پروژه درس فازی

مرحله 1 و 2

استاد براتی

شبیه سازی یک تابع به روش ونگ مندل و بهینه سازی از الگوریتم SA

پویان رضائی

شماره دانشجویی 403149294

**راهنمای اجرای پروژه**

1-لطفا ابتدا فولدر پروژه در محیط برنامه نویسی باز شود

2- سپس فایل fuzzy\_model.py اجرا شود (اگر با خطایی مواجه شدید در اجرای برنامه اول فایل data\_set.py اجرا شود سپس فایل fuzzy\_model.py)

3- تمام فایل های دیتاست های آموزشی و تمرینی و عکس های نمودار های تشکیل شده ابتدا نمایش داده میشوند (باید تک تک نمودار ها بسته شوند پس از نمایش تا برنامه ادامه پیدا کند) سپس در فولدر ذخیره می شوند پس از اجرا

**آماده سازی داده (Data preparation):**

**فایل data\_set.py:**

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

def generate\_data\_and\_plot(num\_points, filename, title, is\_training=True):

    if is\_training:

        x\_values = np.linspace(-5, 5, num\_points)

        x1, x2 = np.meshgrid(x\_values, x\_values)

        output = x1\*\*2 + x2\*\*2

        data = pd.DataFrame({'x1': x1.flatten(), 'x2': x2.flatten(), 'F(x1, x2)': output.flatten()})

    else:

        x1\_values = np.random.uniform(-5, 5, num\_points)

        x2\_values = np.random.uniform(-5, 5, num\_points)

        output = x1\_values\*\*2 + x2\_values\*\*2

        data = pd.DataFrame({'x1': x1\_values, 'x2': x2\_values, 'F(x1, x2)': output})

    data.to\_csv(filename, index=False)

    fig = plt.figure()

    ax = fig.add\_subplot(111, projection='3d')

    if is\_training:

        ax.plot\_surface(x1, x2, output, cmap='viridis')

    else:

        ax.scatter(data['x1'], data['x2'], data['F(x1, x2)'], color='red', marker='o')

    ax.set\_xlabel('X Axis')

    ax.set\_ylabel('Y Axis')

    ax.set\_zlabel('Z Axis')

    plt.title(f'{title} 3D Surface Plot' if is\_training else f'{title} Test Data Scatter Plot')

    plt.savefig(f'{title} {"3D Surface Plot" if is\_training else "Test Data Scatter Plot"}.jpg')

    plt.show()

generate\_data\_and\_plot(41, "training\_data.csv", "Training", is\_training=True)

generate\_data\_and\_plot(168, "test\_data.csv", "Test", is\_training=False)

در این فایل مرحله آماده سازی دیتا اتفاق می افتد که با استفاده از تابع generate\_data\_and\_plot که دارای 4 ورودی میباشد یک ورودی num\_points و filename و title و is\_training . متغیر is\_training که مشخص میکند که داده ساخته شده برای آموزش هست یا برای تست.

Title برای تیتر پلات های رسم شده استفاده می شود

Filename برای مشخص کردن اسم فایل csv ذخیره شده

Num\_points برای داده های تمرینی نشان دهنده مقدار داده ها برای x1 و x2 هست یعنی 41 که ترکیب آن ها 1681 ردیف میشود و برای داده های تست نشان دهنده مقدار داده مورد نیاز است که به صورت همسان پخش شده اند یعنی 168

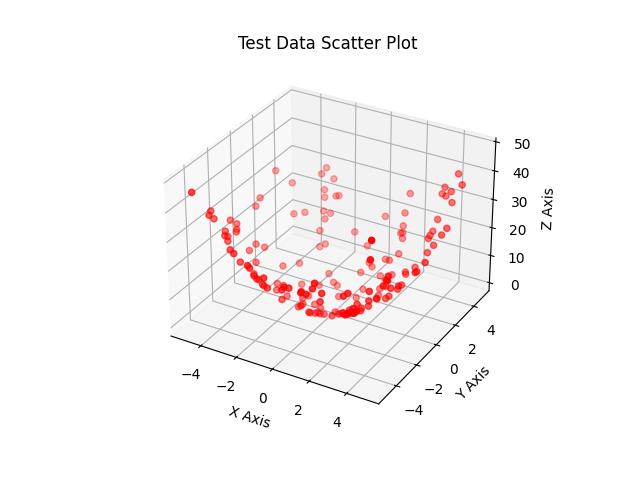
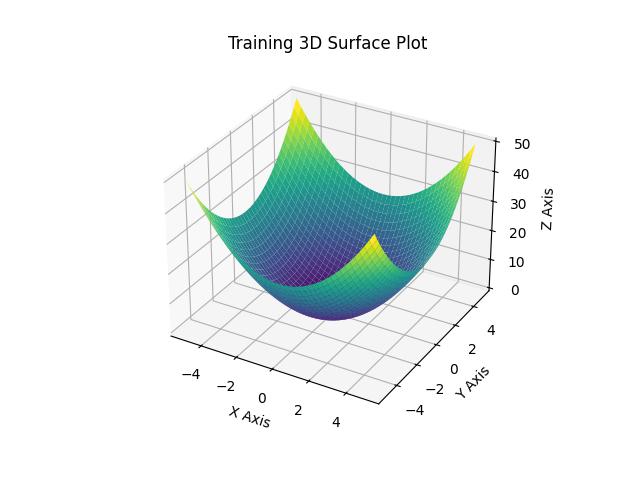
در این تابع از یک if بلاک استفاده شده است که اگر داده های آموزشی باشد با استفاده از تابع کتابخانه numpy یعنی linespace در بازه داده شده دیتا هایی با تعداد داده شده با فواصل یکسان ساخته می شود و با استفاده از تابع meshgrid ترکیب تمام این داده ها به صورت ماتریس ساخته می شود و با حساب کردن مجموع توان 2 متغیر های x1 و x2 خروجی آن ها برای مقایسه با داده های پیش بینی شده ساخته می شود و سپس با استفاده از کتابخانه pandas داده هارا به صورت دیتافریم تبدیل میکنیم و برای اینکار داده ها حتما باید تبدیل به ارایه تک بعدی بشوند به همین دلیل از تابع flatten استفاده شده است.

اگر برای درست کردن داده ای تست بخواهیم از این تابع استفاده کنیم بلاک else رخ میدهد که در آن دو متغیر اولیه با استفاده از numpy مقادیری رندوم و یونیفورم به تعداد 168 ساخته می شود و دوباره مجموع توان 2 آن ها محاسبه شده و در اینجا آرایه ها تک بعدی هستند و به صورت مستقیم بدون نیاز به تبدیل آن ها را تبدیل به دیتافریم کرده و خارج از بلاک های شرطی دیتا ها به صورت csv ذخیره می شوند.

برای تفهیم بیشتر از دیتا ها پلات هایی کشیده می شود که بتوان پخش داده ها را مشاهده کرد

(این بخش صرفا برای آماده سازی csv ها و پلات هایی برای فهمیدن بهتر ساختار دیتا های train و test)

**نمودار های خروجی:**



**خلاصه پیاده سازی:**

**فایل fuzzy\_model.py:**

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)

        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 توضبح 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:

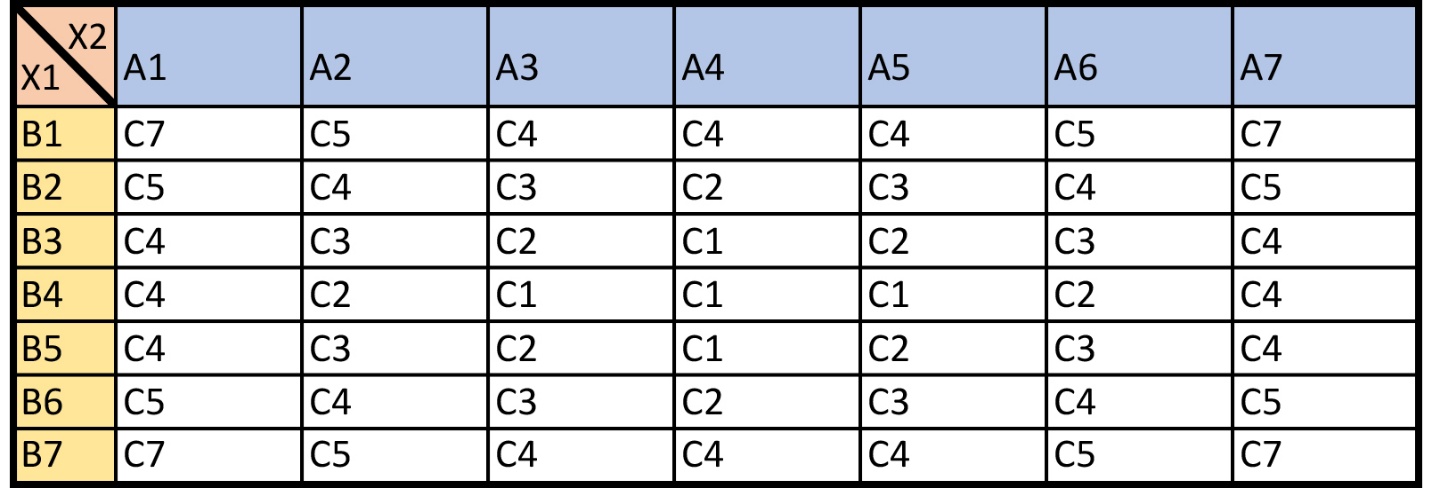
            if antecedent not in rule\_dict or rule\_dict[antecedent][1] < weight:

                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:**

    def predict(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}")

        return mse, predictions, targets, x1\_values, x2\_values

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)=x12+x22 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 ماننده 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 x1 و x2  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.

**تابع plot\_3d\_output:**

    def plot\_3d\_output(self,mse, predictions, targets,x1\_values, x2\_values,title):

        fig1 = plt.figure(figsize=(12, 6))

        ax1 = fig1.add\_subplot(121, projection='3d')

        ax1.scatter(x1\_values, x2\_values, predictions, c='blue', marker='o', label='Predicted')

        ax1.set\_title(f'3D Scatter Plot of Predicted Values Of {title}.png')

        ax1.set\_xlabel('x1')

        ax1.set\_ylabel('x2')

        ax1.set\_zlabel('Predicted Output')

        ax1.legend()

        ax2 = fig1.add\_subplot(122, projection='3d')

        ax2.scatter(x1\_values, x2\_values, targets, c='red', marker='^', label='Target')

        ax2.set\_title(f'3D Scatter Plot of Target Values Of {title}.png')

        ax2.set\_xlabel('x1')

        ax2.set\_ylabel('x2')

        ax2.set\_zlabel('Target Output')

        ax2.legend()

        plt.tight\_layout()

        plt.savefig(f"3D\_Scatter\_Target\_vs\_Predicted\_{title}.jpg")

        plt.show()

        plt.figure(figsize=(10, 6))

        plt.plot(range(len(targets)), targets, label='Target Values', color='red', linestyle='-', marker='o')

        plt.plot(range(len(predictions)), predictions, label='Predicted Values', color='blue', linestyle='--', marker='x')

        plt.title(f'Target vs Predicted {title} Values (MSE: {mse:.4f})')

        plt.xlabel('Test Case Index')

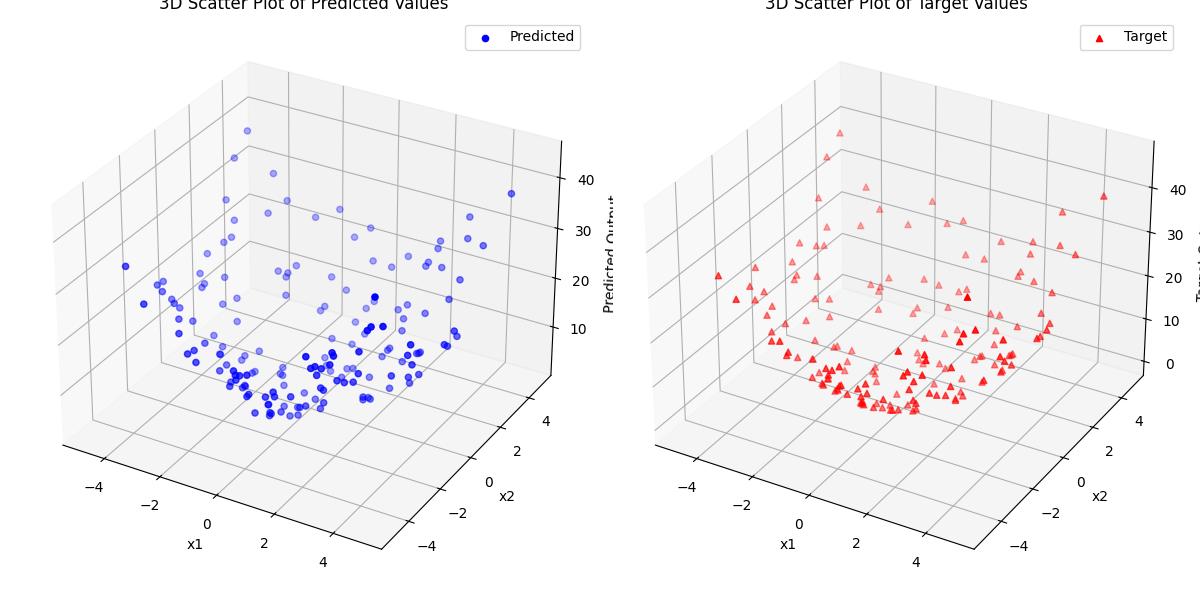
        plt.ylabel('Output Values')

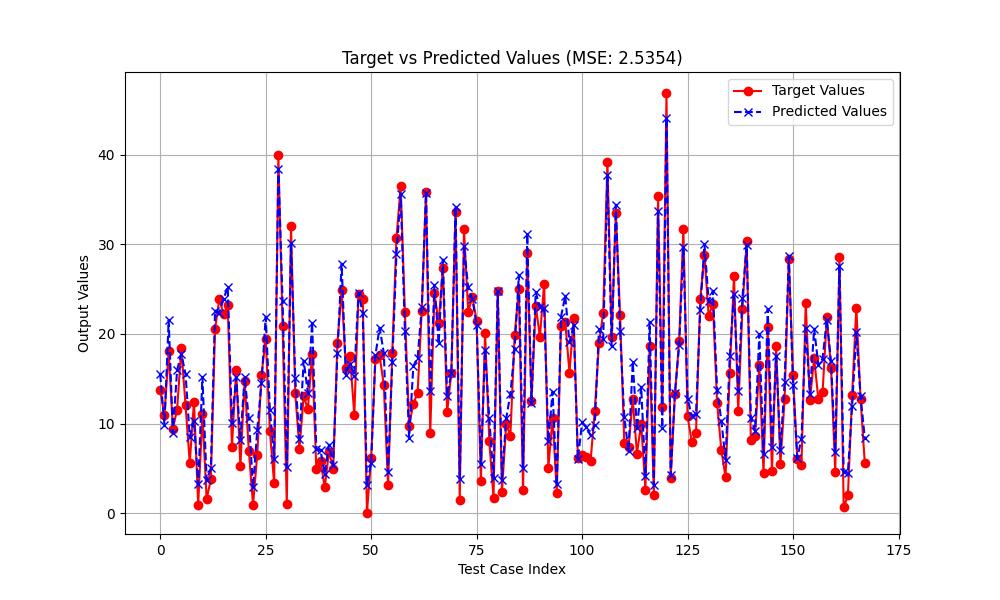
        plt.legend()

        plt.grid(True)

        plt.savefig(f"Line\_Plot\_Target\_vs\_Predicted {title}.jpg")

        plt.show()

در خط های بعد با استفاده از تابع plot\_3d\_ouput و خروجی های گرفته شده از تابع predict برای تفهیم بهتر خروجی دو پلات سه بعدی یکی برای داده های پیش بینی شده و یکی برای داده های حقیقی ساخته می شود با کمک کتابخانه matplotlib که در ان می توان اختلاف های آن را مشاهده کرد. یک نمودار خطی ها از داده های پیش بینی شده و حقیقی نیز ساخته می شود که بتوان اختلاف آن هارا بهتر متوجه شد



**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*

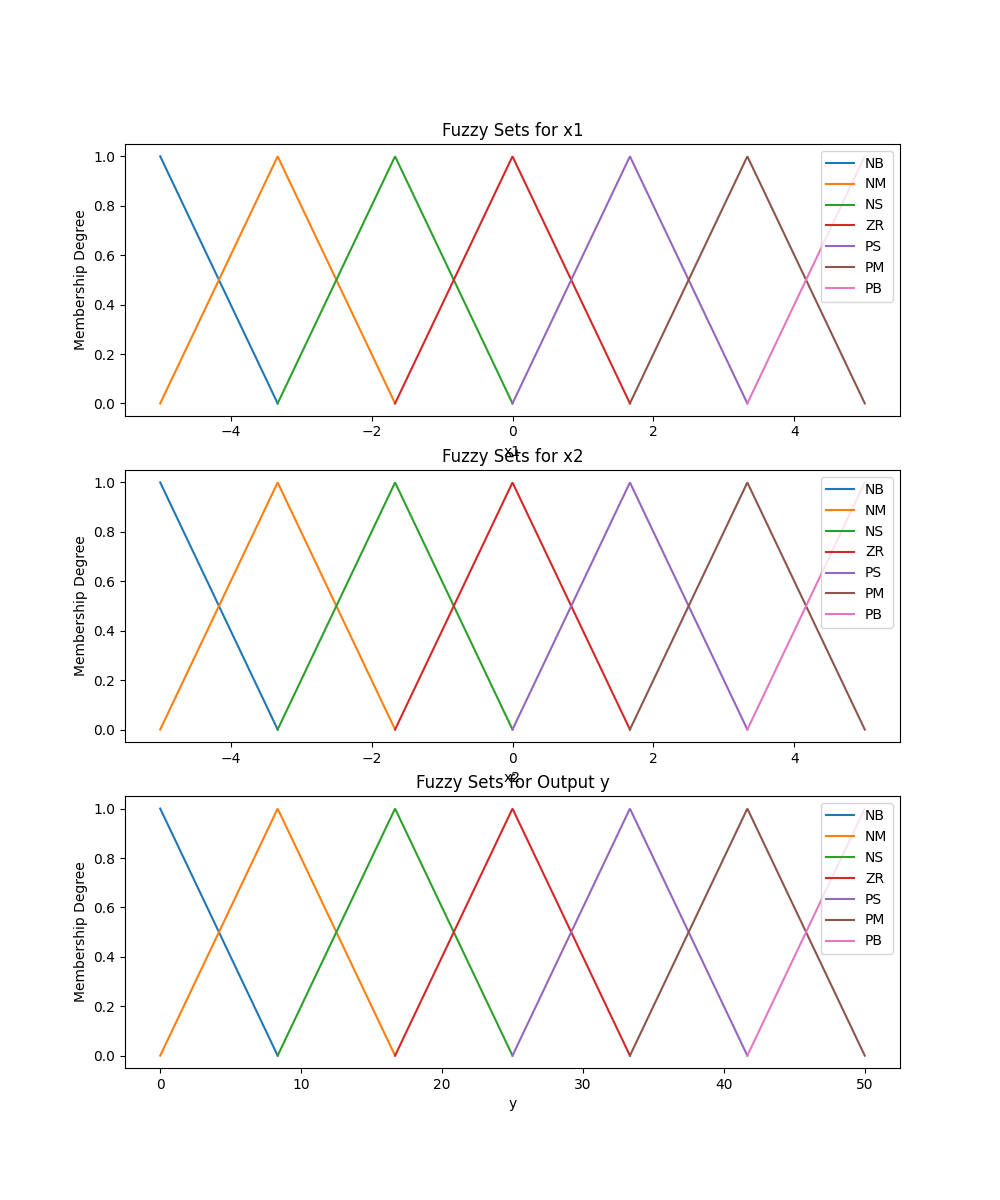
        plt.savefig("Fuzzy sets Plots")

        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 x1 و فازی ست دوم یعنی x2 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 دست بیاوریم. سپس نمودار را براساس x و y به دست میاوریم و از label های فازی ست برای راهنما ساب پلات ها میگیریم.



**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}")

    train\_mse\_before\_optimization, predictions, targets, x1\_values, x2\_values = model.predict(data)

    model.plot\_3d\_output(train\_mse\_before\_optimization, predictions, targets, x1\_values, x2\_values,"Train\_Before\_Optimization")

    optimized\_model, best\_mse = simulated\_annealing(model, data)

    train\_mse\_after\_optimization, predictions, targets, x1\_values, x2\_values = optimized\_model.predict(data)

    optimized\_model.plot\_3d\_output(train\_mse\_after\_optimization, predictions, targets, x1\_values, x2\_values,"Train\_Before\_Optimization")

    print(f"Best MSE after optimization: {best\_mse}")

    print(f"Train MSE Before optimaztion:{train\_mse\_before\_optimization}")

    print(f"Train MSE After optimaztion:{train\_mse\_after\_optimization}")

    model.plot\_fuzzy\_sets("Original Model")

    optimized\_model.plot\_fuzzy\_sets("Optimized Model")

    mse=0

    for i in range(100):

        x1\_values = np.random.uniform(-5, 5, 41)

        x2\_values = np.random.uniform(-5, 5, 41)

        output = x1\_values\*\*2 + x2\_values\*\*2

        data = pd.DataFrame({'x1': x1\_values, 'x2': x2\_values, 'F(x1, x2)': output})

        data = data.to\_records(index=False)

        test\_mse\_before\_optimization, predictions, targets, x1\_values, x2\_values = model.predict(data)

        print(f"The PRE-Optimized MSE {test\_mse\_before\_optimization}")

        mse = mse + test\_mse\_before\_optimization

        if(i%99==0):

            model.plot\_3d\_output(test\_mse\_before\_optimization, predictions, targets, x1\_values, x2\_values,"Test Mse PRE-Optimised")

    test\_mse\_before\_optimization = mse/100

    mse=0

    for i in range(100):

        x1\_values = np.random.uniform(-5, 5, 168)

        x2\_values = np.random.uniform(-5, 5, 168)

        output = x1\_values\*\*2 + x2\_values\*\*2

        data = pd.DataFrame({'x1': x1\_values, 'x2': x2\_values, 'F(x1, x2)': output})

        data = data.to\_records(index=False)

        test\_mse\_optimized, predictions, targets, x1\_values, x2\_values = optimized\_model.predict(data)

        print(f"The Optimized MSE {test\_mse\_optimized}")

        mse = mse + test\_mse\_optimized

        if(i%99==0):

            optimized\_model.plot\_3d\_output(test\_mse\_optimized, predictions, targets, x1\_values, x2\_values,"Test Mse Optimised")

    test\_mse\_optimized = mse/100

    print(f"test\_mse\_before\_optimization:{test\_mse\_before\_optimization}")

    print(f"test\_mse\_optimized: {test\_mse\_optimized}")

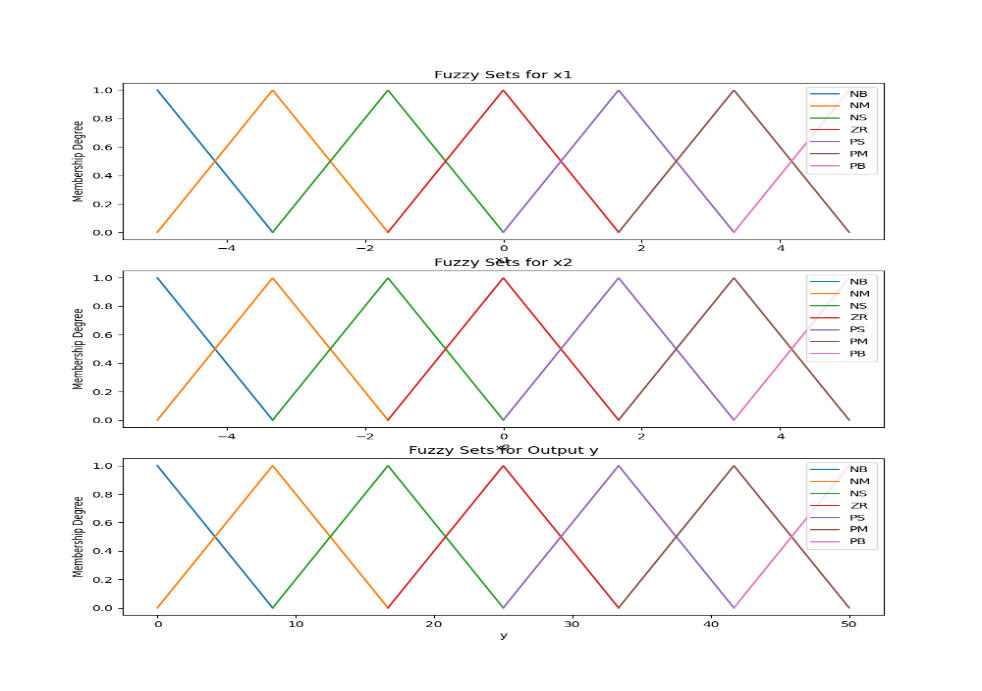
در این بلاک داده های آموزشی که از قبل برای راحتی بیشتر به صورت یک فایل csv ذخیره شده بود را در متغیر data قرار داده ایم و برای اینکه مدل بتواند از داده ها استفاده کند با استفاده از کتابخانه pandas و تابع to\_records داده ها را از دیتافریم به آرایه های نامپای تغییر داده ایم و سپس data را به کلاس پاس میدهیم

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.

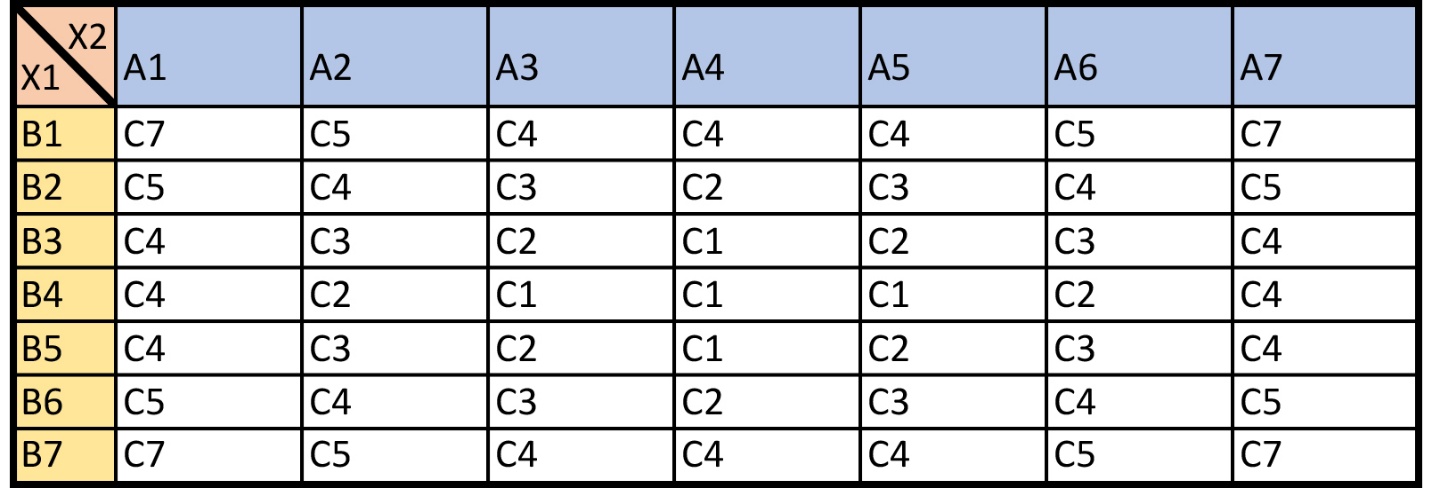
سپس تابع predict را صدا زده و مقادیر crisp پیش بینی شده برای داده های آموزشی را تشکیل میدهیم و با استفاده از مقادیری که تابع به ما برمیگرداند تابع plot\_3d\_output را استفاده کرده و نتایج را به صورت پلات ذخیره و نمایش میدهد.

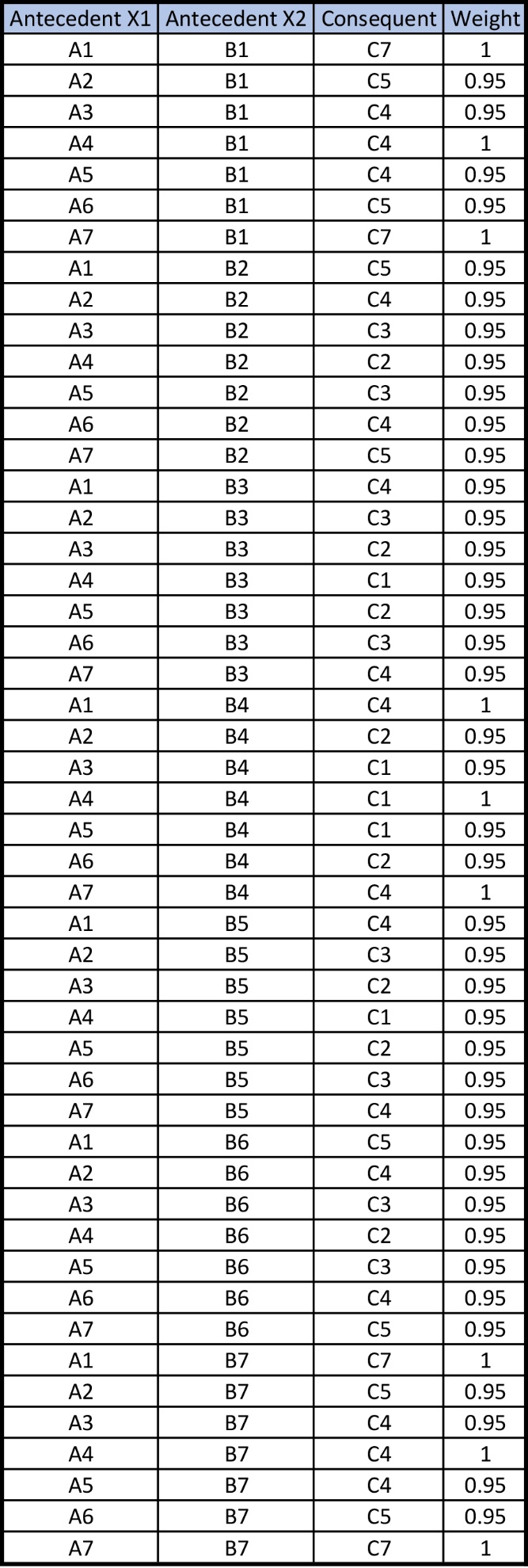
سپس برای داده های تست هم از یک حلقه 100 تایی استفاده کرده که در آن هربار یک سری داده رندوم uniform 168 داده تولید میکند برای x1 و x2 سپس ouput آن هارا حساب کرده و تبدیل به دیتا فریم کرده و سپس دیتافریم را تبدیل به یک ماتریس کرده و آن را به تابع predict کلاس ما داده سپس mse ها را هردور حساب کرده و در دور اخر حلقه برای نمونه پلات های مورد نیاز را توسط تابع plot\_3d\_ouput تشکیل می شوند و در اخر میانگین تمام mse های به دست اماده حساب میشود (اینکار برای این است که تاثیر رندوم بودن داده های تست را در نتیجه mse کمرنگ تر شود). و تمام این مراحل برای مدل optimize شده ما نیز تکرار می شود.

**نتایج:**

**فازی ست های تشکیل شده:**

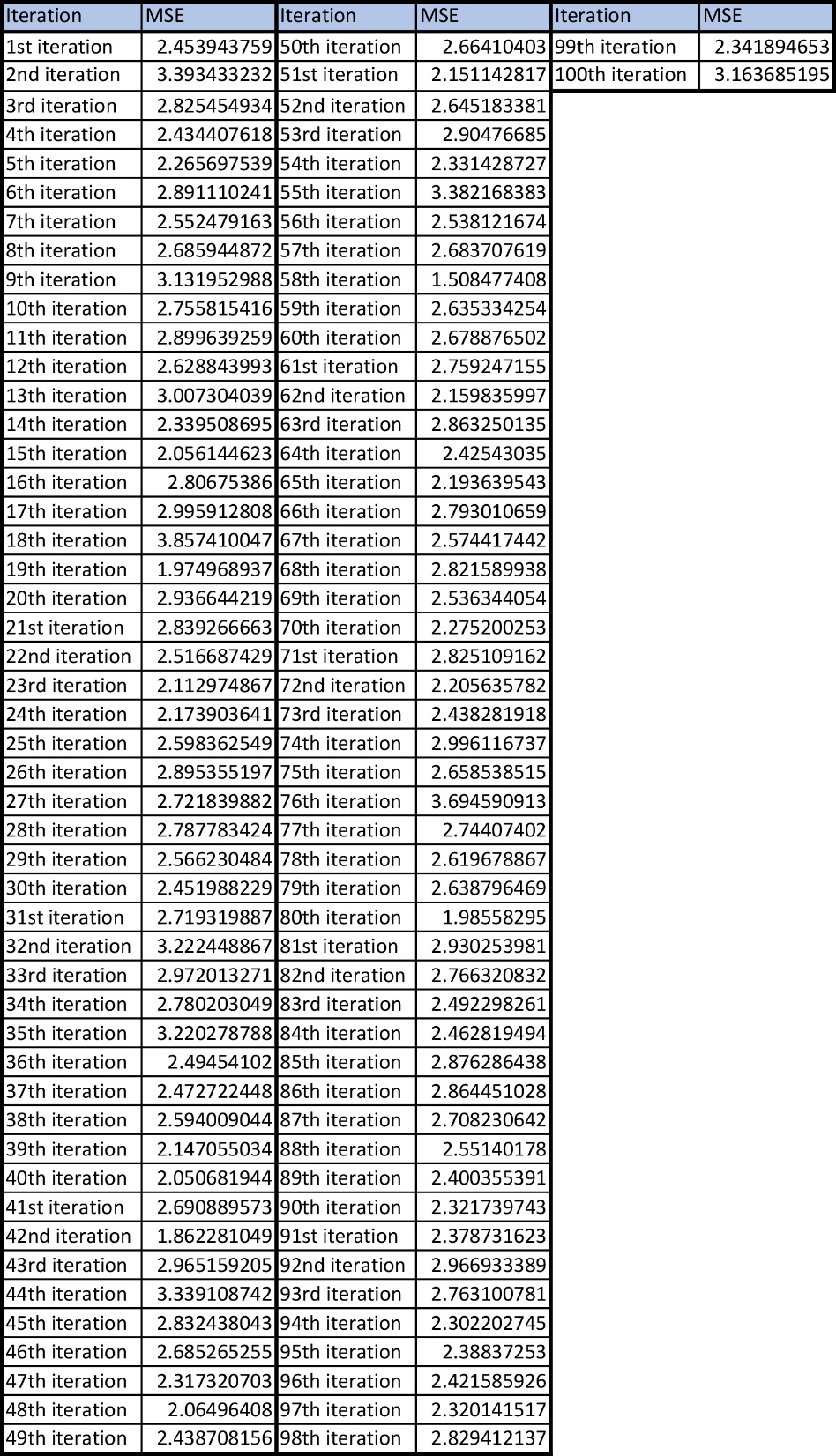
**قوانین ساخته شده:**

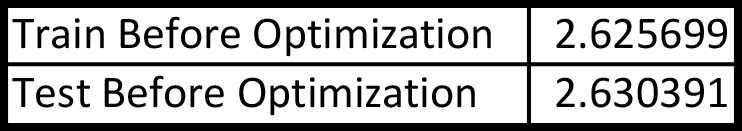
****

****

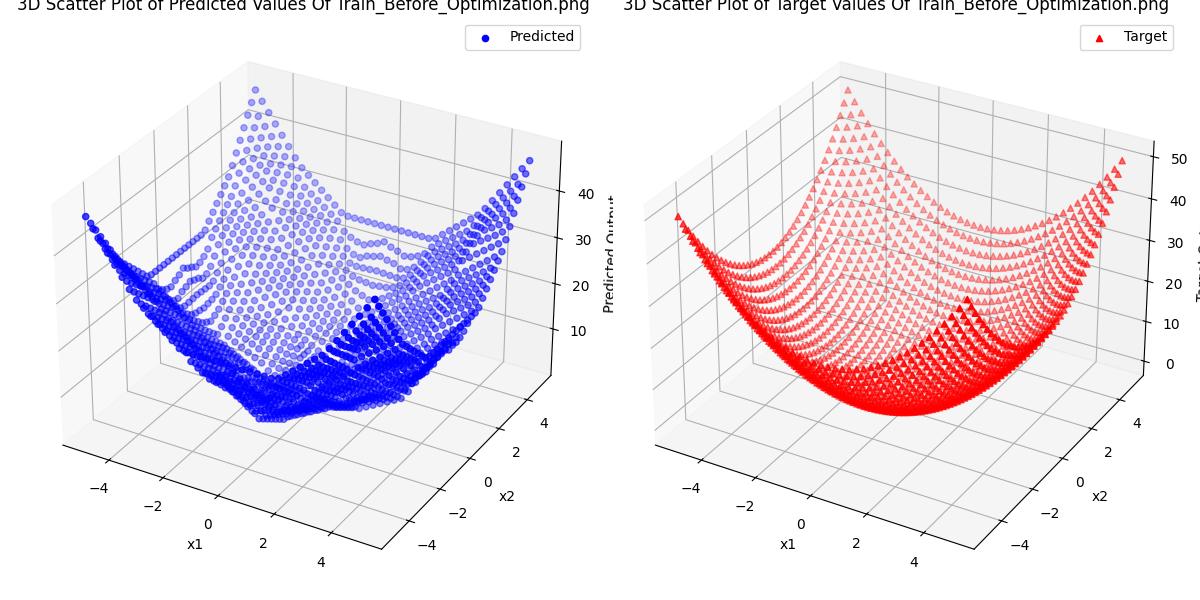
**مقدار خطا یا MSE:**

(با هربار تشکیل داده های تست به صورت رندوم این مقدار عوض میشود اما حدود 2 همیشه هست)

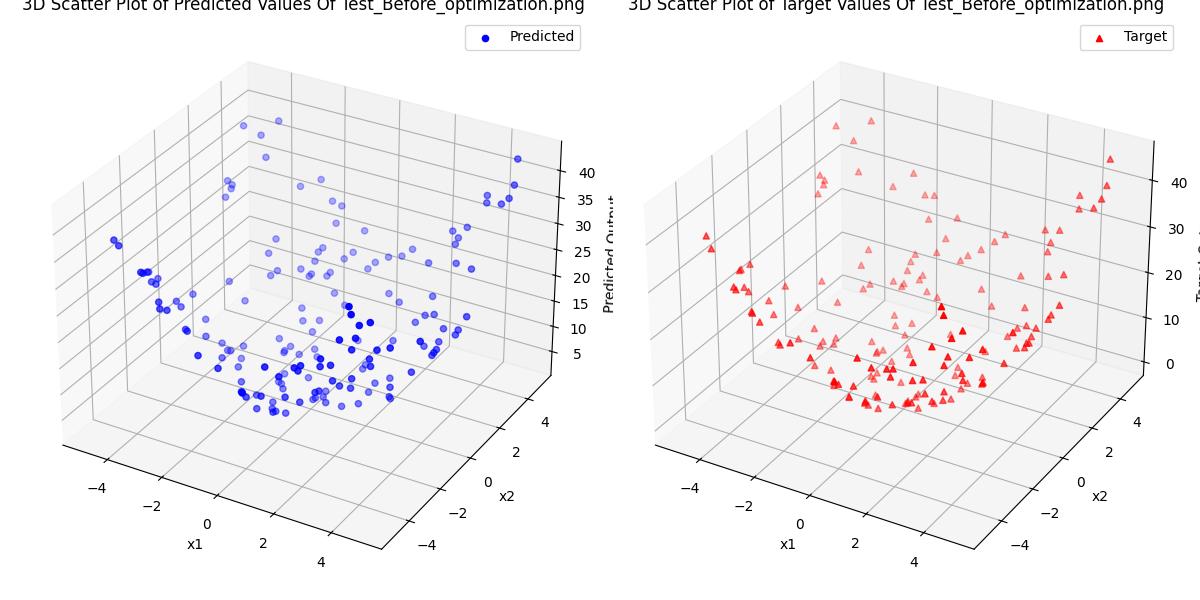




**Crisp value های پیشبینی شده (نمودار):**

**داده های آموزشی:**

**داده های تست:**



**پیاده سازی الگوریتم SA:**

در این قسمت با استفاده از الگوریتم SA تلاش می شود که مدل خود را تنظیم کرده و MSE یا همان خطا را کاهش دهیم که با استفاده از 3 تابع پیاده سازی شده است.

**تابع simulated\_annealing:**

def simulated\_annealing(model, training\_data, initial\_temp=0.1, cooling\_rate=0.98, max\_iterations=500):

*"""*

*Optimize fuzzy sets using the Simulated Annealing (SA) algorithm.*

*Parameters:*

*model: FuzzyModel instance.*

*training\_data: Training dataset.*

*initial\_temp: Initial temperature for SA.*

*cooling\_rate: Cooling rate for temperature decay.*

*max\_iterations: Maximum number of iterations.*

*Returns:*

*Optimized FuzzyModel and the best MSE achieved.*

*"""*

    current\_model = copy.deepcopy(model)

    current\_mse = evaluate\_model(current\_model, training\_data)

    best\_model = copy.deepcopy(current\_model)

    best\_mse = current\_mse

    temperature = initial\_temp

    for iteration in range(max\_iterations):

        new\_model = copy.deepcopy(current\_model)

        choice = random.choice(['input\_x1', 'input\_x2', 'output'])

        if choice == 'input\_x1':

            fuzzy\_sets = new\_model.input\_fuzzy\_sets\_x1

        elif choice == 'input\_x2':

            fuzzy\_sets = new\_model.input\_fuzzy\_sets\_x2

        else:

            fuzzy\_sets = new\_model.output\_fuzzy\_sets

        label = random.choice(list(fuzzy\_sets.keys()))

        fuzzy\_set = list(fuzzy\_sets[label])

        perturbation\_range = 0.1 \* (fuzzy\_set[2] - fuzzy\_set[0])

        fuzzy\_set[0] += random.uniform(-perturbation\_range, perturbation\_range)

        fuzzy\_set[1] += random.uniform(-perturbation\_range, perturbation\_range)

        fuzzy\_set[2] += random.uniform(-perturbation\_range, perturbation\_range)

        fuzzy\_set[0] = max(fuzzy\_set[0], 0)

        fuzzy\_set[1] = max(fuzzy\_set[1], fuzzy\_set[0])

        fuzzy\_set[2] = max(fuzzy\_set[2], fuzzy\_set[1])

        fuzzy\_sets[label] = tuple(fuzzy\_set)

        if iteration % 100 == 0:

            print(f"Iteration {iteration}: Temp={temperature:.4f}, Current MSE={current\_mse:.6f}, Best MSE={best\_mse:.6f}")

            print(f"Current fuzzy set for {label}: {new\_model.input\_fuzzy\_sets\_x1[label]}")

        new\_mse = evaluate\_model(new\_model, training\_data)

        if new\_mse < current\_mse or random.uniform(0, 1) < acceptance\_probability(current\_mse, new\_mse, temperature):

            current\_model = new\_model

            current\_mse = new\_mse

            if new\_mse < best\_mse:

                best\_model = copy.deepcopy(new\_model)

                best\_mse = new\_mse

        temperature = initial\_temp / ( 1 + cooling\_rate \* iteration)

    return best\_model, best\_mse

ورودی های این تابع شامل یک مدل هست (مدل فازی)

داده های تمرینی

انرژی یا Temperature اولیه (به صورت پیش فرض 0.1)

نرخ کاهش Temperature (به صورت پیش فرض 0.98)

و مقدار iterations که باید اتفاق بیوفتد (به صورت پیش فرض 500)

در این کد ابتدا یک کپی عمیق یا همان deepcopy از مدل فازی اولیه گرفته می شود که تغییرات مورد نیاز روی مدل فازی اصلی اتفاق نیافتد و اینکه بتوانیم مدل هارا از این طریق مقایسه و بهترین مدل را پیدا کنیم. سپس از تابع evaluate\_model (در ادامه طرز کار این تابع توضیح داده می شود) استفاده میکنیم تا mse اولیه مدل خود را پیدا کنیم. سپس صرفا برای خوانایی بهتر کد از مدل یک کپی گرفته و در متغیری به نام best\_model ذخیره کرده و mse به دست اورده را به عنوان best\_mse ذخیره میکنیم و temperature اولیه هم را در متغیری به اسم temperature ذخیره میکنیم.

سپس با استفاده از یک حلقه به تعداد iterations های تعریف شده (یعنی 500 بار) شروع میکنیم به ازای هر دور حلقه یک کپی از مدل گرفته و به صورت رندوم از بین فازی ست های ورودی های x1 و x2 و فازی ست خروجی یکی انتخاب شده (اینکار برای جلوگیری از این است که الگوریتم روی فازی ست به خصوصی تمرکز نکند که سبب تغییر بیش از حد و اورفیت شود) و به اندازه حدود 10 درصد فاصله ابتدا و انتها دامنه perturbed خود انتخاب میکنیم که به هرکدام از مقادیر a, b, c به صورت رندوم از منفی شروع دامنه تا مثبت آن عددی انتخاب می شود و به فازی ست مثلثاتی ما اضافه می شود. و برای اینکه مطمئن باشیم که شرط های فازی ست های مثلثاتی برقرار است از تابع max برای پیدا کردن شروع یعنی همان a و قله یعنی b و پایان فازی ست یعنی c استفاده میکنیم.

پس از اینکه فازی ست های جدید برای مدل محاسبه شدند باز با استفاده از تابع evaluate\_model استفاده میکنیم تا mse جدید را به دست بیاوریم سپس با استفاده از یک بلاک if چک میکنیم که آیا mse جدید از قدیم کمتر است یا اینکه با استفاده از تابع acceptance\_probability (در ادامه طرز کار تابع توضیح داده می شود) چک می شود که احتمال قبول کردن مدل با mse بدتر چقدر هست (اینکار برای این هست که الگوریتم در مراحل اول در local minima گیر نکند و با قبول کردن مدلی با mse بدتر در مراحل بعد به یک mse گلوبال بهتر برسیم) اگر شرط ها برقرار باشد، mse current با mse جدید جایگزین می شوند و model\_current هم با مدل جدید جایگزین می شود.

در ادامه در بلاک if توضیح داده شده یک if دیگر نیز وجود دارد که چک میکند که mse جدید آیا از بهترین mse پیدا شده کمتر است و اگر شرط برقرار باشد از مدل یک کپی گرفته و به عنوان best\_model آن را ذخیره کرده و mse آن را به عنوان best\_mse ذخیره میکند (متغیر ریست می شود) و در آخر temperature طبق فرمول داده شده به مقداری کاهش پیدا میکند.

در آخر پس از اتمام حلقه متغیر بهترین mse یا همان best\_mse و بهترین مدل یا همان best\_model برگردانده می شود.

**تابع evaluated\_model:**

def evaluate\_model(model, data):

    predictions = []

    targets = []

    for x1, x2, y in data:

        memberships\_x1 = model.fuzzify\_input(x1, model.input\_fuzzy\_sets\_x1)

        memberships\_x2 = model.fuzzify\_input(x2, model.input\_fuzzy\_sets\_x2)

        output\_memberships = model.evaluate\_rules(memberships\_x1, memberships\_x2)

        crisp\_output = model.defuzzify\_center\_of\_average(output\_memberships)

        predictions.append(crisp\_output)

        targets.append(y)

    return model.calculate\_mse(predictions, targets)

در این تابع که ورودی ها:

مدل هست (model) که مدل فازی مورد نظر به ان پاس داده می شود

و ورودی دوم هم داده آموزشی ما هست

در این تابع صرفا از مدل ران گرفته میشود به خروجی آن با داده حقیقی ما مقایسه می شود و توضیحات این توابع در بخش اول توضیحات پروژه نوشته می شود.

**تابع acceptance\_probability:**

def acceptance\_probability(current\_mse, new\_mse, temperature):

    if new\_mse < current\_mse:

        return 1.0

    return np.exp((current\_mse - new\_mse) / temperature)

ورودی این تابع ها:

Mse موجود یا همان current\_mse

Mse جدید یا همان new\_mse

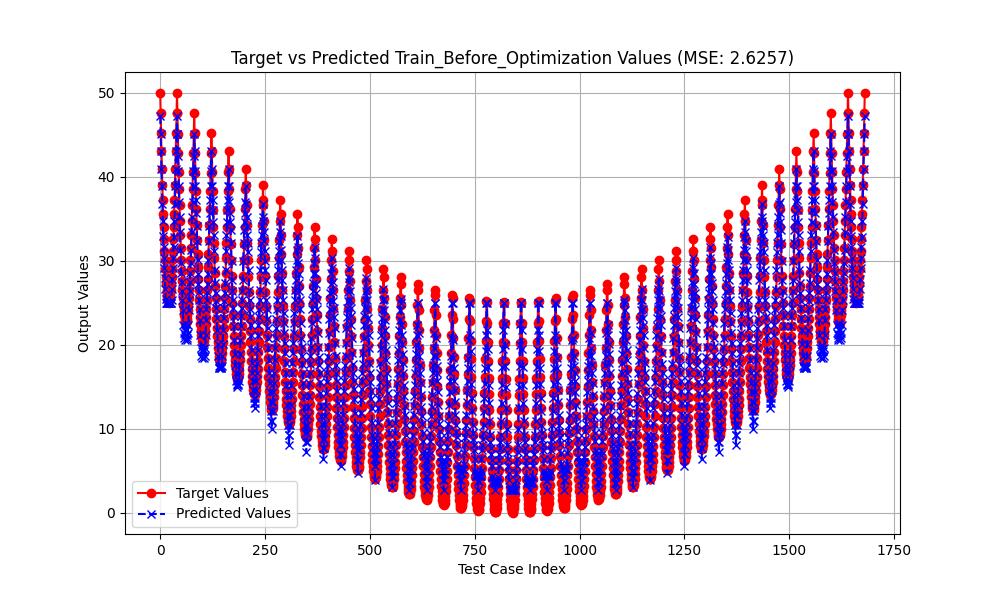
و انرژی یا همان temperature

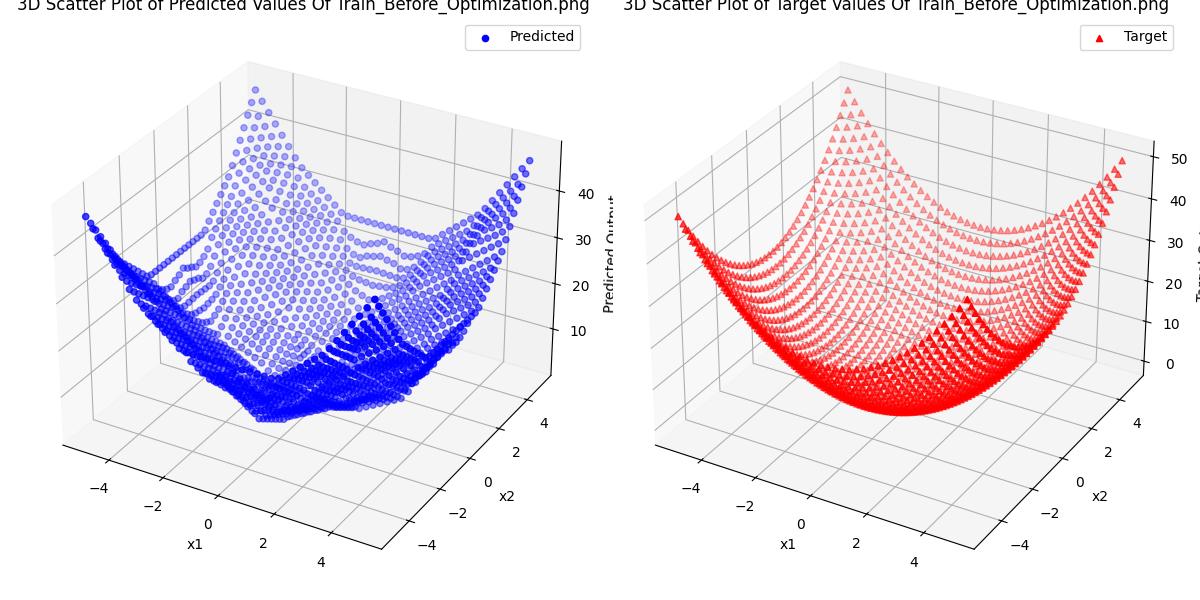
در این تابع اگر mse جدید پایین تر از mse موجود باشد 1 را برمیگرداند در غیر اینصورت طبق فرمول مقدار e به توان اختلاف mse ها تقسیم بر انرژی را برمیگرداند.

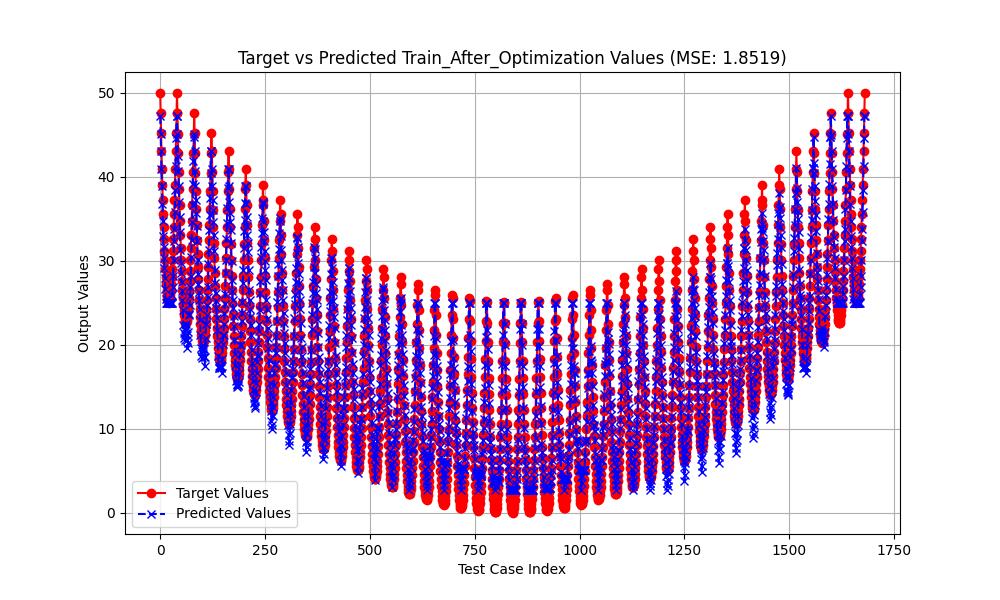
**نتایج:**

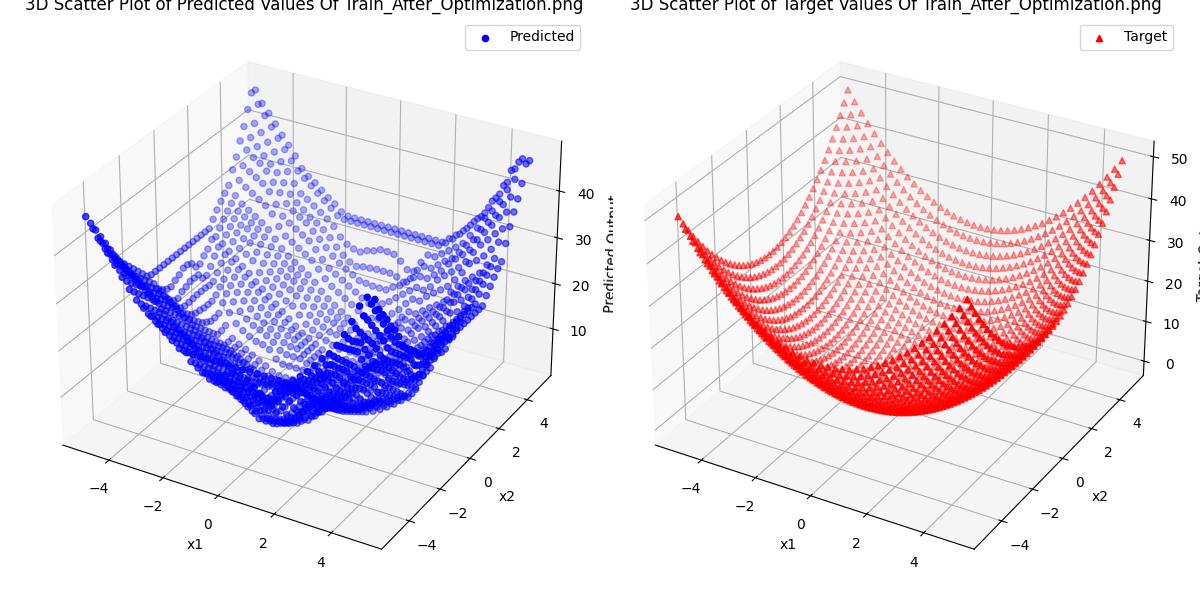
**به صورت کلی با استفاده از الگوریتم SA مقدار mse پایین تر امد و پیش بینی ها به داده های حقیقی نزدیک تر شد. برای سریع تر شدن برنامه تعداد iteration به صورت دیفالت 500 قرار داده شده و اگر بیشتر قرار داده شود مثلا 1000 به مقدار mse مطلوب تری میرسیم.**

**داده های تمرینی:**

**قبل از الگوریتم SA**

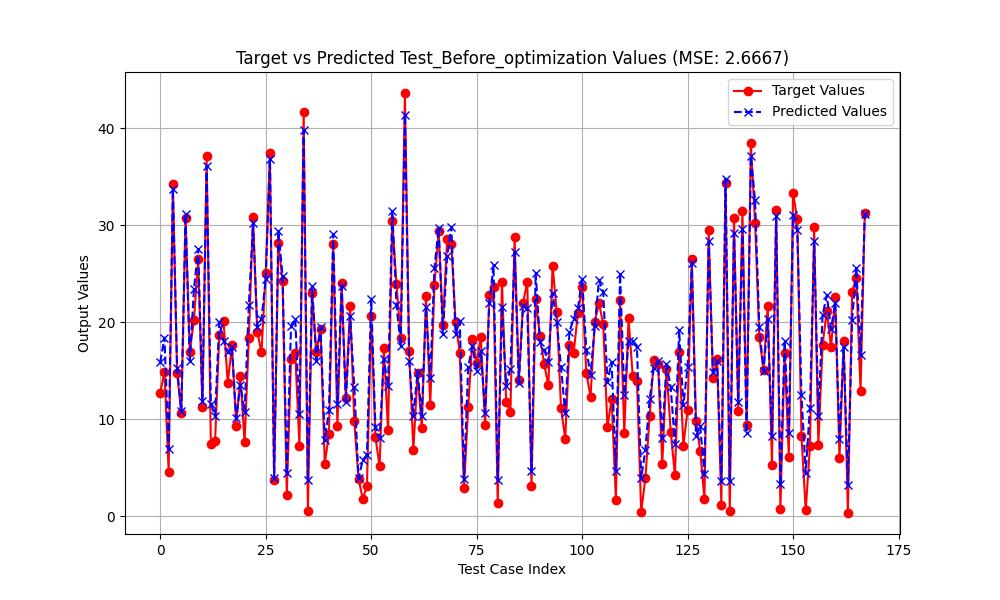
****

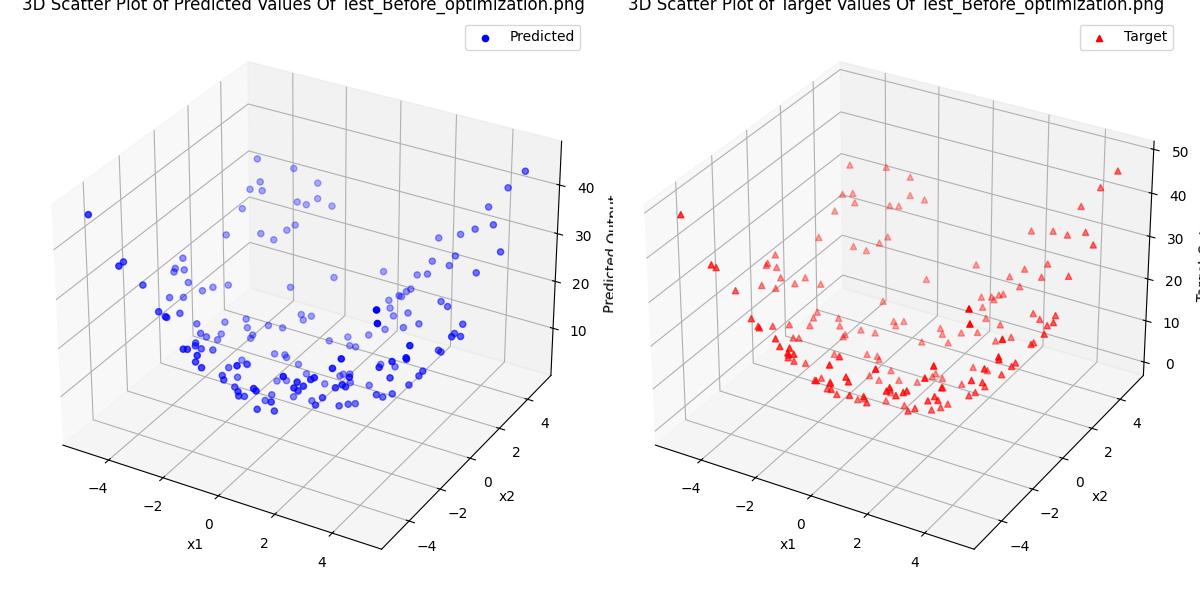
**بعد از الگوریتم SA**

****

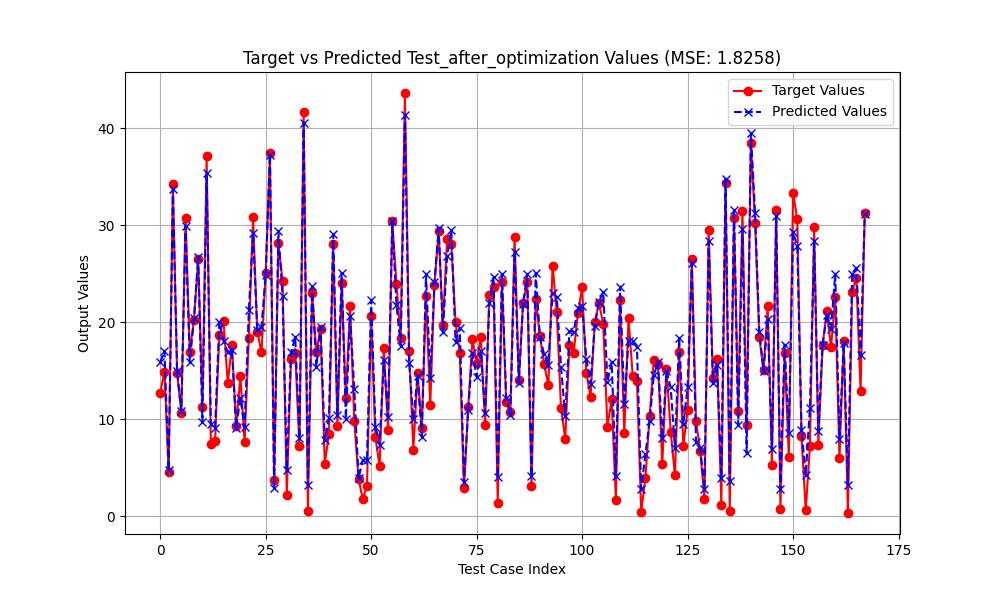
**داده های تست:**

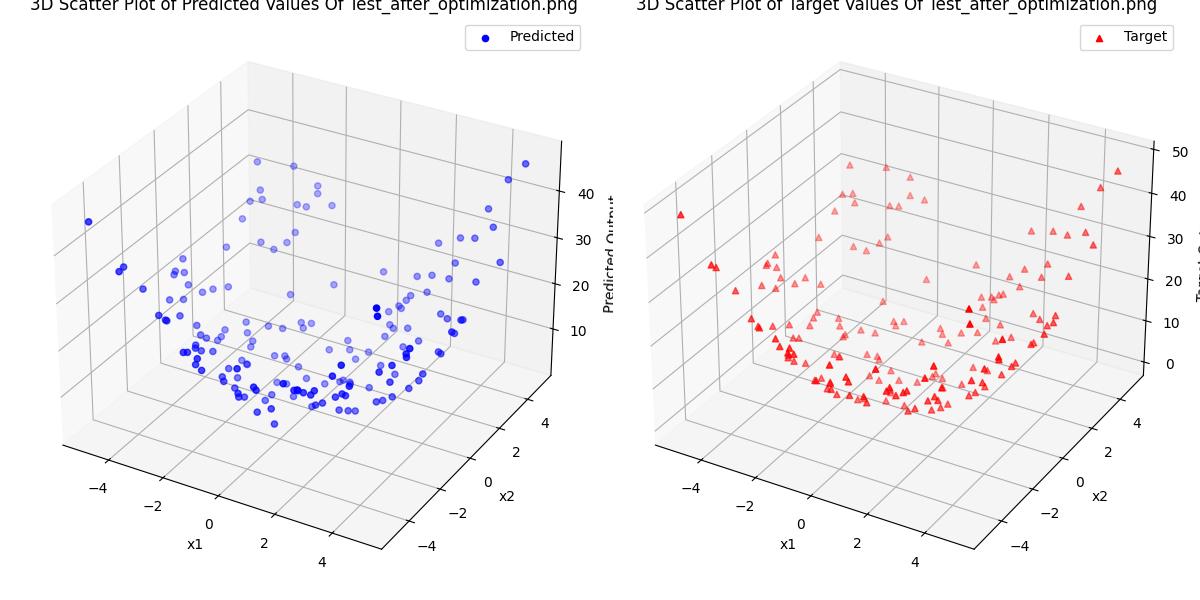
**قبل از الگوریتم SA**



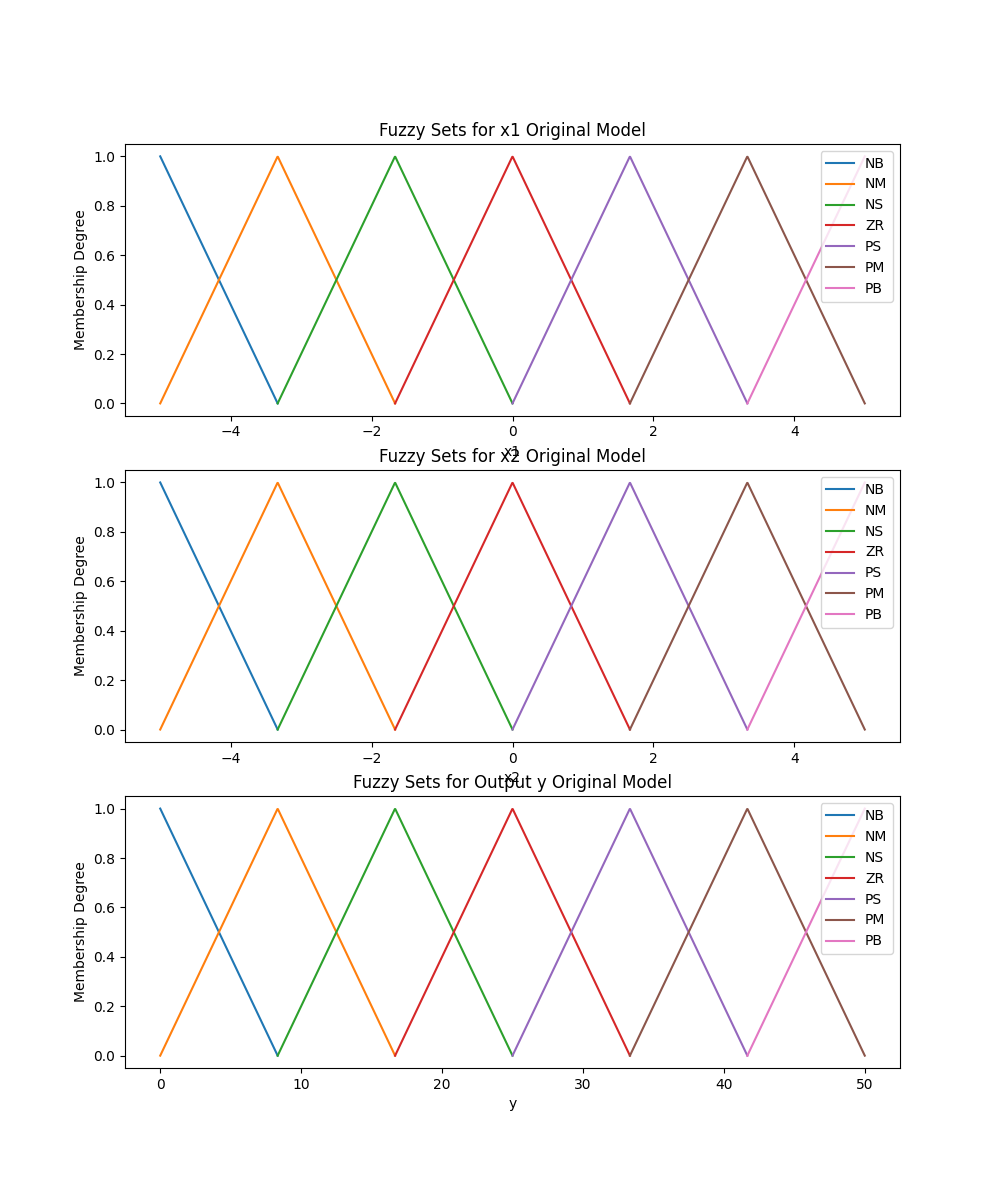


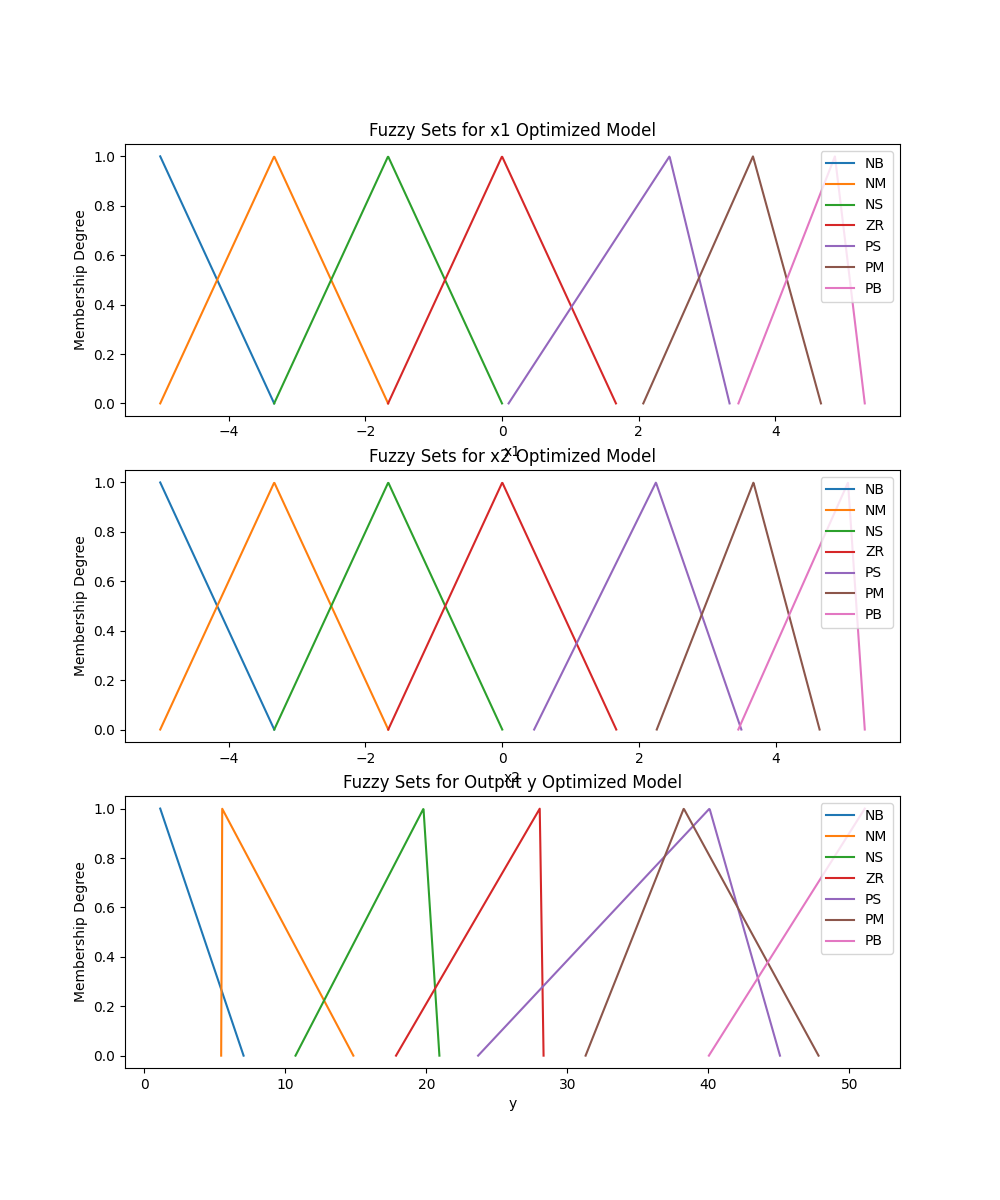
**بعد از الگوریتم SA**





**فازی ست ها:**

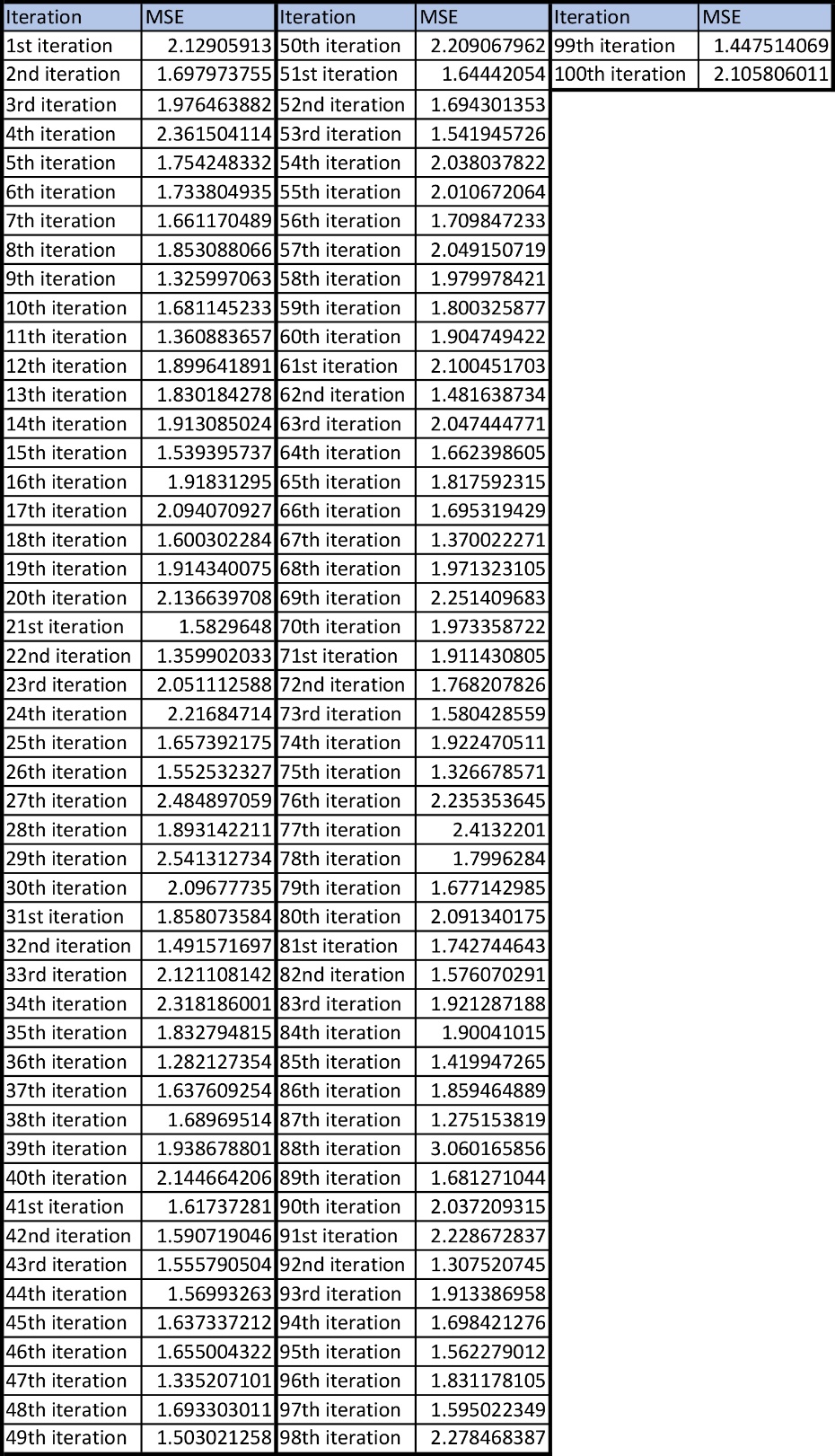
**قبل از الگوریتم SA**

**بعد از الگوریتم SA**

**خطاها (MSE):**

**خروجی SA برای داده های آموزش:**



**خروجی SA برای داده های تست:**