Fuzzy Systems and Neural Networks

Jéssica Consciência e Tiago Leite

October 6, 2024

Part I
Fuzzy System

Firstly we started by deciding between which type of fuzzy system we should implement: Mamdani, Takagi-Sugeno or Tsukamoto. From the project statement we observe that the output *CLPVariation* is not any clear function of the input, rulling out Takagi-Sugeno, also meaning that our output is a **Fuzzy Set**. If we wish for our output to be monotonic then the choice would be Tsukomoto, since we did not want this restriction and decided for starting with a simple approach then later on adding difficulty when needed. (Early on we decided to try to make data-driven decisions with an iterative improving process)

1 Architecture

1.1 First Iterations

In the initial iteration, we selected the variables *ProcessorLoad*, and *MemoryUsage* based on which variables we though were more important. These variables were chosen as inputs, while *CLP* was designated as the output. We opted for triangular membership functions, defining four levels for each input variable: (low, medium, high, critical).

We then defined the range of the membership functions associated with each term of the two linguistic variables, *MemoryUsage* and *ProcessorLoad*. Considering that a device with more than 85% processor load or memory usage is typically unable to perform its basic tasks, it became clear that this threshold would correspond to a specific term, labeled as "critical". The ranges for the other membership function terms were distributed between 0 and 1 based on what we deemed appropriate. We also decided to keep the terms associated with *CLP* straightforward, using only three terms: "decrease", "increase", and "maintain". The values for the membership functions of these terms were distributed between -1 and 1.

The figures below illustrate the membership function graphs for these variables.

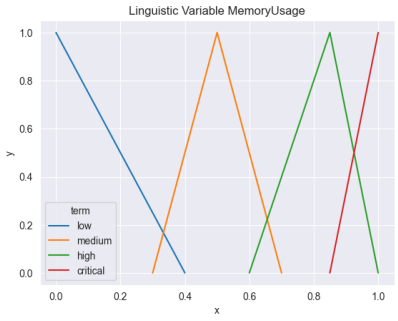


Figure 1: Memory Usage

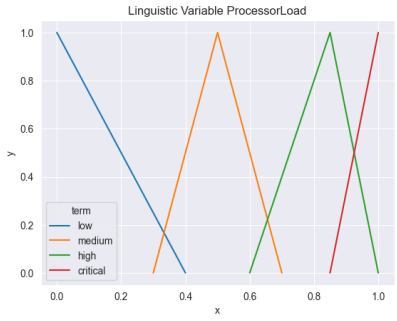


Figure 2: Processor Load

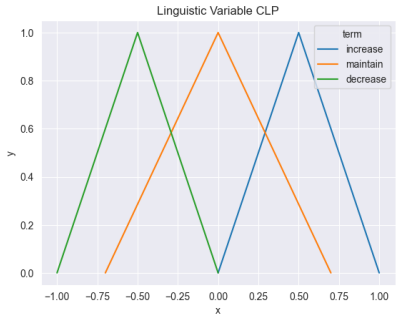


Figure 3: CLP Variation

To design the system's rules, we created a truth table, which can be found below in Table 1 on the following page. The logic behind the table was as follows: when both *MemoryUsage* and *ProcessorLoad* were either "low" or "medium", the *CLP* would increase. When one of them reached "high", the *CLP* remained unchanged (this decision was made to ensure that the node's processing capacity stayed above average). Finally, if any of these variables entered a "critical" state, the *CLP* had to decrease.

CPL		ProcessorLoad			
		low	medium	high	critical
MemoryUsage	low	increase	increase	mantain	decrease
	medium	increase	increase	maintain	decrease
	high	maintain	maintain	maintain	Decrease
	critical	Decrease	Decrease	Decrease	Decrease

Table 1: Truth table

To visualize the system's output, we generated 50 data points for *MemoryUsage* and *ProcessorLoad* ranging between 0 and 1. We then created an interactive 3D plot that showed the evolution of *CLP* based on these two values, this can be seen in Fig. 4. Upon reviewing the graph, we noticed that the variables *ProcessorLoad* and *MemoryUsage* exhibited very similar behavior because intuitively, when designing the system, we had structured the membership functions for each term in the same way for both variables, and the truth table was also symmetric. This indicates that the system should react in the same way to both variables and they could, in fact, be merged into a single variable without losing the system's effectiveness. By combining these two variables, we simplify the model while still accurately representing the system's behavior, as both variables seem to influence the *CLP* in a nearly identical manner.

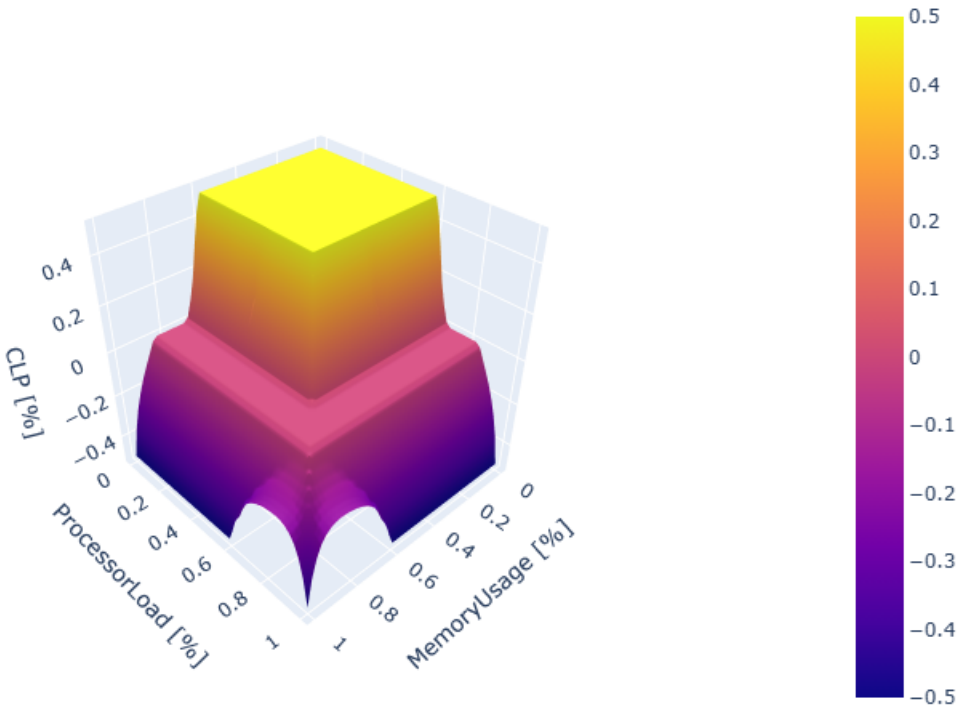


Figure 4: Fuzzy CLP Inference

From the graph we also noticed that the *CLP* only varies between -0.5 and 0.5, which is not the desired range; we aim for it to vary between -1 and 1. This limitation is due to the configurations of the membership functions for the terms "increase" and "decrease" of the *CLP*. Additionally, two constant plateau regions are visible where the *CLP* remains unchanged: when *MemoryUsage* and *ProcessorLoad* are between 0 and 0.6, and when they are between 0.7 and 0.85, which does not make sense in our context. Finally, in the area

of the graph where *CLP* is less than -0.2 and *MemoryUsage*/*ProcessorLoad* is greater than 0.6, there is a "hump" with no *CLP* values, which is undesirable.

Subsequently, we explored the effect of switching the membership functions to a Gaussian distribution because they provide a smoother transition between membership grades. To make a direct comparison with the system we previously developed using triangular membership functions, we retained the same linguistic variables, the same terms for these variables, and the same rules as presented in Table 1.

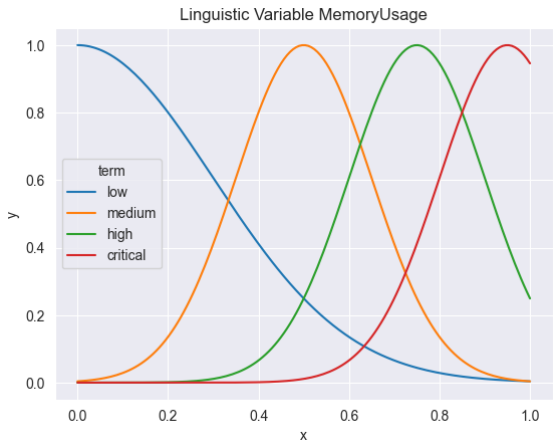


Figure 5: Memory Usage MF

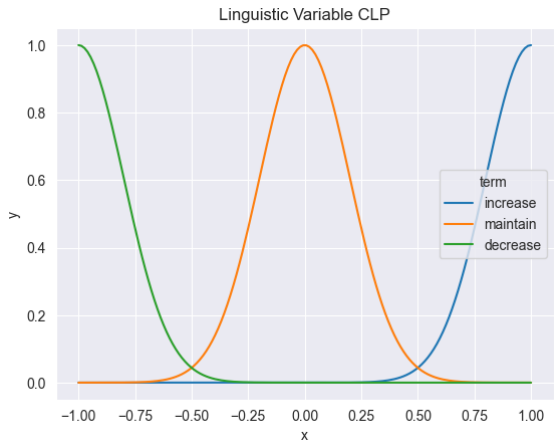


Figure 6: CLP Variation MF

The resulting 3D graph showing the variation of the *CLP* with *MemoryUsage* and *ProcessorLoad* can be seen in Figure 7

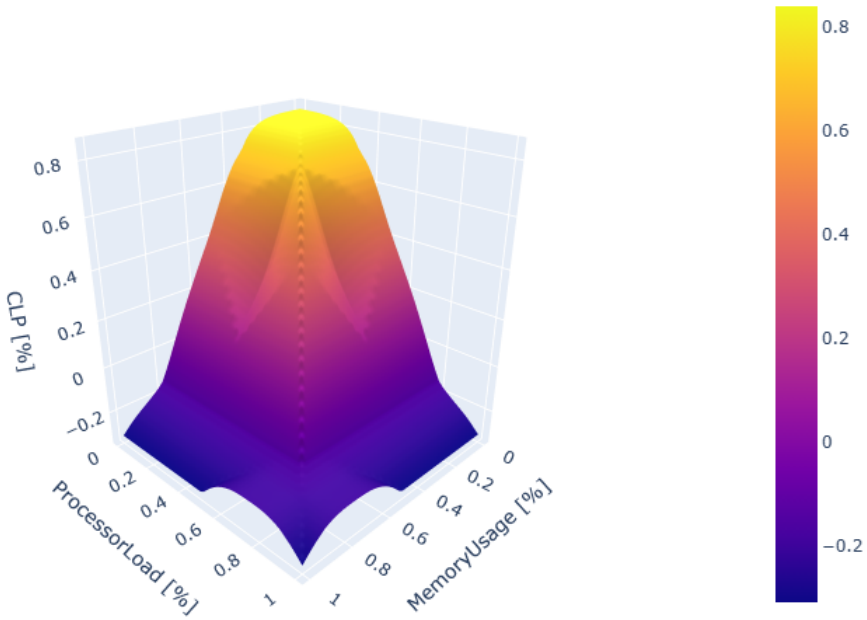


Figure 7: Fuzzy CLP Inference using Guassians as MF

This time, the system achieved higher positive *CLP* values, but it worsened on the negative *CLP* values, only going down to -0.2. To achieve better results, we decided to add more terms to the linguistic variables, merge *MemoryUsage* with *ProcessorLoad* into a single variable, and experiment with different membership functions. By increasing the number of terms, we aim to capture more detailed nuances in the system's behavior, improving its responsiveness to variations in input. Additionally, merging the two load-related variables simplifies the model without losing critical information. The next sections will address these adjustments in detail, explaining the rationale behind these changes and the impact they have on the system's performance.

1.2 Triangle Version Improved

Starting from the system with triangular membership functions presented in Section 1.1, we iteratively built a new system. First, we decided to combine the variables *ProcessorLoad* and *MemoryUsage* into a single variable: *SystemLoad*, which is defined as the maximum value between these two variables. We chose the maximum value because it allows us to capture the most critical resource constraint affecting system performance. By focusing on the highest load, the system can respond to the most demanding condition, ensuring that performance is not compromised under heavy usage.

We decided to assign the same terms we had initially used for the variables *MemoryUsage*/*ProcessorLoad* to the new variable *SystemLoad*, namely: "lo", "medium","high", and "critical." As for the output variable, *CLP*, we decided to add two new terms: "increase_significantly" and "decrease_significantly".

The values a, b, and c that define the triangles of the membership functions were chosen through an iterative process, where we adjusted some of the boundaries (both for the terms related to *SystemLoad* and for the terms of *CLP*) and observed the resulting defuzzified *CLP*. We then evaluated whether the results made sense, given the system's requirements. For example, if the *SystemLoad* input was 0.9, it would not make sense for the resulting *CLP* to be 0.7. Based on this logic, we continued adjusting the limits of the membership functions iteratively.

Later, when we gained access to expert data, this process became easier, as we had a ground truth to reference. This allowed us to test our system and evaluate it using more quantitative metrics. However, this process was quite exhaustive, as improving the error for one set of data sometimes worsened the results for others. Ultimately, we arrived at the following membership functions for each term of each variable.

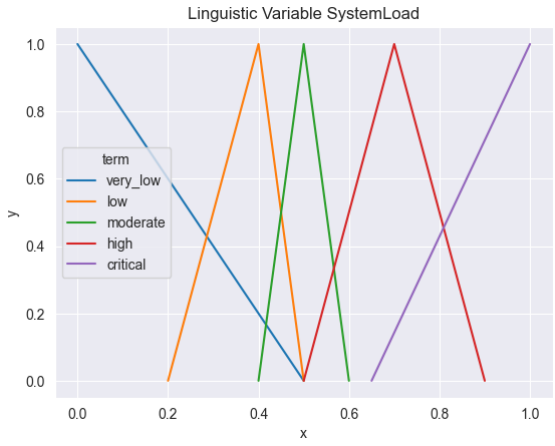


Figure 8: SystemLoad MF's

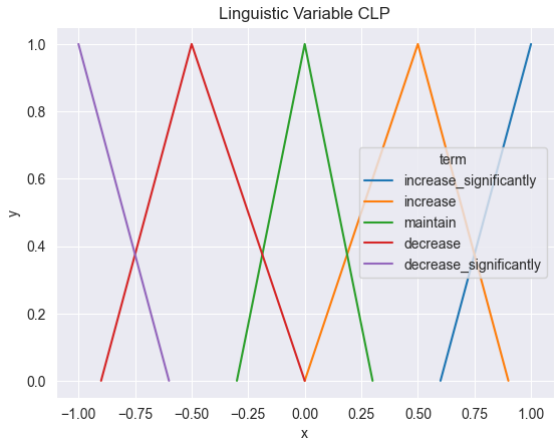


Figure 9: CLP Variation MF's

Initially, the system only considered a single variable, *SystemLoad*. However, with the data provided to us, we realized that this variable alone was not sufficient to capture all the nuances of the real system. There were additional aspects we needed to address using other variables. As a result, we decided to introduce another linguistic variable: *Latency*.

We defined three terms for the linguistic variable *Latency*: "low", "moderate", and "high". Similar to the approach used for *SystemLoad*, the membership functions were established iteratively, using a trial-and-error process based on what we considered reasonable. The membership functions for *Latency* are illustrated in Fig. 10.

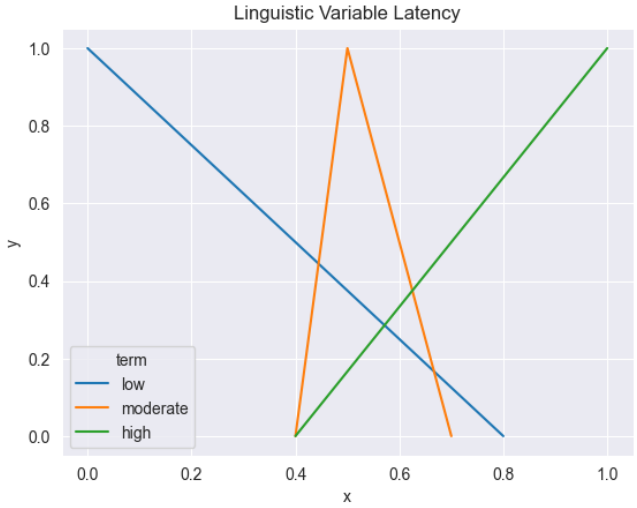


Figure 10: Latency MF

The logic we followed when adding rules was based on the following statement: "high latency means that it's better to process data locally", we determined that in cases where *SystemLoad* is high, the *CLP* could either be maintained at an increasing value or reduced to offload processing to the cloud. However, this decision heavily depends on latency: if latency is low, we can process the data in the cloud (which implies slightly lowering the *CLP*), but if latency is high, it is preferable to continue processing locally on the node since the communication channel is experiencing significant delays.

In the graph shown in Fig. 11, the variation of *CLP* with *SystemLoad* and *Latency* is represented. We can observe that when *SystemLoad* is between 0.6 and 0.75, the *CLP* varies with *Latency*: with low latency, the *CLP* decreases, whereas with high latency, the *CLP* increases.

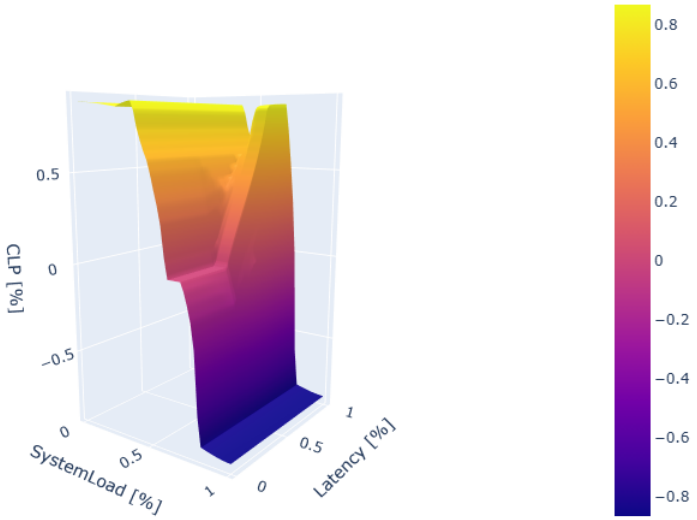


Figure 11: 3D plot of CLP variation with Latency and SystemLoad

2 Generalized Bell

We decided to experiment with a more generic Membership function, so we extended simpful's Base Membership Function class and created Bell_MF [in fuzzy/models/bell_mf.py]. The first results are shown in the figure bellow.

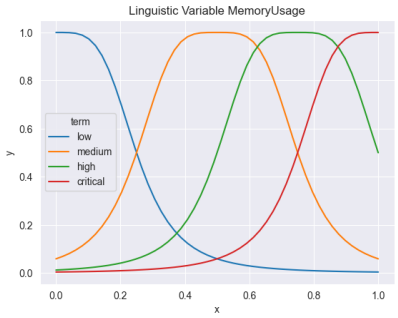


Figure 12: Memory Usage

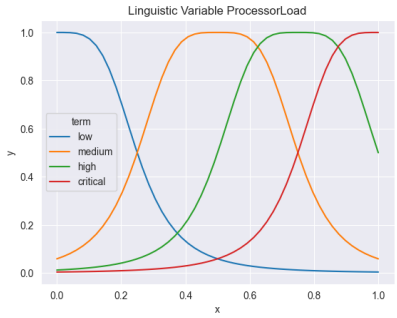


Figure 13: Processor Load

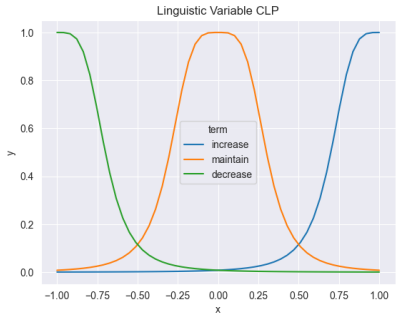


Figure 14: CLP Variation

After some experimentation, we concluded (as foreseen theoretically) that the parameters a, b, and c are responsible for the slope*, width, and center of the function, respectively.

3 Architecture

This should contain choice of architecture and why.

4 Membership Functions

all the membership functions and linguistic terms

5 Rules

rules

CLP Variation		Latency			
		low	moderate	high	very high
System Load	low	IS	IS	I	I
	moderate	I	I	I	I
	high	M	M	D	D
	critical	DS	DS	DS	DS

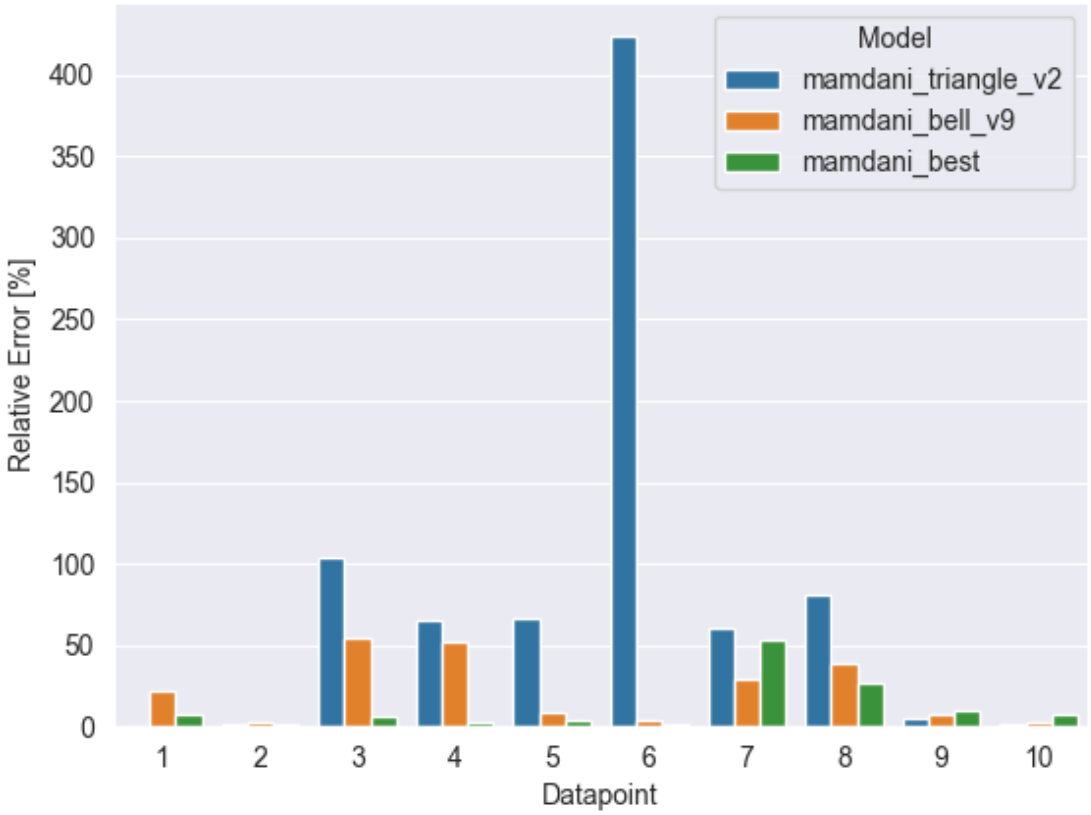
6 Evaluate Models

Now that we had developed several models and established rules, a reliable and straightforward testing mechanism became necessary. To achieve this, we implemented the script `eval_models.py` (`fuzzy/eval_models.py`), which utilizes a dictionary to compare all model predictions on 10 expert sample data points provided in `CINTE24-25_Proj1_SampleData.csv`. Initially, the relative error metric, defined as:

*The slope of the function is influenced by both parameters a and b, where $slope = \frac{a}{2b}$

$$\text{Relative Error} = \frac{|y_{\text{true}} - y_{\text{pred}}|}{y_{\text{true}}}$$

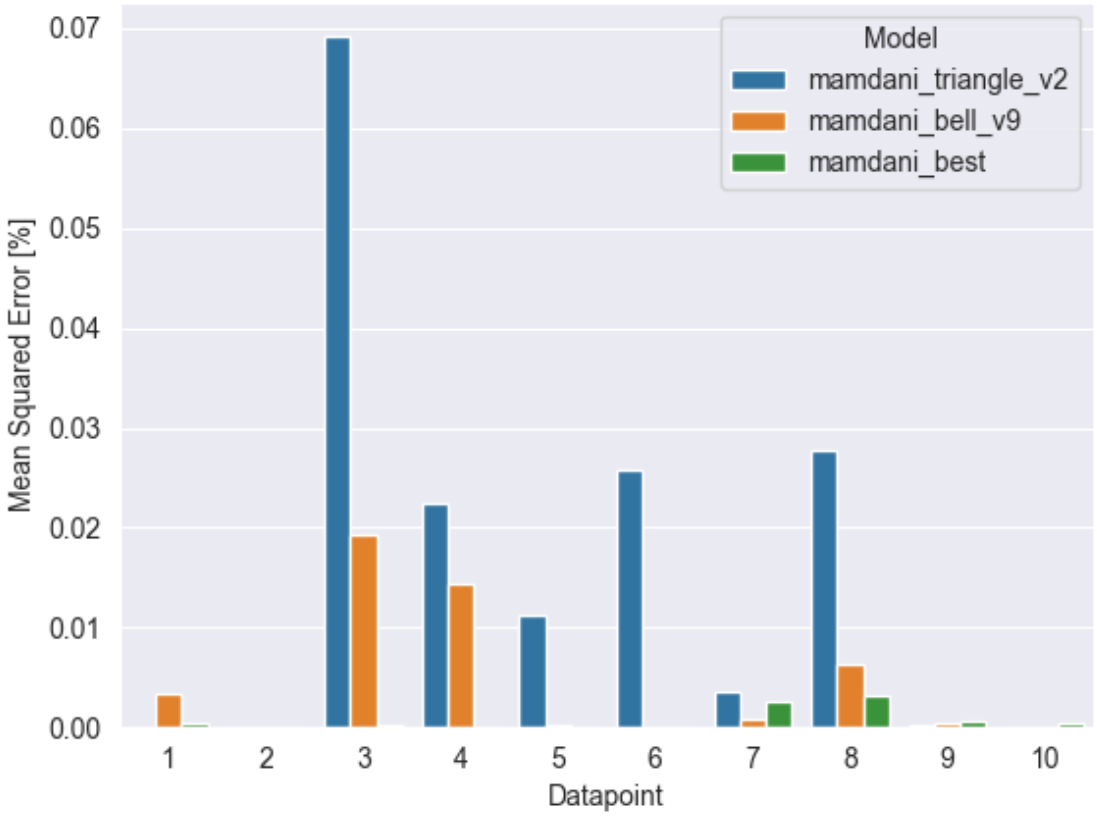
was used for evaluation. However, this metric exhibited instability when $y_{\text{true}} \to 0$, as was the case for data point 6 in the sample. This issue is illustrated in the figure below, where the models `mamdani_triangle_v2`, `mamdani_bell_v9`, and `mamdani_best` are compared.



To address this instability, we switched to the Mean Squared Error (MSE) metric, defined as:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_{\text{true}} - y_{\text{pred}})^2$$

The results of this evaluation are presented below.



This evaluation process was instrumental in improving our models and identifying potential problem areas. For instance, the model consistently underestimated the CLP for data points 3 and 4, this is shown in the table below.

Table 2: Sample data points 3 and 4

	MemoryUsage	ProcessorLoad	Latency	mamdani_bell_v2	CLPVariation
3	0.68	0.60	0.80	0.30	0.73
4	0.50	0.50	0.50	0.66	0.50

These observations suggested, in our view, that the model should account for high latency as a factor contributing to a lower *CLP*. This iterative process led to the development of models like `mamdani_triangle_v2` and `mamdani_bell_v9`.

Many more iterations were tested but not preserved. Some of the deprecated models can still be found in the `fuzzy/models/deprecated` folder, along with their outputs in `fuzzy/output/deprecated`.

The best model, `mamdani_best`, emerged as the result of extensive hyperparameter tuning, which is discussed in detail in the Hyperparameter Tuning section.

7 Hyperparameter Tuning

Since we had manually iterated until a good model was reached, scoring 0.0447 in total MSE on the 10 sample data points, we considered the linguistic variables and rule base as fixed. However, we found ourselves

tirelessly fine-tuning the membership functions, such as the center, slope, and width in the case of the bell membership function. To automate this process, we took the following steps:

- Create a `FuzzySystem` from a set of hyperparameters [`fuzzy/models/mamdani_hparams.py`].
- Define an objective function to minimize/maximize. In our case, we used the total MSE of the 10 sample data points as the objective value to be minimized.
- Sample several hyperparameter trials and find the best. This was done with the help of the Python framework `optuna`, which uses Bayesian Optimization to find optimal hyperparameters.

A simpler approach would have been to use `RandomSearch` or `GridSearch` (possibly using `scikit-learn`). However, we chose to leverage `optuna`'s search algorithm, Tree Parzen Estimation (TPE). In a nutshell, TPE builds an internal model that makes "educated" guesses about which hyperparameters to test and continuously updates that model. The search is then performed in a tree-like manner, and `optuna` can handle both continuous and categorical data.

After 3 hours of hyperparameter tuning, the `mamdani_bell_v9` model, comprising a total of 6228 trials, achieved a final total MSE of 0.00749. This corresponds to a six-fold improvement over the manually obtained MSE of 0.04473. The best hyperparameters from this tuning process were saved as `hparams_007.json`.

8 Conclusion

Part II
Neural Networks

9 Architecture

To build the neural network, we decided to use a simple architecture: 3 layers, 12 input nodes, 32 nodes in the hidden layer, and 1 output node. Since the output should be in the range $[-1, 1]$, we chose the *Tanh* activation function, which produces values within this range. For the optimizer, we used Adam with a learning rate of 1×10^{-3}, leaving the rest of the parameters at their default values: $\epsilon = 1 \times 10^{-8}$, $\beta = (0.9, 0.999)$, and `weight_decay` $= 0$. For the loss function, we chose MSE, allowing us to directly compare the results with those of the Fuzzy System.

We implemented the neural network using the PyTorch framework alongside `pytorch_lightning`. This made it easy to integrate TensorBoard for logging and visualization, apply Early Stopping to prevent unnecessary computation, and enable Model Checkpointing to save the best-performing model. This code can be found in the `nn/models/simple_lightning.py` file.

10 Training Data

To create the training data for the neural network, we began by generating synthetic data for the 12 input features. This was done by sampling 100.000 random uniformly distributed values for each feature. Next, the Fuzzy System was used to predict the CLPVariation, utilizing the best-performing Fuzzy System located in `fuzzy/models/mamdani_best.py`. The results were then stored in a CSV file inside the `gen_input` folder. This code can be found in the `fuzzy/generate_data.py` file. The decision to use random uniform data, as opposed to, for example, a Gaussian distribution, was made to ensure a **balanced** dataset for training the neural network. The choice of the number of training data samples was also carefully considered. We followed the rule of thumb that "a model will often need ten times more data than it has degrees of freedom", where a degree of freedom refers to model parameters or input features. Our model has 449 trainable parameters and 12 input features, which means our training data should consist of more than 4610 samples. We decided to generate a number of samples equal to an order of magnitude above 4610 rounded up, which results in 100,000 samples.

11 Classification

To use the neural network as a classifier, we first defined the ground truth intervals as $[-1, 0.3[$ for *Decrease*, $[0.3, 0.5[$ for *Maintain*, and $[0.5, 1]$ for *Increase*, based on the CLP linguistic variable intervals as shown in the figure below.

Using these intervals, we labeled the training, validation, and test data for the neural network. Next, we labeled the data again using the same intervals but with the results predicted by the neural network. The confusion matrix heatmap showing these results is Fig. 15.

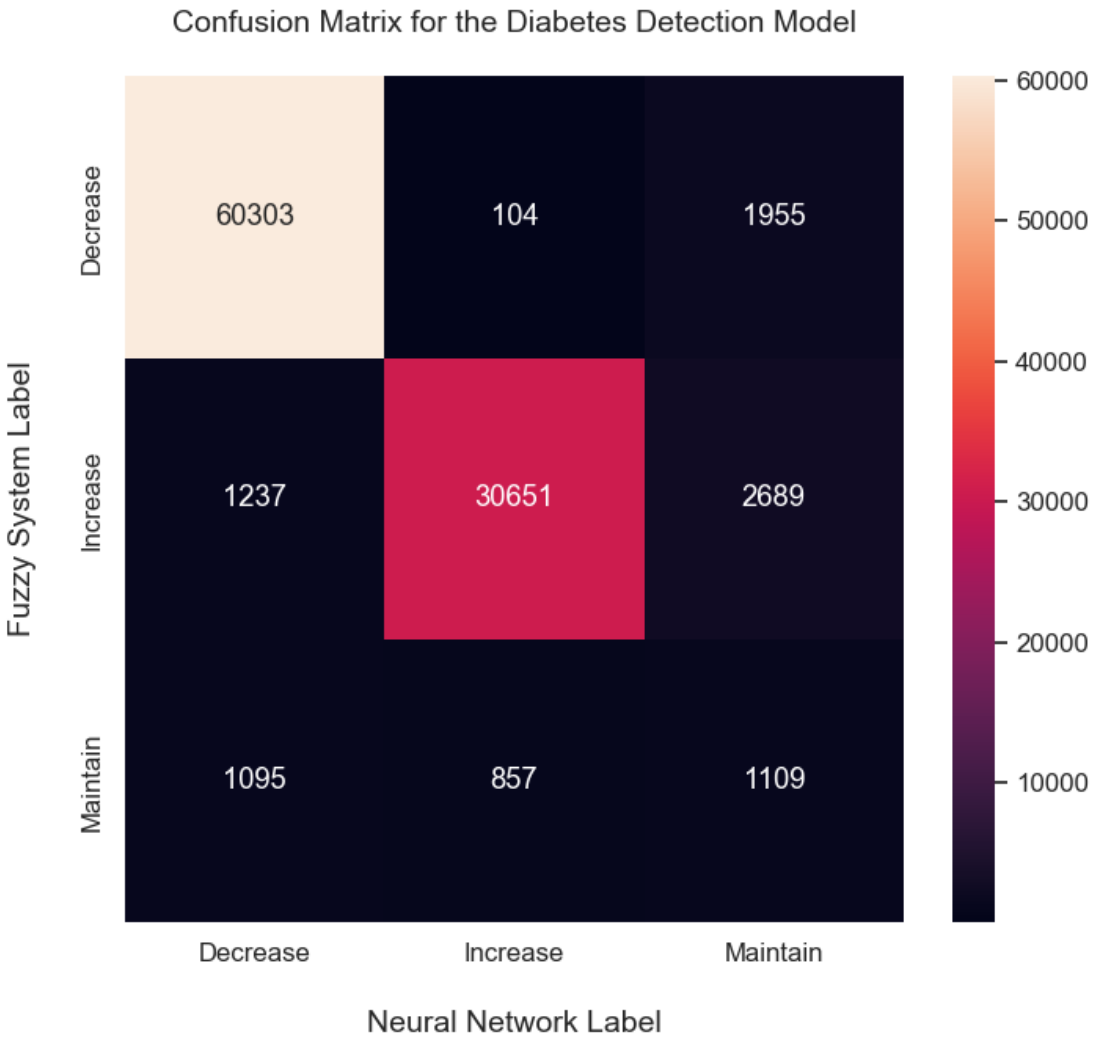


Figure 15: Confusion Matrix Heatmap

To gain a better understanding of the neural networks performance, we calculated the Precision, Recall, and F1-score, which are shown in the table below.

Table 3:

precision	recall	f1-score	label
0.96	0.97	0.96	Decrease
0.97	0.89	0.93	Maintain
0.19	0.36	0.25	Increase