11753 - Computational Intelligence
Intelligent Systems
Universitat de les Illes Balears

GENERATING FUZZY RULES FOR TRAFFIC LIGHT TIMING (VEHICLES) USING NUMERICAL DATA

Charnota, Sofiia
Alsatouf, Abdulrahman
Manasut, Phornphawit
Manrique Villa, Camilo José

Supervisor:
Massanet, Sebastián

1st Semester

January 13, 2024

Contents

1 Introduction

This project revolves around the generation of fuzzy rules for traffic light timing, aiming to improve the efficiency of traffic control systems based on numerical data. Drawing inspiration from an algorithm for generating fuzzy rules from numerical input data [1], our objective is to develop adaptable traffic control rules. These rules will provide guidance for making decisions related to traffic light timing, considering various factors like traffic density and pedestrian activity.

The primary focus of this project is the application of the method proposed in [1] to address the generation of fuzzy rules tailored specifically for traffic light timing. Our main goal is to implement and fine-tune the algorithm to suit a particular traffic management scenario. This comprehensive process includes data generation, algorithm implementation, parameter optimization, and thorough performance evaluation. The end result will be a set of adaptable and effective traffic control rules that can enhance traffic management systems.

2 Implementation

2.1 FuzzyInference Class

`FuzzyInference` class is the implementation of [1] in object-oriented style. The algorithm was implemented to generate fuzzy rules based on the normalized input and output data. The algorithm consists of the following steps:

- Compute membership values for input data using a specified number of clusters (k).
- Calculate degrees of compatibility for each data point.
- Compute weights for each data point.
- Derive consequent real values for the fuzzy rules.
- Derive linguistic labels and predict traffic light timing.
- The Performance Index (PI) was also calculated as a measure of quality of the generated rules.

Each function of the class will be shown in each figure followed by a brief explanation. Since each function is accompanied by the function comment, the explanation in this report about the function will focus on their corresponding section in [1]. The explanation will start from function at the top of the class to the bottom. The string "..." before or after the function implies that there are other code occupying this space.

2.1.1 Initialization Function

Listing 1: FuzzyInference Class `__init__()` Function

```
class FuzzyInference:
    def __init__(self,
                 input_mat,
                 output_vec):
```

```
        self.input_mat = input_mat
        self.output_vec = output_vec
    ...
```

The `__init__()` function which is the function used to instantiate the object of the class in Python. Hereby, the input data and the output data are the parameters to be kept as attribute to use for this object.

2.1.2 Compute Membership Value Function

Listing 2: FuzzyInference Class `compute_membership_value()` Function

```
class FuzzyInference:
    ...
    @staticmethod
    def compute_membership_value(x, j, k):
        """
        Calculate the membership function value for a given input
            value 'x' in a triangular fuzzy set.

        Parameters:
        x (numpy.ndarray or float): The input value or array of
            values of dim (number of sample, 1)
        j (int): The index of the membership function within the
            range [1, k].
        k (int): The total number of membership functions in the set.

        Returns:
        membership (numpy.ndarray): The membership function values
            for the input 'x'.
        """
        a = (j - 1) / (k - 1)
        b = 1 / (k - 1)
        tmp_membership = 1 - np.abs(x - a) / b
        tmp_membership[tmp_membership < 0] = 0
        membership = tmp_membership
        return membership
    ...
```

The `compute_membership_value()` function plays a critical role in fuzzy logic systems, defining the degree of membership of data points to various fuzzy sets. In this implementation, we utilize triangle-shaped membership functions to model the relationship between input variables and fuzzy rule conditions from the paper. This function only computes the membership value for all samples of a single feature and is utilized by the multidimensional `compute_membership_values_multidim()` function to compute the membership values for the entire dataset. This function is a part of equation (3) section 2.2.1 in [1].

2.1.3 Generate Combinations Function

Listing 3: FuzzyInference Class `generate_combinations()` Function

```
class FuzzyInference:
    ...
    @staticmethod
    def generate_combinations(k):
        """
        Generate all possible combinations for different k values
            using itertools.

        Args:
        k (list): A list specifying the number of fuzzy sets for each
            feature.

        Returns:
        combinations (list): A list of lists, where each inner list
            represents a combination of fuzzy set indices for
        each feature.
        num_combinations (int): An integer specifying the number of
            all generated combinations.
        """
        combinations = list(itertools.product(*[range(k_i) for k_i in
            k]))
        num_combinations = len(combinations)

        return combinations, num_combinations
    ...
```

This function generates combinations for calculating the degree of compatibility function and prediction function. It is mainly used to combine fuzzy labels of each feature for other functions to generate the simplified if-then rules which corresponds to the section 2.2.2 of [1].

2.1.4 Generate Partitions Function

Listing 4: FuzzyInference Class `generate_partitions()` Function

```
class FuzzyInference:
    ...
    @staticmethod
    def generate_partitions(k, min_val, max_val):
        """
        Generate an array with a 3-element list for each of the k
            partitions with values of a triangular shape function.

        Args:
        k (int): The number of partitions.
        min_val (int): The starting value of the range.
        max_val (int): The ending value of the range.
```

```
12
13          Returns:
14          spaces (np.ndarray): An array of shape (k, 3) where each row
                represents a triangular shape function with
15          the format [beginning, peak value, ending] points.
16          """
17          if k < 2:
18              raise ValueError("k must be at least 2")
19          spaces = np.zeros((k, 3))
20          portion = (max_val - min_val) / (k - 1)
21          spaces[0] = [min_val, min_val, min_val + portion]
22          spaces[k - 1] = [max_val - portion, max_val, max_val]
23          for i in range(1, k - 1):
24              peak_value = min_val + i * portion
25              spaces[i] = [peak_value - portion, peak_value, peak_value
                    + portion]
26
27          return spaces
28      ...
```

This function generates triangle fuzzy space for each fuzzy labels given the number of fuzzy labels for the feature with minimum and maximum values. This corresponds to the fuzzy subspaces mentioned in the section 2.2.1 of [1].

2.1.5 Compute Weights Function

Listing 5: FuzzyInference Class `compute_weights()` Function

```
1 class FuzzyInference:
2     ...
3     @staticmethod
4     def compute_weights(degree_of_compatibility, alpha):
5         """
6         Compute weights from degrees of compatibility using a power
              transformation.
7
8         Args:
9         degree_of_compatibility (np.ndarray): An array of degrees of
              compatibility.
10        alpha (float): A positive constant that controls the power
              transformation.
11
12        Returns:
13        weights (np.ndarray): An array of weights computed by raising
              the degrees of compatibility to the power of alpha.
14        """
15        return np.power(degree_of_compatibility, alpha)
16    ...
```

Weights are derived based on the degrees of compatibility. A power transformation is applied to these degrees of compatibility, controlled by a parameter (α). This transformation amplifies the influence of well-matched rule conditions and downplays the effect of less compatible rules. The resulting weights reflect the significance of each rule for each data point. The function directly corresponds to equation (9) of section 3.1 in [1].

2.1.6 Compute Membership Values Multidim Function

Listing 6: FuzzyInference Class `compute_membership_values_multidim()` Function

```
class FuzzyInference:
    ...
    def compute_membership_values_multidim(self, x, k):
        """
        Calculate the membership function values for a dataset 'x'
            with triangular fuzzy sets.

        Parameters:
        x (numpy.ndarray): The input dataset of shape (num_samples,
            num_features).
        k (numpy.ndarray): An array specifying the number of
            membership functions for each feature.

        Returns:
        membership_values (numpy.ndarray): A multidimensional array
            of membership function values for the input data 'x'.
                    The shape of the returned array is
                        (num_samples, num_features, max(k)).
        """
        num_samples, num_features = x.shape
        max_member = np.max(k)
        membership_values = np.zeros((num_samples, num_features,
            max_member))
        for i in range(num_features):
            for j in range(max_member):
                membership_values[:, i, j] =
                    self.compute_membership_value(x[:, i], j + 1,
                    k[i])
        return membership_values
    ...
```

This is the version of `compute_membership_value()` that encompasses multiple features. It corresponds to equation (3) in section 2.2.1 in [1] but it involves all the features.

2.1.7 Compute Degrees of Compatibility Function

Listing 7: FuzzyInference Class `compute_degrees_of_compatibility()` Function

```
class FuzzyInference:
    ...
```

```
    def compute_degrees_of_compatibility(self, membership_values, k):
        """
        Compute the degree of compatibility for all possible
            combinations of fuzzy rule conditions.

        Args:
        membership_values (list): A list of membership values for
            each sample and feature.

        Returns:
        degree_of_compatibility (list): A list of lists, where each
            element represents the degree of compatibility for all
        combinations of conditions for each sample.
        """

        def generate_product_values_per_combination(si, comb):
            return np.array([membership_values[si][0][comb[j]] if
                len(membership_values[si]) == 1 \
                                    else
                                        membership_values[si][j][comb[j]]
                                        for j in range(len(k))])

        degree_of_compatibility = []
        combinations = self.generate_combinations(k)[0]
        for i in range(len(membership_values)):
            sample_combinations = [np.product(
                generate_product_values_per_combination(i,
                    combination)) for combination in combinations]
            degree_of_compatibility.append(sample_combinations)
        return degree_of_compatibility
    ...
```

The function calculates the degree of compatibility for each data point with all possible combinations of fuzzy rule conditions given membership values. It assesses how well a data point matches different rule conditions by taking into account the combination of membership values across variables. This step captures the applicability of various rules to each data point. It corresponds directly to equation (8) in section 2.3 of [1].

2.1.8 Compute Consequent Values Function

Listing 8: FuzzyInference Class `compute_consequent_values()` Function

```
class FuzzyInference:
    ...
    @staticmethod
    def compute_consequent_values(weights, output_data):
        """
        Compute the consequent values for different combinations of
            fuzzy rule conditions.

```

```
        Args:
        weights (numpy.ndarray): An array of weights for different
            rule combinations with
        shape [num_samples, num_combinations].
        output_data (numpy.ndarray): The true output values for each
            sample with shape [num_samples, 1].

        Returns:
        consequent_values (numpy.ndarray): An array of consequent
            values for each combination of fuzzy rule conditions
        with shape [num_combinations, 1].
        """
        num_combinations = weights.shape[1]
        numerator, denominator = np.zeros(num_combinations),
            np.zeros(num_combinations)
        for j in range(num_combinations):
            numerator[j] = np.dot(weights[:, j], output_data)
            denominator[j] = np.sum(weights[:, j])
        return (numerator / denominator).reshape(num_combinations, 1)
    ...
```

The function calculates the consequent values for different combinations of fuzzy rule conditions. It does so by considering the weights assigned to each rule for a specific data point and using these to predict the traffic light timing. This step is crucial for obtaining the output values based on the input data and fuzzy rule conditions. The function corresponds to equation (7) in section 2.3 of [1].

2.1.9 Derive Linguistic Labels and Predict Function

Listing 9: FuzzyInference Class `derive_inguistic_labels_and_predict()` Function

```
class FuzzyInference:
    ...
    def derive_linguistic_labels_and_predict(self, x, b, B, k):
        """
        Derive linguistic labels and predict output values using
            fuzzy inference.

        Parameters:
        x (numpy.ndarray): Input data for prediction with shape
            (num_samples, num_features).
        b (numpy.ndarray): Input data representing fuzzy rules with
            shape (num_rules, num_features).
        B (numpy.ndarray): Range of fuzzy sets for each feature with
            shape (num_features, num_fuzzy_sets).
        k (numpy.ndarray): Number of fuzzy sets for each feature with
            shape (num_features,).

        Returns:
        tuple: A tuple containing:
```

```
            - output_inferred (numpy.ndarray): Inferred output values
                with shape (num_samples,).
            - main_table (numpy.ndarray): Main linguistic labels for
                each combination of features with
            shape (num_combinations, num_features).
            - secondary_table (numpy.ndarray): Secondary linguistic
                labels for each combination of features with
            shape (num_combinations, num_features).
        """
        num_samples, num_features = x.shape
        combinations, num_combinations = self.generate_combinations(k)

        membership_values =
            self.compute_membership_values_multidim(b, B)
        membership_values_x =
            self.compute_membership_values_multidim(x, k)

        degree_of_compatibility = np.array(
            self.compute_degrees_of_compatibility(
            membership_values_x, k))

        main_table = np.argmax(membership_values, axis=-1)
        secondary_table = np.argsort(-membership_values, axis=-1)[:,
            :, 1]

        membership_main = np.max(membership_values, axis=-1)
        membership_secondary = np.partition(membership_values, -2,
            axis=-1)[:, :, -2]

        space = self.generate_partitions(B[0], 0, 1)

        numerator = np.zeros([num_samples, num_combinations])
        denominator = np.zeros([num_samples, num_combinations])
        for i in range(num_combinations):
            # Determine the center values of the fuzzy sub-spaces
            B1 = space[main_table[i]]
            B2 = space[secondary_table[i]]

            # Calculate numerator and denominator of the inferred
                output
            numerator[:, i] = (degree_of_compatibility[:, i] *
                B1[0][1] * membership_main[i] +
                                degree_of_compatibility[:, i] *
                                   B2[0][1] * membership_secondary[i])
            denominator[:, i] = (degree_of_compatibility[:, i] *
                membership_main[i] +
                                  degree_of_compatibility[:, i] *
                                     membership_secondary[i])

```

```
        output_inferred = np.sum(numerator, axis=1) /
            np.sum(denominator, axis=1)
        return output_inferred, main_table.reshape(k),
            secondary_table.reshape(k)
    ...
```

With the consequent values in hand, the algorithm proceeds to derive linguistic labels for each combination of fuzzy rule conditions, offering an intuitive representation of the desired output in term of the index of fuzzy set of the output. This function corresponds to the entirety of section 4.1 in [1].

2.1.10 Run Function

This function is the main function to run the entire pipeline from calculation of memberships to the deriving linguistic and prediction of labels. It does not convert the numeric labels to linguistic string. This is left for an external function because so that evaluation of different parameters can be easily done.

Listing 10: FuzzyInference Class `run()` Function

```
class FuzzyInference:
    ...
    def run(self, k=np.array([5, 5]), alpha=5, B=np.array([5])):
        """
        Parameters:
        k (numpy.ndarray): Number of fuzzy sets for each feature with
            shape (num_features,).
        alpha (float): A positive constant that controls the power
            transformation.
        B (numpy.ndarray): Range of fuzzy sets for each feature with
            shape (num_features, num_fuzzy_sets).

        Returns:
        pi
        main_table (numpy.ndarray): Main linguistic labels for each
            combination of features with
             shape (num_combinations, num_features).
        secondary_table (numpy.ndarray): Secondary linguistic labels
            for each combination of features with
             shape (num_combinations, num_features).
        output_inferred (numpy.ndarray): Inferred output values with
            shape (num_samples,).
        consequent_values (numpy.ndarray): An array of consequent
            values for each combination of fuzzy rule conditions
        with shape [num_combinations, 1].
        """

        membership_values =
            self.compute_membership_values_multidim(self.input_mat, k)
```

```
        degree_of_compatibility =
            self.compute_degrees_of_compatibility(membership_values,
            k)
        weights = self.compute_weights(degree_of_compatibility, alpha)
        consequent_values = self.compute_consequent_values(weights,
            self.output_vec)
        output_inferred, main_table, secondary_table =
            self.derive_linguistic_labels_and_predict(self.input_mat,



        pi = (1 / self.input_mat.shape[0]) * np.sum((self.output_vec
            - output_inferred) ** 2)

        return pi, main_table, secondary_table, output_inferred,
            consequent_values
    ...
```

2.1.11 Evaluate Performance Function

Listing 11: FuzzyInference Class evaluate_performance() Function

```
class FuzzyInference:
    ...
    def evaluate_performance(self, k_values, alpha_values):
        """
        Parameters:
        k_values (List[int]): List of values of k to generate k by k
        alpha_values (List[int]): Alpha values to search

        Returns:
        pi (numpy.ndarray): List of performance indices at specific
            alpha and k values in (number of alpha values, number of
            k values)
        best_pi (float):  Best performance index
        best_alpha (float):  Alpha of best performance index run
        best_k (int): K of the best performance index run
        """
        num_k_values = k_values.shape[0]
        num_alpha_values = alpha_values.shape[0]

        pi = np.zeros([num_alpha_values, num_k_values])

        for i in range(num_k_values):
            k = np.array([k_values[i], k_values[i]])
            for j in range(num_alpha_values):
                alpha = alpha_values[j]
                pi[j, i],_,_,_,_ = self.run(k, alpha, k)
```

```
25
26          best_pi = np.min(pi)
27          best_alpha = alpha_values[np.where(pi == np.min(pi))[0]][0]
28          best_k = k_values[np.where(pi == np.min(pi))[1]][0]
29
30          return pi, best_pi, best_k, best_alpha
```

The algorithm was evaluated for different combinations of parameters, including the number of clusters K and the parameter α. The goal was to find the combination of parameters that results in the best PI. The best combination of parameters was determined based on the lowest PI value. The report presents the best combination of parameters and the corresponding PI, which provides insights into the quality of the generated fuzzy rules. Basically, this function executes the `run()` Function multiple times in order to evaluate whole parameter space similarly to grid search.

2.2 Misc Functions

For this section, we shall omit the code pasting, because it takes too much space and the functions detailed here are intuitive.

2.2.1 Table to Linguistic Function

The output table from a `FuzzyInference` object is converted to its linguistic representation using this function. The inputs to this function, which are linguistic labels of each feature and the output, are defined by the user. These inputs should match the dimension of the table and the output labels.

2.2.2 Normalize Data Function

Data is normalized into input for the algorithm. A min-max normalization is performed on the given data and its dynamical range. The dynamical range of a feature can be thought of as the output range we wish the normalization values to be within.

2.2.3 Denormalize Data Function

A reverse of the previous function. Given the dynamical feature range, min-value, max-value, and normalized data, convert the data into the range before being normalized.

3 Application

3.1 Problem

Traffic control plays a crucial role in optimizing the flow of vehicles and ensuring pedestrian safety in urban areas. The timing of traffic lights is a critical aspect of traffic control systems, and it needs to adapt to varying conditions such as traffic density and pedestrian activity. In this context, the problem at hand is to develop a robust and adaptable system for generating **traffic control rules** for traffic light timing.

To address this problem, we employ a fuzzy rule-based system. This system takes numerical input data, considering it in its crisp form without requiring fuzzification. The aim is to generate a set of rules that directly produce numerical values for optimizing traffic light timings.

3.2 Methodology

A fuzzy rule-based system can be used to generate traffic light timing rules based on numerical traffic density (vehicle/minute) and pedestrian activity (pedestrian/minute) by following these steps:

1. Fuzzification: Convert the numerical inputs of traffic density and pedestrian activity into fuzzy linguistic variables. Traffic density is categorized as low, medium, or high, while pedestrian activity is categorized as low, medium, or high.

2. Rule Base: Create a rule base that consists of fuzzy if-then rules. These rules define the relationship between the fuzzy inputs and the desired traffic light timing. For example, a rule is *"IF vehicle density is high AND pedestrian activity is low, THEN green light time medium"*

3. Rule Inference: Use the fuzzy if-then rules with consequent output to determine the appropriate traffic light timing based on the fuzzy inputs. This is done by selecting the (linguistic) output label corresponding to the consequent fuzzy set B^* that is characterized by the largest membership value $\mu_{B^*}(b)$ for a consequent output $b \in \mathbb{R}$. Moreover, the second-largest membership value $\mu_{B^{**}}(b)$ is stored in order to generate a subordinate output label. The inferred linguistic labels are stored into two tables called the main and secondary rule table representing all inferred if-then rules linguistically.

4. Optimization: Fine-tune the fuzzy rule-based system by adjusting the parameters, e.g. the parameter α, to optimize the process of predicting traffic light timing.

5. Output Prediction: Based on the inferred rule tables generated by fuzzy rule-based system, the traffic light timing for vehicles is predicted and the performance is measured using the mean squared error.

Overall, by using a fuzzy rule-based system, traffic light timing rules can be generated based on numerical inputs of traffic density and pedestrian activity, allowing for adaptive and efficient control of traffic signals.

3.3 Data Generation

In the context of our proposed traffic control problem, the generation of synthetic data is a critical step for algorithm development and testing. Listing 12 describes how an artificial dataset is created in order to simulate real-world scenario involving traffic and pedestrian dynamics. The aim of the the synthetic data generation is to observe the capabilities of algorithm to learn rules in the environment that there are clear observable patterns. If we had use the real world data, we can only make hypothesis about what our result should look like and is not able to observe the performance of the algorithm.

Listing 12: Simulation of artificial data for the traffic light timing problem

```
num_data_points = 500
data = []
for _ in range(num_data_points):
    traffic_density = random.randint(0, 45)
    pedestrian_activity = random.randint(0, 50)
    traffic_light_timing =
        np.round(utl.generate_traffic_light_timing(traffic_density,
        pedestrian_activity), 2)
    data.append([traffic_density, pedestrian_activity,
        traffic_light_timing])
data = np.array(data)
```

To determine the green light timing, the function considers the following assumptions for both traffic density and pedestrian activity:

1. Traffic Density Rule:
    - Low (0-12): Corresponds to an area with approximately 0-4 rows of cars, indicating low traffic.
    - Medium (12-30): Represents an area with roughly 4-10 rows of cars, indicating moderate traffic.
    - High (30-45): Implies an area with approximately 10-15 rows of cars, signifying high traffic.
2. Pedestrian Activity Rule:
    - Low (0-15): Denotes low pedestrian activity, with approximately 0-15 people per minute.
    - Medium (15-35): Represents moderate pedestrian activity, with around 15-35 people per minute.
    - High (35-50): Indicates high pedestrian activity, with approximately 35-50 people per minute.

The output green light timing are split into 3 ranges just like the input features. They are as follow:

- Short: 20 to 30 seconds.
- Medium: 30 to 45 seconds.
- Long: 45 to 60 seconds.

We randomized green light timing to those ranges based on the given traffic density and pedestrian activity. Since there are 3 ranges per input features, we hand crafted 9 rules to generate the green light timing. The rules can be referred to in Listing 13.

Listing 13: Simulation of artificial data for the traffic light timing problem

```
def generate_traffic_light_timing(traffic_density,
    pedestrian_activity):
    if traffic_density <= 12 and pedestrian_activity <= 15:
        return random.uniform(30, 45)
    elif traffic_density <= 12 and 15 < pedestrian_activity <= 35:
        return random.uniform(20, 30)
    elif traffic_density <= 12 and pedestrian_activity > 35:
        return random.uniform(20, 30)
    elif 12 < traffic_density <= 30 and pedestrian_activity <= 15:
        return random.uniform(45, 60)
    elif 12 < traffic_density <= 30 and 15 < pedestrian_activity <=
        35:
        return random.uniform(45, 60)
    elif 12 < traffic_density <= 30 and pedestrian_activity > 35:
        return random.uniform(30, 45)
    elif 30 < traffic_density and pedestrian_activity <= 15:
        return random.uniform(45, 60)
    elif 30 < traffic_density and 15 < pedestrian_activity <= 35:
        return random.uniform(45, 60)
    elif 30 < traffic_density and pedestrian_activity > 35:
        return random.uniform(30, 45)
    return random.uniform(20, 60)
```

Table 1 shows part of the data simulated for the problem. The dataset comprises 500 data samples, each representing a specific combination of traffic density, pedestrian activity, and greenlight timing in seconds. These data points are generated based on the description mentioned in previous sections.

ID	Traffic Density (cars/min)	Pedestrian Activity (pedestrian/min)	Greenlight Time (s)
0	42.0	25.0	52.84
1	41.0	46.0	36.88
2	23.0	11.0	51.05
3	35.0	28.0	49.26
4	29.0	37.0	31.12
5	23.0	6.0	52.75
6	4.0	2.0	43.61
7	8.0	29.0	34.04
8	13.0	23.0	54.84
9	39.0	0.0	53.35
10	17.0	16.0	58.99

Figure 1: A set of simulated data samples

On Figure 2, a histogram is created to visualize the distribution of traffic light timing for vehicles. Given the rules defined to simulate the data, the traffic light timing seems to follow a tri-modal, left-skewed distribution.

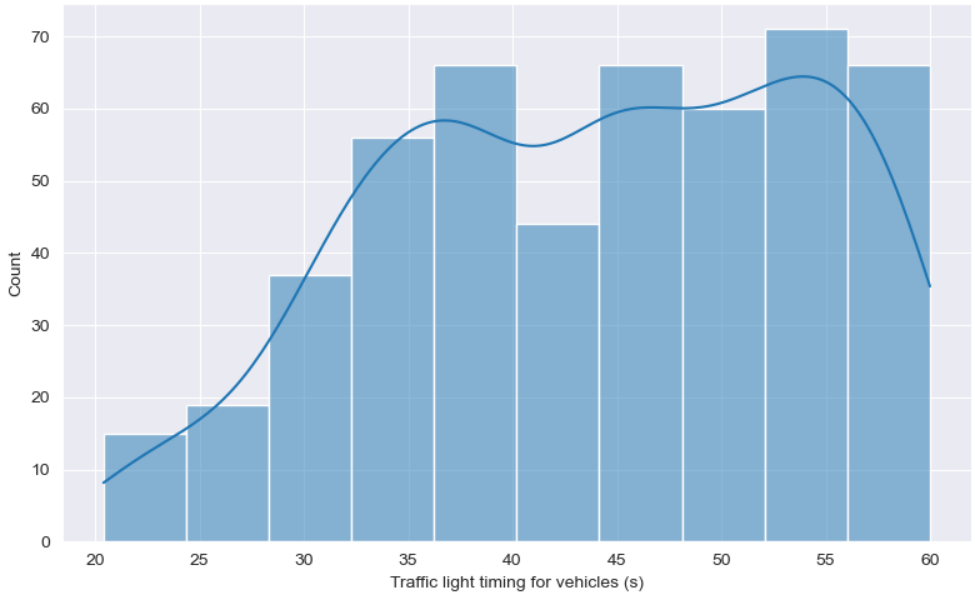

Figure 2: The distribution of the simulated traffic light timing

To ensure consistency and robustness while deploying the algorithm, the data is normalized using the `normalize_data()` function. Normalization scales the data to a common range, facilitating the learning process.

3.4 Evaluation

The Performance Index (PI) for different values of the parameters K and α is indicated in Table 1. The best performance is 0.0225, and is achieved with values $K = 5$ and $\alpha = 2$. The main rule table of the best run is shown in Table 2. The secondary rule table of the best run is shown in Table 3. Each row represents the pedestrian activity while the columns represent traffic density. The labels for each of the features is as follows. Both input features have the same labels - {Very Low, Low, Medium, High, Very High}. Their abbreviations are {VL, L, M, H, VH}, respectively. The output labels are {Very Short, Short, Medium, Long, Very Long} which correspond to {VS, S, M, L, VL}.

α	K ($K_1 = K_2$)			
	$K = 2$	$K = 3$	$K = 4$	$K = 5$
0.1	0.0614	0.0397	0.0295	0.0240
0.5	0.0532	0.0346	0.0275	0.0232
1	0.0469	0.0306	0.0262	0.0227
2	0.0418	0.0269	0.0253	**0.0225**
5	0.0471	0.0249	0.0258	0.0241
10	0.0568	0.0255	0.0294	0.0283
20	0.0598	0.0272	0.0357	0.0332
50	0.0604	0.0292	0.0389	0.0365
100	0.0618	0.0297	0.0382	0.0375

Table 1: Performance Index (PI) for different values of the parameters K and α

In Table 2 (aka the main table), we can observe that there are only four output labels instead of five. Seemingly, the algorithm has determined that only the label "Long" for greenlight timing can be inferred with high membership value. This set of data allows the fuzzy model to learn the trend that the higher the traffic density, the longer the greenlight timing is. However, it also intuitively learns that if there is a higher number of pedestrians, some concession will have to be made and the greenlight timing has to be reduced to either medium, short, or very short.

On the other hand, Table 3 (aka the secondary table) includes all five possible output labels. However, the rules are not very consistent as compared to the main rule table. For example, when there is very low vehicle density and high pedestrian activity, the secondary table results in medium greenlight timing. This is despite the fact that at medium pedestrian activity and the same vehicle density, the greenlight timing is short. In addition, the secondary table gives high pedestrian activity at the same vehicle density a short greenlight timing. However, this is not the case according to the main table where the output moves from medium to short and then very short greenlight timing when the pedestrian activity increases from medium to very high at a very low traffic density. Thus, this shows an inconsistent change in the output labels for the secondary rule table.

Moreover, the inconsistent changes in the output labels for the secondary rule table also result in a large change in the inferred output labels. For example, if we move from medium to low traffic density at very low pedestrian activity in the secondary table, the greenlight timing jumps from very long to medium instead of changing to long. This observation would not be observed at all for the case of one step feature label change in the main rule table. It can be noted that these inconsistencies may have arisen given the fact that the data was generated artificially. After all, the rules are learnt from the data.

Pedestrian activity \ Traffic density	VL	L	M	H	VH
VH	VS	S	M	M	M
H	S	M	M	M	M
M	M	M	L	L	L
L	M	M	L	L	L
VL	M	L	L	L	L

Table 2: Main rule table

Pedestrian activity \ Traffic density	VL	L	M	H	VH
VH	S	VS	S	S	S
H	M	S	L	L	S
M	S	L	VL	VL	VL
L	S	L	VL	VL	VL
VL	S	M	VL	VL	VL

Table 3: Secondary rule table

4 Conclusion

This project demonstrates the process of generating fuzzy rules for traffic light timing for vehicles based on numerical data. The inferred rules take into account traffic density and pedestrian activity to make decisions about greenlight timing for vehicles. The data used was generated artificially based on predefined rules and is given to a rule-based fuzzy inference system in order to generate consequent real outputs according to the method proposed in [1]. The consequent real outputs are then converted to linguistic labels using the main and secondary rule tables. The performance of the algorithm is evaluated and the best combination of parameters was determined for optimal fuzzy rule generation.

In conclusion, the algorithm was able to generate satisfying if-then rules based on the main rule table while the secondary table resulted in some inconsistencies. The generated rules can be useful for optimizing traffic light control systems, as they consider the real-world factors that influence traffic flow and pedestrian safety. Further analysis and testing may be necessary in a real-world context to fine-tune and validate the rules.

References

[1] K. Nozaki, H. Ishibuchi, and H. Tanaka, "A simple but powerful heuristic method for generating fuzzy rules from numerical data," *Fuzzy Sets and Systems*, vol. 86, pp. 251–270, 3 1997.