Explainable AI with Fuzzy Inference System: An attempt to derive a TSK model from data

Group 9

Husø, Tancred Heyerdahl
s348842@oslomet.no

Jørgensen, Kristian
s344189@oslomet.no

Olsen, Bernt Moritz Schmid
s341528@oslomet.no

***Abstract*— The lack of transparency in deep learning models used in smart farming makes them challenging to deploy in the industry. To better understand the reasoning behind the decisions these models make, the importance of Explainable AI has become more and more apparent. Ryo [1] used post-hoc methods to interpret results from “black-box” models. We build on this work by applying a Takagi-Sugeno-Kang (TSK) Adaptive Neuro-Fuzzy Inference System (ANFIS) to the same problem to investigate the interpretability of fuzzy inference systems. For that we created a TSK implementation that derives the rule base from data using k-means clustering and fine tunes parameters using the Bees Algorithm. The result was compared with the results from the pyFume library and the anfis library in matlab. The performance was observed to be similar, but the pyFume library displays a better ability to avoid overfitting.**

Keywords—TSK, Fuzzy Controller, XAI, BA, Python, MATLAB

I. Introduction

This project aims to investigate how Fuzzy systems can be applied to a regression problem. We try to find out how well a Takagi-Sugeno-Kang (TSK) Fuzzy Controller can solve a regression problem compared to other models. An important property of the Fuzzy Controller is its interpretability [2], [3]. Thus, we also investigate the transparency of the models created.

The field of Machine Learning has seen significant developments in the past years. Models have been successfully applied to problems in important fields such as Health care, economy, or agriculture. One of the most successful models is called Deep Learning (DL) models. Those are models that are based on the concept of neural networks and are characterised by an architecture consisting of a large amount of hidden layer and nodes. This can be called a deep neural network. One of the most important drawbacks of DL models and neural networks, is that the complexity increases with the number of hidden layers added. DL is considered a Black Box (BB) model because it is difficult to interpret how the model derives the dependent variable from the independent input variables [2].

Agriculture is one of the most important industries worldwide. It is providing society with a basic human need in food. There are many examples of machine learning usage in agriculture. The learning methods can be used to help gain valuable insights that can help farmers optimize their production [4]. Use cases can be found in the entire process from sowing to selling. In the stage called pre-harvesting, models can be used to find the ideal crop to be sown based on variables that describe the environment and the market [4].

During the growing stage, crops and plants can develop diseases. It is therefore important for the farmer to inspect the plants regularly. Manual inspection is prone to errors, leading to false positives of false negatives. A false negative can mean that the farmer applies more pesticide than necessary, causing exposing plants and human health to unnecessary risk of damage. Models have thus been developed to detect deceases and determine the correct amount of pesticide to be applied [4].

When a farmer takes a decision based on the knowledge from machine learning, there are several reasons for why a transparent model is more convenient. For one the model might have a weakness that has not been detected during model validation. If the farmer can inspect how the model derives the result, what the line of reasoning is, the chance of avoiding error is increased. It is also giving the farmers confidence in performing the action. Secondly, inspecting a transparent model can yield valuable new insights that can be investigated further.

Models such as neural networks, Support Vector Machines (SVM) or Random Forest have been applied to problems in agriculture according to [1], [4]. Those models are to some extent considered to be Black Box models, especially for large and complex models. It is difficult for the user to find out why a model produces a certain result. Because of this, current ML research is concerned with finding methods that can provide the means to explain those complex models or produce interpretable models. Those models are typically called White Box (WB) models [5].

Achieving both interpretability and accuracy for the same model is difficult. Those two objectives can be considered as conflicting objectives [1], [5]. Many White Box models like Logistic regression, decision tree or k-nearest neighbour do not provide the same possibility that neural networks have to fit complex models. It is thus important to consider the importance of interpretability and accuracy, when deciding on a model for a learning problem. There can for example be cases where the accurate results are not usable if it cannot be explained how this result came to be [5]. In that case a more interpretable model can be better suited than more accurate BB models.

There are many approaches to XAI. The paper by Ismail et. al [5] categorizes the interpretable machine learning models into Posthoc and Antehoc. Posthoc models are used to analyse a BB model after it has been trained. An Antehoc model is typically an interpretable WB model. Other papers use the term model-based approach [1]. Ryo [1] also categorizes the XAI models into model generality and explanation scale. A model can either be model-specific, meaning that a method can only be used for a certain type of model, or model-agnostic which means it can be applied to several model types. Explainability scale is either global or local. An XAI model is global if it uses all available input data to find out something about how the model is deriving its

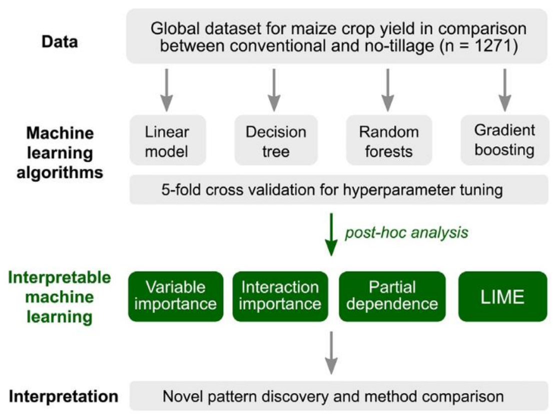

Data
Global dataset for maize crop yield in comparison between conventional and no-tillage (n = 1271)
Machine learning algorithms
Linear model
Decision tree
Random forests
Gradient boosting
5-fold cross validation for hyperparameter tuning
post-hoc analysis
Interpretable machine learning
Variable importance
Interaction importance
Partial dependence
LIME
Interpretation
Novel pattern discovery and method comparison

Fig 1. The workflow described by Ryo [4]

result. It could for example be what variable is most impactful on the dependent variable. A local model is a model where the modelling process is analysed for each individual input combination. For example, if a model is given an image of a plant and detects a decease, the aim of local models would be to find out on what bases that happened.

In this project, we worked specially on the paper *Explainable artificial intelligence and interpretable machine learning for agricultural data analysis* by Masahiro Ryo. The paper describes how four Posthoc models were applied to the four machine learning models: Linear Regression, Decision Tree, Random Forest and Gradient Boosting. The models were applied to a regression problem derived from the global dataset of crop production with or without conventional tillage. The dataset was presented by Su et al. [6]. The dataset can be downloaded here: https://figshare.com/articles/dataset/A_global_dataset_for_crop_production_under_conventional_tillage_and_conservation_agriculture/12155553?file=25625990

The goal of the work by Ryo [1] was to find out under what circumstances maize crop has a positive yield change when going from conventional tillage to no-tillage. This was done by training the models mentioned above, to predict the relative yield change variable from 17 input variables. After the models had been trained, four Posthoc methods were applied to explain the models: Variable Importance, Interaction importance, Partial Dependence and Local Interpretable Model-agnostic Explanation (LIME). All of them are model-agnostic and Posthoc. But the three first models are global methods that represent three different ways of describing the impact of all the variables on the dependent variable. The LIME model is local and used to explain why certain results are produced [1].

The paper also evaluates how well the models can fit the problem by using the R^2-metric and the prediction error by using the Root Mean Squared Error (RMSE) metric. The results showed that Random Forest was able to fit the problem best and had the lowest prediction error (see *Table 1*).

When it comes to explainability, the Variable Importance suggested that "conventional tillage yield" is most important for Gradient Boosting and Random Forest. For the linear model and decision tree "Soil Texture" was regarded most important. Those features were further investigated with Interaction Importance with other features scoring high on importance. The result is that "Yield_CT" and "Tmax" has the most significant interactino strength for GB and RF. The two variables were further investigated, and the author found that no-tillage can produce more yield in cases where the yield under conventional tillage is lower than 5000 [kg/ha] and the max temperature higher than 32 degrees [1].

Model	Model evaluation metric	
	R^2	$RMSE$
Linear Regression	0.11	0.236
Decision Tree	0.18	0.225
Random Forest	0.42	0.199
Gradient Boosting	0.33	0.200

Table 1. Performance metrics for the machine learning models presented in [4].

The scripting done by Ryo [1], was done with R. The code and the dataset is made available on github: https://github.com/masahiroryo/2022_IML_ Agriculture.git

Our goal for this project is to create a TSK fuzzy controller that solves the same regression problem described in [1]. This involves using methods to learn a TSK model from a dataset. For this project we investigated using Adaptive Neuro-Fuzzy Inference Systems (ANFIS) or Evolutionary Fuzzy control. After doing some research around TSK and ANFIS, we ended up implementing an evolutionary Fuzzy control approach in python. This model was then compared to the results from using existing Python and Matlab libraries.

The models were evaluated using the same performance metrics that were used in [1]. Then the goal was to see if the same conclusions about relative yield change can be drawn from the model by analysing the rule base.

In the second section (Method) we describe how the data was handled (data cleaning and feature engineering) models were developed and defined. In the third section (Results), we present the result we found and in the fourth section (Discussion) we discuss the results and how the results of this project could be improved with further work.

II. Method

In this section the work that was done during the project is presented. We started by performing data cleaning on the dataset and feature engineering. Then we decided on which algorithms to use for solving the regression problem. We used three different approaches: Implementing an evolutionary approach to tuning and deriving TSK from data, using pyFume to derive TSK from data and use ANFIS with grid partition in Matlab. The next subsections describe how these algorithms/libraries were worked with. Lastly, we evaluated the performance of the models. Fig 1 shows the workflow of the project.

Python is the most used programming language by data practitioners. It has a large community and there are many powerful libraries like Pandas, NumPy, matplotlib, Keras, Tensorflow or PyTorch available to perform data analysis and machine learning tasks [3]. Therefore, it was decided to

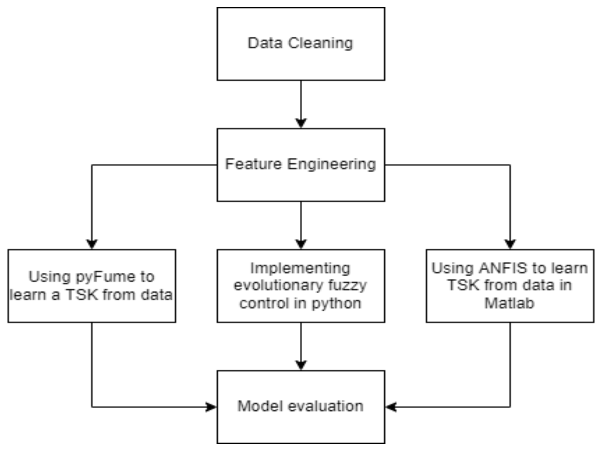

Data Cleaning
Feature Engineering
Using pyFume to learn a TSK from data
Implementing evolutionary fuzzy control in python
Using ANFIS to learn TSK from data in Matlab
Model evaluation

Fig 3. The workflow applied in this project.

use python for this project. It is also the case that there is a lack of libraries for FIS tuning in python. In Matlab tuning FIS is supported by libraries and even a GUI. This is why we also tried to apply the ANFIS library in Matlab to the problem. To collaborate as a team, it was decided to use GitHub and the version control system Git.

A. *Data cleaning and feature engineering*

The data cleaning began with extracting 18 columns that were used in the original analysis from the original dataset of 53 columns. From there filter it by crop-type = Maize reducing the 4403 rows to 1434. Afterwards Boolean data was encoded to binary i.e., yes = 1, no = 0. Then basic cleaning was done I.e., checking for missing values and removing them. The process of encoding Boolean values to binary is necessary for random forests to work since most feature selection models require numerical values. Leading the final dataset to be 14 columns by 1434 rows with no missing data.

One issue repeatedly encountered was an error message stating that a certain column was not found in the dataset while being present. The fix became to drop the columns in question. The feature selection was done via random forests, although PCA was first attempted, but there was no need for reduction given the already limited size of the dataset. Random forests (RF) was used due to the random selection of features during feature selection meaning that it selects a few features at random with equal chance for each forest, thus avoiding bias towards stronger features as explained in [7]. RF produced the following feature importance as shown in *Fig 4*.

B. *Picking models*

While finding the right approach of using Fuzzy Inference to solve the regression problem, it is important to take note of the input and output variables. A FIS produces only one output value. Since we are trying to solve a regression problem with only one dependent variable, that requirement is met.

Secondly, it is important to understand the cost of adding input features to a FIS. With many input variables, the number of rules and fuzzy sets increase (more model

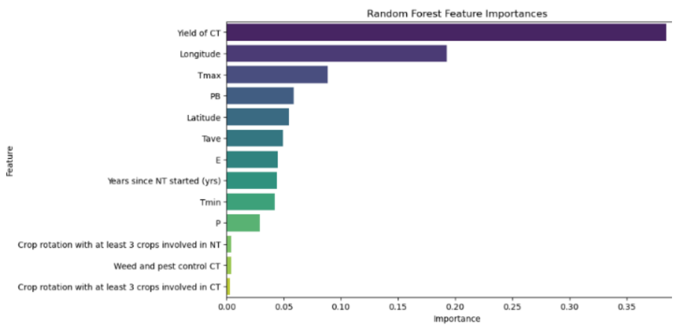


Fig 4. Feature importance, higher values mean higher importance.

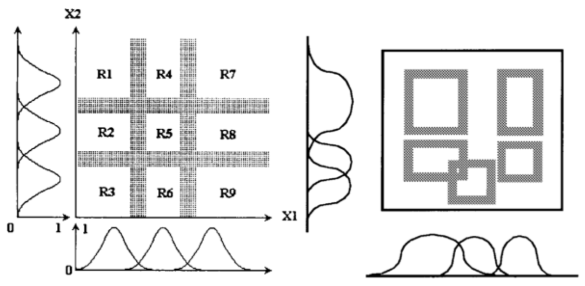


Fig 2. An example of how two input variables are partitioned. The left figure demonstrates Grid Partition. The right figure demonstrates the clustering based approach. Figures taken from [8] and [7] respectively.

parameters), causing the model to be less transparent. Larger models also require considerable amounts of computational power to tune parameters [5].

One of the approaches used to derive the fuzzy sets and consequents from the data is called Grid Partition. With this method the derived rule base covers the entire output space. The result is that each possible combination of fuzzy sets is represented in the rule base. For example, if we have two input variables sorted into 3 different fuzzy sets, 9 different rules are defined. Each rule has an antecedent consisting of two sets from the two input variables. Generally, the number of rules is given by (1) where m is the number of fuzzy sets for each input and k is the number of inputs.

$$m^k = n_{rules}$$

(1)

In this project the number of input variables is 17. For a model with 3 fuzzy sets we would end up with $3^{17} =$ 129140163 rules. If we now consider that there are 17 + 1 different parameters for each rule consequence in TSK, we get a very large model.

The number of rules can be reduced by using clustering as an approach to derive the rules. For example, the k-means algorithm can be applied to derive the most significant clusters in the output space. Each of those clusters are then represented by one rule. This will cover less of the output space and probably lead to less accurate models compared to Grid Partition. Fig 2 illustrates the difference between grid partitioning and clustering.

Commonly ANFIS models are initialized by grid partition. One example is the ANFIS module in Matlab. We therefore started looking into alternative methods to derive the TSK model from data. A python library created by [3]

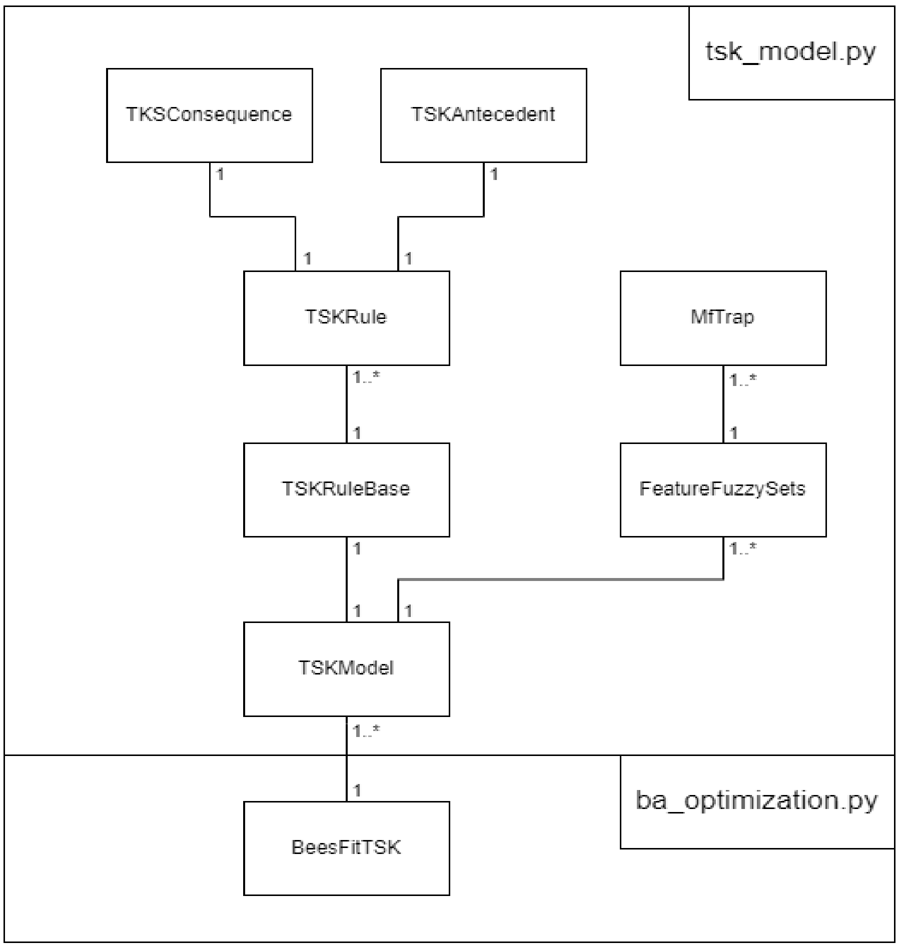


Fig 5. The Domain diagram for the implemented code. Note the file names indicating where the objects are defined.

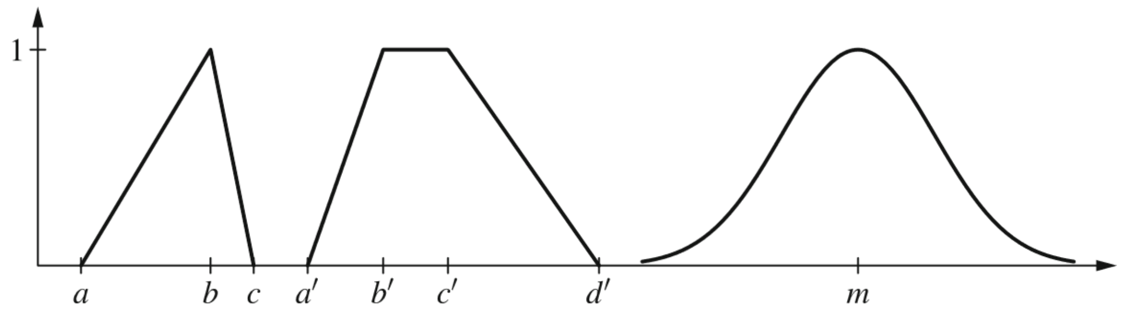


Fig 6. The graphs of the Triangular, Trapezoidal and Bell membership functions. Figure is taken from [8, p.337]

uses C-Means Clustering to derive a TSK rule base. The library is called pyFume.

Since this project is an assignment to a university course, we also looked for a project with great learning outcomes. It was therefore decided to develop a program for defining a TSK model with k-means clustering and the Bees Algorithm (BA). The next subsections describe the model that was developed.

There exists several different Fuzzy Controllers. Two very common types are the Mamdani Type and Takagi-Sugeno-Kang type. An important feature of ANFIS models is that it is easier to represent a TSK type rule base in a neural network, allowing parameter tuning with Gradient decent [8, p. 214]. The disadvantage with TSK type is that the consequents in the rules can become less interpretable with a large input size. We decided to use TSK to keep the option to implement gradient decent because we lack the experience working with this models and don`t fully know the difficulty of implementing the evolutionary learning process.

C. Developing learning framework for TSK

The development of the TSK learning framework can be divided into three parts: Creating the TSK model, using k-means to derive the fuzzy sets and rules and using implementing the BA algorithm to tune the parameters.

1) Developing the TSK model object

The first step in developing a custom learning framework for TSK, was to implement a TSK model in python. It was decided to use object-oriented programing to make the code clear and easy to expand. Fig 1 shows the objects that were created for the TSK model. They are all defined in the *tsk_model.py* file.

A Fuzzy Controller typically consists of a rule base containing multiple IF, THEN rules. From those rules a single output value can be found by feeding the rules with input data and performing fuzzy logic. The rules can be divided into two parts: Antecedent and Consequence.

The Antecedents of a TSK model consists of a list of fuzzy sets. One for each input to the model. For each model input given to the rule, the membership degree to the corresponding Fuzzy set is calculated. Next the membership degrees are aggregated by taking using a t-norm operator. This corresponds to performing the and operator in classical logic where the membership degrees are either 1 or 0. It is also important to note how each fuzzy set can represent a linguistic expression. The following sentence is an example of such an expression: IF t_min is cold AND t_max is cold. Here both t_min and t_max are inputs and the two fuzzy sets which both represent the word cold are the corresponding fuzzy sets. Note that despite representing the same expression, the sets are most likely different as they cover different features.

The first step in implementing the antecedent was to define the Fuzzy set that should be used. A fuzzy set are sets where a value has a certain degree of membership to that set. The degree of membership can be in the interval [0, 1]. To define such a set, a membership function is required that maps input values to the degree of membership. A classical set, where a value is only either fully in the set, is defined by a membership function that is 0 for all values not in the set and 1 for all values in the set. A classical set is a special case of a fuzzy set.
There exist different options for defining a fuzzy set. It could for example be defined by a triangular function, Trapezoidal function or a bell curve. The difference between the functions is the number of parameters. A triangular function is defined by three parameters, bell curve has two parameters (mean and standard deviation) and the trapezium has four parameters. In our case we used a trapezium. This option requires more parameters to tune, but at the same time the number of possible sets that can be defined is higher. Equation *(2)* shows the MF used in this project. Note that it is important that the value of the parameters must be ordered such that *a' < b' ≤ c' < d'*.

$$\Pi_{a',b',c',d'} : \mathbb{R} \to [0,1], \qquad x \mapsto \begin{cases} \frac{x-a'}{b'-a'} & \text{if } a' \le x \le b' \\ 1 & \text{if } b' \le x \le c' \\ \frac{d'-x}{d'-c'} & \text{if } c' \le x \le d' \\ 0 & \text{otherwise,} \end{cases}$$

(2) formula from *[9, p. 337]*

The membership function was defined in the object called *MFTrap*. The object is initialized with the parameter values a, b, c, and d, as well as the name/expression of the set (see __*init*__ method). The method called *get_membership_degree* contains the code for calculating the function in *(2)*.

Each *MFTrap* objects defined for a model is added to a *FeatureFuzzySets* object. This object contains all fuzzy sets for a certain input variable for the model. The *TSK_Model* object that defines the model contains a list of *FeatureFuzzySets* objects.

To calculate the firing strength of the antecedents of rules, we defined an object called TSKAntecedent. This object defines the antecedent of one rule in the rule base. It contains a list of indexes for the fuzzy set in the FeatureFuzzySets list. Furthermore, the method *calculate_firing_strenght* is used to calculate the firing strength of the antecedent. It does so by finding the membership values for each input values x. Next, the product t-norm operator is applied to the object. It is possible to use different t-norm operators for this operation but, the product operator is a common choice for TSK models defined in ANFIS models [8, p. 216], [9, p. 418].

The consequent part of the rules is defined in the *TKSConsequence* object. A consequence in the TSK controller consists of a function that maps the input variables to a value y. The functions used are often polynomials with a specific order. For example, a zero-order TSK, is a controller where the consequent functions consist of only constants. For first-order TSK, the function is a linear combination of the input values given to the rules. Each variable has a coefficient. Additionally, a constant is added to the result. In the code, it was decided to go for a first-order TSK as it this gives allow approximation of more complex functions. The downside is that the transparency probably will be reduced. The coefficients and the constant are stored in lists and a variable (*self.params_list* and *self.const*). The consequent object also has a *calculate_consequence* method used to calculate the function value of the consequent.

$$f_i = p_1 \cdot x_1 + p_2 \cdot x_2 + \cdots + p_i \cdot x_i + r$$

(3) An example first-order consequent of a TSK controller.

To group together the antecedent and consequent object in one object, the *TSKRule* object was created. It contains the consequent and antecedent objects as well as a method to print out the rule in the terminal. The *TSKRule* objects are grouped together in a *TSKRuleBase* object containing a list: *self.rules.* This object has the responsibility to add and remove rules from the rule base and create a printout for the entire rule base.

Lastly the full representation of the model is defined with the *TSKModel* class. It contains important methods like *calculate_output* which produces the defuzzyfied output value from the controller, *create_rulebase_kmeans* and *create_rulebase_kmeans_advanced* which derives the fuzzy sets and rules from data. The methods *set_feature_fuzzy_sets* and *set_rulebase* are used to add or updated the fuzzy sets and rule base. They can also be used to create a model from expert knowledge.
The defuzzification in TSK is simpler than with the Mamdani Controller. In case of the TSK, the firing strength w from the antecedent and the function value from the consequent. The products, from each rule is then added up and normalized by dividing by the total firing strength from all antecedents (see equation (4)). Note that R is the rule

index, $\mu_{R,a_1,\ldots,a_n}$ denotes the firing strength for the current rule R with input values a_1 to a_n.

$$y = \frac{\sum_R \mu_{R,a_1,\ldots,a_n} \cdot f_R(x_1, \ldots, x_n)}{\sum_R \mu_{R,a_1,\ldots,a_n}}$$

(4) Formula is taken from *[9, p. 405]*

In order to evaluate the TSK controller, we added a method to calculate the RSME and the R^2 of the model given a DataFrame with test data. The RMSE is a performance measure that describes the average error of the predictor. This is a common measure used in regression problems. The RMSE is calculated with the formula given in (5) [10, p. 39].

$$RMSE = \sqrt{\frac{\sum_{i=k}^{N}(y_i - \hat{y}_i)^2}{N}}$$

(5)

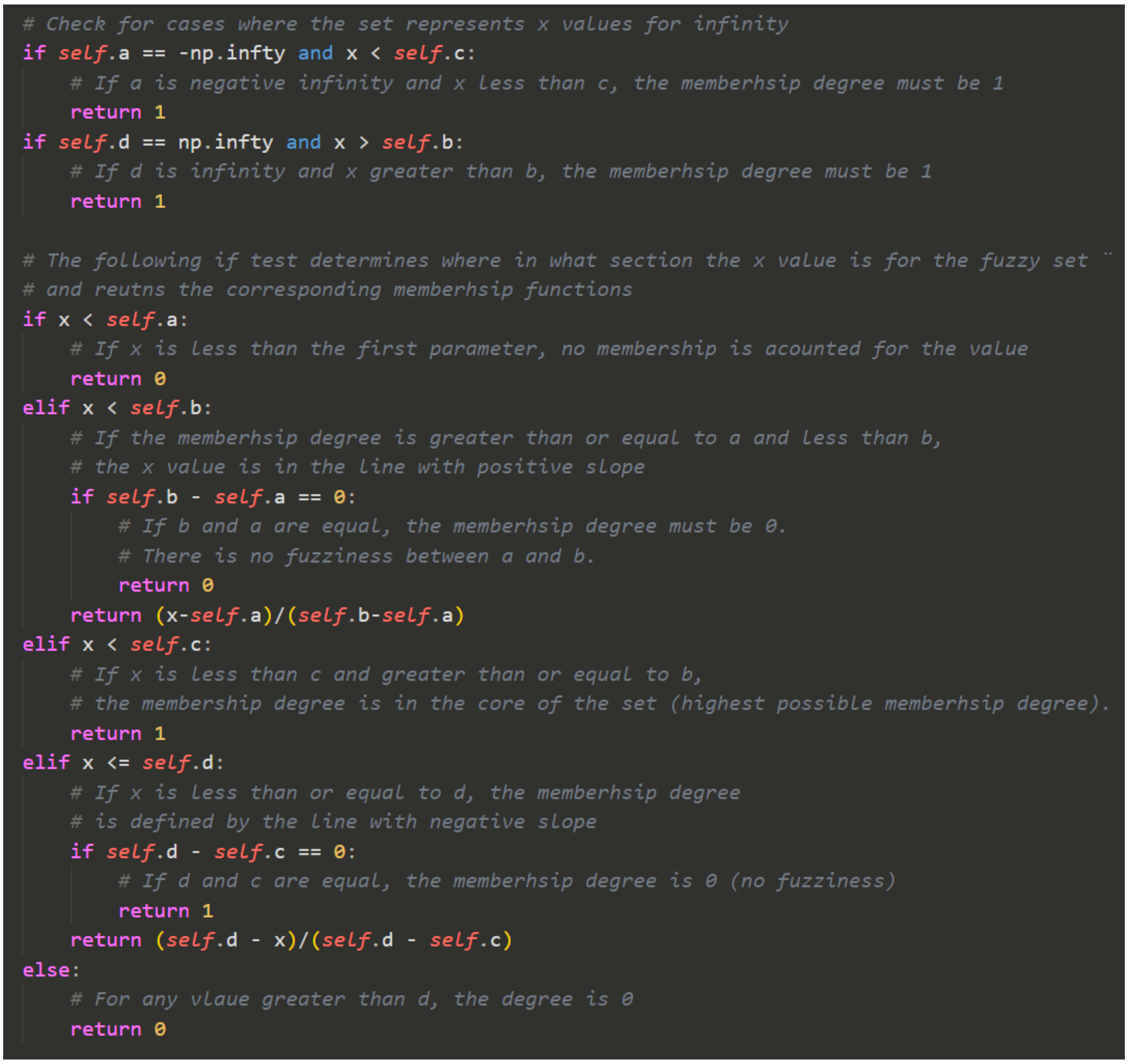
The R^2 measure describes how well the regression model fits the problem. Equation (6) is the equation used for the calculation [11]. The calculations are done by the methods *calculate_r_squared* and *calculate_rmse*.

$$R^2 = 1 - \frac{\sum_{i=k}^{N}(y_i - \hat{y}_i)^2}{\sum_{i=k}^{N}(y_i - \bar{y}_i)^2}$$

(6)

2) *Using k-means clustering to derive TSK rules*

After implementing the TSK model and testing it, the methods *create_rulebase_kmeans* and *create_rulebase_kmeans_advanced* were created. Both methods use K-Means clustering and least square fitting to determine the rule base of the TSK from a dataset. The dataset is provided to the methods as a Pandas DataFrame. The first column of that DataFrame is the target value (y) while the rest are independent variables (x_i).



```python
# Check for cases where the set represents x values for infinity
if self.a == -np.infty and x < self.c:
    # If a is negative infinity and x less than c, the memberhsip degree must be 1
    return 1
if self.d == np.infty and x > self.b:
    # If d is infinity and x greater than b, the memberhsip degree must be 1
    return 1

# The following if test determines where in what section the x value is for the fuzzy set
# and reutns the corresponding memberhsip functions
if x < self.a:
    # If x is less than the first parameter, no membership is acounted for the value
    return 0
elif x < self.b:
    # If the memberhsip degree is greater than or equal to a and less than b,
    # the x value is in the line with positive slope
    if self.b - self.a == 0:
        # If b and a are equal, the memberhsip degree must be 0.
        # There is no fuzziness between a and b.
        return 0
    return (x-self.a)/(self.b-self.a)
elif x < self.c:
    # If x is less than c and greater than or equal to b,
    # the membership degree is in the core of the set (highest possible memberhsip degree).
    return 1
elif x <= self.d:
    # If x is less than or equal to d, the memberhsip degree
    # is defined by the line with negative slope
    if self.d - self.c == 0:
        # If d and c are equal, the memberhsip degree is 0 (no fuzziness)
        return 1
    return (self.d - x)/(self.d - self.c)
else:
    # For any vlaue greater than d, the degree is 0
    return 0
```

Fig 7. The code of the *get_membership_degree* method.

The clustering is done by using the KMeans class in the scikit-learn library. The number of clusters is given as argument to the method and the number of clustering rounds is set to 10. The result is a 2d-array where each feature in the DataFrame (including y) is assigned a cluster label.

In the method called *create_rulebase_kmeans_advanced* the number of rules that should be created is also given as argument. The purpose of this is to let the user have the option to cover more of the output space with the same number of sets. In that case the dataset is first clustered into the number of rules and then each feature is again clustered into the number of fuzzy sets. Note that the number of fuzzy sets must be less than the number of rules specified. This produces a mapping between the rule clusters and the set clusters where a group of rule clusters in each feature is covered by the same set clusters. The method *create_rulebase_kmeans* performs the procedure where the number of fuzzy sets equals to the number of rules. Only one clustering is performed.

The Trapezoidal Fuzzy Sets are derived from the cluster centres and the quantile of each feature among the inputs. The features are clustered so that each fuzzy set has its own set of datapoints that should be covered. The outer most parameter a is determined by subtracting the absolute difference of the center and the minimum value for that feature in the cluster. For d the absolute difference between centre and max is added to the center. For b the absolute difference between the mean and the value for the given quantile is subtracted from the center. For c the same absolute difference is added (see equation (7).

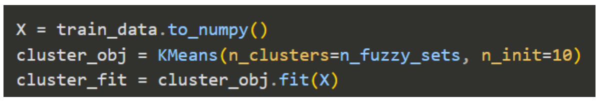
$$\begin{aligned} a &= center - |\bar{x} - min(x)| \\ b &= center - |\bar{x} - quantile(x)| \\ c &= center + |\bar{x} - quantile(x)| \\ d &= center + |\bar{x} - max(x)| \end{aligned}$$

(7)

After defining the fuzzy sets, the firing strength can be calculated. This is important for deriving the last part of the rules: the consequences. In ANFIS models, the training of the fuzzy sets are performed with backpropagation, while the training of the consequence parameters are commonly done with the least squares method [9, p. 418]. This is why it was decided to use the LinearRegression class of the scikit-learn library to train the coefficients/parameters of the first-order TSK consequents.

The consequent parameters need to be fitted to the target variable so that equation (4) produces the correct values. In order to use the *LinearRegression* method, each row of input values in the dataset is copied R times (for each rule) and multiplied with the firing strength for that rule divided by the total firing strength for all rules. The result is a dataset with R*(n+1) features (n is the number of features and +1 for the constant). This can be viewed as one single regression problem that is fitted with the *fit()* method of the *LinearRegression* object. The resulting parameters are the partitioned and added to the corresponding rules.

In the *create_rulebase_kmeans* method the above-described approach was slightly changed. There the dataset was modified for each rule. That way the parameters of the consequent were approximated for each rule separately. Another measure taken was to remove the datapoints that did



```
X = train_data.to_numpy()
cluster_obj = KMeans(n_clusters=n_fuzzy_sets, n_init=10)
cluster_fit = cluster_obj.fit(X)
```

Fig 8. Code snipet from create_rulebase_kmeans, showing the clustering instructions.

not have any firing strength for the rule before doing the least square fit. The hope was to make the fitting more efficient.

3) *Using bees algorithm to tune TSK parameters*

The field of evolutionary computing is concerned about bio inspired optimization problems. In this project we decided to look into the usage of Swarm Intelligence to tune a TSK model. Specifically, we tried to use the Bees Algorithm (BA) to train the model. Using evolutionary algorithms to optimize Fuzzy inference systems or ANFIS models is something that has been researched in the past. The paper by Karaboga & Kaya [12] mentioned cases where Genetic algorithms (GA), Ant Colony Optimization (ACO) or Particle Swarm Optimization (PSO) are among many algorithms that have been applied to training ANFIS models.

The BA algorithm is an algorithm inspired by the foraging behavior of bee colonies. In a bee colony some scout bees probe the search space (fields with plants) for food sources. When the scout returns to the hive, it performs the Waggle dance. This is a means to communicate information about a food source to the other bees in the hive (the worker bees). The dance communicates the Quality of the source (high rhythm, high quality), the direction relative to the sun (angle of the dance) and the travel distance (the length of the dance).

This behavior has been the inspiration for the BA, which consists of the following steps (Fig 9):

1. The scout bees are initialized. Each scout represents a feasible solution.
2. The scouts are sorted based on their fitness value. In this case RMSE is used as fitness function.
3. The *n_elite_sites* (number of elites) first scouts are selected as elite sites.
4. The remaining best sites (*n_elite_sites* – *n_best_sites*) are selected as best sites.
5. Local search is performed by sending *n_workers_beset* number of workers to the selected best sites and *n_workers_elite* to the elite sites.
    - Local search is performed by making small changes (size depending on the neighborhood size). This is done for every worker. Thus, search is performed in the proximity of the scout bee in the search space.
    - If one of the workers have a better fitness than the scout, it becomes the new scout of the site.
6. The remaining scouts, not selected, are reinitialized.
7. While the termination condition is not meet, repeat the steps from 2. In this case the termination condition is max number of generations/iterations.

One of the important design aspects in population-based evolutionary computing is the representation of the solutions that make up the populations. The representation has

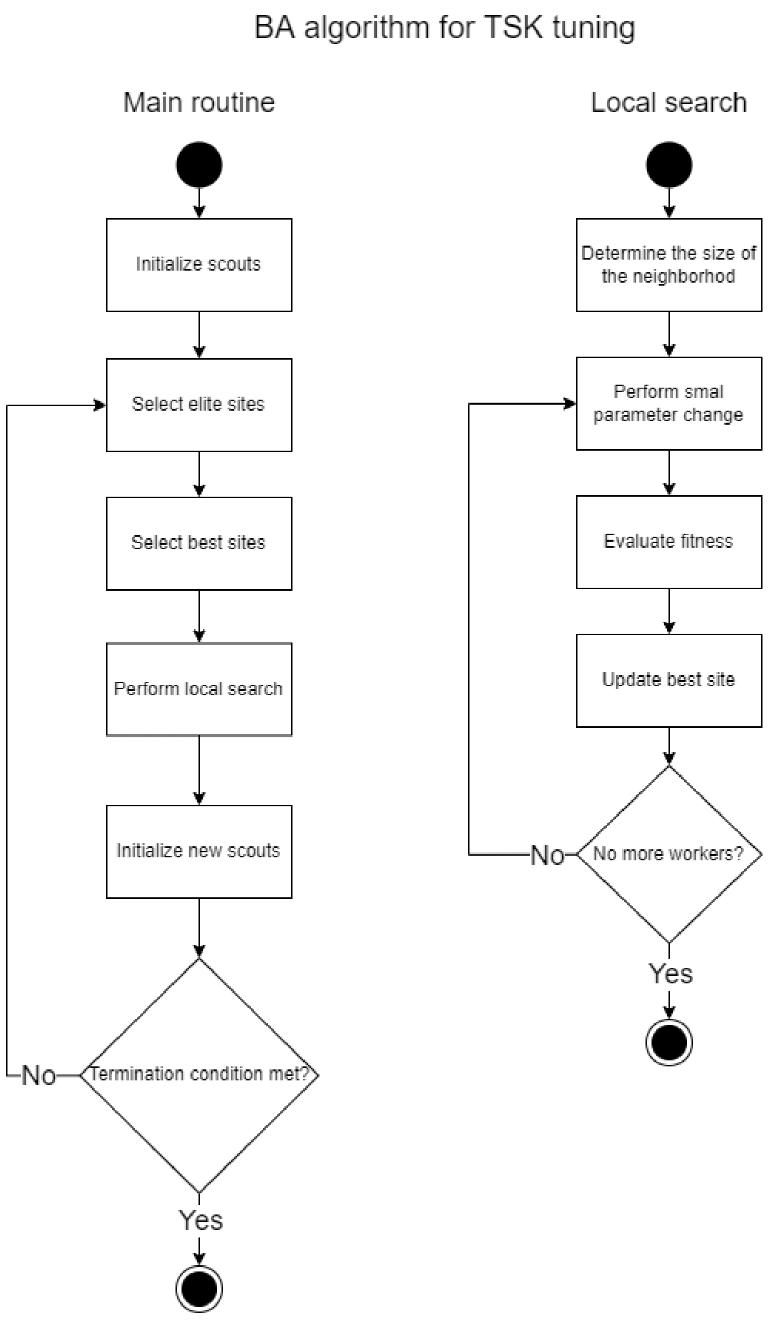


Fig 9. The flow diagram for BA.

implications for how any other step in the algorithm can be performed. In this case the solution is a TSKModel object. This is possible because the antecedent and consequent parameters are accessible as attributes in the TSKModel. This can be considered a relatively direct representation. Another option is the indirect approach where the model is encoded as for example a bit string that must be decoded each time the fitness is evaluated. The indirect approach is probably more relevant if the changes done to the solutions in the local search step are more sophisticated than those performed in this project.

The next important design decision is how the local search should be done. What changes can be applied to effectively search the area around the scout? In this project, we decided to use two different approaches for updating the antecedent parameters and the consequent parameters. In both cases the parameter to be updated is selected randomly.

For the consequence parameters, we added the product of a stochastic value (normal distributed), the increment value and the old parameter value. The increment value is the same as the shrinking factor for the site. Equations (8) and (9) are the equations used. Note that n is the number of iterations without improvement and *slimit* is the threshold for n. If n is higher than *slimit*, the site is abandoned. The symbol o stands for the old parameter value and N(0, 1) denotes a random value drawn from the standard normal distribution. Note how using the normal distribution probably will lead to doing more smaller random changes. The formula is defined in the *update_set_param* method.

$$ShrinkingFactor = 1 - \frac{n}{slimit} \tag{8}$$

$$UpdConsequence = o(1 + ShrinkingFactor \cdot N(0,1)) \tag{9}$$

$$UpdFuzzySet = o + o \cdot increment \cdot N(0,1) \tag{10}$$

For the fuzzy set parameters, the formula is defined in *update_consequence_param*. The equation used, see (10), is resembles the one used for the fuzzy set. But in this case the increment factor is derived by taking the difference between the min and max value divided by 10 for each feature in the dataset. The idea behind this approach is that the possible changes now limited to max one tenth relative to the values for the feature. This avoids too large changes while small changes are made more common because of the normal distribution.

The code for the BA optimization can be found in *ba_optimization.py*. The object can be used by instantiating the *BeesFitTSK* object with all the required parameters and call the *initialize_population* to create the initial TSK models with k-means clustering. Here it is important to note that by deriving all models in the population based on the same dataset, leads to a population with very little diversity. This means that only the local search abilities of BA are exploited. The idea was then to randomize the initialization of the models by assigning random numbers for the number of fuzzy sets (0 to 9) and a random number for the quantile used to define the fuzzy set.

The parameters that must be specified for the algorithm is listed in Table 2. For initialization the number of sets (greater than 0) and the quantile must be specified (real number in [0, 0.7]).

D. Describing usage of pyFume

To create a good experiment to evaluate the program we implemented, we used the pyFume library. This is a python library that consists of methods and classes to estimate TSK model from data [3]. This library is very similar to our implementation in that it also uses clustering to partition the dataset and then derive the fuzzy sets from that. It also uses the leas squared fitting method to estimate the first-order TSK consequence parameters [3].

The library can easily be used by installing pyFume and then specify a *pyFUME* object with the dataset (pandas dataframe) and the number of clusters/sets as second parameter. It is then possible to inspect the TSK controller

Table 2. The BA parameters.

Name	Description
n_scout_bees	The number of solutions in the populations
n_best_sites	The number of best sites selected from population
n_elite_sites	The number of elite sites selected
n_workers_best	The number of worker bees deployed to the best sites
n_workers_elite	The number of worker bees deployed to elite sites
n_neighborhod_size	The number of changes applied to a model by one worker
slimit	The number of iterations a site is searched without improvement until the site is abandoned.

by calling *get_model* and then call *produce_figure* on the returned model object. This plots the membership functions. A difference from our implementation is that only sigmoid and gaussian membership functions can be used, while our implementation only supports trapezoidal membership functions [3].

E. Fuzzy Inference System in Matlab

In the implementation of ANFIS in Matlab, objects and functions from the existing *Fuzzy Logic Toolbox* for fuzzy logic and FIS tuning were used. This toolbox makes it easy to train an ANFIS model with relatively few lines of code.

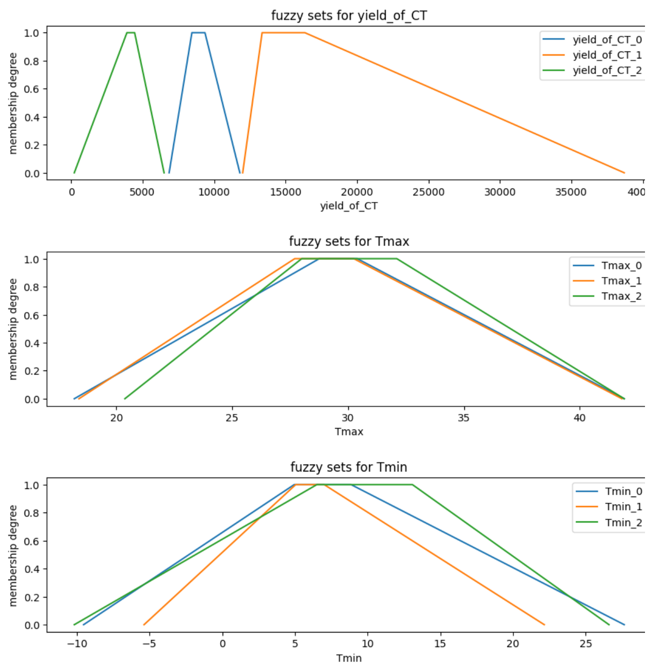


Fig 11. Plots of the fuzzy sets from our implemented TSK estimation.

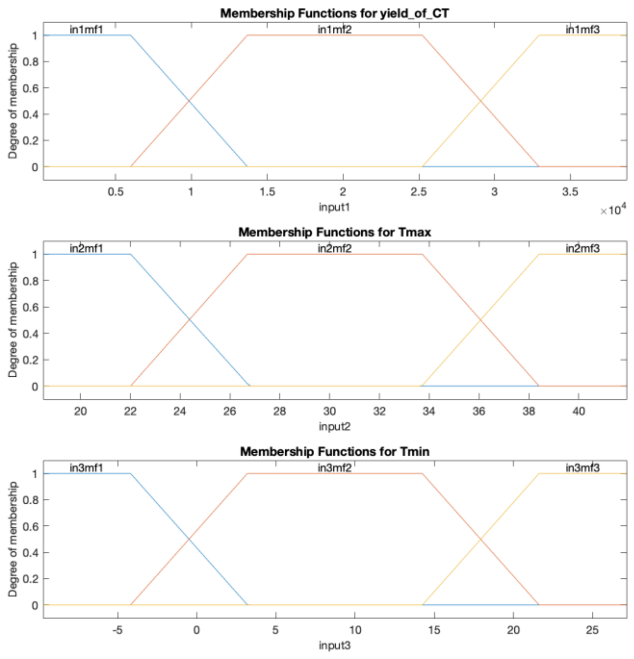


Fig 12. Plots of the fuzzy sets produced with MATLAB.

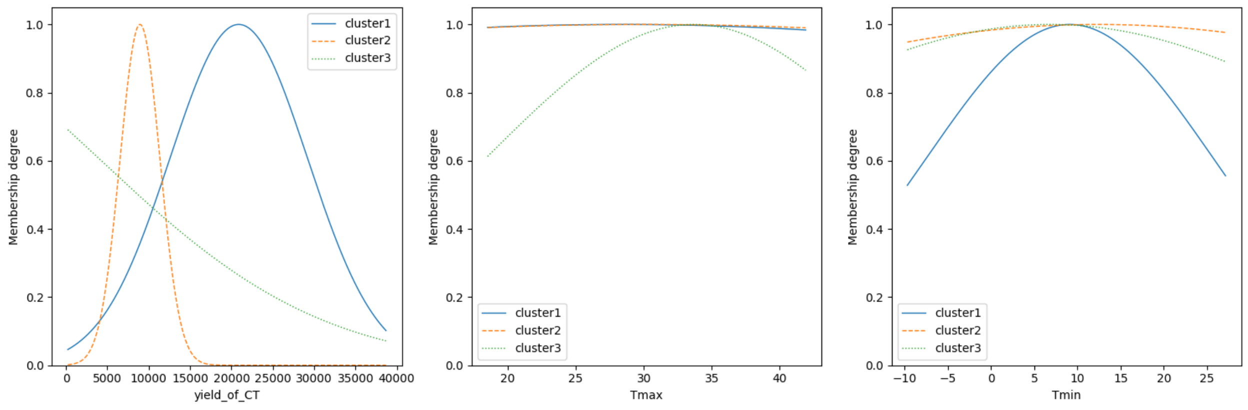


Fig 10. The visualizatoin of the fuzzy sets from the pyFume module (trained on yield_of_ct, t_max and t_min).

The dataset was split into two sets where 20% of the original dataset is for testing and 80% is for training. *Relative_yieald_change* is the output (*y*) for each run, while the input variables (*x*) were changed for different runs.

After importing the required datasets, the specific columns in each set are assigned to the variables *x_train* and *y_train* which is the input and output of the training data, and *x_test* and *y_test*, which is the input and output of the test data.

To specify the structure of the FIS model, the *genfis()* function is used to create an initial FIS object. This function takes input and output data along with an option set from the *genfisOptions* object. Here, the number of membership functions is set to 3 with *NumMembershipFunctions* and membership type is set to *'trapmf'*, a trapezoid, with *InputMembershipFunctionType*. The initial FIS is then created as *inFIS* using *genfis()* with the specified options.

The *anfisOptions* object is used to specify tuning options for the *anfis()* function. *EpochNumber* is the number of training epochs, and the training process stops when this value is reached. *anfisOptions* also have options for showing information about the system and the results. *DisplayANFISInformation* displays information about the system like the number of nodes, number of linear and nonlinear parameters, number of fuzzy rules and more. *DisplayErrorValues* shows the error values for each epoch of the training process. The initial FIS object *inFIS* is assigned to *InitalFIS* in *anfisOptions*.

The training input and output data, *x_train* and *y_train* is converted from table data to numeric arrays and fed to the *anfis()* function along with the specified options in the *anfisOptions* assigned to the *opt* variable. The *anfis()* function tunes a TSK-type system from training data. It automatically generates the system using grid partitioning. The resulting trained ANFIS model is then stored in the *test_fis* variable.

The *evalfis()* function is used to evaluate the system, taking the ANFIS model *test_fis* and test input *x_test* as arguments. The output can then be compared with the original output data *y_test.*

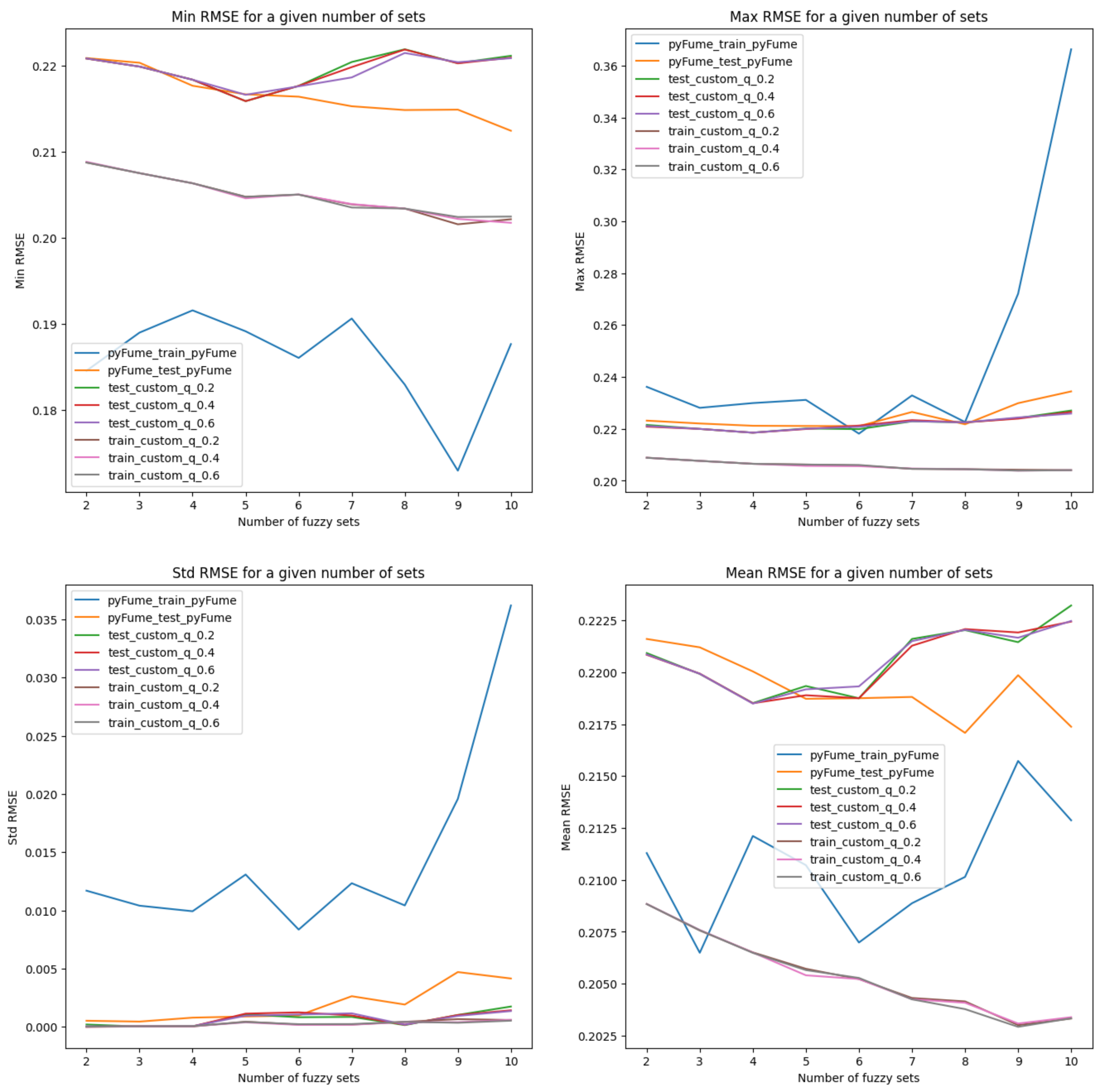


Fig 13. The results from the RMSE performance testing of the pyFume implementation and our implementation.

III. Results

There are multiple interesting test cases in this project. We started by inspecting the TSK model fuzzy sets and consequences for our implementation, pyFume and MATLAB. Unfortunately, it became clear that the goal of running estimating a TSK model from 17 input variables would be unfeasible with the resources we have in hand. The first attempt was with the anfis function in MATLAB. This function uses grid partition and the training process failed because of lack of available memory. We thus shifted focus on training the models with only 3 to 5 input variables. The performance measures will not be comparable with the result in [1], but it made us able to evaluate the performance of our implemented model against the pyFume library and the ANFIS library in MATLAB.

First the estimation of TSK controllers for our implementation was compared with the pyFume library. This was done in the *evaluation.ipynb* notebook. In order to observe the ability to generalize for the models, the dataset was divided into a training and a test set (the test set consists of 20% of the data). The training was also done for several different parameter sets consisting of the parameters: number of fuzzy sets and the quantile to determine the fuzzy set core. For each parameter set, the training and measurement was repeated 20 times.

We tried several different input combinations. In the paper by Ryo [1], the features “yield_of_CT” and “T_max” are highlighted as the most important. To those two, we also added "T_min.". The performance measure used is RMSE, the same measure used by Ryo. The results from the testing were ordered in a table that is added as Appendix II. It is important to note that the BA optimization method was not used for this test.

In Fig 13 the table is visualized in four plots. There is one plot for each statistical measure that was applied to aggregate the rmse values from the runs (min, max, standard deviation and mean). Each plot consists of several graphs for different combinations of test and training set, quantile (only for our model) and model type (pyFume or our model). The x axis is the number of fuzzy sets derived from the data and the y axis is the rmse number.

The first observation that can be made is that the different quantile values (in this case 0.2, 0.4, 0.6) do not have a visible effect on the accuracy of the model for this problem because all the test_custom plots are grouped together similar to the train_custom plots. This might be because the changes are not significant. It could also be an indication of a bug in the code, but a visual inspection of the fuzzy sets created for different quantile values, a visual difference can be observed. The plots that visualize this can be found in *evaluation.ipynb*.

Another observation is that the pyFume model has a standard deviation that is clearly higher compared to the custom model for the training set. But it is also clear that that is not the case for the test set, because it has the same deviation as the custom model and better mean rmse than the test set plots for the custom mode. It therefore looks like the pyFume implementation is better in avoiding overfitting than our model.

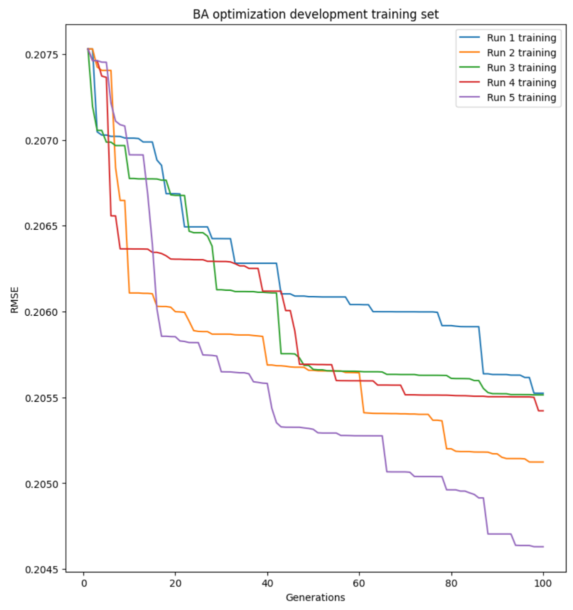
It can also be observed that the mean rmse for our model is decreasing with increase of fuzzy sets for the plots of the training set. For the test set, however, a negative development can be seen, indicating that our model is prone to overfitting when the number of fuzzy sets increase. The pyFume plots show the opposite development where the test set plot has a negative slope. This again indicates that the methods used in pyFume are solving the generalization problem better than our model.

Next, the BA optimization implementation was tested. The tests were conducted with the same setup as for the TSK estimation. The BA parameters were set focus on exploitation/local search because the initialization of the scout produce very similar results, and thus preventing global search. This was achieved by setting the numbers of scouts to 5, the number of elite sites to 2 and 2 sites as best sites. Next the number of elite workers were set to 30 and best site workers to 10. The neighborhood size was set to 3 and the stagnation limit was set to 100.

The first results showed that the optimization is very slow. It takes multiple seconds to do a generation. It looks like the main time consumption is the fitness evaluation where the models are tested against the training and test set. We also see that after 100 generations, the optimization is still reducing the rmse, this is why the stagnation limit was set higher to avoid premature abandonment.

For about every second generation, a very small improvement in RMSE can be observed. In Fig 14 the performance development of 5 tsk models are visualized during the BA optimization. The program is only doing 100 iterations per model and this takes on average 239 seconds. The models creates a tsk from 3 clusters (3 rules/fuzzy sets are created) and a trap_quantile of 0.4.

From the plot, it can be observed that in all cases there is performance improvement continuously. It also looks like the improvements get smaller towards 100 generations, indication that the algorithm is converging towards a local optimum. Another observation is that the performance on the test set also is improved proportional to the training performance. The test performance is consistently at 0.01 rmse greater than the training performance, indicating that the fine tuning does not lead to more overfitting for the first 100 generations. From this it can be concluded that the TSK models have room for improvement in terms of accuracy and generalization.



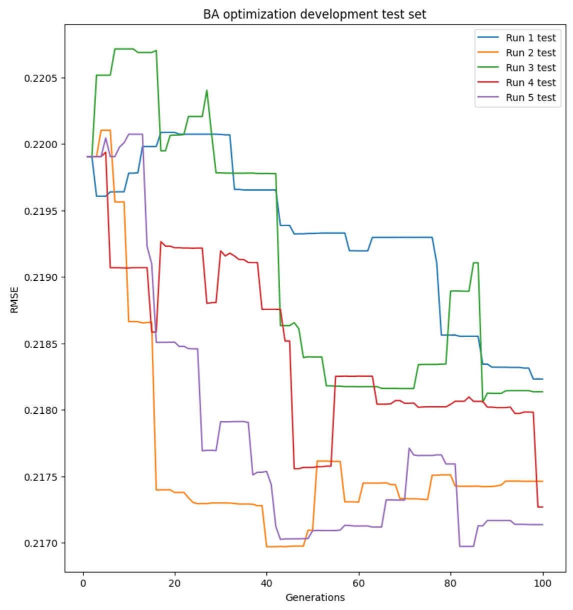


Fig 14. Plots from the BA evaluation. Left plot contains the rmse values on the training set (used to evaluate fitness). Right plot contains the rmse values on the test set.

Because of the observation that the convergence is slow, it was attempted to increase the changes made in the local search. This was done by setting the neighboorhod change parameter to 10. There was however not much improvement in time consumption because the time per generations increased.

In the MATLAB implementation, the tsk was trained with the *anfis()* function. The result after 10 epochs is a rmse of 0.200304. The fuzzy sets created by the model can be seen in *Fig 12* while the fuzzy sets from pyFume and our implementation can be found in *Fig 11* and *Fig 10*. By comparing the plots, the different approaches to partitioning can be seen. MATLAB uses grid partition and covers the entire variables with fuzzy sets. The other two implementations use clustering which becomes clear by looking at the *t_max* and *t_min* variable where all clusters seem to be grouped together. There can be observed some similarities in the generated fuzzy sets for pyFume and our implementation.

One of the goals of the project was to assess the interpretability of fuzzy controllers compared to other methods in XAI. This was in this case not possible because we were not able to train on the same number of parameters as it was done in [1]. From this we can see that the methods applied by Ryo [1] are better suited for this use case compared to basic TSK models.

Despite this we also looked at the fuzzy sets and the consequences. It can be observed that there is created a fuzzy set for values between 0 and 5000 in the *yield_of_CT* variable for both the tsk models derived with cluster partitioning. A look at the consequences of the rule base shows that the coefficients of the *t_min* and the *t_max* are positive and greater compared to the other rules. From this, we are able to see the same tendency as mentioned in [1], namely that a *yield_of_ct* variable less than 5000 can lead to positive relative yield change. However, it is difficult to confirm the correlation between *tmax* and relative yield change. If we look closely, it can be seen that the fuzzy set belonging to the same rule as the 0 to 5000 fuzzy set in *yield_of_CT*.

IV. Discussion

Throughout this project it became clear that the project we had envisioned was more time consuming in practice. The development of the custom TSK estimator brought with it some problems and bugs that required time. At the end of the project there are numerous things we would like to continue working on.

For one, the TSK estimator has not been tested as thorough as we see it as necessary. There are also some week parts that can cause some bugs, like the generation of fuzzy sets, that we would like to improve. The code base should also be refactored because there are some parts that are not as flexible. An improvement would be to structure the code in a way that allows to extend the TSK implementation with other types of membership functions.

We also would like to spend more time on finding ways to make our implementation more interpretable. An idea we have is to create surface plots with two input variables. This could give valuable insight about when the yield change is positive and possibly disclose a pattern.

With regards to the Bee algorithm, it can be argued if using another form of local search would be better suited in this use case. In the current state we are only using the local search functionality of the algorithm. It would also be great to spend some time improving the convergence rate and make the entire implementation more user friendly.

The problem with the number of input variables causing too many rules can be solved with various methods. One Method often seen in literature, especially for cases where XAI is applied to CNNs, Fuzzy Rough Feature Selection (FRFS) algorithm [5]. Til would be interesting to look into in order to use ANFIS with backpropagation and gradient decent as learning methods.

V. References

[1] M. Ryo, ‘Explainable artificial intelligence and interpretable machine learning for agricultural data analysis’, *Artif. Intell. Agric.*, vol. 6, pp. 257–265, Jan. 2022, doi: 10.1016/j.aiia.2022.11.003.

[2] N. Talpur, S. J. Abdulkadir, H. Alhussian, M. H. Hasan, N. Aziz, and A. Bamhdi, ‘Deep Neuro-Fuzzy System application trends, challenges, and future perspectives: a systematic survey’, *Artif. Intell. Rev.*, vol. 56, no. 2, pp. 865–913, Feb. 2023, doi: 10.1007/s10462-022-10188-3.

[3] C. Fuchs, S. Spolaor, M. S. Nobile, and U. Kaymak, ‘pyFUME: a Python Package for Fuzzy Model Estimation’, in *2020 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*, Jul. 2020, pp. 1–8. doi: 10.1109/FUZZ48607.2020.9177565.

[4] V. Meshram, K. Patil, V. Meshram, D. Hanchate, and S. D. Ramkteke, ‘Machine learning in agriculture domain: A state-of-art survey’, *Artif. Intell. Life Sci.*, vol. 1, p. 100010, Dec. 2021, doi: 10.1016/j.ailsci.2021.100010.

[5] M. Ismail, C. Shang, and Q. Shen, ‘Towards a Framework for Interpretation of CNN Results with ANFIS’, in *Advances in Computational Intelligence Systems*, T. Jansen, R. Jensen, N. Mac Parthaláin, and C.-M. Lin, Eds., in Advances in Intelligent Systems and Computing. Cham: Springer International Publishing, 2022, pp. 153–166. doi: 10.1007/978-3-030-87094-2_14.

[6] Y. Su, B. Gabrielle, and D. Makowski, ‘A global dataset for crop production under conventional tillage and no tillage systems’, *Sci. Data*, vol. 8, no. 1, Art. no. 1, Jan. 2021, doi: 10.1038/s41597-021-00817-x.

[7] J. Rogers and S. Gunn, ‘Identifying Feature Relevance Using a Random Forest’, in *Subspace, Latent Structure and Feature Selection*, C. Saunders, M. Grobelnik, S. Gunn, and J. Shawe-Taylor, Eds., in Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2006, pp. 173–184. doi: 10.1007/11752790_12.

[8] H. Singh and Y. A. Lone, *Deep Neuro-Fuzzy Systems with Python: With Case Studies and Applications from the Industry*. Berkeley, CA: Apress, 2020. doi: 10.1007/978-1-4842-5361-8.

[9] R. Kruse, C. Borgelt, C. Braune, S. Mostaghim, and M. Steinbrecher, *Computational Intelligence*. in Texts in Computer Science. London: Springer, 2016. doi: 10.1007/978-1-4471-7296-3.

[10] A. Géron, *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. Sebastopol, UNITED STATES: O’Reilly Media, Incorporated, 2019. Accessed: Nov. 27, 2023. [Online]. Available: http://ebookcentral.proquest.com/lib/hioa/detail.action?docID=5892320

[11] ‘Numeracy, Maths and Statistics - Academic Skills Kit’. Accessed: Nov. 27, 2023. [Online]. Available: https://www.ncl.ac.uk/webtemplate/ask-assets/external/maths-resources/statistics/regression-and-correlation/coefficient-of-determination-r-squared.html

[12] D. Karaboga and E. Kaya, ‘An adaptive and hybrid artificial bee colony algorithm (aABC) for ANFIS training’, *Appl. Soft Comput.*, vol. 49, pp. 423–436, Dec. 2016, doi: 10.1016/j.asoc.2016.07.039.

[13] M.-C. Su, C.-W. Liu, and S.-S. Tsay, ‘Neural-network-based fuzzy model and its application to transient stability prediction in power systems’, *Syst. Man Cybern. Part C Appl. Rev. IEEE Trans. On*, vol. 29, pp. 149–157, Mar. 1999, doi: 10.1109/5326.740677.

[14] C.-T. Lin, I.-F. Chung, H.-C. Pu, T.-H. Lee, and J.-Y. Chang, ‘Genetic algorithm-based neural fuzzy decision tree for mixed scheduling in ATM networks’, *IEEE Trans. Syst. Man Cybern. Part*

B Cybern. Publ. IEEE Syst. Man Cybern. Soc., vol. 32, pp. 832–45, Feb. 2002, doi: 10.1109/TSMCB.2002.1049617.

Appendix I

Instructions for how to navigate this repository and the source code of the most important files.

Link to repo: https://github.com/B-Moritz/ACIT4620_XAI_with_ANFIS.

This project is the exam assignment for the ACIT4100 class at OsloMet. The repository contains the code of a TSK estimation implementation. The following files are important:

- The file used to produce the training and test is the preliminary_analysis.ipynb. It is important that this notebook is executed whenever the other parts of the project are tested.
- The TSK fuzzy controller is defined in the file: tsk_model.py. If the file is executed, a simple TSK model is defined. This part was used for testing under the development of the TSK code. A 3D-plot describing the model is showed.
- The methods for deriving the TSK from a dataset is found in the TSKModel class in tsk_model_py.
- The Bees Algorithm implemented for fine tuning the TSK model can be found in the file ba_optimizatino.py. This file can be executed to test the optimization. The output will be the evolving rmse values.
- The evaluation of the tsk models were done in the file evaluation.ipynb. By running this notebook the table and plot of the evaluation can be produced.
- Testing of the Bees algorithm can be found in test_bees_optimization.ipynb.
- The Matlab ANFIS implementation can be found in Anfis_matlab_1.mlx

The dataset used in this project is available here:

- https://doi.org/10.6084/m9.figshare.12155553

Please make sure the csv containing the dataset is located in the dataset folder.

Create the environment

To keep track of dependencies an environment management system was used: venv. Please make sure to create a venv environment and install the requirements by following these steps:

- Open the terminal from the root folder of the github repository (assuming the repository is cloned to your local computer).
- Run the command: python -m venv test_env to create the virtual environment.
- Activate the environment:
  - Mac: source test_env/bin/activate
  - Windows: ./test_env/Script/activate
- Install libraries: pip install -r requirements.txt

To make the environment available in jupyter, a ipykernel needs to be installed: python -m ipykernel install --name=test_env

Running CI_RF_mm in jupyter

The code can be run as is simply hitting shift+enter will run the selected cell and select the next, repeat until there are no more code segments. any cells with only df or df.info() can be rerun without causing any error since it is only used to display current df.

Code from tsk_model.py:

```
"This file contains the program for optimizing ANFIS with the bees algorithm"

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib import cm
from sklearn.cluster import KMeans
from sklearn.linear_model import LinearRegression
import pdb


class MfTrap():
    """The membership function used to define a trapezoidal fuzzy set.
    """

    def __init__(self, a, b, c, d, name):
        # The four parameters used to define the function
        self.a = a
        self.b = b
        self.c = c
        self.d = d
        # The name of the set (linguistic expression) is defined as an attribute to the
object
        self.name = name
        # The set is validated. Note how the following must be satisfied a < b <= c < d
        self.validate_set()

    def validate_set(self):
        if self.a > self.b:
            print("a:" + str(self.a) + ", b:" + str(self.b))
            raise Exception("Parameter a is larger than b.")
        if self.b > self.c:
            raise Exception("Parameter b is larger than c.")
        if self.b > self.c:
            raise Exception("Parameter c is larger than d.")

    def get_membership_degree(self, x):
        """This method produces the membership degree of the value x to the current
defined trapezoidal set (a,b,c,d)

        Parameters
        ----------
        x : float

        Returns
        -------
        float in the universeral interval [0,1]
            The membership degree of the input value.
        """
        # Check for cases where the set represents x values for infinity
        if self.a == -np.infty and x < self.c:
            # If a is negative infinity and x less than c, the memberhsip degree must
be 1
            return 1
        if self.d == np.infty and x > self.b:
            # If d is infinity and x greater than b, the memberhsip degree must be 1
            return 1

        # The following if test determines where in what section the x value is for the
fuzzy set ¨
        # and reutns the corresponding memberhsip functions
        if x < self.a:
            # If x is less than the first parameter, no membership is acounted for the
value
            return 0
        elif x < self.b:
            # If the memberhsip degree is greater than or equal to a and less than b,
            # the x value is in the line with positive slope
            if self.b - self.a == 0:
                # If b and a are equal, the memberhsip degree must be 0.
                # There is no fuzziness between a and b.
                return 0
            return (x-self.a)/(self.b-self.a)
        elif x < self.c:
            # If x is less than c and greater than or equal to b,
            # the membership degree is in the core of the set (highest possible
memberhsip degree).
            return 1
        elif x <= self.d:
            # If x is less than or equal to d, the memberhsip degree
            # is defined by the line with negative slope
            if self.d - self.c == 0:
                # If d and c are equal, the memberhsip degree is 0 (no fuzziness)
                return 1
            return (self.d - x)/(self.d - self.c)
        else:
            # For any vlaue greater than d, the degree is 0
            return 0

    def get_param_list(self):
        return [self.a, self.b, self.c, self.d]

    def set_param_list(self, param_list):
        self.a = param_list[0]
        self.b = param_list[1]
        self.c = param_list[2]
        self.d = param_list[3]
        self.validate_set()

class FeatureFuzzySets():
    "This object represents all input features and their fuzzy sets"
    def __init__(self, set_list, feature_name="Unknown"):
        self.fuzzy_sets : [MfTrap] = np.array(set_list)
        self._feature_name = feature_name

    def __setitem__(self, index, new_set):
        self.fuzzy_sets[index] = new_set

    def __getitem__(self, index):
        return self.fuzzy_sets[index]

    def remove_set(self, index):
        self.fuzzy_sets[index] = None
        return True

class TKSConsequence():
    """ A TSK consequence of first order with three parameters
        f(x1, x2) = x1 * p + x2 * q + r
    """
    def __init__(self, params, const):
        self.params_list = params # Corresponds to the coefficients of input values of 
the rule
        self.const = const # this coresponds to r

    def calculate_consequence(self, x_list, feature_indexes):
        # This method is used to calculate the function value of the TSK consequent
        cur_sum = self.const
        for counter, x_ind in enumerate(feature_indexes):
            cur_sum += x_list[x_ind]*self.params_list[counter]

        return cur_sum

class TSKAntecedent():
    """This class represents the antecedent part of a TSK rule and can calculate the 
firing strength for the rule.
       The t-norm used is product.
    """
    def __init__(self, fuzzy_sets_indexes : []):
        self.fuzzy_sets_indexes : [] = fuzzy_sets_indexes
        self.w = None

    def calculate_firing_strenght(self, x_list:[], fuzzy_sets:[FeatureFuzzySets]):
        # This method calculates the firing strenght of the antecedents
        # Start by setting the firing strenght to the identity value for the product
operator
        w = 1
        for feature_number, set_number in self.fuzzy_sets_indexes:
            # For each set in the antecedent, calculate the membership degree and find
the product of degree and the firing strength calculated up until now.
            w = w *
fuzzy_sets[feature_number][set_number].get_membership_degree(x_list[feature_number])

        self.w = w
        return self.w

    def get_firing_strenght(self):
        return self.w

class TSKRule():
    """A class used to defining a TSK rule.
    """
    def __init__(self, fuzzy_sets_indexes, params : [float], const : float):
        # The antecedent of the rule
        self.antecedent = TSKAntecedent(fuzzy_sets_indexes)
        # The consequent of the rule
        self.consequent = TKSConsequence(params, const)
        # The indexes of the sets making up the antecedents
        self.feature_indexes = fuzzy_sets_indexes

    def calculate_consequence_func(self, x_list):
        return self.consequent.calculate_consequence(x_list, self.feature_indexes[:,
0])

    def to_string(self, fuzzy_sets, expressions) -> str:
        antecedent_str = ""
        for i, feature in enumerate(self.feature_indexes):
            antecedent_str += f"{expressions[feature[0]]} in " +
fuzzy_sets[feature[0]][feature[1]].name
            if i < (len(self.feature_indexes)-1):
                antecedent_str += " and "

        consequent_str = "y = "
        for i, feature in enumerate(self.feature_indexes[:,0]):
            consequent_str += f"{expressions[feature]} *
{self.consequent.params_list[feature]:.4f}"
            if i < (len(self.feature_indexes)-1):
                consequent_str += " + "
        return f"IF {antecedent_str} THEN {consequent_str} +
{self.consequent.const:.4f}"

class TSKRuleBase():
    """This class represents a TSK rule base. It has the responsibility to add and
remove rules.
        It also has the responsibility to print out the rules.
    """
    def __init__(self, feature_names : [str], expressions : {str:str}):
        self.rules : [TSKRule] = []
        # Storing feature names and creating expressions for printout.
        self.feature_names = feature_names
        self.expressions = []
        for i, feature in enumerate(self.feature_names):
            if feature in expressions:
                self.expressions.append(expressions[feature])
            else:
                self.expressions.append("x"+str(i))

    def appendRule(self, new_rule : TSKRule):
        self.rules.append(new_rule)

    def removeRule(self, index : int):
        self.rules.pop(index)

    def print_rulebase(self, fuzzy_sets):
        for rule in self.rules:
            print(rule.to_string(fuzzy_sets, self.expressions))

    def write_to_csv(self, fileName="Last_rulebase.csv"):
        """This method writes the rule base to csv
        """
        rows = []
        columns = [np.concatenate(self.feature_names, ["p"+i for i in
range(len(self.feature_names))], ["r"])]
        rule : TSKRule
        for rule in self.rules:
            rows.append([np.concatenate(rule.feature_indexes[:,1],
rule.consequent.params_list, [rule.consequent.const])])

        self.rulebase_df = pd.DataFrame(rows, columns=columns)
        self.rulebase_df.to_csv(fileName)


class TSKModel():

    def __init__(self, debug:bool = False):
        self._feature_fuzzy_sets = [MfTrap]
        self.test_actuals = []
        self.r_squared = None
        self.test_data = None
        self.debug = debug
        self.generation = 0


    def randomize_model(self,
                        n_fuzzy_sets : int,
                        n_fuzzy_rules : int,
                        feature_boudaries : {},
                        feature_names : [],
                        max_expressions : int=2):

        if n_fuzzy_sets < 2:
            # Too few fuzzy sets
            raise Exception(f"Too few fuzzy sets: n_fuzzy_sets < {2}")
        # Model attributes
        # Numbe rof fuzzy sets per feature
        self.n_fuzzy_sets : int = n_fuzzy_sets
        # Number of rules (should be n_fuzzy_sets**n_features for grid partition)
        self.n_fuzzy_rules : int = n_fuzzy_rules
        # A list of feature names
        self.feature_names = feature_names
        # Feature max min list
        self.feature_boundaries = feature_boudaries
        # Maximum number of expressions in the rules (should be same as number of
independent features)
        self.max_expressions = max_expressions

        # Create the random fuzzy sets
        overlap_const = 0.5 # 1 full overlapping between sets, 0 there cannot be anny
overlaps
        for feature_number, boundaries in self.feature_boundaries.items():
            cur_feature_sets = []
            portion_size = (boundaries[1] - boundaries[0]) / self.n_fuzzy_sets
            for set_num in range(self.n_fuzzy_sets):
                if set_num == 0:
                    # Make sure the first set is defined to -infinity
                    cur_params = np.sort(np.random.uniform(boundaries[0] +
portion_size*(set_num - overlap_const), boundaries[0] + portion_size*(set_num + 1 +
overlap_const), size=2))
                    cur_feature_sets.append(MfTrap(-np.infty,-
np.infty,cur_params[0],cur_params[1],
f"{self.feature_names[feature_number]}{set_num}"))
                elif set_num == (self.n_fuzzy_sets-1):
                    cur_params = np.sort(np.random.uniform(boundaries[0] +
portion_size*(set_num - overlap_const), boundaries[0] + portion_size*(set_num + 1 +
overlap_const), size=2))
                    cur_feature_sets.append(MfTrap(cur_params[0],cur_params[1],
np.infty, np.infty, f"{self.feature_names[feature_number]}{set_num}"))
```

```
                else:
                    cur_params = np.sort(np.random.uniform(boundaries[0] + 
portion_size*(set_num - overlap_const), boundaries[0] + portion_size*(set_num + 1 + 
overlap_const), size=4))
                    cur_feature_sets.append(MfTrap(cur_params[0],cur_params[1],cur_para
ms[2],cur_params[3], f"{self.feature_names[feature_number]}{set_num}"))

            # Add the sets to the input feature
            self._feature_fuzzy_sets.append(FeatureFuzzySets(cur_feature_sets, 
self.feature_names[feature_number]))

        feature_list = np.array(list(self.feature_boundaries.keys()))
        self.rulebase : TSKRuleBase = TSKRuleBase(self.feature_names, expressions={})
        for rule_n in range(self.n_fuzzy_rules):
            # For each rule created

            # pick some of the input features for the rule
            feature_indexes = np.random.choice(len(feature_list), self.max_expressions, 
replace=False)
            cur_MF_list = [] # The list containing the index of the mfs for the rule
            params = []
            for feature in feature_indexes:
                # For each feature, pick one of the sets
                cur_set_number = np.random.choice(self.n_fuzzy_sets, 1)[0]
                cur_MF_list.append([feature, cur_set_number])
                params.append(1)

            self.rulebase.appendRule(TSKRule(np.array(cur_MF_list), params, 1))

    def calculate_output(self, x_list, activate_debug=False):
        """This method is used to apply the trained model on some input values.

        Parameters
        ----------
        x_list : [int]
            A list of one value for each input variable x

        Returns
        -------
        float
            The predicted value
        """
        a_out = 0
        w_total = 0
        rule : TSKRule = None

        for rule in self.rulebase.rules:
            # Calculate the total firing strenght for normalization
            cur_firing_strenght = rule.antecedent.calculate_firing_strenght(x_list, 
self._feature_fuzzy_sets)
```

```
            w_total += cur_firing_strenght
        if w_total == 0:
            # The output is 0 if there is no firing strength
            #print(x_list)
            return 0

        for i, rule in enumerate(self.rulebase.rules):
            cur_firing_strenght = rule.antecedent.get_firing_strenght()
            cur_consequent_value = rule.calculate_consequence_func(x_list)
            if activate_debug:
                # Print out the rule calculation results for transparency
                print(f"Rule {i}: \t Firing strength: {cur_firing_strenght}, Consequent
value: {cur_consequent_value}")
            a_out += (cur_firing_strenght / w_total)*cur_consequent_value

        return a_out

    def set_rulebase(self, rulebase : TSKRuleBase):
        # Check that the number of rules and max expressions match the previous
configuration
        if len(rulebase.rules) > self.n_fuzzy_rules:
            raise Exception("The number of rules are not matching the previous
configuration.")
        if len(rulebase.rules[0].antecedent.fuzzy_sets_indexes) > self.max_expressions:
            raise Exception("The number of expressions per rules are too high.")

        self.rulebase = rulebase
        if self.debug:
            self.rulebase.print_rulebase(self._feature_fuzzy_sets)

    def set_feature_fuzzy_sets(self, feature_fuzzy_sets : [FeatureFuzzySets]):
        # Check that there are enough sets
        if len(feature_fuzzy_sets) != len(self.feature_names):
            raise Exception("The number of features are not matching with the existing
feature set.")
        if len(feature_fuzzy_sets[0].fuzzy_sets) != self.n_fuzzy_sets:
            raise Exception("The number of sets per feature is not matching existing
configuration.")

        self._feature_fuzzy_sets = feature_fuzzy_sets

    def create_rulebase_kmeans_advanced(self,
                                   train_data : pd.DataFrame,
                                   n_fuzzy_sets : int=3,
                                   n_rules : int=10,
                                   trap_quantile : float=0.4,
                                   expressions={}) -> None:
        """This method derives the TSK rulebase by performing K-Means clustering and
Multi-Linear Regression.
```

```
        Parameters
        ----------
        train_data : pandas.DataFrame
            The training data. Dependent variable is column 1 (index 0). Independent
features follow
        n_fuzzy_sets : int, optional
            Number of fuzzy sets per independent feature, by default 3
        inner_bound_factor : int, optional
            Factor for setting the core of the trapezoidals. center +/-
outer_bound_factor*std, by default 1
        """
        # Verify that there are enough features in dataframe (feature 1 is the
target/dependent variable)
        if len(train_data.columns) < 2:
            raise Exception("The provided input data does not have a valid dimention:
dimention < 2")

        # Verfiy the given number of fuzzy sets
        if n_fuzzy_sets < 1:
            raise Exception(f"Cannot create model with a number of fuzzy sets of
{n_fuzzy_sets} for each input.")

        self.n_fuzzy_sets = n_fuzzy_sets

        X = train_data.to_numpy()
        cluster_obj = KMeans(n_clusters=n_rules, n_init=10)
        cluster_fit = cluster_obj.fit(X)
        # Add a feature at the end of the dataframe, containing the cluster labels
        merged_data = pd.merge(train_data, pd.DataFrame({"Subcluster_number" :
cluster_fit.labels_}), left_index=True, right_index=True)
        self.merged_dataset = merged_data
        # Resulting dataframe has target in first column, independent features in the
middle and the cluster labels in the last colummn
        # Extract the featurenames by ignoring the first and last column
        self.feature_names = merged_data.columns[1:-1]
        # Extract the target name
        self.target_name = merged_data.columns[0]
        self.max_expressions = len(self.feature_names)

        # Extracting all unique subclusters/rules
        subclusters = merged_data["Subcluster_number"].unique()

        # Derive the membership functions from the clusters (trapeziums)
        cluster_center : [float] = []
        mfs : [FeatureFuzzySets] = []
        # Initialization of the rule list dict
        clusters : {} = {i : [] for i in np.unique(cluster_fit.labels_)}

        for k in range(1, len(merged_data.columns)-1):
            # Iterate over all features (independent variables), starts at 1 because
first column is not an independent variable
            # k - feature index (column index)

            # Get the cluster centers of the subclusters. Note that the
cluster_fit.cluster_centers_ matrix has cluster on axis 0 and feature cluster center
along axis 1
            raw_mfs = cluster_fit.cluster_centers_[:,k].reshape(-1, 1)
            # Reduce amount of mfs by clustering in one dimension
            reduced_mfs_middle = KMeans(n_clusters=n_fuzzy_sets, n_init=5).fit(raw_mfs)

            # Storing cluster centers of the current feature
            cluster_center.append(np.sort(reduced_mfs_middle.cluster_centers_.ravel()))
            # Preparing list for mapping what rules each set is used in (index is set
number, values are lists with rule index)
            subcluster_in_clusters = [[] for i in
np.unique(reduced_mfs_middle.labels_)]
            for subcluster_number in range(len(raw_mfs)):
                # Iterate ower the raw_mfs (rule clusters)
                cur_cluster_number = reduced_mfs_middle.labels_[subcluster_number]
                # Add the feature set to the rule list dictionary.
                # Note how this is reapeated for each feature in the outer loop and in
this loop the current feature is added to all rules
                clusters[subcluster_number].append(cur_cluster_number)
                # Keep list of which subcluster are in the root clusters for the
feature
                subcluster_in_clusters[cur_cluster_number].append(subcluster_number)

            # Preparing the mfs list
            cur_mfs = []
            for cluster, subclusters in enumerate(subcluster_in_clusters):
                # For each set derived for this feature, create the fuzzy trapezodial
by calculating the quantile
                cur_mean =
merged_data[merged_data["Subcluster_number"].isin(np.unique(subclusters))].iloc[:,
k].mean()
                cur_quantile_val =
merged_data[merged_data["Subcluster_number"].isin(np.unique(subclusters))].iloc[:,
k].quantile(trap_quantile)
                dis_from_center = abs(cur_mean - cur_quantile_val)
                cur_min =
merged_data[merged_data["Subcluster_number"].isin(np.unique(subclusters))].iloc[:,
k].min()
                cur_max =
merged_data[merged_data["Subcluster_number"].isin(np.unique(subclusters))].iloc[:,
k].max()
                cur_center = cluster_center[k-1][cluster]
                cur_mfs.append(MfTrap(cur_center - (abs(cur_mean - cur_min)),
                                      cur_center - dis_from_center,
                                      cur_center + dis_from_center,
                                              cur_center + (abs(cur_mean - cur_max)),
                                              f"{merged_data.columns[k]}_{cluster}"))

            # Add the feature collection of sets
            mfs.append(FeatureFuzzySets(cur_mfs, merged_data.columns[k]))
        # Add the fuzzy sets to model
        self.set_feature_fuzzy_sets(mfs)


        consequent_train_data = []

        antecedents = []
        for rule_num, rule in clusters.items():
            cur_index_matrix = np.array([np.arange(len(rule)), rule]).transpose()
            antecedents.append(TSKAntecedent(cur_index_matrix))
        # Create the dataset to fit the consequents with least squeare method
        for row in range(merged_data.shape[0]):
            cur_x_row = merged_data.iloc[row, 1:-1].to_numpy()
            cur_row_expanded = np.array([])
            w_total = 0
            cur_w_list = []
            # Find firing strength for each rule
            for rule in antecedents:
                rule_w = rule.calculate_firing_strenght(cur_x_row,
self._feature_fuzzy_sets)
                w_total += rule_w
                cur_w_list.append(float(rule_w))

            if w_total == 0:
                w_total = 1

            for w in cur_w_list:
                cur_firing_normalized = w/w_total
                if len(cur_row_expanded) == 0:
                    cur_row_expanded =
np.concatenate(((cur_x_row*cur_firing_normalized), np.array([cur_firing_normalized])),
dtype=float)
                else:
                    cur_row_expanded = np.concatenate((cur_row_expanded,
(cur_x_row*cur_firing_normalized), np.array([cur_firing_normalized])), dtype=float)

            consequent_train_data.append(cur_row_expanded)

        cur_Y = merged_data.iloc[:, 0].to_numpy()
        cur_X = consequent_train_data
        # Estimate the consequence parameters
        linear_estimate = LinearRegression(fit_intercept=False, copy_X=True).fit(cur_X,
cur_Y)
        self.r2_scores = linear_estimate.score(cur_X, cur_Y)
        # Extract the parameters
        self.extracted_params = linear_estimate.coef_
        # The next lines partition the parameters into the rules. Each rule has
parameters and one intercept which is the last parameter (1 + the last index of the
rule)
        rule_params = [[] for i in range(len(clusters.keys()))]
        rule_intercepts = np.empty(len(clusters.keys()), dtype=float)
        param_counter = 0
        for rule_num, rule in clusters.items():
            # For each rule iterate over all features in rule variable
            for param_nr in range(len(rule)):
                rule_params[rule_num].append(self.extracted_params[param_counter +
param_nr])

            rule_intercepts[rule_num] = self.extracted_params[param_counter +
len(rule)]
            param_counter += len(rule) + 1

        self.n_fuzzy_rules = len(clusters.keys())
        new_rulebase = TSKRuleBase(self.feature_names, expressions)
        # Create rulebase
        for rule_num, rule in clusters.items():
            # Creating a matrix containing the mapping between antecedent place and the
fuzzy set index
            cur_index_matrix = np.array([np.arange(len(rule)), rule]).transpose()
            # Defining the rule
            new_rulebase.appendRule(TSKRule(cur_index_matrix,
                                    rule_params[rule_num],
                                    rule_intercepts[rule_num])
                                    )
        # Add the rulebase
        self.set_rulebase(new_rulebase)

    def create_rulebase_kmeans(self,
                               train_data : pd.DataFrame,
                               n_fuzzy_sets : int=3,
                               trap_quantile : float=0.4,
                               expressions={}) -> None:
        """This method derives the TSK rulebase by performing K-Means clustering and
Multi-Linear Regression.


        Parameters
        ----------
        train_data : pandas.DataFrame
            The training data. Dependent variable is column 1 (index 0). Independent
features follow
        n_fuzzy_sets : int, optional
            Number of fuzzy sets per independent feature, by default 3
        inner_bound_factor : int, optional
```

```
            Factor for setting the core of the trapezoidals. center +/-
outer_bound_factor*std, by default 1
        """
        # Verify that there are enough features in dataframe (feature 1 is the
target/dependent variable)
        if len(train_data.columns) < 2:
            raise Exception("The provided input data does not have a valid dimention:
dimention < 2")

        # Verfiy the given number of fuzzy sets
        if n_fuzzy_sets < 1:
            raise Exception(f"Cannot create model with a number of fuzzy sets of
{n_fuzzy_sets} for each input.")

        self.n_fuzzy_sets = n_fuzzy_sets

        X = train_data.to_numpy()
        cluster_obj = KMeans(n_clusters=n_fuzzy_sets, n_init=10)
        cluster_fit = cluster_obj.fit(X)
        # Add a feature at the end of the dataframe, containing the cluster labels
        merged_data = pd.merge(train_data, pd.DataFrame({"Subcluster_number" :
cluster_fit.labels_}), left_index=True, right_index=True)
        self.merged_dataset = merged_data
        # Resulting dataframe has target in first column, independent features in the
middle and the cluster labels in the last colummn
        # Extract the featurenames by ignoring the first and last column
        self.feature_names = merged_data.columns[1:-1]
        # Extract the target name
        self.target_name = merged_data.columns[0]
        self.max_expressions = len(self.feature_names)

        # Extracting all unique subclusters/rules
        unique_clusters = merged_data["Subcluster_number"].unique()

        # Derive the membership functions from the clusters (trapeziums)
        cluster_center : [float] = []
        mfs : [FeatureFuzzySets] = []
        # Initialization of the rule list dict
        clusters : {} = {i : [] for i in np.unique(cluster_fit.labels_)}

        for k in range(1, len(merged_data.columns)-1):
            # Iterate over all features (independent variables), starts at 1 because
first column is not an independent variable
            # k - feature index (column index)

            # Get the cluster centers of the subclusters. Note that the
cluster_fit.cluster_centers_ matrix has cluster on axis 0 and feature cluster center
along axis 1
            raw_mfs = cluster_fit.cluster_centers_[:,k]
            # Preparing list for mapping what rules each set is used in (index is set
number, values are lists with rule index)
            subcluster_in_clusters = [[] for i in range(len(raw_mfs))]
            for subcluster_number in range(len(raw_mfs)):
                # Iterate ower the raw_mfs (rule clusters)
                # Add the feature set to the rule list dictionary.
                # Note how this is reapeated for each feature in the outer loop and in
this loop the current feature is added to all rules
                clusters[subcluster_number].append(subcluster_number)
                # Keep list of which subcluster are in the root clusters for the
feature
                subcluster_in_clusters[subcluster_number].append(subcluster_number)

            # Preparing the mfs list
            cur_mfs = []
            for cluster, subclusters in enumerate(subcluster_in_clusters):
                # For each set derived for this feature, create the fuzzy trapezodial
by calculating the quantile
                cur_mean =
merged_data[merged_data["Subcluster_number"].isin(np.unique(subclusters))].iloc[:,
k].mean()
                cur_quantile_val =
merged_data[merged_data["Subcluster_number"].isin(np.unique(subclusters))].iloc[:,
k].quantile(trap_quantile)
                dis_from_center = abs(cur_mean - cur_quantile_val)
                cur_min =
merged_data[merged_data["Subcluster_number"].isin(np.unique(subclusters))].iloc[:,
k].min()
                cur_max =
merged_data[merged_data["Subcluster_number"].isin(np.unique(subclusters))].iloc[:,
k].max()
                cur_center = raw_mfs[cluster]
                cur_mfs.append(MfTrap(cur_center - (abs(cur_center - cur_min)),
                                      cur_center - dis_from_center,
                                      cur_center + dis_from_center,
                                      cur_center + (abs(cur_center - cur_max)),
                                      f"{merged_data.columns[k]}_{cluster}"))

            # Add the feature collection of sets
            mfs.append(FeatureFuzzySets(cur_mfs, merged_data.columns[k]))
        # Add the fuzzy sets to model
        self.set_feature_fuzzy_sets(mfs)


        self.r2_scores = []
        self.extracted_params = []
        rule_params = [[] for i in range(len(clusters.keys()))]
        rule_intercepts = np.empty(len(clusters.keys()), dtype=float)
        antecedents = []
        cur_y = merged_data.iloc[:, 0].to_numpy().ravel()
        for rule_num, rule in clusters.items():
            cur_index_matrix = np.array([np.arange(len(rule)), rule]).transpose()
            antecedents.append(TSKAntecedent(cur_index_matrix))
        # Create the dataset to fit the consequents with least squeare method
        for w_rule in range(len(antecedents)):
            # For each rule, calculate the updated dataset
            consequent_train_data = []
            reduced_y = []
            for row in range(merged_data.shape[0]):
                cur_x_row = merged_data.iloc[row, 1:-1].to_numpy()
                w_total = 0
                cur_w_list = []
                # Find firing strength for each rule
                for rule in antecedents:
                    rule_w = rule.calculate_firing_strenght(cur_x_row, 
self._feature_fuzzy_sets)
                    w_total += rule_w
                    cur_w_list.append(float(rule_w))

                if w_total > 0 and cur_w_list[w_rule] > 0:
                    cur_firing_normalized = cur_w_list[w_rule]/w_total
                    consequent_train_data.append(np.concatenate((cur_x_row*cur_firing_n
ormalized, np.array([cur_firing_normalized]))))
                    reduced_y.append(cur_y[row])

            cur_linear_estimate = LinearRegression(fit_intercept=False, 
copy_X=True).fit(consequent_train_data, np.array(reduced_y).reshape(-1, 1))
            self.r2_scores.append(cur_linear_estimate.score(consequent_train_data, 
np.array(reduced_y).reshape(-1, 1)))
            cur_parameters = cur_linear_estimate.coef_.ravel()
            self.extracted_params.append(cur_parameters)
            rule_params[w_rule] = cur_parameters[:-1]
            rule_intercepts[w_rule] = cur_parameters[-1]


        self.n_fuzzy_rules = len(clusters.keys())
        new_rulebase = TSKRuleBase(self.feature_names, expressions)
        # Create rulebase
        for rule_num, rule in clusters.items():
            # Creating a matrix containing the mapping between antecedent place and the 
fuzzy set index
            cur_index_matrix = np.array([np.arange(len(rule)), rule]).transpose()
            # Defining the rule
            new_rulebase.appendRule(TSKRule(cur_index_matrix,
                                    rule_params[rule_num],
                                    rule_intercepts[rule_num])
                                    )
        # Add the rulebase
        self.set_rulebase(new_rulebase)
```

```
    def test_model(self, test_data):
        self.test_data = test_data
        # Calculate the output values
        cur_actuals = np.empty(test_data.shape[0])
        for i, x_vals in test_data.iloc[:, 1:].iterrows():
            cur_actuals[i] = self.calculate_output(x_vals.to_numpy())

        self.test_actuals = cur_actuals

    def calculate_r_squared(self, test_data=None):
        # This method calculates the r-squared: 1 - RSS/TSS
        self.r_squared = 0
        if len(self.test_actuals) == 0 and len(test_data) > 0:
            # If the actuals have not been calculated, apply the
            self.test_data = test_data
            self.test_model(test_data)
        elif len(self.test_actuals) == 0:
            raise Exception("No test data is available, please provide test data!")

        self.test_model(test_data)
        # Total sum of squares (denominator)
        y = self.test_data.iloc[:, 0].to_numpy()
        y_mean = np.mean(self.test_actuals)
        tss = np.sum((y - y_mean)**2)
        rss = np.sum((y - self.test_actuals)**2)

        self.r_squared = 1 - (rss/tss)
        return self.r_squared

    def calculate_rmse(self, test_data=pd.DataFrame()):
        # This method calculates the r-squared: 1 - RSS/TSS
        self.rmse = 0
        if len(self.test_actuals) == 0 and len(test_data) > 0:
            # If the actuals have not been calculated, apply the
            self.test_data = test_data
            self.test_model(test_data)
        elif len(self.test_actuals) == 0:
            raise Exception("No test data is available, please provide test data!")

        self.test_model(test_data)
        # Total sum of squares (denominator)
        y = self.test_data.iloc[:, 0].to_numpy()

        rmse = np.sqrt(np.sum((y - self.test_actuals)**2)/len(y))

        self.rmse = rmse
        return self.rmse

    def increment_training_counter(self):
        self.generation += 1
    def get_training_counter(self):
        return self.generation

    def show_fuzzy_sets(self):
        """This method is used to plot the fuzzy sets for each feature"""
        fig, axes = plt.subplots(len(self.feature_names), 1, figsize=(10, 10))
        if len(self.feature_names) == 1:
            axes = [axes]

        fig.tight_layout(pad=5)
        for id, feature in enumerate(self.feature_names):
            axes[id].set_title(f"fuzzy sets for {feature}")
            axes[id].set_xlabel(feature)
            axes[id].set_ylabel("membership degree")
            for cur_set in self._feature_fuzzy_sets[id].fuzzy_sets:
                cur_param_list = cur_set.get_param_list()

                if cur_param_list[0] == -np.infty:
                    # Add display boundaries
                    cur_param_list[0] = cur_param_list[2] - 2
                    cur_param_list[1] = cur_param_list[2] - 1

                if cur_param_list[2] == np.infty:
                    # Add display boundaries
                    cur_param_list[2] = cur_param_list[1] + 2
                    cur_param_list[3] = cur_param_list[1] + 1

                deg_list = []
                for param in cur_param_list:
                    deg_list.append(cur_set.get_membership_degree(param))

                axes[id].plot(cur_param_list, deg_list, label=cur_set.name)

            axes[id].legend()

        plt.show()


if __name__ == '__main__':
    # Test features
    test_dict = {0 : (0, 6), 1 : (0, 6)}
    test_names = ["A", "B"]
    test_tsk_model = TSKModel()
    #test_tsk_model.randomize_model(4, 4*4, test_dict, test_names, max_expressions=2)

    cur_feature_sets_A = []
    cur_feature_sets_A.append(MfTrap(-np.infty, -np.infty, 0, 1, f"Very small"))
    cur_feature_sets_A.append(MfTrap(1, 2, 2, 3, f"Small"))
    cur_feature_sets_A.append(MfTrap(3, 4, 4, 5, f"Large"))
    cur_feature_sets_A.append(MfTrap(5, 6, np.infty, np.infty, f"Very large"))

    cur_feature_sets_B = []
    cur_feature_sets_B.append(MfTrap(-np.infty, -np.infty, 0, 1, f"Very small"))
    cur_feature_sets_B.append(MfTrap(1, 2, 2, 3, f"Small"))
    cur_feature_sets_B.append(MfTrap(3, 4, 4, 5, f"Large"))
    cur_feature_sets_B.append(MfTrap(5, 6, np.infty, np.infty, f"Very large"))

    feature_fuzzy_sets = {0 : FeatureFuzzySets(cur_feature_sets_A, "A"), 1 :
FeatureFuzzySets(cur_feature_sets_B, "B")}
    test_tsk_model.set_feature_fuzzy_sets(feature_fuzzy_sets)

    new_rulebase = TSKRuleBase(test_names, {"A" : "test1", "B" : "test2"})
    possible_params = [[1,1],[0,0], [1,1], [0,0]]
    possible_r = [0,2,-1,9]
    for i in range(4):
        for j in range(4):
            cur_params = possible_params[max(i, j)]
            cur_r = possible_r[max(i, j)]
            cur_feature_mapping = np.array([[0, i], [1, j]])
            new_rulebase.appendRule(TSKRule(cur_feature_mapping, cur_params, cur_r))

    #new_rulebase.appendRule(TSKRule(np.array([[0, 0], [1, 0]]), [1, 1], 0))
    #new_rulebase.appendRule(TSKRule(np.array([[0, 1], [1, 1]]), [0, 0], 1))
    #new_rulebase.appendRule(TSKRule(np.array([[0, 2], [1, 2]]), [1, 1], -2))
    #new_rulebase.appendRule(TSKRule(np.array([[0, 3], [1, 3]]), [0, 0], 3))

    test_tsk_model.set_rulebase(new_rulebase)

    #test_tsk_model.show_fuzzy_sets()

    X = np.linspace(0,6, 100)
    Y = np.linspace(0, 6, 100)
    X, Y = np.meshgrid(X, Y)
    X_flat, Y_flat = X.ravel(), Y.ravel()
    Z = np.empty(100*100)
    for i, x, y in zip(np.arange(len(X_flat)), X_flat, Y_flat):
        Z[i] = test_tsk_model.calculate_output([x, y])


    fig, ax = plt.subplots(subplot_kw={"projection": "3d"})
    ax.plot_surface(X, Y, Z.reshape(X.shape), cmap=cm.coolwarm,
                       linewidth=0, antialiased=False)
    plt.show()
```

Code from ba_optimization.py:

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib import cm
from sklearn.cluster import KMeans
from sklearn.linear_model import LinearRegression
import copy
import math

from tsk_model import TSKModel


# Representation? - 2 fold - one array for the sets, one for the consequence parameters

class BeesFitTSK():

    def __init__(self,
                 n_scout_bees,
                 n_best_sites,
                 n_elite_sites,
                 n_workers_best,
                 n_workers_elite,
                 n_neighborhod_size,
                 slimit,
                 training_data, test_data,
                 activate_debuging=False):
        # Initialize the swarm algorithm with a population and the parameters
        self.n_scout_bees = n_scout_bees
        self.n_best_sites = n_best_sites
        self.n_elite_sites = n_elite_sites
        self.n_workers_best = n_workers_best
        self.n_workers_elite = n_workers_elite
        self.n_neighborhod_size = n_neighborhod_size
        self.slimit = slimit
        self.train_data = training_data
        self.test_data = test_data
        self.activate_debuging = activate_debuging
        self.gen_counter = 0
        self.training_log = [[],[],[]] # gen_number, training_rmse, test_rmse
        # Calculate the mutation increment for each feature
        self.mutation_increments = ((self.train_data.iloc[:,1:].max() -
self.train_data.iloc[:,1:].min()) / 100).to_dict()
        # Initialize the population
        self.bee_population = np.empty(n_scout_bees,dtype=object)

    def initialize_population(self, n_rules=20, n_fuzzy_sets=15, trap_quantile=0.5,
expressions={}, random_init=False):
        """Initialization of the population is done acording to the input parameters.
        Parameters
        ----------
        n_rules : int, optional
            The number of rules for TSK model, by default 20
        n_fuzzy_sets : int, optional
            Number of fuzzy sets in the TSK model, by default 15
        trap_quantile : float, optional
            The quantile that indicate the boundaries of the core for each fuzzy set,
by default 0.5
        expressions : dict, optional
            The expression that is used to describe the input variables, by default {}
        random_init : bool, optional
            Flag for activating random initialization (quantile and number of sets are
randomized), by default False
        """
        self.n_rules = n_rules
        self.n_fuzzy_sets = n_fuzzy_sets
        self.expressions = expressions
        self.trap_quantile = trap_quantile
        self.random_init = random_init
        for i in range(len(self.bee_population)):
            # For each solution in population, initialize model with kmeans
            cur_tsk_model = TSKModel()
            cur_tsk_model.create_rulebase_kmeans(self.train_data,
                                                 #n_rules=n_rules,
                                                 n_fuzzy_sets= self.n_fuzzy_sets if not
self.random_init else np.random.choice(np.arange(1, self.n_fuzzy_sets)),
                                                 expressions=self.expressions,
                                                 trap_quantile=np.random.uniform(0.1,
0.7))

            self.bee_population[i] = cur_tsk_model
            # Evaluate fitness
            self.bee_population[i].calculate_rmse(self.train_data)
        # Sort population on fitness
        self.sort_population()
        self.all_time_best_model = self.bee_population[0]

    def sort_population(self):
        self.bee_population = sorted(self.bee_population, key=lambda x: x.rmse)
        if self.activate_debuging:
            print("Fitness in population")
            for i, site in enumerate(self.bee_population):
                print(f"Position {i}: {site.rmse}")

    def main_loop(self):
        # Select the scouts for elite sites and perform local search
        for i, elite_site in enumerate(self.bee_population[:self.n_elite_sites]):
            if self.activate_debuging: print(f"Work on model {i} (elite)")
            self.bee_population[i] = self.site_exploitation(elite_site,
self.n_workers_elite, self.n_neighborhod_size)

        for i, remaining_best_site in
enumerate(self.bee_population[self.n_elite_sites:self.n_best_sites]):
            if self.activate_debuging: print(f"Work on model {self.n_elite_sites + i}
(remaining best)")
            self.bee_population[self.n_elite_sites + i] =
self.site_exploitation(remaining_best_site, self.n_workers_best,
self.n_neighborhod_size)

        for i, abandoned_site in enumerate(self.bee_population[self.n_best_sites:]):
            if self.activate_debuging: print(f"Work on model {self.n_best_sites + i}
(abandoned)")
            cur_tsk_model = TSKModel()
            cur_tsk_model.create_rulebase_kmeans(self.train_data,
                                                  n_fuzzy_sets=self.n_fuzzy_sets if not
self.random_init else np.random.choice(np.arange(1, self.n_fuzzy_sets)),
                                                  expressions=self.expressions,
                                                  trap_quantile=np.random.uniform(0.1,
0.7))
            cur_tsk_model.calculate_rmse(self.train_data)
            self.bee_population[self.n_best_sites + i] = cur_tsk_model

        self.sort_population()
        # Check if there is a new best site
        if self.all_time_best_model.rmse > self.bee_population[0].rmse:
            self.all_time_best_model = copy.deepcopy(self.bee_population[0])
            print(f"Current best fitness: {self.all_time_best_model.rmse} generation
{self.gen_counter}")

        self.gen_counter += 1
        self.training_log[0].append(self.gen_counter)
        self.training_log[1].append(self.all_time_best_model.rmse)
        self.training_log[2].append(self.all_time_best_model.calculate_rmse(self.test_d
ata))
        return self.all_time_best_model.rmse

    def update_set_param(self, old_param_val, increment, previous_parameter,
next_parameter):
        # This method updates the set parameter
        new_param_value = old_param_val + np.random.normal(loc=0, scale=1)*increment
        if new_param_value < previous_parameter:
            return previous_parameter
        elif new_param_value > next_parameter:
            return next_parameter
        else:
            return new_param_value

    def update_consequence_param(self, old_param, shrinking_factor):
        # This method updates the coeficient in the consequence
        return old_param + np.random.normal(0, 1)*old_param*shrinking_factor

    def site_exploitation(self, site : TSKModel, n_followers, n_changes:int):
        cur_best = site
        improvement_falg = False
        n_changes = math.ceil(n_changes * site.get_training_counter() / self.slimit)
        shrinking_factor = 1 - (site.get_training_counter()/self.slimit)
        set_param_size = len(site._feature_fuzzy_sets)
        for worker in range(n_followers):
            # Apply change to the current fuzzy sets
            cur_copy:TSKModel = copy.deepcopy(site)
            for neighborhod_change in range(n_changes):
                # Iterate over the number of changes that should be made to the site
                if np.random.randint(2):
                    # Change consequent of anticedent
                    selected_rules = np.random.randint(len(site.rulebase.rules))
                    selected_param =
np.random.randint(len(cur_copy.rulebase.rules[selected_rules].consequent.params_list)+1
)

                    if selected_param ==
len(cur_copy.rulebase.rules[selected_rules].consequent.params_list):
                        old_param =
cur_copy.rulebase.rules[selected_rules].consequent.const
                        cur_copy.rulebase.rules[selected_rules].consequent.const =
self.update_consequence_param(old_param, shrinking_factor)
                    else:
                        old_param =
cur_copy.rulebase.rules[selected_rules].consequent.params_list[selected_param]
                        cur_copy.rulebase.rules[selected_rules].consequent.params_list[
selected_param] = self.update_consequence_param(old_param, shrinking_factor)
                else:
                    set_to_modify = np.random.randint(set_param_size)
                    subset_to_modify =
np.random.randint(len(cur_copy._feature_fuzzy_sets[set_to_modify].fuzzy_sets))
                    param_to_modify = np.random.randint(4)
                    # Get the current param list of the selected set
                    cur_param_list =
cur_copy._feature_fuzzy_sets[set_to_modify].fuzzy_sets[subset_to_modify].get_param_list
()
                    # Get the selected param value
                    cur_selected_param = cur_param_list[param_to_modify]
                    if param_to_modify == 0:
                        next_param = cur_param_list[param_to_modify+1]
                        previous_param = -np.inf
                    elif param_to_modify == 3:
                        previous_param = cur_param_list[param_to_modify-1]
                        next_param = np.inf
                    else:
                        next_param = cur_param_list[param_to_modify+1]
                        previous_param = cur_param_list[param_to_modify-1]

                    # Get the current value of the parameter to be changed
                    cur_feature_name =
cur_copy._feature_fuzzy_sets[set_to_modify]._feature_name
                    cur_param_list[param_to_modify] = self.update_set_param(
                                                                                cur_sel
ected_param,
                                                                                self.mu
tation_increments[cur_feature_name],
                                                                                previou
s_param,
                                                                                next_pa
ram
                                                                            )

                    cur_copy._feature_fuzzy_sets[set_to_modify].fuzzy_sets[subset_to_mo
dify].set_param_list(cur_param_list)


            # Calculate fitness of the worker
            cur_fitness = cur_copy.calculate_rmse(self.train_data)
            if cur_fitness < self.all_time_best_model.rmse and self.activate_debuging:
                if self.activate_debuging: print("Found fitness " + str(cur_fitness) +
" while site rmse is " + str(site.rmse))
            if cur_fitness < site.rmse:
                cur_best = cur_copy
                improvement_falg = True

        if not improvement_falg:
            cur_best.increment_training_counter()

        if not improvement_falg and self.slimit <= cur_best.get_training_counter():
            # abandon the site if it has not been improved for n number of generations
            if self.activate_debuging: print("Abandoning site")
            cur_tsk_model = TSKModel()
            cur_tsk_model.create_rulebase_kmeans(self.train_data,
                                                 #n_rules=self.n_rules,
                                                 n_fuzzy_sets=self.n_fuzzy_sets if not
self.random_init else np.random.choice(np.arange(1, self.n_fuzzy_sets)),
                                                 expressions=self.expressions,
                                                 trap_quantile=np.random.uniform(0.1,
0.7))

            cur_tsk_model.calculate_rmse(self.train_data)
            return cur_tsk_model

        return cur_best
```

```
if __name__=="__main__":
    expressions = {"P":"Precipitation", "E":"Potential evapotranspiration",
"PB":"Precipation balance", "Tave":"Tave"}
    selected_features = ["Relative_yield_change", "Tave", "Tmax", "Tmin"]
    train_data = pd.read_csv("dataset/matlab_1_train.csv")[selected_features]
    test_data = pd.read_csv("dataset/matlab_1_test.csv")[selected_features]
    training_model:BeesFitTSK = BeesFitTSK(8, 5, 3, 20, 50, 10, 1000, train_data,
test_data, activate_debuging=True)
    training_model.initialize_population(10, 3, trap_quantile=0.4)

    for i in range (10000):
        training_model.main_loop()
```

Code from the evaluation notebook:
# Evaluation of the TSK model
In this notebook the tsk models are tested agains the train and test set from preliminary_analysis.ipynb

```
 import pandas as pd
 import numpy as np
 import matplotlib.pyplot as plt
 from tsk_model import TSKModel
 from pyfume import *
 import time
 expressions = {"P":"Precipitation", "E":"Potential evapotranspiration",
"PB":"Precipation balance", "Tave":"Tave"}
 #selected_features_custom = ["Relative_yield_change", "Latitude", "Tmax", "Tmin",
"Longitude"]
 #selected_features_pyfume = ["Latitude", "Tmax", "Tmin", "Longitude",
"Relative_yield_change"]
 #selected_features = ['Relative_yield_change','yield_of_CT', 'Latitude', 'Longitude',
 #        'Years_since_CT_started', 'Crop_rotation_CT', 'Crop_rotation_NT', 'ST',
 #        'Soil_cover_in_CT', 'Soil_cover_in_CT', 'Weed_pest_control_CT',
 #        'Weed_pest_control_NT', 'P', 'E', 'PB', 'Tave', 'Tmax', 'Tmin']

 # Note that we need copies of the datasets since the two models require different
column ordering.
 selected_features_custom = ["Relative_yield_change", 'yield_of_CT', "Tmax", "Tmin"]
 selected_features_pyfume = ['yield_of_CT', "Tmax", "Tmin", "Relative_yield_change"]
 train_data_custom =
pd.read_csv("dataset/matlab_1_train.csv")[selected_features_custom]
 test_data_custom = pd.read_csv("dataset/matlab_1_test.csv")[selected_features_custom]
 train_data_pyfume =
pd.read_csv("dataset/matlab_1_train.csv")[selected_features_pyfume]
 test_data_pyfume = pd.read_csv("dataset/matlab_1_test.csv")[selected_features_pyfume]
```

Definition of test parameters

```
 cluster_numbers = [2, 3, 4, 5, 6, 7, 8, 9, 10]
 quantile_numbers = [0.2, 0.4, 0.6]
 n_runs = 20
 test_data_pyfume
```

The choice of different trap_quantile parameter value has shown litle effect on the performance of the models. Below we plot four different configurations to see that it actualy has an effect on the fuzzy membership functions.

```
 cur_tsk_model = TSKModel()
 cur_tsk_model.create_rulebase_kmeans(train_data_custom, n_fuzzy_sets=3,
expressions=expressions, trap_quantile=0.2)
 cur_tsk_model.show_fuzzy_sets()
```

```
 cur_tsk_model = TSKModel()
 cur_tsk_model.create_rulebase_kmeans(train_data_custom, n_fuzzy_sets=3,
expressions=expressions, trap_quantile=0.4)
 cur_tsk_model.show_fuzzy_sets()
```

```
 cur_tsk_model = TSKModel()
 cur_tsk_model.create_rulebase_kmeans(train_data_custom, n_fuzzy_sets=3,
expressions=expressions, trap_quantile=0.6)
 cur_tsk_model.show_fuzzy_sets()
```

```
 cur_tsk_model.rulebase.print_rulebase(cur_tsk_model._feature_fuzzy_sets)
```

The above plots clearly show that the core is greater for smaler trap_quantile parameters.
Next the test runs are performed. Warning: this operation might take several minutes.

```
 cluster_list_pyfume_training = [[] for n_sets in cluster_numbers]
 cluster_list_pyfume_test = [[] for n_sets in cluster_numbers]
 pyfume_timer = time.time()
 for run in range(n_runs):
      print(f"Starting run {run}")
      cur_clusters_pyfume = {}
      cur_clusters_custom = {}
      for i, n_fuzzy_sets in enumerate(cluster_numbers):
           print(f"n_sets = {n_fuzzy_sets}")
           FIS = pyFUME(dataframe=train_data_pyfume, nr_clus=n_fuzzy_sets)
           cur_model = FIS.get_model()
           cluster_list_pyfume_training[i].append(FIS.calculate_error(method="RMSE"))
```

```
            test = SugenoFISTester(model=cur_model, test_data=test_data_pyfume.iloc[:,:-
1].to_numpy(),
                                   variable_names=np.array(selected_features_pyfume[:-
1], dtype='<U22'),
                                   golden_standard=test_data_pyfume.iloc[:,-
1].to_numpy())
            cluster_list_pyfume_test[i].append(test.calculate_RMSE())

 pyfume_timer = time.time() - pyfume_timer
 cluster_list_custom_training = [[] for n_sets in cluster_numbers]
 cluster_list_custom_test = [[] for n_sets in cluster_numbers]
 for quantile_number in quantile_numbers:
      for i in range(len(cluster_numbers)):
            cluster_list_custom_training[i].append([])
            cluster_list_custom_test[i].append([])
```

```
 custom_timer = time.time()
 for run in range(n_runs):
      print(f"Starting run {run}")
      cur_clusters_pyfume = {}
      cur_clusters_custom = {}
      for i, n_fuzzy_sets in enumerate(cluster_numbers):
            print(f"n_sets = {n_fuzzy_sets}")
            for j, quantile_number in enumerate(quantile_numbers):
                  cur_tsk_model = TSKModel()
                  cur_tsk_model.create_rulebase_kmeans(train_data_custom,
n_fuzzy_sets=n_fuzzy_sets, expressions=expressions, trap_quantile=quantile_number)

cluster_list_custom_training[i][j].append(cur_tsk_model.calculate_rmse(train_data_custo
m))

cluster_list_custom_test[i][j].append(cur_tsk_model.calculate_rmse(test_data_custom))

 custom_timer = time.time() - custom_timer
```

```
 np.array(cluster_list_custom_training).shape
```

```
 np.array(cluster_list_custom_test).shape
```

```
 np.array(cluster_list_pyfume_training).shape
```

```
 np.array(cluster_list_pyfume_test).shape
```

The model results are aggregated and cleaned up

```
log = {
    ("labels","n_sets") : [],
    ("labels", "statistic_measures",) : [],
    ("pyFume", "train_pyFume") : [],
    ("pyFume", "test_pyFume") : [],
    }
for q in quantile_numbers:
    log[("test_custom", f"q_{q}")] = []

for q in quantile_numbers:
    log[("train_custom", f"q_{q}")] = []

for i, cluster_number in enumerate(cluster_numbers):
    for name, measure in {"Min" : np.min, "Max" : np.max, "Std" : np.std, "Mean" :
np.mean}.items():
        log[("labels", "n_sets")].append(cluster_number)
        log[("labels", "statistic_measures")].append(name)
        log[("pyFume",
"train_pyFume")].append(measure(cluster_list_pyfume_training[i]))
        log[("pyFume", "test_pyFume")].append(measure(cluster_list_pyfume_test[i]))
        for j, q in enumerate(quantile_numbers):
            log[("train_custom",
f"q_{q}")].append(measure(cluster_list_custom_training[i][j]))
            log[("test_custom",
f"q_{q}")].append(measure(cluster_list_custom_test[i][j]))
```

Printing results

```
results = pd.DataFrame(log)
print(f"Runtime for custom implementation: {custom_timer:.4f} s")
print(f"Runtime for pyFume implementation: {pyfume_timer:.4f} s")
print(f"Number of runs per parameterset: {n_runs}")
```

```
results[["pyFume", "test_custom",
"train_custom"]].set_index(pd.MultiIndex.from_frame(results["labels"]))
results[["pyFume", "test_custom",
"train_custom"]].set_index(pd.MultiIndex.from_frame(results["labels"])).to_csv("first_r
esults.csv", float_format="%.5f")
results.to_csv("first_results_raw.csv")
```

```
results[["pyFume", "test_custom", "train_custom"]].columns.to_numpy()
fig, axes = plt.subplots(2,2, figsize=(16, 16))
measure_list = results["labels"]["statistic_measures"].unique()
for i, measure_label in enumerate(measure_list):
    cur_axis = axes[int(i/(len(measure_list)/2)),int(i%(len(measure_list)/2))]
    cur_axis.set_title(f"{measure_label} RMSE for a given number of sets")
    cur_axis.set_xlabel(f"Number of fuzzy sets")
    cur_axis.set_ylabel(f"{measure_label} RMSE")
    for parent, column in results[["pyFume", "test_custom", 
"train_custom"]].columns.to_numpy():
        cur_y = results[results["labels"]["statistic_measures"] == 
measure_label][parent][column]
        cur_x = results[results["labels"]["statistic_measures"] == 
measure_label]["labels"]["n_sets"]
        cur_axis.plot(cur_x, cur_y, label=f"{parent}_{column}")

    cur_axis.legend()
```

Code from the test_ba_optimization.ipynb. Note that this is a notebook with text cells. Please select the code cells carefully when trying to run the code:

```
## Evaluating and testing the BA algorithm for tuning TSK
from ba_optimization import BeesFitTSK
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
import copy
expressions = {"P":"Precipitation", "E":"Potential evapotranspiration", "PB":"Precipation balance", "Tave":"Tave"}
#selected_features = ["Relative_yield_change", "Tave", "Tmax", "Tmin"]
#selected_features = ['Relative_yield_change','yield_of_CT', 'Latitude', 'Longitude',
#       'Years_since_CT_started', 'Crop_rotation_CT', 'Crop_rotation_NT', 'ST',
#       'Soil_cover_in_CT', 'Soil_cover_in_CT', 'Weed_pest_control_CT',
#       'Weed_pest_control_NT', 'P', 'E', 'PB', 'Tave', 'Tmax', 'Tmin']
selected_features = ["Relative_yield_change", 'yield_of_CT', "Tmax", "Tmin"]
train_data = pd.read_csv("dataset/matlab_1_train.csv")[selected_features]
test_data = pd.read_csv("dataset/matlab_1_test.csv")[selected_features]


training_model:BeesFitTSK = BeesFitTSK(5, 2, 2, 30, 10, 3, 100, train_data, test_data)
training_model.initialize_population(3, 3, trap_quantile=0.4)

for i in range (3):
    training_model.main_loop()
Plotting the optimization development
run_n = 1
plt.plot(training_model.training_log[0], training_model.training_log[1], label=f"Run {run_n}")
plt.plot(training_model.training_log[0], training_model.training_log[2], label=f"Run {run_n}")
training_model.all_time_best_model._feature_fuzzy_sets[0].fuzzy_sets[0].get_param_list()
training_model.all_time_best_model.show_fuzzy_sets()
training_model.all_time_best_model.rulebase.print_rulebase(training_model.all_time_best_model._feature_fuzzy_sets)
training_model.all_time_best_model.extracted_params
training_model.all_time_best_model.r2_scores
intercepts = [-3.561233e-01, -3.798248e-01, -9.774359e-01]
training_model.all_time_best_model.rulebase.rules[0].consequent.params_list = [4.588443e-02, -1.339733e-02, -1.892194e-
02]
training_model.all_time_best_model.rulebase.rules[1].consequent.params_list = [4.569108e-03, 9.736062e-03,-7.339191e-
04]
training_model.all_time_best_model.rulebase.rules[2].consequent.params_list = [5.483614e-02, 2.399846e-03, -2.256654e-
02]
training_model.all_time_best_model.rulebase.rules[0].consequent.const = intercepts[0]
training_model.all_time_best_model.rulebase.rules[1].consequent.const = intercepts[1]
training_model.all_time_best_model.rulebase.rules[2].consequent.const = intercepts[2]
training_model.all_time_best_model.rulebase.rules.append(copy.deepcopy(training_model.all_time_best_model.rulebase.rule
s[0]))
```

```
training_model.all_time_best_model.rulebase.rules.append(copy.deepcopy(training_model.all_time_best_model.rulebase.rule
s[1]))
training_model.all_time_best_model.rulebase.rules.append(copy.deepcopy(training_model.all_time_best_model.rulebase.rule
s[2]))
print(training_model.all_time_best_model.rulebase.rules)
training_model.all_time_best_model.calculate_rmse()
training_model.all_time_best_model.rulebase.print_rulebase(training_model.all_time_best_model._feature_fuzzy_sets)
```
The measured error is not good, indicating that the model underfits and there could be some bug in the code.
```
test_series = pd.Series(abs(train_data.iloc[:, 0].to_numpy() - training_model.all_time_best_model.test_actuals))
test_series
training_model.all_time_best_model.test_actuals[605]
train_data.iloc[605,:]
```
Running more thorough testing
```
n_runs = 5
import time

time_list = []
log_list = {}
fig, ax = plt.subplots(1,2,figsize=(20, 10))
for run in range(1,n_runs+1):
    start_time = time.time()
    training_model:BeesFitTSK = BeesFitTSK(5, 2, 2, 10, 30, 10, 1000, train_data, test_data)
    training_model.initialize_population(3, 3, trap_quantile=0.4)

    for i in range (100):
        training_model.main_loop()

    print(f"Finish time: {time.time() - start_time:.3f}")
    time_list.append(time.time() - start_time)
    ax[0].plot(training_model.training_log[0], training_model.training_log[1], label=f"Run {run} training")
    ax[1].plot(training_model.training_log[0], training_model.training_log[2], label=f"Run {run} test")
    log_list[(f"Run{run}", "gen")] = training_model.training_log[0]
    log_list[(f"Run{run}", "train")] = training_model.training_log[1]
    log_list[(f"Run{run}", "test")] = training_model.training_log[2]

ax[0].legend()
ax[1].legend()
ax[0].set_xlabel("Generations")
ax[0].set_ylabel("RMSE")
ax[0].set_title(f"BA optimization development training set")
ax[1].set_xlabel("Generations")
ax[1].set_ylabel("RMSE")
ax[1].set_title(f"BA optimization development test set")

print(f"Mean runtime: {np.mean(time_list):.3f} s")

pd.DataFrame(log_list).to_csv("BA_log_2.csv")
```
Trying with a simple aproximation
```
def benchmark_funciton(x):
    if x < 10:
        return 5.0
    if x < 20:
        return x*0.5
    if x < 30:
        return 10.0
    if x < 50:
        return 17.44 + x*-0.25
    return 5
test_x = np.linspace(0, 60, 100)
vec_func = np.vectorize(benchmark_funciton)
cur_y = vec_func(test_x)
plt.plot(test_x, cur_y)
```

train_x = np.random.choice(np.linspace(0, 60, 1000), 1000, replace=True)
train_y = vec_func(train_x)
plt.scatter(train_x, train_y)
train_df = pd.DataFrame({"X" : train_x, "Y" : train_y})
training_model2:BeesFitTSK = BeesFitTSK(7, 5, 2, 20, 10, 1, 100, train_df, train_df)
training_model2.initialize_population(4, 4)

for i in range (10):
    training_model2.main_loop()
training_model2.all_time_best_model.show_fuzzy_sets()
first = training_model2.all_time_best_model.show_fuzzy_sets()

Code from preliminary_data_analysis.ipynb. Note that this is a notebook with text cells. Please select the code cells carefully when trying to run the code:

# Data cleaning and feature engineering
This notebook contain the code used to prepare the dataset for analysis
import pandas as pd

```
from sklearn.model_selection import train_test_split
```

Importing and defining the features needed for the project

```
main_df = pd.read_csv("dataset/Database.csv")
variables_used_by_ryo = ["Relative yield change",
                         "Crop","Yield of CT",
                         "Latitude", "Longitude",
                         "Years since NT started (yrs)",
                         "Crop rotation with at least 3 crops involved in CT",
                         "Crop rotation with at least 3 crops involved in NT",
                         "ST", "Soil cover in CT", "Soil cover in NT",
                         "Weed and pest control CT", "Weed and pest control NT ",
                         "P", "E", "PB","Tave", "Tmax", "Tmin"]
main_df.columns
```

The features used by Ryo are extracted. Note that we are only interested in the columns for maize

```
ryo_df = main_df[variables_used_by_ryo]
ryo_df = ryo_df[ryo_df["Crop"] == "maize"]
```

Ryo also removed extreme values in the target variable by removing all values over the quantile=0.975

```
print(ryo_df.shape)
quantile = ryo_df["Relative yield change"].quantile(0.975)
mean = ryo_df["Relative yield change"].mean()
deviation = abs(mean - quantile)
removed_df = ryo_df[(ryo_df["Relative yield change"] >= quantile) | (ryo_df["Relative
yield change"] <= (mean - deviation))]
ryo_df = ryo_df[(ryo_df["Relative yield change"] < quantile) & (ryo_df["Relative yield
change"] > (mean - deviation))]
print(ryo_df.shape)
```

There are 56 rows that are removed because they are regarded as outliers.

```
removed_df.shape
removed_df["Relative yield change"]
```

```
 The variables used by Ryo are 17. But with crop and target, the number of features
should be 19:
 len(variables_used_by_ryo)
```

Create a basic summary of the features with pandas

```
 ryo_df.describe()
 ryo_df
```

We see that there are 3 columns containing nan values, that we need to deal with. We also need to convert the categorical features to contain numeric values.

```
 ryo_df[ryo_df.columns[ryo_df.dtypes == object]].describe()
 Change categories
 unique_values = ryo_df["Crop rotation with at least 3 crops involved in CT"].unique()
 mapping_for_Crop_rotation = {}
 for i, val in enumerate(unique_values):
     mapping_for_Crop_rotation[val] = i

 ryo_df["Crop rotation with at least 3 crops involved in
CT"].replace(list(mapping_for_Crop_rotation.keys()),
list(mapping_for_Crop_rotation.values()), inplace=True)
 ryo_df["Crop rotation with at least 3 crops involved in
NT"].replace(list(mapping_for_Crop_rotation.keys()),
list(mapping_for_Crop_rotation.values()), inplace=True)
 ryo_df["Weed and pest control CT"].replace(list(mapping_for_Crop_rotation.keys()),
list(mapping_for_Crop_rotation.values()), inplace=True)
 ryo_df["Weed and pest control NT "].replace(list(mapping_for_Crop_rotation.keys()),
list(mapping_for_Crop_rotation.values()), inplace=True)
 mapping_for_Crop_rotation
 unique_values = ryo_df["ST"].unique()
 mapping_for_ST = {}
 for i, val in enumerate(unique_values):
     mapping_for_ST[val] = i

 ryo_df["ST"].replace(list(mapping_for_ST.keys()), list(mapping_for_ST.values()),
inplace=True)
 mapping_for_ST

 unique_values = ryo_df[~ryo_df["Soil cover in CT"].isnull()]["Soil cover in CT"]
 mapping_for_soil = {}
 for i, val in enumerate(unique_values):
     mapping_for_soil[val] = i

 ryo_df["Soil cover in CT"].replace(list(mapping_for_soil.keys()),
list(mapping_for_soil.values()), inplace=True)
 ryo_df["Soil cover in NT"].replace(list(mapping_for_soil.keys()),
list(mapping_for_soil.values()), inplace=True)
 mapping_for_soil
 ryo_df[ryo_df.columns[ryo_df.dtypes == object]].describe()
 ryo_df.shape
```

Removing rows with null values in any of the columns
Rows to remove

```
 ryo_df[ryo_df.isnull().any(axis=1)]
 ryo_df[~ryo_df.isnull().apply(lambda x: any(x), axis=1)].isnull().any()
 Remove the rows containing null values
 cleaned_df = ryo_df[~ryo_df.isnull().apply(lambda x: any(x),
axis=1)].rename(columns={"Relative yield change" : "Relative_yield_change",

"Yield of CT" : "yield_of_CT",
"Years since NT started (yrs)" : "Years_since_CT_started",
"Crop rotation with at least 3 crops involved in CT" : "Crop_rotation_CT",
"Crop rotation with at least 3 crops involved in NT" : "Crop_rotation_NT",
"Soil cover in CT" : "Soil_cover_in_CT",
"Soil cover in NT" : "Soil_cover_in_CT",
"Weed and pest control CT" : "Weed_pest_control_CT",
"Weed and pest control NT " : "Weed_pest_control_NT"}, inplace=False)
 cleaned_df
```

Preparing data for matlab

```
 cleaned_df.columns
 matlab_dataset = cleaned_df[['Relative_yield_change','yield_of_CT', 'Latitude',
'Longitude',
        'Years_since_CT_started', 'Crop_rotation_CT', 'Crop_rotation_NT', 'ST',
        'Soil_cover_in_CT', 'Soil_cover_in_CT', 'Weed_pest_control_CT',
        'Weed_pest_control_NT', 'P', 'E', 'PB', 'Tave', 'Tmax', 'Tmin']]
 matlab_train_df, matlab_test_df = train_test_split(matlab_dataset, test_size=0.2,
random_state=10, shuffle=True)
 matlab_train_df.to_csv("dataset/matlab_1_train.csv")
 matlab_test_df.to_csv("dataset/matlab_1_test.csv")
```

Code from the CI_rf_mm.ipynb file. Note that this is a notebook with text cells. Please select the code cells carefully when trying to run the code:

```
 import pandas as pd
 df = pd.read_csv('DataFrame.csv')
 selected_columns = ['Yield of CT', 'Latitude', 'Longitude', 'Years since NT started
(yrs)',
                     'Crop rotation with at least 3 crops involved in CT',
                     'Crop rotation with at least 3 crops involved in NT', 'ST',
                     'Soil cover in CT', 'Soil cover in NT', 'Weed and pest control
CT',
                     'P', 'E', 'PB', 'Tave', 'Tmax', 'Tmin', 'Crop']
```

```
 df = df[selected_columns]

 df = df[df['Crop'] == 'maize']
```

```
columns_to_drop = ['ST_encoded','ST','Soil cover in CT','Soil cover in NT', 'Crop']
df=df.drop(columns = columns_to_drop)
df.info()
# Check for missing values in the entire DataFrame
missing_values = df.isna().sum()

# Display the count of missing values for each column
print("Missing Values:")
print(missing_values)
```

```
# Check if there are any missing values in the entire DataFrame
print("Any missing values in the DataFrame:", df.isna().any().any())
```

```
df = df.dropna()
df.reset_index(drop=True, inplace=True)
from sklearn.ensemble import RandomForestRegressor
import pandas as pd


X = df.drop(columns=['Yield of CT'])
y = df['Yield of CT']

# Fit a Random Forest model
rf_model = RandomForestRegressor(random_state=42)
rf_model.fit(X, y)

# Extract feature importances
feature_importances = rf_model.feature_importances_

# Create a DataFrame with feature names and importances
feature_importance_df = pd.DataFrame({'Feature': X.columns, 'Importance':
feature_importances})

# Sort features based on importance
sorted_features = feature_importance_df.sort_values(by='Importance',
ascending=False)['Feature']

# Print sorted features
print("Sorted Features based on Random Forest Feature Importance:")
print(sorted_features)
```

```
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.ensemble import RandomForestRegressor
```

```
 import pandas as pd
```

```
 X = df.drop(columns=['Yield of CT'])
 y = df['Yield of CT']
```

```
 # Fit a Random Forest model
 rf_model = RandomForestRegressor(random_state=42)
 rf_model.fit(X, y)
```

```
 # Extract feature importances
 feature_importances = rf_model.feature_importances_
```

```
 # Create a DataFrame with feature names and importances
 feature_importance_df = pd.DataFrame({'Feature': X.columns, 'Importance':
feature_importances})
```

```
 # Sort features based on importance
 sorted_features = feature_importance_df.sort_values(by='Importance', ascending=False)
```

```
 # Plot feature importances
 plt.figure(figsize=(10, 6))
 sns.barplot(x='Importance', y='Feature', data=sorted_features, palette='viridis')
 plt.title('Random Forest Feature Importances')
 plt.xlabel('Importance')
 plt.ylabel('Feature')
 plt.show()
```

Code from the test_pyfume.ipynb notebook. Note that this is a notebook with text cells. Please select the code cells carefully when trying to run the code:
# PyFume

In this notebook the pyFume libary is investigated

```
 import pandas as pd
 import numpy as np
 from pyfume import *
 #selected_features = ["Tave", "Tmax", "Tmin", "Relative_yield_change"]
 selected_features = ['yield_of_CT', "Tmax", "Tmin", "Relative_yield_change"]
 #selected_features = ['Relative_yield_change','yield_of_CT', 'Latitude', 'Longitude',
 #       'Years_since_CT_started', 'Crop_rotation_CT', 'Crop_rotation_NT', 'ST',
 #       'Soil_cover_in_CT', 'Soil_cover_in_CT', 'Weed_pest_control_CT',
 #       'Weed_pest_control_NT', 'P', 'E', 'PB', 'Tave', 'Tmax', 'Tmin']
```

```
train_data = pd.read_csv("dataset/matlab_1_train.csv")[selected_features]
test_data = pd.read_csv("dataset/matlab_1_test.csv")[selected_features]
train_data
```

```
# Generate the Takagi-Sugeno FIS
FIS = pyFUME(dataframe=train_data, nr_clus=3)
```

```
# Calculate and print the accuracy of the generated model
RMSE=FIS.calculate_error(method="RMSE")
print ("The estimated RMSE of the developed model is:", RMSE)
test = FIS.get_model()
test.produce_figure()
test.get_rules()
test._outputfunctions
```

Code from Anfis_matlab_1.mlx

```
test = readtable("matlab_1_test.csv");
train = readtable("matlab_1_train.csv");
x_train = train(:, {'yield_of_CT', 'Tmax', 'Tmin'});
y_train = train(:, {'Relative_yield_change'});
x_test = test(:, {'yield_of_CT', 'Tmax', 'Tmin'});
y_test = test(:, {'Relative_yield_change'});
x_array = table2array(x_train);
y_array = table2array(y_train);

genOpt = genfisOptions('GridPartition');
genOpt.NumMembershipFunctions = 3;
genOpt.InputMembershipFunctionType = 'trapmf';
inFIS = genfis(x_array, y_array, genOpt);

opt = anfisOptions;
opt.InitialFIS = inFIS;
opt.EpochNumber = 10;
opt.DisplayANFISInformation = 1;
opt.DisplayErrorValues = 1;
opt.DisplayStepSize = 1;
opt.DisplayFinalResults = 1;

test_fis = anfis([x_train{:,:}, y_train{:,:}],opt);
x_test = table2array(x_test);
y_test = table2array(y_test);
```

Plot Membership Functions

```matlab
y_predicted = evalfis(test_fis, x_test);

subplot(3, 1, 1);
plotmf(test_fis, 'input', 1);
title('Membership Functions for yield\_of\_CT');

subplot(3, 1, 2);
plotmf(test_fis, 'input', 2);
title('Membership Functions for Tmax');

subplot(3, 1, 3);
plotmf(test_fis, 'input', 3);
title('Membership Functions for Tmin');
```

APPENDIX II

A table containing the RMSE evaluation results for the pyFume and custom TSK implementation. The independent variables are *yield of conventional tillage*, *minimum temperature* and *maximum temperature. n_sets* stand for number of Fuzzy sets (same as number of rules in this case). pyFume columns contain test results for the pyFume TSK while *train_custom* and *test_custom* columns contain the test results from our proposed TSK implementation. The orange color indicates a column based on the test set.

		pyFume		train_custom			test_custom		
n_sets	statistic_measures	train_pyFume	test_pyFume	q_0.2	q_0.4	q_0.6	q_0.2	q_0.4	q_0.6
2	Min	0.18458	0.22089	0.22085	0.22084	0.22085	0.20877	0.20884	0.20877
	Max	0.23619	0.22316	0.22151	0.22085	0.22109	0.20888	0.20885	0.20884
	Std	0.01170	0.00052	0.00020	0.00000	0.00005	0.00002	0.00000	0.00002
	Mean	0.21129	0.22161	0.22093	0.22084	0.22086	0.20884	0.20884	0.20884
3	Min	0.18901	0.22035	0.21990	0.21990	0.21990	0.20753	0.20753	0.20753
	Max	0.22810	0.22206	0.21997	0.21997	0.21997	0.20765	0.20765	0.20765
	Std	0.01041	0.00045	0.00003	0.00003	0.00003	0.00005	0.00006	0.00006
	Mean	0.20649	0.22120	0.21992	0.21994	0.21993	0.20757	0.20759	0.20758
4	Min	0.19160	0.21768	0.21838	0.21838	0.21838	0.20635	0.20635	0.20635
	Max	0.22994	0.22118	0.21852	0.21852	0.21852	0.20652	0.20652	0.20652
	Std	0.00993	0.00079	0.00003	0.00004	0.00006	0.00004	0.00005	0.00007
	Mean	0.21212	0.22004	0.21852	0.21851	0.21850	0.20651	0.20650	0.20649
5	Min	0.18916	0.21668	0.21589	0.21589	0.21663	0.20479	0.20461	0.20477
	Max	0.23115	0.22110	0.22014	0.21994	0.22003	0.20623	0.20572	0.20623
	Std	0.01308	0.00089	0.00107	0.00114	0.00096	0.00045	0.00039	0.00042
	Mean	0.21071	0.21873	0.21934	0.21890	0.21918	0.20572	0.20541	0.20566
6	Min	0.18608	0.21641	0.21766	0.21766	0.21761	0.20503	0.20505	0.20505
	Max	0.21815	0.22104	0.21988	0.22124	0.22078	0.20587	0.20560	0.20606
	Std	0.00835	0.00098	0.00083	0.00125	0.00106	0.00019	0.00017	0.00023
	Mean	0.20699	0.21875	0.21875	0.21875	0.21932	0.20523	0.20523	0.20528
7	Min	0.19065	0.21530	0.22044	0.21984	0.21864	0.20390	0.20394	0.20354
	Max	0.23288	0.22650	0.22291	0.22335	0.22294	0.20458	0.20470	0.20458
	Std	0.01234	0.00263	0.00086	0.00098	0.00116	0.00020	0.00018	0.00024
	Mean	0.20888	0.21881	0.22161	0.22128	0.22150	0.20431	0.20425	0.20425
8	Min	0.18297	0.21486	0.22191	0.22188	0.22148	0.20341	0.20341	0.20341
	Max	0.22262	0.22176	0.22248	0.22248	0.22248	0.20445	0.20445	0.20445

	Std	0.01043	0.00192	0.00015	0.00018	0.00019	0.00043	0.00041	0.00043
	Mean	0.21015	0.21708	0.22203	0.22208	0.22205	0.20415	0.20408	0.20378
9	Min	0.17298	0.21490	0.22030	0.22028	0.22041	0.20159	0.20221	0.20244
	Max	0.27207	0.22989	0.22415	0.22394	0.22437	0.20423	0.20388	0.20391
	Std	0.01958	0.00471	0.00102	0.00102	0.00094	0.00066	0.00040	0.00035
	Mean	0.21573	0.21986	0.22145	0.22192	0.22167	0.20301	0.20308	0.20292
10	Min	0.18768	0.21245	0.22115	0.22094	0.22088	0.20217	0.20176	0.20249
	Max	0.36632	0.23443	0.22705	0.22649	0.22589	0.20406	0.20406	0.20406
	Std	0.03619	0.00415	0.00175	0.00142	0.00136	0.00059	0.00061	0.00053
	Mean	0.21287	0.21738	0.22322	0.22244	0.22247	0.20332	0.20339	0.20334