

FuzzPy

Autor: [Leynier Gutiérrez González](#)

Grupo: C412

Correo: <mailto:l.gutierrez@estudiantes.matcom.uh.cu>

Tutor: Dr. Yudivian Almeida, Universidad de La Habana

Código: <https://github.com/leynier/fuzzpy>

FuzzPy es una biblioteca de **Python** para implementar **Sistemas de Inferencia Difusa**.

Características del Sistema de Inferencia

La biblioteca contiene implementados los métodos de inferencia *Mamdani* y *Larsen*. Pero es posible implementar partiendo de una base común otros métodos de inferencia.

Los métodos de inferencia reciben una función de *defuzzificación*. La biblioteca contiene implementadas *Centroide*, *Bisectriz*, *Máximo Central*, *Máximo más pequeño* y *Máximo más grande*.

Durante el proceso de definición de los conjuntos difusos esto requieren una función de membresía que puede ser implementada o utilizar una de las disponibles en la biblioteca.

Funciones de membresía implementadas en FuzzPy:

- Función Gamma
- Función Lambda o Triangular
- Función Pi o Trapezoidal
- Función S
- Función Z
- Función Gaussiana

La *T-conorm* y *T-norm* utilizadas en las reglas de inferencia, así como el método de agregación de los conjuntos son posibles de sobrescribir, por defecto, son *mínimo*, *máximo* y *máximo* respectivamente.

Es posible definir más de una variable de salida para el sistema de inferencia difusa implementado en la biblioteca.

Estructura de la Implementación

La implementación se sostiene sobre 7 clases fundamentales:

- Membership
- BaseSet
- BaseVar
- BaseRule
- Predicate
- InferenceSystem

Membership

Es la clase encargada de representar una función de membresía junto a los puntos (llamados *items* internamente)

```
1 class Membership:
2     def __init__(self, function: Callable[[Any], Any], items: list):
3         self.function = function
4         self.items = items
5
6     def __call__(self, value: Any):
7         return self.function(value)
```

BaseSet

Es la clase encargada de representar un conjunto difuso. Recibe como parámetros un objeto de tipo *Membership* representando la función de membresía del conjunto y un método de agregación.

```
1 class BaseSet:
2     def __init__(
3         self,
4         name: str,
5         membership: Membership,
6         aggregation: Callable[[Any, Any], Any],
7     ):
8         self.name = name
9         self.membership = membership
10        self.aggregation = aggregation
11
12    def __add__(self, arg: "BaseSet"):
13        memb = Membership(
14            lambda x: self.aggregation(
15                self.membership(x),
16                arg.membership(x),
17            ),
18            self.membership.items + arg.membership.items,
19        )
20        return BaseSet(
21            f"({self.name})_union_({arg.name})",
22            memb,
23            aggregation=self.aggregation,
24        )
```

BaseVar

Es la clase encargada de representar una variable lingüística. Recibe como parámetros una función de unión, una función de intercepción y una lista de objetos de tipo *BaseSet* representando los conjuntos difusos de la variable.

```
1 class BaseVar:
2     def __init__(
3         self,
4         name: str,
5         union: Callable[[Any, Any], Any],
6         inter: Callable[[Any, Any], Any],
7         sets: Optional[List[BaseSet]] = None,
8     ):
9         self.name = name
10        self.sets = {set.name: set for set in sets} if sets else {}
11        self.union = union
12        self.inter = inter
13
14    def into(self, set: Union[BaseSet, str]) -> VarSet:
15        set_name = set.name if isinstance(set, BaseSet) else set
16        if set_name not in self.sets:
17            raise KeyError(f"Set {set_name} not found into var {self.name}")
18        temp_set = self.sets[set_name]
19        return VarSet(self, temp_set, self.union, self.inter)
```

BaseRule

Es la clase encargada de representar una regla de inferencia. Recibe como parámetro un objeto de tipo *Predicate* representando el antecedente de la regla.

```
1 class BaseRule:
2     def __init__(self, antecedent: Predicate):
3         self.antecedent = antecedent
4
5     def __call__(self, values: dict):
6         raise NotImplementedError()
```

BaseRule no contiene consecuencias porque las consecuencias de todos los tipos de reglas no son de la misma estructura. La clase *Rule* hereda de *BaseRule* y representa las reglas en los que el sistema produce un conjunto o más como resultado.

```
1 class Rule(BaseRule):
2     def __init__(self, antecedent: Predicate, consequences: List[VarSet]):
3         super(Rule, self).__init__(antecedent)
4         self.consequences = consequences
5
6     def aggregate(self, set: BaseSet, value: Any) -> BaseSet:
7         raise NotImplementedError()
8
9     def __call__(self, values: dict):
10        value = self.antecedent(values)
11        return {
12            consequence.var.name: self.aggregate(
13                consequence.set,
14                value,
15            )
16            for consequence in self.consequences
17        }
```

Predicate

Es la clase encargada de representar a los antecedentes. De ella heredan cuatro clases: *AndPredicate*, *OrPredicate*, *NotPredicate* y *VarSet*. Las primeras tres para representar las relaciones lógicas de unión, intercepción y negación; y la última representa la inclusión de una variable en un determinado conjunto, siendo esta la clase básica para representar a los antecedentes.

```
1 class Predicate:
2     def __init__(
3         self,
4         union: Callable[[Any, Any], Any],
5         inter: Callable[[Any, Any], Any],
6     ) -> None:
7         self.union = union
8         self.inter = inter
9
10    def __call__(self, values: dict):
11        raise NotImplementedError()
12
13    def __and__(self, other: "Predicate"):
14        return AndPredicate(self, other, self.union, self.inter)
15
16    def __or__(self, other: "Predicate"):
17        return OrPredicate(self, other, self.union, self.inter)
18
19    def __invert__(self):
20        return NotPredicate(self, self.union, self.inter)
```

VarSet

```
1 class VarSet(Predicate):
2     def __init__(
3         self,
4         var: "BaseVar",
5         set: BaseSet,
6         union: Callable[[Any, Any], Any],
7         inter: Callable[[Any, Any], Any],
8     ):
9         super(VarSet, self).__init__(union, inter)
10        self.var = var
11        self.set = set
12
13    def __call__(self, values: dict):
14        return self.set.membership(values[self.var.name])
```

InferenceSystem

Es la clase encargada de representar el sistema de inferencia. Recibe como parámetros las reglas y una función de defuzzificación y con el método *infer* permite realizar la inferencia según los valores proveídos.

```
1 class InferenceSystem:
2     def __init__(
3         self,
4         rules: Optional[List[BaseRule]] = None,
5         defuzz_func: Optional[Callable[[BaseSet], Any]] = None,
6     ):
7         self.rules = rules if rules else []
8         self.defuzz_func = defuzz_func
9
10    def infer(
11        self,
12        values: dict,
13        defuzz_func: Optional[Callable[[BaseSet], Any]] = None,
14    ) -> Dict[str, Any]:
15        if not self.rules:
16            raise Exception("Empty rules")
17        if self.defuzz_func is None and defuzz_func is None:
18            raise Exception("Defuzzification not found")
19        func = self.defuzz_func if defuzz_func is None else defuzz_func
20        set: Dict[str, BaseSet] = self.rules[0](values)
21        for rule in self.rules[1:]:
22            temp: Dict[str, BaseSet] = rule(values)
23            for key in temp:
24                set[key] += temp[key]
25        result: Dict[str, Any] = {}
26        for key in set:
27            result[key] = func(set[key])
28        return result
```

Ejemplo de cómo utilizar FuzzPy

Como ejemplo se utilizará el siguiente problema.

Se desea inferir el por ciento de la cantidad de un determinado producto que se ha vendido en un día en un restaurante, cafetería, etc.

Por ejemplo, el producto *Pollo*, se desea conocer bajo determinadas condiciones que por ciento del *Pollo* sacado del almacén dispuesto para venderse ese día se termina vendiendo.

Para la implementación se seleccionaron 4 variables lingüísticas. Las primeras 3 de entrada y la última de salida.

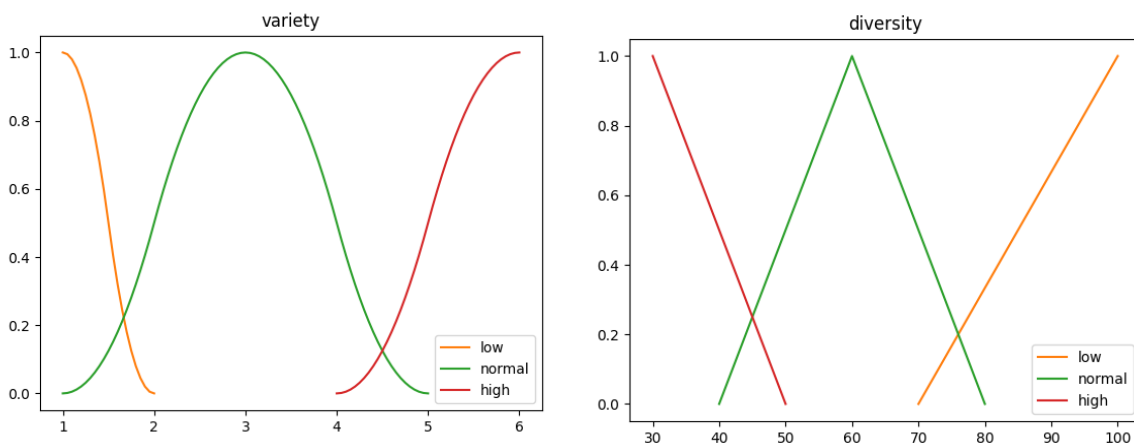
1. Cantidad de platos o derivados del producto que se vende. Por ejemplo, retomando el ejemplo del *Pollo*, si se vendería *Pollo Frito* y *Pollo Asado*, la variable valdría 2. A esta variable le llamaremos *variety*.
 - Baja: *low* \leq 2. Función de Membresía: Z
 - Normal: $1 \leq$ *normal* \leq 5. Función de Membresía: Gaussiana
 - Alta: *high* \geq 4. Función de Membresía: S
2. Por ciento que representa la variable *variety* del total de platos o derivados de productos que se vende. Por ejemplo, si se vende *Pollo Frito*, *Pollo Asado*, *Pescado* y *Cerdo* la variable valdría 50. A esta variable se le llamará *diversity*.
 - Baja: *low* \geq 70. Función de Membresía: Gamma
 - Normal: $40 \leq$ *normal* \leq 80. Función de Membresía: Lambda
 - Alta: *high* \leq 50. Función de Membresía: L
3. Por ciento de la utilización del local, si es 100 es que el local siempre está lleno, si es 0 es que no asiste ningún cliente al establecimiento. A esta variable se le llamará *clients*.
 - Baja: *low* \leq 40. Función de Membresía: L
 - Normal: $30 \leq$ *normal* \leq 90. Función de Membresía: Lambda
 - Alta: *high* \geq 80. Función de Membresía: Gamma
4. Por ciento de la cantidad del producto que se vendió en el día, si es 100 fue se vendió todo al final del día, si es 50 fue que no se vendió la mitad de la cantidad. A esta variable se le llamará *sales*.
 - Baja: *low* \leq 60. Función de Membresía: L
 - Normal: $30 \leq$ *normal* \leq 90. Función de Membresía: Lambda
 - Alta: *high* \geq 90. Función de Membresía: Gamma

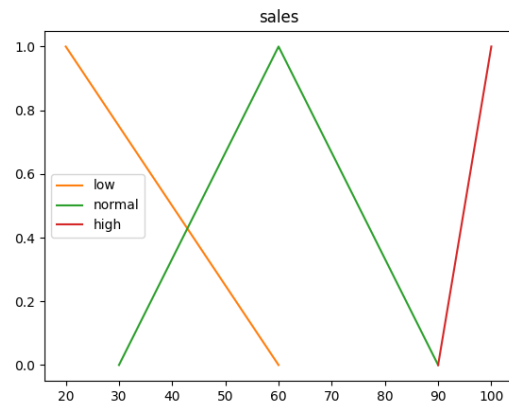
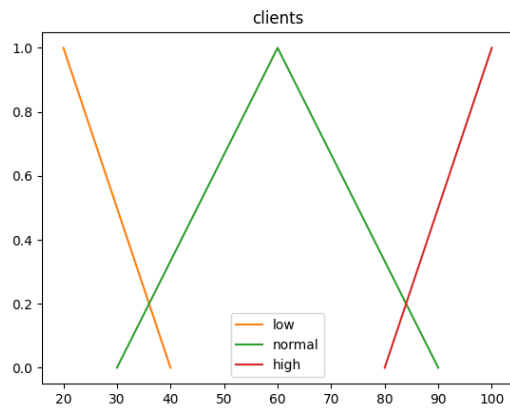
Declaración de las variables lingüísticas y sus conjuntos difusos en FuzzPy



```
1 variety_var = Var("variety")
2 variety_var += "low", ZMembership(1, 2)
3 variety_var += "normal", GaussianMembership(3, 2)
4 variety_var += "high", SMembership(4, 6)
5
6 diversity_percent_var = Var("diversity")
7 diversity_percent_var += "low", GammaMembership(70, 100)
8 diversity_percent_var += "normal", LambdaMembership(40, 60, 80)
9 diversity_percent_var += "high", LMembership(30, 50)
10
11 clients_percent_var = Var("clients")
12 clients_percent_var += "low", LMembership(20, 40)
13 clients_percent_var += "normal", LambdaMembership(30, 60, 90)
14 clients_percent_var += "high", GammaMembership(80, 100)
15
16 sales_percent_var = Var("sales")
17 sales_percent_var += "low", LMembership(20, 60)
18 sales_percent_var += "normal", LambdaMembership(30, 60, 90)
19 sales_percent_var += "high", GammaMembership(90, 100)
```

Gráficos de pertenencia de los conjuntos por cada variable





Reglas de Inferencia

1	variety	diversity	clients	sales
2	:-----:	:-----:	:-----:	:-----:
3	low	low	low	low
4	low	low	normal	normal
5	low	low	high	high
6	low	normal	low	low
7	low	normal	normal	low
8	low	normal	high	normal
9	low	high	low	low
10	low	high	normal	low
11	low	high	high	normal
12	normal	low	low	low
13	normal	low	normal	normal
14	normal	low	high	high
15	normal	normal	low	low
16	normal	normal	normal	normal
17	normal	normal	high	normal
18	normal	high	low	low
19	normal	high	normal	low
20	normal	high	high	normal
21	high	low	low	low
22	high	low	normal	normal
23	high	low	high	high
24	high	normal	low	low
25	high	normal	normal	low
26	high	normal	high	high
27	high	high	low	low
28	high	high	normal	low
29	high	high	high	normal

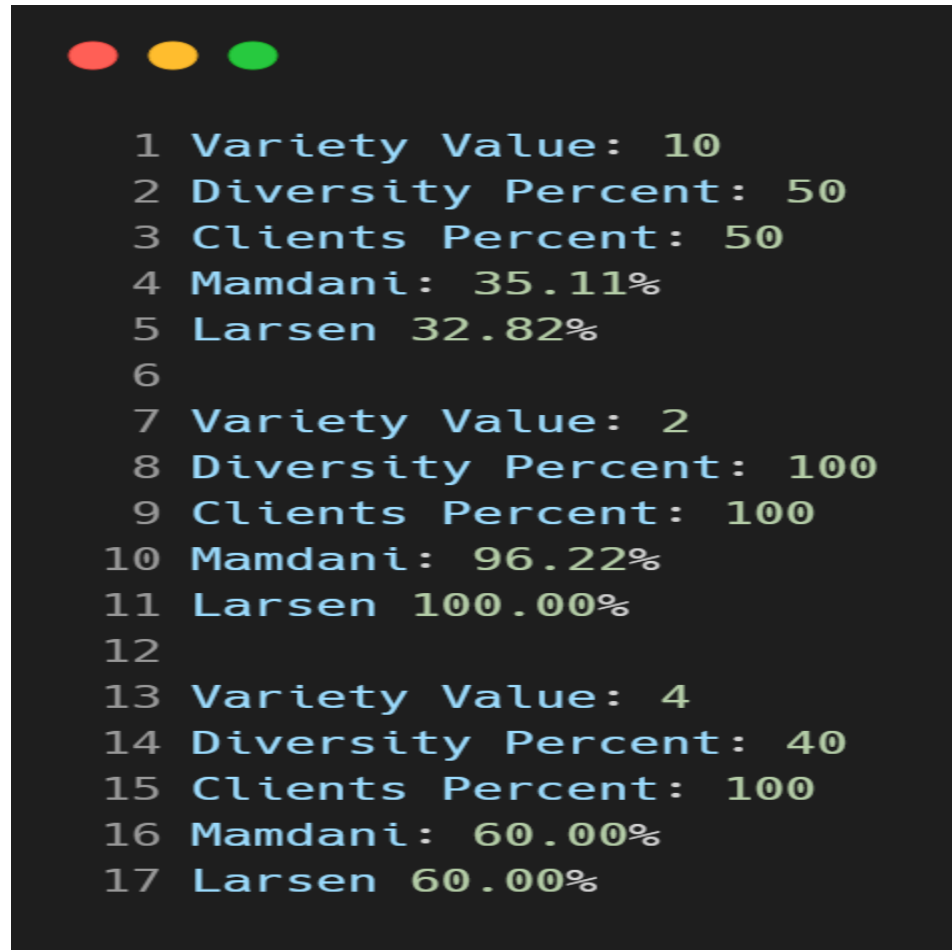
Declaración de las Reglas de Inferencia en FuzzPy



```
1 mamdani = MamdaniSystem(  
2     defuzz_func=centroid_defuzzification,  
3 )  
4 mamdani += (  
5     variety_var.into("low")  
6     & diversity_percent_var.into("low")  
7     & clients_percent_var.into("low")  
8 ), sales_percent_var.into("low")  
9 mamdani += (  
10    variety_var.into("low")  
11    & diversity_percent_var.into("low")  
12    & clients_percent_var.into("normal")  
13 ), sales_percent_var.into("normal")  
14 mamdani += (  
15    variety_var.into("low")  
16    & diversity_percent_var.into("low")  
17    & clients_percent_var.into("high")  
18 ), sales_percent_var.into("high")  
19 mamdani += (  
20    variety_var.into("low")  
21    & diversity_percent_var.into("normal")  
22    & clients_percent_var.into("low")  
23 ), sales_percent_var.into("low")  
24 mamdani += (  
25    variety_var.into("low")  
26    & diversity_percent_var.into("normal")  
27    & clients_percent_var.into("normal")  
28 ), sales_percent_var.into("low")  
29  
30 ...
```

De manera análoga sería para Larsen utilizando la clase *LarsenSystem*.

Resultados



```
1 Variety Value: 10
2 Diversity Percent: 50
3 Clients Percent: 50
4 Mamdani: 35.11%
5 Larsen 32.82%
6
7 Variety Value: 2
8 Diversity Percent: 100
9 Clients Percent: 100
10 Mamdani: 96.22%
11 Larsen 100.00%
12
13 Variety Value: 4
14 Diversity Percent: 40
15 Clients Percent: 100
16 Mamdani: 60.00%
17 Larsen 60.00%
```

Análisis de los Resultados

De los resultados, se puede observar que los métodos de Mamdani y Larsen obtienen resultados similares. A primera vista no es posible validar si los resultados se asemejan a la realidad, para esto es imprescindible la colaboración de un experto en el tema para la correcta definición de las variables, la asignación de las funciones de membresía más correctas así como la definición de las reglas asociadas.

Conclusiones

En este escrito se muestra las líneas generales de cómo utilizar FuzzPy, además de que muestra la capacidad de los sistemas de inferencia difusos para afrontar problemáticas donde la definición utilizando la lógica clásica no esté clara o que la solución utilizando esta sea demasiado engorrosa.

Referencias

1. Sistemas de Control con Lógica Difusa: Métodos de Mamdani y de Takagi-Sugeno-Kang (TSK). Autor: Samuel Diciembre Sanahuja
2. Temas de Simulación. Autor: Dr. Luciano García Garrido
3. First Course on Fuzzy Theory and Applications. Autor: Kwang H. Lee