        selected = tournament_selection(fuzzy_population, fitness_scores, tournament_size)

        while len(next_generation) < population_size:
            if random.random() < crossover_rate and len(selected) > 1:
                parent1, parent2 = random.sample(selected, 2)
                offspring1, offspring2 = one_point_crossover([parent1, parent2])
                next_generation.extend([offspring1, offspring2])
            else:
                next_generation.extend(random.sample(selected, 2))

        for idx in range(len(next_generation)):
            if random.random() < mutation_rate:
                next_generation[idx] = mutate_consequent(next_generation[idx])

        population = next_generation
        best_scores.append(best_score)

        print(f"Generation {generation + 1}, Best Score: {best_score}")

    return best_scores, best_chromosomes

```

Diagram 3.2.5

After defining all the required parameters, the "genetic_algorithm_A" is executed with these parameters to optimize the fuzzy rule base. Finally, the best scores over generations are plotted. The evolutionary optimization process is conducted to refine the fuzzy rule base for a specified dataset and parameters, with the best fitness scores visualized across 50 generations.

```
# Define your dataset and labels
data = df.drop(['total_UPDRS'], axis=1)
labels = df['total_UPDRS']

# Define parameters
crossover_rate = 0.8
tournament_size = 5
num_generations = 50
num_features = 11
mutation_rate = 0.5
population_size = 30
threshold = 0.3
num_states_per_feature = [3, 3, 3, 3, 3,
                           3, 3, 3, 3, 3, 3] # List containing the number of states for each feature

# Run genetic algorithm for Fitness Function A
best_scores, best_chromosomes = genetic_algorithm_A(population_size, data, labels, num_generations, num_features, mutation_rate, num_states_per_feature, tournament_size, crossover_rate)

# Plotting the best scores
plt.plot(best_scores)
plt.title('Best Fitness Score per Generation')
plt.xlabel('Generation')
plt.ylabel('Best Fitness Score')
plt.show()
```

Diagram 3.2.6

Finally, we perform unsupervised clustering on the best chromosomes obtained from the genetic algorithm to identify important features. It starts by extracting all features from the best chromosomes(rule base) and converting them into a numerical representation. Then,we applies unsupervised clustering using K-means with 2 clusters. Next, we analyzes each cluster to determine feature importance by counting occurrences of each feature within the rules belonging to that cluster. After calculating the feature importance scores for each cluster, we plots the importance scores for each feature within each cluster. This process helps in identifying which features are most significant within different clusters of rules, aiding in the interpretation of the fuzzy rule base.

```

all_features = [[condition["feature"] for condition in rule["conditions"]] for rule in rules]

# Convert features into numerical representation
unique_features = list(set(feature for rule_features in all_features for feature in rule_features))
numerical_representation = np.zeros((len(rules), len(unique_features)))

for i, rule_features in enumerate(all_features):
    for j, feature in enumerate(unique_features):
        if feature in rule_features:
            numerical_representation[i, j] = 1

# Perform unsupervised clustering
num_clusters = 2 # Adjust as needed
kmeans = KMeans(n_clusters=num_clusters, random_state=0)
cluster_labels = kmeans.fit_predict(numerical_representation)

# Analyze clusters for feature importance
cluster_feature_counts = {cluster: {} for cluster in range(num_clusters)}
for i, cluster_label in enumerate(cluster_labels):
    rule_features = all_features[i]
    for feature in rule_features:
        cluster_feature_counts[cluster_label][feature] = cluster_feature_counts[cluster_label].get(feature, 0) + 1

# Calculate feature importance scores for each cluster
cluster_feature_importance = {}
for cluster, feature_counts in cluster_feature_counts.items():
    total_occurrences = sum(feature_counts.values())
    cluster_feature_importance[cluster] = {feature: count / total_occurrences for feature, count in feature_counts.items()}

# Plot feature importance scores for each cluster
for cluster, importance_scores in cluster_feature_importance.items():
    sorted_importance_scores = dict(sorted(importance_scores.items(), key=lambda item: item[1], reverse=True))
    features = list(sorted_importance_scores.keys())
    importances = list(sorted_importance_scores.values())

plt.figure(figsize=(10, 6))
plt.barh(features, importances, color='skyblue')
plt.xlabel('Importance')
plt.title(f'Cluster {cluster} Feature Importance Scores')
plt.gca().invert_yaxis()
plt.show()

```

Diagram 3.2.7

3.3 Evolutionary Strategy 2

We first implement another set of “fitness_function_B” that calculates the fitness value for a chromosome based on fuzzy rule base coverage and complexity. It first converts the antecedents of the rules in the chromosome into a NumPy array and checks if it's a 1D array; if so, it reshapes it to a 2D array. Then, it calculates the coverage by counting the non-zero elements in the antecedent array, representing the number of conditions covered. The complexity is determined by the number of rules in the chromosome. It assigns weights (w1 and w2) to coverage and complexity, respectively, and calculates the fitness value as the weighted sum of coverage minus complexity. The fitness value is printed for each chromosome, and it's returned as the output. This function is used in evolutionary algorithms to guide the selection and evolution of chromosomes toward solutions that maximize coverage while minimizing complexity.

```

def fitness_function_B(chromosome):
    chromosome_array = np.array([rule.antecedent for rule in chromosome])
    if len(chromosome_array.shape) == 1: # Check if it's a 1D array
        chromosome_array = np.array([chromosome_array]) # Reshape to 2D array if necessary
    coverage = np.count_nonzero(chromosome_array[:, :-1] != -1)
    complexity = len(chromosome)
    print('coverage: ' + str(coverage))
    print('complexity: ' + str(complexity))
    fitness_value = (coverage * 0.9) - (0.1 * complexity)
    print("Fitness B fitness value : " + str(fitness_value))
    return fitness_value

```

Diagram 3.3.1

For this evolutionary strategy, we develop a `roulette_wheel_selection` function that chooses individuals from a population based on their fitness scores using a probabilistic method. It calculates the total fitness of the population by adding up all the fitness scores and then assigns selection probabilities to each chromosome that are proportional to their fitness scores. It initiates by creating an empty 'selected' list and iterates through the population to randomly pick a number using the 'random.random()' function. It initializes current as 0 before using it to keep track of the cumulative probability. When the cumulative probability exceeds the random number, the corresponding chromosome is selected and added to the list of selected individuals until the desired number of individuals is selected. The function then returns the 'selected' list.

```
def roulette_wheel_selection(population, fitness_scores):
    total_fitness = sum(fitness_scores)
    probabilities = [f / total_fitness for f in fitness_scores]
    selected = []

    for _ in range(len(population)):
        pick = random.random()
        current = 0
        for individual, probability in zip(population, probabilities):
            current += probability
            if current > pick:
                selected.append(individual)
                break

    return selected
```

Diagram 3.3.1

For the crossing-over, we perform uniform cross-over to generate two offspring from two parent chromosomes by randomly swapping genes. We first determine the length of the shorter parent and perform gene swaps with a 50% probability for each position up to this length. For any remaining genes in the longer parent, we append them to the corresponding offspring. This process ensures that the offspring inherit a mix of genes from both parents while handling differing parent lengths appropriately.

```
def uniform_crossover(parent1, parent2):
    min_length = min(len(parent1), len(parent2))
    offspring1, offspring2 = [], []

    # Crossover for the length of the shorter parent
    for i in range(min_length):
        if random.random() < 0.5:
            offspring1.append(parent1[i])
            offspring2.append(parent2[i])
        else:
            offspring1.append(parent2[i])
            offspring2.append(parent1[i])

    # Handle excess genes in the longer parent
    if len(parent1) > min_length:
        offspring1.extend(parent1[min_length:])
    if len(parent2) > min_length:
        offspring2.extend(parent2[min_length:])

    return offspring1, offspring2
```

8

Diagram 3.3.3

The provided code defines two functions, "mutate_rule" and "mutate_chromosome", to introduce genetic mutations into fuzzy rules and chromosomes. In the "mutate_rule" function, we mutate a rule's antecedent with a certain probability by randomly selecting a feature and a new label ('low', 'medium', or 'high'), then combining the new antecedent with the existing one using the OR operator. In the "mutate_chromosome" function, we apply mutate_rule to each rule within a chromosome (a list of fuzzy rules), resulting in a new chromosome with potentially mutated rules. This process adds genetic diversity to the evolutionary algorithm.

```
def mutate_rule(rule, fuzzy_vars, mutation_rate):
    if random.random() < mutation_rate:
        # Mutate antecedent
        chosen_feature_index = np.random.randint(len(fuzzy_vars) - 1)
        new_label = random.choice(['low', 'medium', 'high'])
        # Create a new antecedent term based on random selection
        new_antecedent = fuzzy_vars[chosen_feature_index][new_label]
        combined_antecedent = rule.antecedent | new_antecedent
        # Create a new rule with the mutated antecedent and consequent
        new_rule = ctrl.Rule(antecedent=combined_antecedent, consequent=rule.consequent)
        return new_rule # Return the new, mutated rule
    else:
        return rule

# Example of applying mutation across a chromosome (list of rules)
def mutate_chromosome(chromosome, fuzzy_vars, mutation_rate):
    return [mutate_rule(rule, fuzzy_vars, mutation_rate) for rule in chromosome]
```

Diagram 3.3.4

A 'genetic_algorithm_B' function is developed to optimize a population of fuzzy logic rule-based systems over multiple generations by applying genetic algorithms. We first initialized 'fuzzy_population' based on parameters: 'population_size', 'num_rules', 'num_features', 'num_states_per_feature'. It tracks the best fitness score and corresponding chromosome using 'best_score' and 'best_chromosome', while 'best_scores' and 'best_chromosomes' store the best scores and chromosomes for each generation. A 'stagnation_counter' monitors periods without improvement to dynamically adjust the mutation rate within a range of 0.1 to 1.0. For each generation, the function calculates fitness scores, updates the best solutions, and adjusts the mutation rate based on stagnation. Elitism is employed by preserving the best chromosome in the next generation. Parents are selected using 'roulette_wheel_selection', and offspring are generated through 'uniform_crossover' and mutation via 'mutate_chromosome'. The function appends the best score for each generation to 'best_scores' and prints the progress. Finally, it returns the lists 'best_scores' and 'best_chromosomes', representing the optimization history.

```

def genetic_algorithm_B(population_size, num_generations, num_rules, num_features, mutation_rate, num_states_per_feature, crossover_rate, fuzzy_vars):
    # Initialize the population
    fuzzy_population = initialize_population(population_size, num_rules, num_features, num_states_per_feature)

    # Initialize variables to store the best solution and its fitness
    best_score = float('-inf')
    best_chromosome = None

    # Track the best score and best chromosome for each generation
    best_scores = []
    best_chromosomes = []

    # Track stagnation to adjust mutation rate
    stagnation_counter = 0

    # Main loop for each generation
    for generation in range(num_generations):
        # Calculate fitness scores for the current population
        fitness_scores = [fitness_function_B(chromosome) for chromosome in fuzzy_population]

        # Find the maximum fitness score in this generation
        max_fitness = max(fitness_scores)

        # Update best score and best chromosome if a better solution is found
        if max_fitness > best_score:
            best_score = max_fitness
            best_chromosome = fuzzy_population[fitness_scores.index(max_fitness)]
            best_chromosomes.append(best_chromosome)
            stagnation_counter = 0
        else:
            stagnation_counter += 1

        # Adjust mutation rate based on stagnation
        if stagnation_counter > 5:
            mutation_rate = min(1.0, mutation_rate + 0.05)
        else:
            mutation_rate = max(0.1, mutation_rate - 0.01)

        # Elitism: Keep the best chromosome in the next generation
        next_generation = [best_chromosome]

        # Select parents using roulette wheel selection
        selected = roulette_wheel_selection(fuzzy_population, fitness_scores)

        # Crossover and mutate to generate offspring
        while len(next_generation) < population_size:
            if random.random() < crossover_rate and len(selected) > 1:
                parent1, parent2 = random.sample(selected, 2)
                offspring1, offspring2 = uniform_crossover(parent1, parent2)
                next_generation.extend([offspring1, offspring2])
            else:
                next_generation.extend(random.sample(selected, 2))

        next_generation = next_generation[:population_size]

        next_generation = [mutate_chromosome(chromosome, fuzzy_vars, mutation_rate) for chromosome in next_generation]

        # Store the best score for this generation
        best_scores.append(best_score)

        # Print progress
        print(f"Generation {generation + 1}, Best Score: {best_score}")

    # Return the list of best scores and best chromosomes for each generation
    return best_scores, best_chromosomes

```

Diagram 3.3.5

It starts by preparing the dataset, where data contains all features except the target variable 'total_UPDRS', and labels contain the 'total_UPDRS' values. The parameters for the genetic algorithm are defined, including crossover_rate, num_generations, num_features, num_parents, mutation_rate, population_size, and num_states_per_feature, which specifies the number of states for each feature. The genetic algorithm is then executed using the genetic_algorithm_B function with these parameters, optimizing the fuzzy logic system according to Fitness Function B and returning the best scores and chromosomes for each generation. Finally, the best fitness scores per generation are plotted to visualize the optimization progress.

```

# Define your dataset and labels
data = df.drop(['total_UPDRS'], axis=1)
labels = df['total_UPDRS']

# Define parameters
crossover_rate = 0.8
num_generations = 50
num_features = 11
num_parents = 4
mutation_rate = 0.5
population_size = 30
num_states_per_feature = [3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3] # List containing the number of states for each feature

# Run genetic algorithm for Fitness Function B
best_scores, best_chromosomes = genetic_algorithm_B(population_size, num_generations, num_max_rules, num_features, mutation_rate, num_states_per_feature, crossover_rate, fuzzy_vars)

# Plotting the best scores
plt.plot(best_scores)
plt.title('Best Fitness Score per Generation')
plt.xlabel('Generation')
plt.ylabel('Best Fitness Score')
plt.show()

```

Diagram 3.3.6

4. Chromosome Analysis

4.1 Best Chromosomes Extraction

Initially, the code below extracts features from the conditions of each rule and converts them into numerical representation.

```

# Extract features from the structured rules
all_features = [[condition["feature"] for condition in rule["conditions"]] for rule in rules]

# Convert features into numerical representation
unique_features = list(set(feature for rule_features in all_features for feature in rule_features))
numerical_representation = np.zeros((len(rules), len(unique_features)))

for i, rule_features in enumerate(all_features):
    for j, feature in enumerate(unique_features):
        if feature in rule_features:
            numerical_representation[i, j] = 1

```

Diagram 4.1

4.2 Unsupervised Clustering Feature Importance

Next, the code applies unsupervised clustering using the KMeans algorithm to group the rules into a specified number of clusters. Subsequently, it analyzes each cluster to determine the importance of features within them, calculating the proportion of times each feature appears in the cluster. Finally, it visualizes the feature importance scores for each cluster through horizontal bar charts, providing insights into which features contribute most significantly within each rule cluster.

```
# Perform unsupervised clustering
num_clusters = 2 # Adjust as needed
kmeans = KMeans(n_clusters=num_clusters, random_state=0)
cluster_labels = kmeans.fit_predict(numerical_representation)

# Analyze clusters for feature importance
cluster_feature_counts = {cluster: {} for cluster in range(num_clusters)}
for i, cluster_label in enumerate(cluster_labels):
    rule_features = all_features[i]
    for feature in rule_features:
        cluster_feature_counts[cluster_label][feature] = cluster_feature_counts[cluster_label].get(feature, 0) + 1

# Calculate feature importance scores for each cluster
cluster_feature_importance = {}
for cluster, feature_counts in cluster_feature_counts.items():
    total_occurrences = sum(feature_counts.values())
    cluster_feature_importance[cluster] = {feature: count / total_occurrences for feature, count in feature_counts.items()}

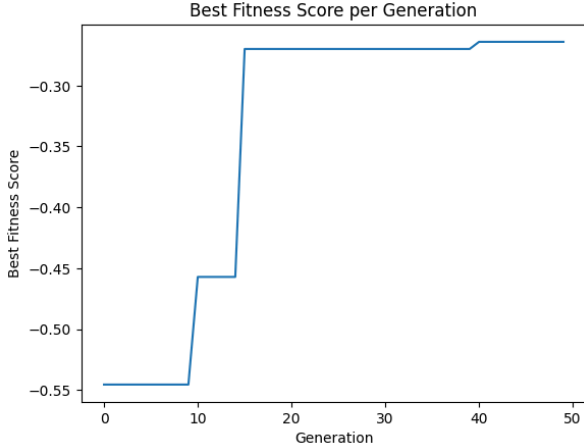
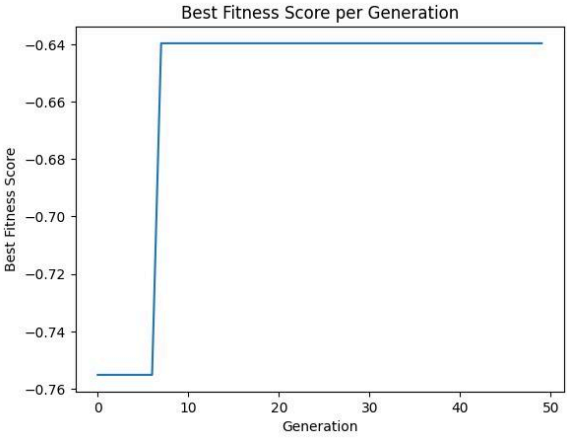
# Plot feature importance scores for each cluster
for cluster, importance_scores in cluster_feature_importance.items():
    sorted_importance_scores = dict(sorted(importance_scores.items(), key=lambda item: item[1], reverse=True))
    features = list(sorted_importance_scores.keys())
    importances = list(sorted_importance_scores.values())

    plt.figure(figsize=(10, 6))
    plt.barh(features, importances, color='skyblue')
    plt.xlabel('Importance')
    plt.title(f'Cluster {cluster} Feature Importance Scores')
    plt.gca().invert_yaxis()
    plt.show()
```

Diagram 4.2

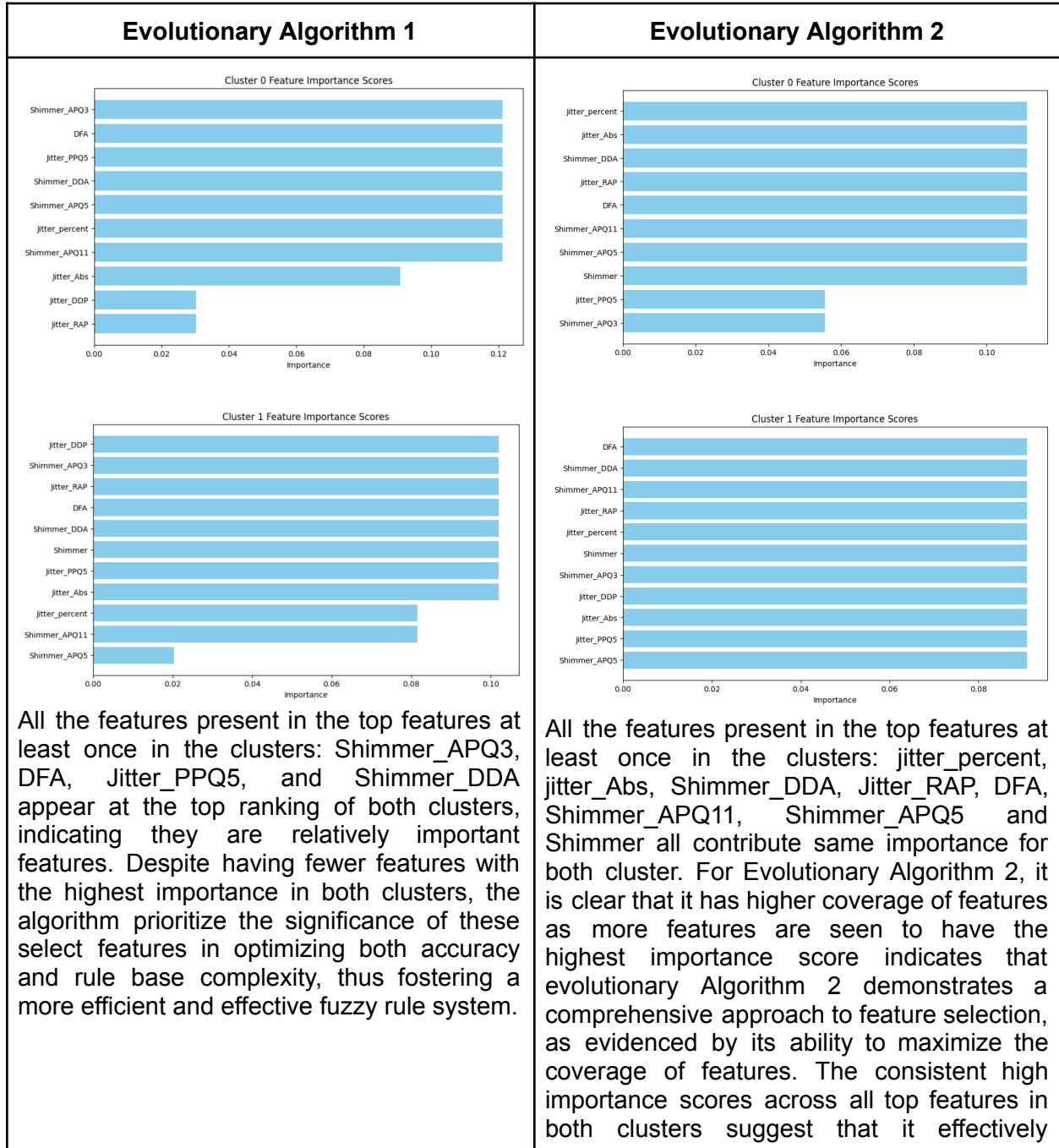
5. Result and Discussions

5.1 Compare EA Result

Evolutionary Algorithm 1	Evolutionary Algorithm 2
 <p>At the start (generation 0), the fitness score is approximately -0.55. In the first few generations, there is a sharp increase in the fitness score. This behavior aligns with the expected performance of optimization algorithms, where each generation aims to improve upon the previous one. The initial boost suggests that the algorithm successfully explores the solution space and identifies better solutions.</p> <p>Around generation 15, the fitness score levels off at approximately -0.28. This plateau indicates that further improvements are not significant. The lack of progress suggests that the algorithm might be trapped in a local optimum or that the problem space has inherent limitations, causing it to converge prematurely to a suboptimal solution. Nevertheless, it still demonstrates a good evolutionary algorithm with the increasing fitness value across the 50 generations.</p>	 <p>At the beginning (generation 0), the fitness score is approximately -0.75, indicating that the initial set of rules is not effective or efficient in achieving the desired outcome.</p> <p>Around generation 7, the score sharply rises to 0.64. This sudden improvement suggests that the genetic algorithm has successfully optimized the rules, leading to a significant enhancement in their performance. The rules generated around this point are likely better suited to capture the underlying patterns in the data, resulting in a substantial increase in their effectiveness.</p> <p>Following this sharp increase, the fitness score stabilizes and becomes constant. This indicates that the genetic algorithm has reached a plateau where further iterations or generations do not significantly improve the performance of the rules. At this stage, the rules have likely converged to a solution that adequately represents the data and achieves the desired outcome.</p>

In both scenarios of the evolutionary process, the fitness score initially starts low but shows a notable improvement over generations. Subsequently, the score stabilizes, suggesting convergence to a stable solution. Despite differences in the specific trajectories, both scenarios demonstrate the iterative refinement process of the genetic algorithm, leading to improved rule performance and eventual convergence to solutions that adequately represent the data and achieve desired outcomes.

5.2 Compare Feature Importance in EA



	captures the diverse aspects of the data, ensuring a robust representation of the underlying patterns and characteristics.
--	--

The reason that all the features have high feature importance values might be the feature dropping process conducted during data preprocessing for computational efficiency purposes. Besides, the rules may exhibit highly similar patterns, where each feature plays an equally important role in defining the behavior of the rules. This uniformity in feature importance suggests that the rules within the cluster are driven by a balanced combination of features, without any single feature dominating the decision-making process.

Through unsupervised clustering using the KMeans algorithm, rules with similar feature patterns are grouped into distinct clusters. Subsequently, within each cluster, the occurrence of features is meticulously analyzed to gauge their importance. Features that recurrently appear within a cluster are deemed more influential in defining the characteristics of rules within that cluster. By calculating feature importance scores as the proportion of feature occurrences relative to the total occurrences of all features within a cluster, a normalized measure of importance is obtained.

Visualization of these scores through horizontal bar charts offers a clear understanding of which features hold significant sway within each cluster. Thus, the evolutionary process effectively discerns important features by systematically clustering rules and analyzing feature occurrences within these clusters, ultimately providing valuable insights into the underlying patterns and determinants of rule behavior.

5.3 Compare Methodology EA1 and EA2

Fitness Function	
<p>Function A is used to maximize the fuzzy rule base accuracy and minimize the number of fuzzy rules.</p> <p>Pros:</p> <ul style="list-style-type: none"> • Direct Optimization for Accuracy: Directly targets improving the predictive performance of the fuzzy rule base, which is often the primary objective. • Simplicity Encouragement: By minimizing the number of rules, it promotes simpler and more interpretable models. 	<p>Function B is used to maximize the fuzzy rule base coverage with the minimum number of features.</p> <p>Pros:</p> <ul style="list-style-type: none"> • Encourages Generalization: Maximizing coverage ensures that the rules apply to a broad range of cases, which can lead to better generalization to new data. • Feature Reduction: By minimizing the number of features, it promotes simpler models that are easier to interpret and computationally less

<ul style="list-style-type: none"> Balanced Trade-off: Combines both accuracy and complexity into the fitness evaluation, aiming to find an optimal balance between the two. <p>Cons:</p> <ul style="list-style-type: none"> Risk of Overfitting: Focusing too heavily on accuracy can lead to models that perform well on training data but poorly on unseen data. Potential Loss of Important Rules: Simplifying the rule base by minimizing the number of rules might exclude some necessary rules, affecting the model's effectiveness. Complex Fitness Landscape: The fitness landscape can be complex, making it difficult for the evolutionary algorithm to navigate and find the global optimum. 	<p>intensive.</p> <ul style="list-style-type: none"> Enhanced Interpretability: Fewer features and broader coverage can make the fuzzy rule base more understandable and usable. <p>Cons:</p> <ul style="list-style-type: none"> Indirect Accuracy Optimization: Coverage does not always correlate with accuracy, potentially leading to less effective predictive performance. Risk of Overgeneralization: Broad coverage might produce rules that are too generic, missing specific but important nuances in the data. Balancing Act: The need to balance coverage and feature minimization might complicate the fitness function, requiring careful calibration to avoid suboptimal models.
Selection Strategies	
<p><u>Tournament Selection</u></p> <p>Pros:</p> <ul style="list-style-type: none"> Maintains diversity by selecting the best among a random subset of the population. Reduces the chance of premature convergence to local optima. Simple to implement and effective in maintaining competitive pressure. <p>Cons:</p> <ul style="list-style-type: none"> Can be computationally expensive with large populations. Selection pressure might be too high if the tournament size is too large, leading to a loss of diversity. May require fine-tuning of tournament size to balance exploration and exploitation. 	<p><u>Roulette Wheel Selection</u></p> <p>Pros:</p> <ul style="list-style-type: none"> Probability of selection is proportional to fitness, promoting better solutions. Maintains diversity by giving all individuals a chance to be selected. Simple and intuitive to implement. <p>Cons:</p> <ul style="list-style-type: none"> This can lead to premature convergence if high-fitness individuals dominate too early. Less effective if fitness values have small differences, leading to random selection. May struggle with maintaining diversity in later generations.
<p><u>One Point Crossover</u></p>	<p><u>Uniform Crossover</u></p>

<p>Pros:</p> <ul style="list-style-type: none"> • Simple and easy to implement. • Preserves large blocks of genes, maintaining useful building blocks. • Can introduce new genetic combinations, promoting diversity. <p>Cons:</p> <ul style="list-style-type: none"> • May disrupt useful gene combinations if the crossover point is poorly chosen. • Limited exploration of the search space due to single crossover points. • Can lead to less diverse offspring if parent chromosomes are similar. 	<p>Pros:</p> <ul style="list-style-type: none"> • Promotes higher diversity by mixing genes from both parents more thoroughly. • Provides more exploration of the search space, potentially finding better solutions. • Flexibility in gene combination, not restricted by crossover points. <p>Cons:</p> <ul style="list-style-type: none"> • Can disrupt beneficial gene combinations more frequently. • May require more offspring to find optimal solutions due to higher disruption. • More complex to implement compared to one-point crossover.
<p><u>Mutation at Consequent</u></p> <p>Pros:</p> <ul style="list-style-type: none"> • Fine-tunes the output of rules, improving accuracy. • Helps escape local optima by introducing small changes. • Maintains genetic diversity, crucial for evolutionary progress. <p>Cons:</p> <ul style="list-style-type: none"> • May not significantly alter the rule's behavior if consequences have small impact. • Can be computationally expensive if applied too frequently. • Requires careful balancing to avoid disrupting effective rules. 	<p><u>Mutation at Rule</u></p> <p>Pros:</p> <ul style="list-style-type: none"> • Introduces significant changes, potentially discovering new useful rules. • Can greatly increase diversity, essential for exploring the search space. • Helps avoid stagnation by introducing entirely new rules. <p>Cons:</p> <ul style="list-style-type: none"> • High risk of creating ineffective or harmful rules. • May disrupt beneficial rule sets, reducing overall fitness. • Requires careful rate adjustment to balance exploration and exploitation

In short, EA1 uses Fitness Function A to maximize accuracy and minimize the number of fuzzy rules, employing tournament selection, one-point crossover, and mutation at the consequent. EA2 uses Fitness Function B to maximize coverage and minimize the number of features, employing roulette wheel selection, uniform crossover, and mutation at the rule. Evolutionary algorithm 1 (EA1)'s direct optimization for accuracy and simplicity suits scenarios

requiring high predictive performance and model interpretability. However, its risk of overfitting and need for careful balance between accuracy and rule simplicity must be managed. On the other hand, evolutionary algorithm 2 (EA2) focuses on generalization and feature reduction makes it ideal for creating broadly applicable, interpretable models. Still, its indirect accuracy optimization and the potential for overgeneralization require careful calibration. Ultimately, the choice between EA1 and EA2 depends on the specific needs of the application, such as the importance of accuracy versus interpretability and the complexity of the fitness landscape.

6. Conclusion

In conclusion, we have explored the application of evolutionary algorithms and fuzzy control systems to predict the progression of Parkinson's disease through telemonitoring of voice measurements. We have implemented two different evolutionary strategies, Evolutionary Algorithm 1 (EA1) focused on maximizing accuracy while minimizing the number of fuzzy rules, and Evolutionary Algorithm 2 (EA2) aimed at maximizing rule coverage with a reduced number of features. By comparing two distinct evolutionary strategies, we identified that both approaches have unique advantages and challenges. Through comparative analysis, it was found that EA1's direct optimization for accuracy is beneficial for scenarios requiring high predictive performance, although it poses a risk of overfitting. On the other hand, EA2's emphasis on generalization and feature reduction provides broader applicability and interpretability, albeit at the cost of indirect accuracy optimization. This assignment demonstrates the potential of evolutionary computing in enhancing telemonitoring frameworks for Parkinson's disease, paving the way for more efficient and scalable clinical assessments.

Despite the promising results, several limitations should be considered. One major limitation is the risk of overfitting, particularly with EA1, which focuses on maximizing accuracy. Overfitting can result in models that perform exceptionally well on training data but fail to generalize to new, unseen data, necessitating careful tuning of the algorithm to achieve a balance between accuracy and generalizability. Another limitation is the computational complexity involved in both evolutionary algorithms. The iterative processes of population generation, fitness evaluation, selection, crossover, and mutation can be computationally intensive and time-consuming, particularly when dealing with large datasets.

Additionally, the performance of EAs is highly sensitive to the choice of parameters such as population size, mutation rate, crossover rate, and the number of generations. Inappropriate parameter settings can significantly impact the effectiveness of the algorithms. EAs can also converge prematurely to local optima, failing to find the global optimal solution. This limitation is particularly evident in complex fitness landscapes where the search space has many suboptimal

peaks. Furthermore, although EA2 aims to minimize the number of features for simpler models, the interpretability of the generated fuzzy rules can still be challenging, especially when dealing with high-dimensional data. The rules may become complex, making it difficult to draw clear and actionable insights.

In summary, while evolutionary algorithms offer significant potential in enhancing telemonitoring frameworks for Parkinson's disease, careful consideration of their limitations is essential. Addressing these challenges will be crucial for the success

References

- Tsanas, A., & Little, M. (2009). *Parkinsons Telemonitoring Data Set*. UC Irvine Machine Learning Repository. <https://archive.ics.uci.edu/dataset/189/parkinsons+telemonitoring>
- Tata, V. (2018, August 18). *An Introduction to Evolutionary Algorithms and Code with Genetic Algorithm in Unity*. Medium. <https://venkateshtata9.medium.com/an-introduction-to-evolutionary-algorithms-and-code-part-1-theory-behind-genetic-algorithm-df75af08d5d6>
- Kharrazi, H., & Khanmohammadi, S. (2008). *Genetic Algorithm Combined with H^∞ Filtering for Optimizing Fuzzy Rules and Membership Functions*. *Journal of Applied Sciences*, 8(20), 3439-3445. <https://doi.org/10.3923/jas.2008.3439.3445>
- Korzhakin, D. A., & Sugiharti, E. (2021). *Implementation of Genetic Algorithm and Adaptive Neuro Fuzzy Inference System in Predicting Survival of Patients with Heart Failure*. *Sistemasi*, 8(2). <https://doi.org/10.15294/sji.v8i2.32803>