پروژه درس فازی

مرحله 1 و 2

استاد براتی

شبیه سازی یک تابع به روش ونگ مندل

پویان رضائی

خلاصه پیاده سازی:

این پروژه با استفاده از زبان برنامه نویسی پایتون (python) پیاده سازی شده و از کتابخانه های:

Matplotlib

برای کشیدن پلات های مورد نیاز

Pandas

برای ذخیره دیتا به صورت دیتافریم و فایل (csv) (تنها برای بخش data preparation استفاده شده)

Numpy

برای انجام عملیات های ریاضی و ماتریسی استفاده شده (امکان پردازش سریع تر میدهد)

مدل به صورت یک کلاس شامل چند متغیر و تابع پیاده سازی شده است

```
class FuzzyModel():
```

Constructor این کلاس شامل 4 متغیر می شود ، متغیر اول فازی ست های تعریفی برای x1_data و x2_data و y_data و مرکز فازی ست خروجی (output_centers) و ورودی constructor شامل دیتا آموزشی و تعداد فازی ست های مورد نیاز که به صورت پیش فرض 7 قرار گرفته

```
    def __init__(self, training_data, num_fuzzy_sets=7):
        x1_data = [x[0] for x in training_data]
        x2_data = [x[1] for x in training_data]
        y_data = [x[2] for x in training_data]

        self.input_fuzzy_sets_x1 = self.calculate_fuzzy_sets(x1_data,
num_fuzzy_sets)
        self.input_fuzzy_sets_x2 = self.calculate_fuzzy_sets(x2_data,
num_fuzzy_sets)
        self.output_fuzzy_sets = self.calculate_fuzzy_sets(y_data,
num_fuzzy_sets)

        # Calculate centers for output fuzzy sets
        self.output_centers = {label: (a + b + c) / 3 for label, (a, b, c) in
self.output_fuzzy_sets.items()}
```

بعد با استفاده از3 تعریف شده و تابع calculate_fuzzy_sets (در بخش تابع ها طرز کار این تابع توضیح داده شده است) فازی ست های مورد نیاز برای دو ورودی و خروجی محاسبه می شود.
و در متغیر output_centers مرکز خروجی ها یا همان centroid ها برای defuzzification از روش center average استفاده می شود ذخیره می شود.

تابع ها

تابع calculate_fuzzy_sets

```
    def calculate_fuzzy_sets(self, data, num_sets=7):
        min_val = np.min(data)
        max_val = np.max(data)
        sets = ["NB","NM","NS","ZR","PS","PM","PB"]

        step = (max_val - min_val) / (num_sets - 1)

        fuzzy_sets = {}
        for i in range(len(sets)):
            label = f"{sets[i]}"
            if i == 0:
                fuzzy_sets[label] = (min_val, min_val, min_val + step)
            elif i == num_sets - 1:
                fuzzy_sets[label] = (max_val - step, max_val, max_val)
            else:
                fuzzy_sets[label] = (min_val + (i - 1) * step, min_val + i *
step, min_val + (i + 1) * step)

        return fuzzy_sets
```

در این تابع دو ورودی دارد که یکی data هست که ورودی داده های برای انجام پروسه تعریف فازی ست ها هست و دیگر num_sets هست که مقدار فازی ست های مورد نیاز است در این تابع در ابتدا مقدار کمینه و بیشینه داده ها حساب میشود سپس مقدار فاصله (step) ست ها محاسبه میشود.
و لیستی تعریف شده برای ذخیره کردن نام های فازی ست ها.
در ادامه یک دیکشنری تعریف میکنیم به نام fuzzy_sets و با استفاده از یک حلقه به تعداد فازی ست های مورد نیاز یعنی 7 لوپ انجام می شود که در این لوپ چک میشود که اگر فازی ست اول (در این پروژه یعنی NB) مقدار شروع فازی ست با مینیمم فازی ست برابر است و به اندازه یک step پایان فازی ست نسبت به

مینیمم فاصله دارد و جلو تر است و اگر فازی ست آخر باشد مقدار پایانی فازی ست با ماکزیمم فازی ست آخر (در این پروژه یعنی فازی ست PB) برابر است و مقدار شروع فازی ست برابر با یک step عقب تر از فازی ست آخر هست. و در غیر این صورت یعنی اگر فازی ست اول یا آخر نباشد به اندازه step ضربدر i (یعنی iterator) به اضافه مقدار منیمم پیدا شده میشود مقدار ماکزیمم فازی ست پیدا می شود و یک step قبل تر از آن میشود شروع فازی ست و یک step جلوتر میشود اتمام فازی ست. و در اخر دیکشنری تشکیل شده برگردانده میشود.

تابع Triangular MF:

```
    def triangular_mf(self, x, a, b, c):
        return np.maximum(0, np.minimum(
            (x - a) / (b - a) if b != a else (1 if x >= a else 0),
            (c - x) / (c - b) if b != c else (1 if x <= c else 0)
        ))
```

تابع بعدی تابع ممبرشیپ مثلثاتی است که به صورت زیر نوشته شده که اگر x منفی شود 0 برمیگردد و اگر بیشتر از بزرگتر از c یا همان پایان فازی ست باشد 0 برمیگردانند در غیر این صورت از فرمول گفته شده در تابع های تعریف شده در Triangular MF استفاده میکنیم. در این تابع به علت دو فازی ست NB و PB که مثلث کامل نیستن شرایط آن ها هم در نظر گرفته شده.

تابع fuzzify_input:

```
    def fuzzify_input(self, x, fuzzy_sets):
        membership_degrees = {}
        for label, (a, b, c) in fuzzy_sets.items():
            membership_degrees[label] = self.triangular_mf(x, a, b, c)
        return membership_degrees
```

در این تابع ورودی ها x و fuzzy_sets هستند، که بر اساس فازی ست های تعریف شده آرگومان های آن به تابع ممبرشیپ مثلثاتی پاس داده می شود و یه درجه عضویت محاسبه میشود و در دیکشنری تعریف شده ذخیره میشود در این دیکشنری لیبل فازی ست key هست و درجه عضویت value

تابع generate_rules:

```
    def generate_rules(self, training_data):
        rules = []
```

```
        for x1, x2, y in training_data:
            memberships_x1 = self.fuzzify_input(x1, self.input_fuzzy_sets_x1)
            memberships_x2 = self.fuzzify_input(x2, self.input_fuzzy_sets_x2)

            memberships_y = self.fuzzify_input(y, self.output_fuzzy_sets)

            x1_label = max(memberships_x1, key=memberships_x1.get)
            x2_label = max(memberships_x2, key=memberships_x2.get)
            y_label = max(memberships_y, key=memberships_y.get)

            weight = min(memberships_x1[x1_label], memberships_x2[x2_label])


            rules.append(((x1_label, x2_label), y_label, weight))

        rule_dict = {}
        for (antecedent, consequent, weight) in rules:
            if antecedent not in rule_dict or rule_dict[antecedent][1] < weight:
                rule_dict[antecedent] = (consequent, weight)

        self.rule_base = {k: v[0] for k, v in rule_dict.items()}

        return self.rule_base
```

در تابع generate_rules ورودی ما داده آموزشی ما هست که در این تابع قوانین بر اساس این داده ها ساخته می شود در ابتدا درجه عضویت ورودی ها و خروجی محاسبه می شود سپس فازی ستی که دارای بیشترین درجه عضویت هست انتخاب می شود و وزن قانون بر اساس مینیمم دو قانون یا همان intersection آن ها محاسبه می شود و در لیستی اضافه میشوند سپس باید conflict های قوانین آن حل شود در این قسمت در حلقه تعریف شده مطمئن میشویم که قوانین antecedent یا پیش آمد های غیر تکراری دارای بالاترین مقدار وزن هستند و اگر یک antecedent وجود داشته باشد در دیکشنری که وزن کمتری نسبت به قانونی که تازه محاسبه شده داشته باشد وزن آن قانون آپدیت و با وزن جدید جایگزین می شود. و در اخر یک لیست از قوانین نهایی برگردانده می شود. که مدل فازی ما بر اساس این قوانین کار میکنند.

تابع evalutate rules:

```
    def evaluate_rules(self, input_memberships_x1, input_memberships_x2):
```

```
        output_memberships = {label: 0 for label in
self.output_fuzzy_sets.keys()}

        for (input1_label, input2_label), output_label in self.rule_base.items():
            if input1_label in input_memberships_x1 and input2_label in
input_memberships_x2:
                firing_strength = min(input_memberships_x1[input1_label],
input_memberships_x2[input2_label])

                output_memberships[output_label] =
max(output_memberships[output_label], firing_strength)

        return output_memberships
```

در این تابع که ورودی آن درجه عضویت متغیر اول و درجه عضویت متغیر دوم هست، طبق rule base ساخته شده درجه عضویت خروجی به ازای ورودی ها ساخته می شود. ابتدا لیبل فازی ست هارا از فازی ست خروجی که در constructor ساختیم پیدا کرده و در یک دیکشنری به عنوان key استفاده میکنیم و تمام value هارا 0 قرار میدهیم (این 0 ها همان درجه عضویت های خروجی هستند که در ادامه آپدیت می شود) سپس یک حلقه داریم که وظیفه iteration در لیست rule base را دارد که با استفاده از یک بلاک if چک میکند که طبق antecedent مورد نظر (antecedent ها یعنی ورودی اول و دوم) مطابق کدام قانون ساخته شده است. سپس اول با استفاده از تابع min قدرت شلیک یا اجرا یا همان firing strength بین دو ورودی محاسبه می شود و سپس با استفاده از تابع max درجه عضویت های خروجی مورد نیاز از از 0 به مقدار firing strength آپدیت می شود و اینکار باعث می شود که اگر چند قانون فازی منتهی به یک خروجی شوند درجه عضویت بالاتر ذخیره می شود.

تابع defuzzify_center_of_average:

```
    def defuzzify_center_of_average(self, memberships):
        numerator = sum(membership * self.output_centers[label] for label,
membership in memberships.items())
        denominator = sum(memberships.values())

        if denominator == 0:
            return 0

        return numerator / denominator
```

این تابع همان تابع دیفازیفیکیشن ما هست که از طریق روش center average انجام میشود. ورودی این تابع درجه عضویت های بدست امده هست (برای توضیح بیشتر درجه عضویت های بدست آمده از تابع evaluate

rules) . در این تابع ابتدا numerator حساب می شود که می شود جمع وزن دار ممرشیپ ها یا همان درجه های عضویت که درجه عضویت ضربدر مرکز یا center به دست امده میشود (centroid ها در بخش constructor محاسبه شدند). سپس باید denominator را محاسبه کنیم که می شود جمع تمام درجه عضویت های بدست آمده. سپس برای به دست اوردن crisp value باید numerator بر denominator تقسیم شود که برای جلوگیری از خطای تقسیم بر صفر یک if بلاک قرار داده ایم که اگر مخرج 0 بود 0 را برگرداند.

تابع calculate_mse:

```
    def calculate_mse(self, predictions, targets):
        predictions = np.array(predictions)
        targets = np.array(targets)
        squared_differences = (predictions - targets) ** 2

        mse = np.sum(squared_differences) / (2 * len(targets))
        return mse
```

تابع محاسبه خطا MSE دارای دو ورودی هست ورودی اول پیش بینی های مدل هست و پیش بینی بعدی داده های واقعی و تارگت های ما هست. اول دو لیست ورودی را به آرایه های نامپای تبدیل میکنیم (برای تسهیل و تسریع محاسبه) سپس طبق فرمول گفته شده ابتدا مقدار اختلاف مقادیر پیش بینی شده با مقادیر واقعی به توان 2 را محاسبه کرده (squared error) سپس طبق فرمول داده شده مجموع تمام این مقادیر را تقسیم بر دو برابر تعداد مقادیر محاسبه میکنیم و تابع یک عدد به عنوان معیار برمیگرداند.

تابع predict_plot_3d_output:

```
    def predict_plot_3d_output(self, test_cases):
        x1_values = [case[0] for case in test_cases]
        x2_values = [case[1] for case in test_cases]
        actual_output = [case[2] for case in test_cases]

        predictions = []
        targets = []
        for input_value_x1, input_value_x2, actual_output in zip(x1_values,
x2_values, actual_output):
            memberships_x1 =
self.fuzzify_input(input_value_x1,self.input_fuzzy_sets_x1)
```

```
            memberships_x2 =
self.fuzzify_input(input_value_x2,self.input_fuzzy_sets_x2)
            output_memberships =
self.evaluate_rules(memberships_x1,memberships_x2)
            crisp_output = self.defuzzify_center_of_average(output_memberships)
            predictions.append(crisp_output)
            targets.append(actual_output)

        mse = model.calculate_mse(predictions, targets)
        print(f"Mean Squared Error (MSE): {mse}")
        X1_grid, X2_grid = np.meshgrid(np.unique(x1_values),
np.unique(x2_values))
        Z_grid = np.array(predictions).reshape(X1_grid.shape)

        fig = plt.figure(figsize=(10, 8))
        ax = fig.add_subplot(111, projection='3d')

        ax.plot_surface(X1_grid, X2_grid, Z_grid, cmap='viridis',
edgecolor='none')

        ax.set_title(f'Crisp Output Surface Plot, MSE: {mse}')
        ax.set_xlabel('x1')
        ax.set_ylabel('x2')
        ax.set_zlabel('Crisp Output')

        plt.show()
```

در این تابع یک ورودی به عنوان test_cases میگیرد که شامل یک ستون که مقادیر X1 را دارد و ستون بعدی که مقادیر X2 را دارد و ستون اخر که خروجی $f(x)=x_1^2+x_2^2$ را در بر دارد این تابع به صورت خلاصه تمام تابع های نوشته شده در بالا استفاده می شود (به غیر از generate rule از این تابع مانند تابع train استفاده میکنیم برای استفاده از تابع predict_plot_3d_output اول باید یک دور تابع generate rule را فرخوانی کنیم که قوانین ساخته شود سپس میتوان از تابع predict استفاده کرد) در این تابع بعد از ساخته شدن قوانین ستون های ورودی را به سه لیست مجزا تقسیم میکند سپس دو لیست تعریف کرده یکی برای مقادیر پیش بینی شده و یکی برای مقادیر واقعی (لیست targets صرفا برای خوانایی بهتر کد تعریف شده و میتوان همان استفاده را از طریق سومین لیست تشکیل شده از ورودی ها یعنی actual ouput برد) در ابتدا به ازای هر ردیف ورودی های x_1 و x_2 مقادیر را فازیفای میکنیم طبق توابع تعریف شده سپس با استفاده از تابع evaluate rules و ممبرشیپ های به دست امده و با استفاده قوانین تولید شده درجه عضویت خروجی را به دست میاوریم سپس با استفاده از تابع defuzzifying_center_of_average درجه عضویت خروجی را به یک crisp value تبدیل میکنیم و در انتها آن را به لیست predictions اضافه کرده و مقادیر واقعی را به

targets اضافه میکنیم و در انتها با استفاده از دو لیست نهایی شده مقدار خطا را توسط تابع calculate mse به دست میاوریم.

در خط های بعد برای تفهیم بهتر خروجی یک پلات سه بعدی از داده های ورودی x_1 و x_2 و خروجی crisp کشیده می شود که توسط کتابخانه matplotlib اینکار انجام میشود.

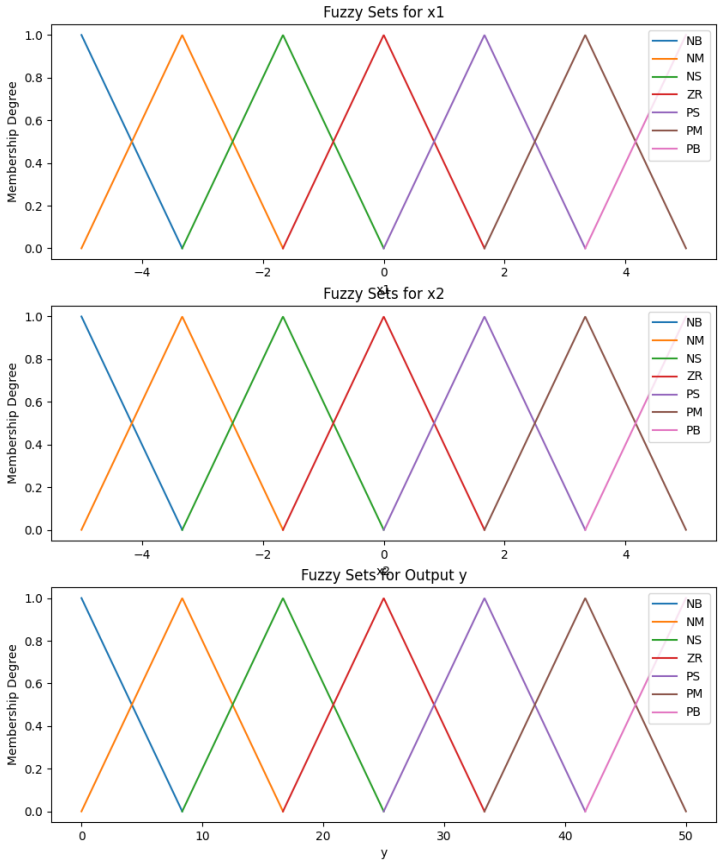
تابع plot_fuzzy_sets:

```
    def plot_fuzzy_sets(self):
        vectorized_mf = np.vectorize(self.triangular_mf)

        fig, ax = plt.subplots(3, 1, figsize=(10, 12))

        ax[0].set_title('Fuzzy Sets for x1')
        for label, (a, b, c) in self.input_fuzzy_sets_x1.items():
            x = np.linspace(a , c , 500)
            y = vectorized_mf(x, a, b, c)
            ax[0].plot(x, y, label=f'{label}')
        ax[0].legend(loc='upper right')
        ax[0].set_xlabel('x1')
        ax[0].set_ylabel('Membership Degree')

        ax[1].set_title('Fuzzy Sets for x2')
        for label, (a, b, c) in self.input_fuzzy_sets_x2.items():
            x = np.linspace(a , c, 500)
            y = vectorized_mf(x, a, b, c)
            ax[1].plot(x, y, label=f'{label}')
        ax[1].legend(loc='upper right')
        ax[1].set_xlabel('x2')
        ax[1].set_ylabel('Membership Degree')

        ax[2].set_title('Fuzzy Sets for Output y')
        for label, (a, b, c) in self.output_fuzzy_sets.items():
            x = np.linspace(a , c , 500)
            y = vectorized_mf(x, a, b, c)
            ax[2].plot(x, y, label=f'{label}')
        ax[2].legend(loc='upper right')
        ax[2].set_xlabel('y')
        ax[2].set_ylabel('Membership Degree')

        # Show the plots
        plt.savefig("Fuzzy sets Plots")
        plt.tight_layout()
        plt.show()
```

در این تابع که باز هم برای تفهیم بهتر و مطمئن شدن از اینکه فازی ست ها به درستی تعریف شده اند نوشته شده. در این تابع، تابع درجه عضویت مثلثاتی نوشته شده برای فازی ست های ورودی اول یعنی x_1 و فازی ست دوم یعنی x_2 و فازی ست خروجی با 3 ساب پلات (subplot) کشیده میشود .اول با استفاده از کتابخانه نامپای و تابع vectorize (یک تابع wrapper هست که به ما اجازه می دهد که تابع درجه عضویت مثلثاتی خود را روی تموم ورودی ها بدون اینکه از حلقه ها استفاده کنیم اجرا کنیم و تمام خروجی هارا به صورت یک لیست تحویل میگیریم) تابع vectorized_mf را میسازیم.

سپس یک صفحه تعریف میکنیم با سایز 10,12 و دارای 3 ساب پلات برای هر ساب پلات مراحلی که گفته می شود اجرا می شود. اول تیتر ساب پلات را مشخص میکنیم به دلخواه سپس برای فازی ست هایی که در کلاس تعریف شده (در constructor کلاس) و مقادیر تعریف شده و لیبل های هر فازی ست (NB,NM,NS,ZR,PS,PM,PB) را از متغیر آن با یک حلقه گرفته و برای هر کدام از a تا c (یعنی از شروع آن تا پایان آن) 500 مقدار با فاصله یکسان تعریف و در x گذاشته و برای y از تابع vectoriced_mf که تعریف کردیم استفاده میکنیم که خروجی های آن به درست بیاوریم سپس نمودار را براساس x و y به دست میاوریم و از label های فاز

<u>پلات خروجی:</u>



بلاک اجرایی برنامه یا همان main:

```
if __name__ == "__main__":
    data = pd.read_csv("training_data.csv")
    data= data.to_records(index=False)
    model = FuzzyModel(data)

    # Generate rules using Wang-Mendel method
    model.generate_rules(data)
    print("\nGenerated Rule Base:")
    for rule, output in model.rule_base.items():
        print(f"If x1 is {rule[0]} and x2 is {rule[1]}, then output is {output}")

    # Define test cases (with corresponding actual target outputs)
    test_cases = pd.read_csv("test_data.csv")
    test_cases = test_cases.to_records(index=False)


    model.plot_fuzzy_sets()
    model.predict_plot_3d_output(test_cases)
```

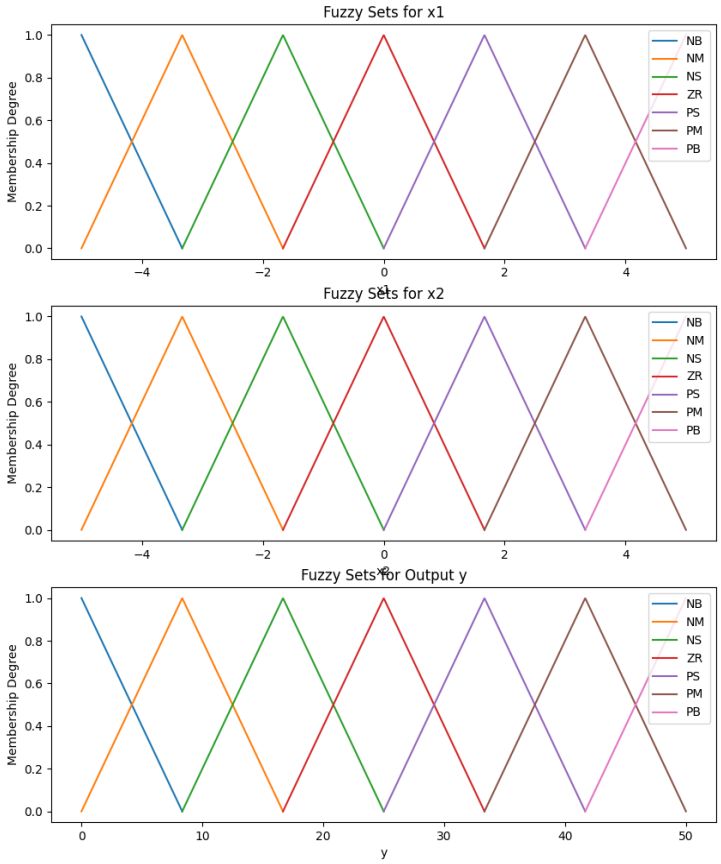
در این بلاک داده های آموزشی که از قبل برای راحتی بیشتر به صورت یک فایل csv ذخیره شده بود را در متغیر data قرار داده ایم و برای اینکه مدل بتواند از داده ها استفاده کند با استفاده از کتابخانه pandas و تابع to_records داده ها را از دیتافریم به آرایه های نامپای تغییر داده ایم و سپس data را به کلاس پاس میدهیم تا فازی ست های مورد نیاز را برای کلاس بسازد و در آبجکت ساخته شده را در model ذخیره میکنیم، سپس با همان داده های آموزشی تابع generate rules را صدا میزنیم که قوانین مورد نیاز ساخته شوند و با استفاده از یک حلقه برای تفهیم بهتر قوانین را پرینت میگیریم.

در انتها برای تست مدل و قوانین ساخته شده از دیتاست تست که ساخته شده بود استفاده میکنیم و دوباره از حالت دیتافریم اول آن را به ارایه های نامپای تبدیل میکنیم.

سپس اول تابع plot_fuzzy_sets را صدا زده که از تعریف درست فازی ست ها مطمئن شویم و به صورت نمودار آن ها را مشاهده کنیم. سپس داده های تست را به تابع predict_plot_3d_output پاس داده ایم که به صورت سلسله وار بقیه تابع هارا صدا زده و داده های crisp را تولید میکند و در اخر نمودار از داده های پیش بینی شده به ما میدهد

نتایج:

فازی ست های تشکیل شده:



قوانین ساخته شده:

Generated Rule Base:

If x1 is NB and x2 is NB, then output is PB
If x1 is NM and x2 is NB, then output is PS
If x1 is NS and x2 is NB, then output is ZR
If x1 is ZR and x2 is NB, then output is ZR
If x1 is PS and x2 is NB, then output is ZR
If x1 is PM and x2 is NB, then output is PS

If x1 is PB and x2 is NB, then output is PB
If x1 is NB and x2 is NM, then output is PS
If x1 is NM and x2 is NM, then output is ZR
If x1 is NS and x2 is NM, then output is NS
If x1 is ZR and x2 is NM, then output is NM
If x1 is PS and x2 is NM, then output is NS
If x1 is PM and x2 is NM, then output is ZR
If x1 is PB and x2 is NM, then output is PS
If x1 is NB and x2 is NS, then output is ZR
If x1 is NM and x2 is NS, then output is NS
If x1 is NS and x2 is NS, then output is NM
If x1 is ZR and x2 is NS, then output is NB
If x1 is PS and x2 is NS, then output is NM
If x1 is PM and x2 is NS, then output is NS
If x1 is PB and x2 is NS, then output is ZR
If x1 is NB and x2 is ZR, then output is ZR
If x1 is NM and x2 is ZR, then output is NM
If x1 is NS and x2 is ZR, then output is NB
If x1 is ZR and x2 is ZR, then output is NB
If x1 is PS and x2 is ZR, then output is NB
If x1 is PM and x2 is ZR, then output is NM
If x1 is PB and x2 is ZR, then output is ZR

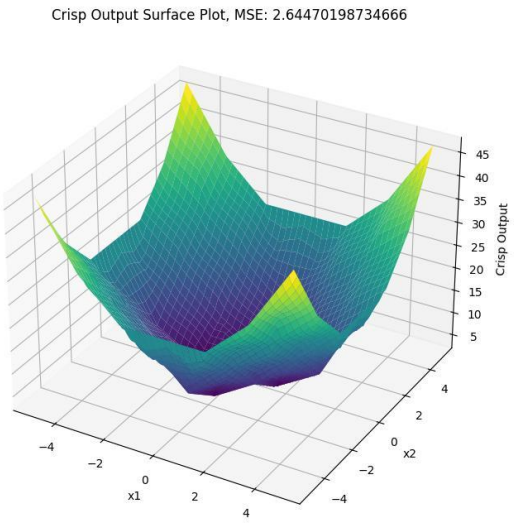
If x1 is NB and x2 is PS, then output is ZR
If x1 is NM and x2 is PS, then output is NS
If x1 is NS and x2 is PS, then output is NM
If x1 is ZR and x2 is PS, then output is NB
If x1 is PS and x2 is PS, then output is NM
If x1 is PM and x2 is PS, then output is NS
If x1 is PB and x2 is PS, then output is ZR
If x1 is NB and x2 is PM, then output is PS
If x1 is NM and x2 is PM, then output is ZR
If x1 is NS and x2 is PM, then output is NS
If x1 is ZR and x2 is PM, then output is NM
If x1 is PS and x2 is PM, then output is NS
If x1 is PM and x2 is PM, then output is ZR
If x1 is PB and x2 is PM, then output is PS
If x1 is NB and x2 is PB, then output is PB
If x1 is NM and x2 is PB, then output is PS
If x1 is NS and x2 is PB, then output is ZR
If x1 is ZR and x2 is PB, then output is ZR
If x1 is PS and x2 is PB, then output is ZR
If x1 is PM and x2 is PB, then output is PS
If x1 is PB and x2 is PB, then output is PB

مقدار خطا یا MSE:

Mean Squared Error (MSE): 2.64470198734666

```
If x1 is PM and x2 is PB, then output is PS
If x1 is PB and x2 is PB, then output is PB
Mean Squared Error (MSE): 2.64470198734666
PS F:\Work\Fuzzy Learning>
```

Crisp value های پیشبینی شده (نمودار):



مقادیر آموزشی (نمودار):