ECE 449 - Intelligent Systems Engineering
Python Supplement

1 Jupyter Notebook

All labs will be completed in a **Jupyter notebook**, which is a document that allows you to create and run Python code, while including visualizations and explanatory texts in the same document. Consequently, the lab reports will also be completed in these documents. The Jupyter notebook templates for each lab can be found on eClass.

To begin, go to `https://cybera.syzygy.ca` and log-in with your ualberta e-mail address. Start your server, upload a Jupyter notebook file, and open it. In order to run a cell, select it, and press SHIFT+ENTER. Each cell has access to variables and functions in cells that were already run. The `In []` to the left of a code cell indicates its current state:

`In [ ]` - the cell has not yet been run
`In [*]` - the cell is currently being run
`In [#]` - the cell has been run and in the order indicated by the number

A cell has access to all of the variables and functions in previously run cells. Therefore, **make sure that you run the cells with code that are initially provided in the template before proceeding with your own coding**. Keyboard shortcuts can be found under HELP → KEYBOARD SHORTCUTS.

When you have completed your notebook and are ready to submit it, convert the file to PDF by printing it to PDF. This is accomplished by pressing CTRL + P or RIGHT CLICK → PRINT. Under the "Destination" option, change it to "Save to PDF", and then click Save. Make sure that all of the necessary plots are displayed in the final document that you submit. Document submission will be done through eClass, and each lab report is due a day before the next lab.

2 Fuzzy Logic

Assume that the following import statements are used:

```
import numpy as np                          # General math operations
import matplotlib.pyplot as plt             # Data visualization
from mpl_toolkits.mplot3d import Axes3D     # 3D data visualization
import skfuzzy as fuzz                      # Fuzzy toolbox
```

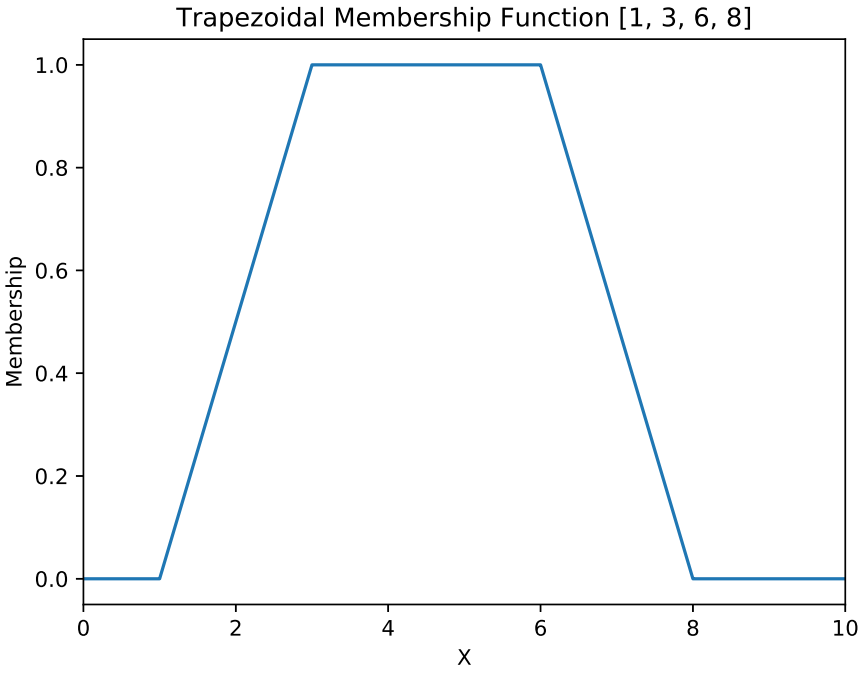
2.1 Defining a membership function

A membership function can be defined in several ways. A trapezoidal membership function requires four parameters: `a`, `b`, `c`, and `d`. `a` and `d` specify the ends of the base, and `b` and `c` specify the ends of the top.

Example:

```
X = np.linspace(0, 10, num = 11) # 0 to 10 with 11 elements
Y = fuzz.trapmf(X, [1, 3, 6, 8])
plt.plot(X, Y)
plt.xlabel('X')
plt.ylabel('Membership')
```

```
plt.title('Trapezoidal Membership Function [1, 3, 6, 8]')
plt.show()
```



2.2 Concentration and dilatation

Concentration emphasizes points with higher membership values. For example, concentration of "small numbers" turns into "**very** small numbers". This is quantitatively represented by squaring the membership values:

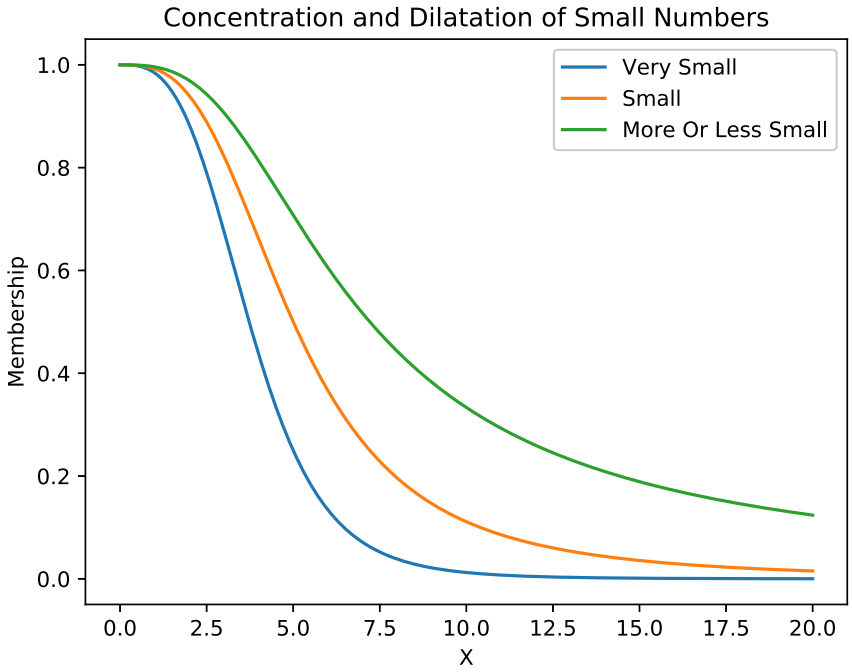
$$Con(A) = A^2(x)$$

Dilatation of "small numbers" turns into "**more or less** small numbers", and can be quantitatively represented by taking the square root of the membership values:

$$Dil(A) = \sqrt{A(x)}$$

Example:

```
X = np.linspace(0, 20, num = 201)
A = 1 / (1 + (X / 5 )** 3) # "small numbers"
verySmall = A ** 2
moreOrLessSmall = A ** (0.5)
verySmallHandle ,= plt.plot(X, verySmall, label = 'Very Small')
smallHandle ,= plt.plot(X, A, label = 'Small')
moreOrLessSmallHandle ,= plt.plot(X, moreOrLessSmall, label = 'More Or Less Small')
plt.xlabel('X')
plt.ylabel('Membership')
plt.title('Concentration and Dilatation of Small Numbers')
plt.legend(handles = [verySmallHandle, smallHandle, moreOrLessSmallHandle])
plt.show()
```



2.3 Union, intersection, and complement

The union ($A \cup B$), intersection ($A \cap B$), and complement (A^C) are common operations performed on fuzzy sets. There are various realizations of the union and intersection operations, but the most common forms are maximum and minimum, respectively. The complement operation is always defined as $1 - A(x)$

Example:

```
X = np.linspace(0, 20, num = 201)
A = 1 / (1 + (X / 5 ) ** 3) # "small numbers"
B = 1 / (1 + 0.3 * (X - 8) ** 2) # "about 8"
[Y, union]= fuzz.fuzzy_or(X, A, X, B) # Y, new universe of discourse; union, new MF
[Z, intersection]= fuzz.fuzzy_and(X, A, X, B)
complement = fuzz.fuzzy_not(A)
```

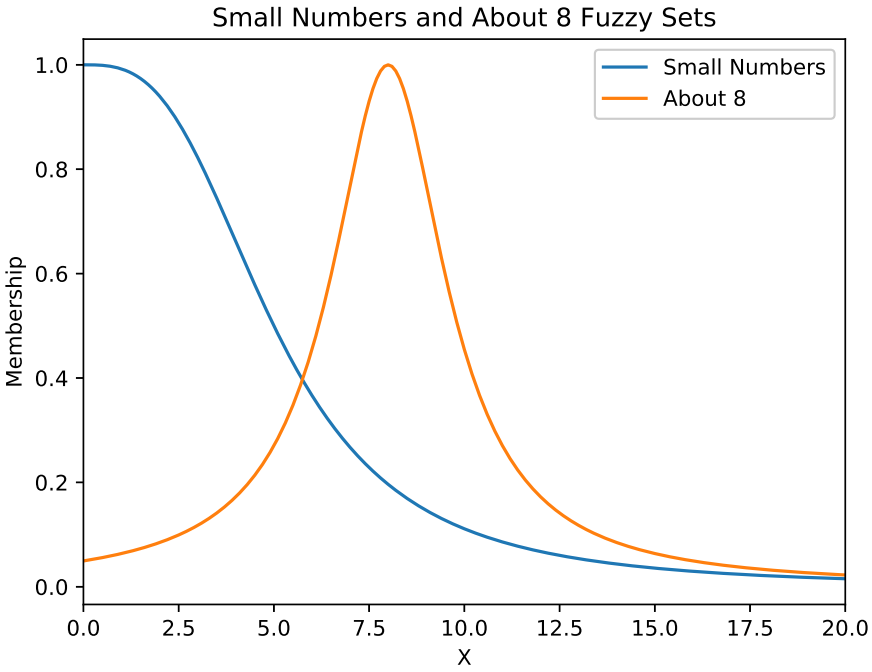
2.4 α-cuts

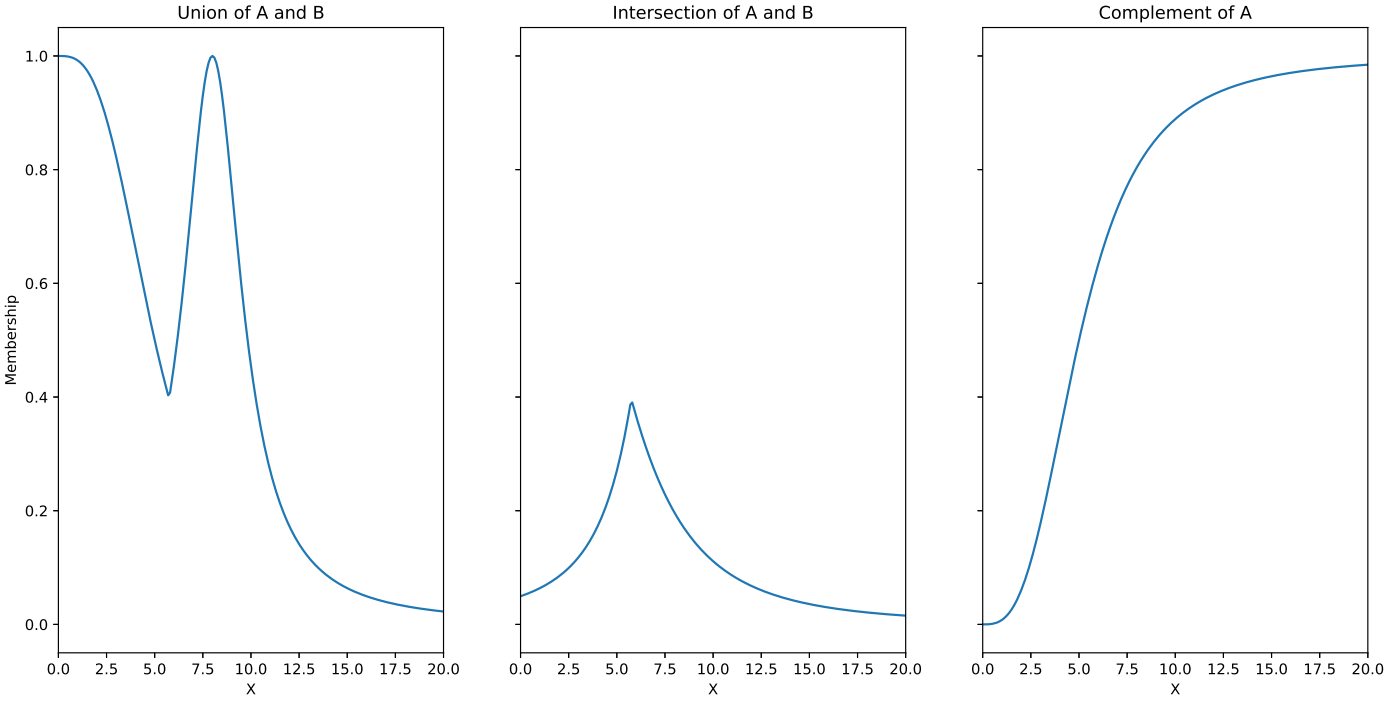
An α-cut is a crisp set that contains the elements that have membership greater than or equal to a certain value, α. Consider a fuzzy set with membership function:

$$A(x) = \frac{1}{1+0.01(x-50)^2}$$

Example:

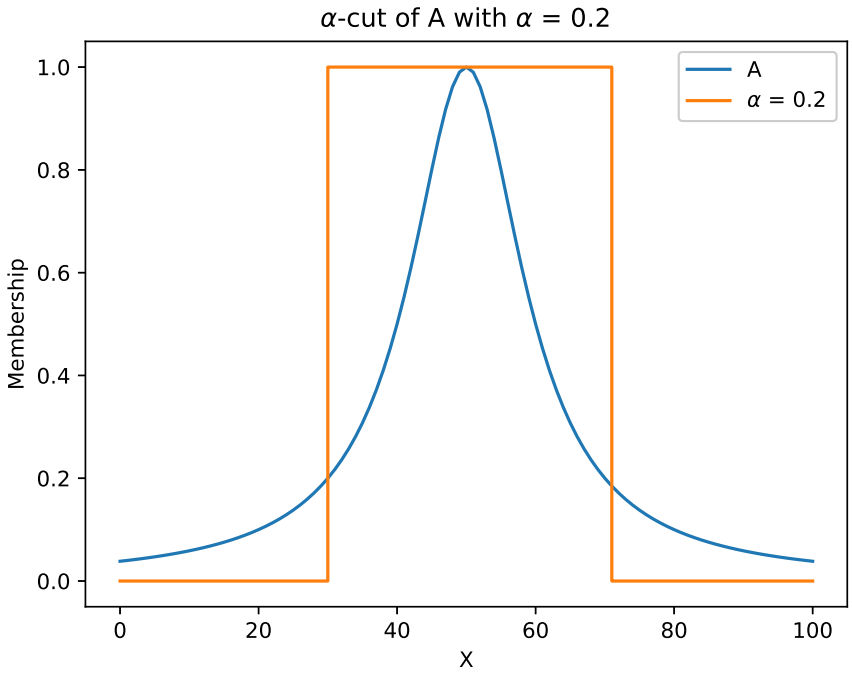
```
X = np.linspace(0, 100, num = 101)
A = 1 / (1 + 0.01 * (X - 50) ** 2)
```






```
alphaCut = (A >= 0.2) # alpha-cut with alpha = 0.2
AHandle ,= plt.plot(X, A, label = 'A')
aCutHandle ,= plt.step(X, alphaCut, where = 'post', label = r'$\alpha$ = 0.2')
plt.xlabel('X')
plt.ylabel('Membership')
```

```
plt.title(r'$\alpha$-cut of A with $\alpha$ = 0.2')
plt.legend(handles = [AHandle, aCutHandle])
plt.show()
```



2.5 Relations

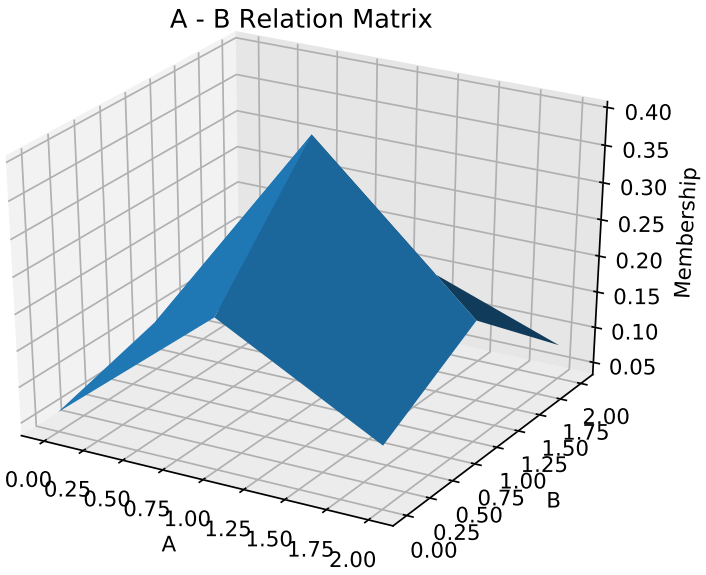
Fuzzy relations between two fuzzy sets are performed using implication operators. Common operators include Larsen implication (product) and Mamdani implication (minimum). The relation can be expressed as:

$$R_{A \to B}(x, y) = [A(x)B(y)]$$

Example:

```
uniA = np.array([0, 1, 2])
uniB = np.array([0, 1, 2])
A = np.array([0.3, 0.5, 0.2])
B = np.array([0.2, 0.8, 0.4])
larsen = fuzz.relation_product(A, B) # Larsen implication
fig = plt.figure()
[gX, gY]= np.meshgrid(uniA, uniB)
ax = fig.gca(projection = '3d')
ax.plot_surface(gX, gY, larsen)
ax.set_xlabel('A')
ax.set_ylabel('B')
ax.set_zlabel('Membership')
ax.set_title('A - B Relation Matrix')
plt.show()
```

$$larsen = \begin{bmatrix} 0.06 & 0.24 & 0.12 \\ 0.10 & 0.40 & 0.20 \\ 0.04 & 0.16 & 0.08 \end{bmatrix}$$



2.6 Projections

The x and y projections of $R(x, y)$ can be respectively expressed as:

$$ProjR_x(x) = sup_{y \in Y} R(x, y)$$
$$ProjR_y(y) = sup_{x \in X} R(x, y)$$

The supremum (sup) operator often employed is the maximum operation.

Example:

```
xProj = np.amax(larsen, axis = 0, keepdims = True)
yProj = np.amax(larsen, axis = 1, keepdims = True)
```

$$xProj = \begin{bmatrix} 0.10 & 0.40 & 0.20 \end{bmatrix}$$

$$yProj = \begin{bmatrix} 0.24 \\ 0.40 \\ 0.16 \end{bmatrix}$$

2.7 Reconstruction

The reconstruction of a fuzzy relation is expressed as the Cartesian product of its projections.

Example:

```
reconstruction = yProj * xProj
```

$$reconstruction = \begin{bmatrix} 0.024 & 0.096 & 0.048 \\ 0.040 & 0.160 & 0.080 \\ 0.016 & 0.064 & 0.032 \end{bmatrix}$$

2.8 Cylindrical extension

Cylindrical extension is defined as:

$$Cyl(A)(x, y) = A(x) \ \forall \ y \in Y$$

Example:

```
cyl = np.concatenate([xProj, xProj, xProj])
```

$$cyl = \begin{bmatrix} 0.10 & 0.40 & 0.20 \\ 0.10 & 0.40 & 0.20 \\ 0.10 & 0.40 & 0.20 \end{bmatrix}$$

2.9 Sup-t composition

Using relations R_1 and R_2, the sup-t composition can be performed with various **t** operations. Some common combinations are max-min and max-product.

Example:

```
R1 = np.array([[0.3, 0.5], [0.7, 0.2]])
R2 = np.array([[0.4, 0.9], [0.8, 1.0]])
R = fuzz.maxmin_composition(R1, R2)
```

$$R = \begin{bmatrix} 0.5 & 0.5 \\ 0.4 & 0.7 \end{bmatrix}$$

2.10 Compositional rule of inference

Generalized modus ponens states that when a rule's antecedent is satisfied to some degree, its consequent can be inferred to the same degree.

IF x **IS** A **THEN** y **IS** B
IF x **IS** A' **THEN** y **IS** B'

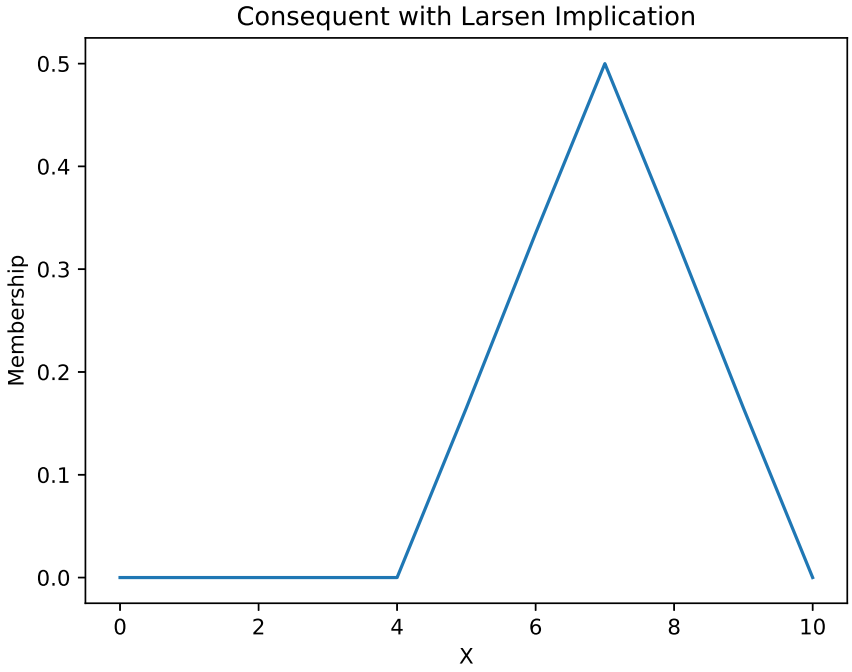
This can be written using the implication relation ($R(x, y)$) as in the max-min composition:

$$B' = A' \circ R(x, y)$$

Consider the rule "**IF** x **IS** ABOUT 3 **THEN** y **IS** ABOUT 7" with fuzzy sets FN3 = ABOUT 3 and FN7 = ABOUT 7 defined on the universe of discourse, X. Applying a compositional rule of inference to a fuzzy singleton A = 4, using Larsen implication, yields the following results.

Example:

```
X = np.linspace(0, 10, num = 11)
FN3 = np.array([0, 0, 0.5, 1.0, 0.5, 0, 0, 0, 0, 0, 0])
FN7 = np.array([0, 0, 0, 0, 0, 0.33, 0.67, 1.0, 0.67, 0.33, 0])
larsen = fuzz.relation_product(FN3, FN7)
A = np.zeros(11) # Define fuzzy singleton
A[4] = 1
fuzzySet = fuzz.maxprod_composition(A, larsen)
```



2.11 Defuzzification

There are various methods of determining the crisp value from a fuzzy set. The most frequently used are the centroid and mean of maxima (MOM) methods.

Example:

```
X = np.linspace(0, 1, num = 6)
fuzzySet = np.array([0, 0.165, 0.335, 0.5, 0.335, 0.165])
crispValue = fuzz.defuzz(X, fuzzySet, 'mom') # Mean Of Maximum defuzzification method
```

$$crispValue = 0.6$$

3 Neural Networks

Assume that the following import statements are used:

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

```
from sklearn import preprocessing
from sklearn.linear_model import Perceptron
from sklearn import MLPRegressor
from minisom import MiniSom
```

3.1 Preprocessing

The Python library, `sklearn`, contains some common methods of data preprocessing. This includes normalization, standard scaling, and minimum-maximum scaling.

3.1.1 Normalization

Using this function on an array of data normalizes each input vector in the array (along the x-axis) to unit length.

Example:

```
X = np.array([[0, 1], [3, 4], [0.5, 0.5]])
normX = preprocessing.normalize(X)
```

$$normX = \begin{bmatrix} 0 & 1 \\ 0.6 & 0.8 \\ 0.707 & 0.707 \end{bmatrix}$$

3.1.2 Standard scaling

This function standardizes features in the input data by removing the mean and scaling to unit variance. When the input data is transformed, the scaler allows for the user to perform an inverse transform to revert the data back to its original form.

Example:

```
X = np.array([[0, 1], [3, 4], [0.5, 0.5]])
scaler = preprocessing.StandardScaler()
stdX = scaler.fit_transform(X)
invX = scaler.inverse_transform(stdX) # X = invX
```

$$stdX = \begin{bmatrix} -0.889 & -0.539 \\ 1.40 & 1.40 \\ -0.508 & -0.863 \end{bmatrix}$$

3.1.3 Minimum-maximum scaling

Minimum-maximum scaling transforms the input data by scaling each feature to a given range, typically [0, 1]. Each column of the input array is scaled to this range, independently of the other columns. Similar to standard scaling, an inverse transform function is provided.

Example:

```
X = np.array([[0, 1], [3, 4], [0.5, 0.5]])
scaler = preprocessing.MinMaxScaler()
minmaxX = scaler.fit_transform(X)
invX = scaler.inverse_transform(minmaxX) # X = invX
```

$$minmaxX = \begin{bmatrix} 0 & 0.143 \\ 1 & 1 \\ 0.167 & 0 \end{bmatrix}$$

3.2 Perceptron

`sklearn` provides a way of building a simple perceptron, with the ability to set how many iterations of training it completes, and appropriate methods to employ the neural unit.

Example:

```
# Logical "OR" data
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]]) # Inputs
Y = np.array([0, 1, 1, 1]) # Targets
percep = Perceptron(n_iter = 50)
percep.fit(X, Y) # Train the perceptron
Z = percep.predict(X)
```

$$Z = \begin{bmatrix} 0 & 1 & 1 & 1 \end{bmatrix}$$

3.3 Multi-layer perceptron

MLPRegressor and MLPClassifier in `sklearn` build a multi-layer perceptron, with the ability to customize various parameters, such as the number of units in each hidden layer, the activation function, and the initial learning rate. It has similar training and prediction methods to the perceptron class. Additionally, if the output values after each iteration are required, a `partial_fit` method is provided that trains the MLP for one iteration.

Example:

```
# Logical "XOR" data
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]]) # Inputs
Y = np.array([0, 1, 1, 0]) # Targets
mlp = MLPClassifier(
    hidden_layer_sizes = (100,), # One hidden layer of 100 units
    activation = 'relu',
    max_iter = 1000
    )
mlp.fit(X, Y) # Train the MLP
Z = mlp.predict(X)
```

$$Z = \begin{bmatrix} 0 & 1 & 1 & 0 \end{bmatrix}$$

3.4 Self-organizing map (Kohonen network)

`MiniSom` is a Python library that implements self-organizing maps. The constructor permits for customization of size, neighborhood function, and learning rate.

```
som = MiniSom(
    x, # X dimension of SOM
    y, # Y dimension of SOM
    input_len, # Number of elements in an input vector
    sigma = 1.0, # Spread of neighborhood function
    learning_rate = 0.5, # Initial learning rate
    neighborhood_function = 'gaussian' # Neighborhood function type
)
```

Next, the weights must be randomly initialized, and then the SOM needs to be trained. The network can be trained using either a random method (`train_random`) or a batch method (`train_batch`).

```
som.random_weights_init(inputData) # Initialize weights randomly
som.train_random(inputData, noIter) # Train randomly noIter times
```

Finally, the SOM results can be visualized using various methods, such as `distance_map`, which displays how close each neuron is to its neighboring neurons, and `activation_response`, which displays how many times each neuron was activated to an input array of data.

```
plt.pcolor(som.distance_map().T)
plt.pcolor(som.activation_response(inputData).T)
```

4 Genetic Algorithms

Assume that the following import statements are used:

```
import numpy as np
from pyeasyga import pyeasyga
```

`pyeasyga` is a Python library that allows for the creation and customization of genetic algorithms. Only the input data needs to be specified when creating an algorithm, but numerous other parameters can be specified.

```
ga = pyeasyga.GeneticAlgorithm(
    inputData,
    population_size = 50,
    generations = 100,
    crossover_probability = 0.8,
    mutation_probability = 0.2,
    maximise_fitness = True # Minimizes fitness if False
)
```

Additionally, functions to implement the individual creation, crossover, mutation, selection, and fitness functions need to be assigned to the GA.

```
def create_individual(data):
    individual = ...
    return individual

def crossover(parent_1, parent_2):
    child_1 = ...
    child_2 = ...
    return child_1, child_2

def mutate(individual):
    individual = ...

def fitness(individual, data):
    fitness = ...
    return fitness

ga.create_individual = create_individual
ga.crossover_function = crossover
ga.mutate_function = mutate
ga.selection_function = ga.tournament_selection # pyeasyga's implemented selection function
ga.fitness_function = fitness
```

Finally, the GA can be run, and the best solution can be obtained.

```
ga.run()
bestSoln = ga.best_individual()
print("Fitness = ", bestSoln[0])
print("Solution = ", bestSoln[1])
```