به نام خداوند رنگین کمان

دانشکده مهندسی برق

مینی پروژه دوم

مبانی سیستم های هوشمند

شیرین مهدی حاتم

پاییز ۱۴۰۲

سوال اول)

ابتدا دیتاست داده شده را آپلود کرده و بعد خواسته سوال که گفته است مجموع داده را به صورت ۲۰ به ۸۰ تقسیم کنید؛ تقسیم میکنیم:

```
[ ]  # 1
     x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2)
     x_train.shape, x_test.shape, y_train.shape, y_test.shape

     ((320, 2), (80, 2), (320,), (80,))
```

در ادامه باید با قاعده پرسپترون یک نورون را روی این دیتاست آموزش دهیم:

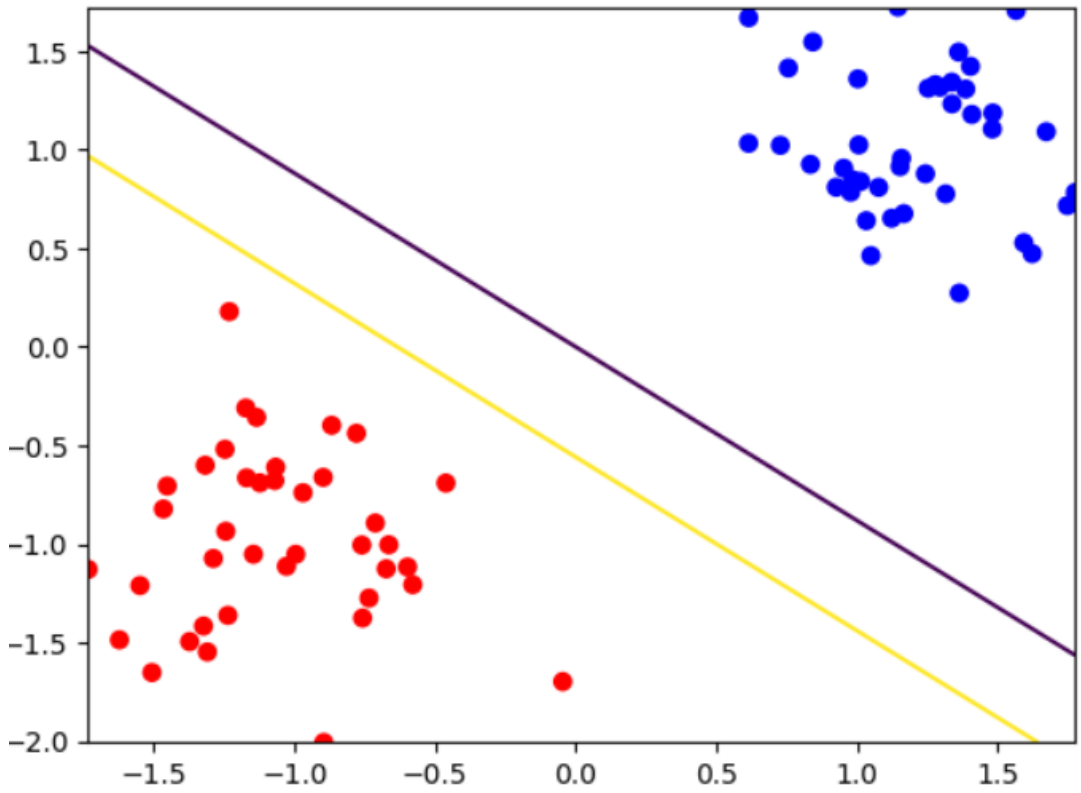
```
[ ]  # Initialize Perceptron classifier with a different threshold (example: 0.5)
     model = Perceptron()

     # Train the perceptron with the new threshold
     model.fit(x_train, y_train)
```

نتیجه رو مجموعه داده آزمون و آموزش برابر خواهد بود با:

```
[ ]  # 2
     # Accuracy on train and test data with the new threshold
     train_accuracy = model.score(x_train, y_train)
     test_accuracy = model.score(x_test, y_test)

     print(f"Accuracy on train set with new threshold: {train_accuracy}")
     print(f"Accuracy on test set with new threshold: {test_accuracy}")
     # Convert y_test to integer type
     y_test = y_test.astype(np.int)

     Accuracy on train set with new threshold: 1.0
     Accuracy on test set with new threshold: 1.0
```

ترشهولد تنظیم شده در این قسمت برابر با ۰.۵ بوده است.

در ادامه خواسته شده است تا مرز تصمیم بین داده های تست را رسم کنیم ، که این کار را با دو ویژگی x1 ,x2 باید انجام دهیم:



در قسمت بعدی خواسته شده تا مراحل بالا را با آستانه یا ترشهولد دیگری انجام دهیم و نتایج را مقایسه کنیم:

برای اینکار باید نورون را بصورت خودمان تعریف کنیم، پس به سراغ ساختن نورون از ابتدا می‌رویم:

```
def relu(x):
    return np.maximum(0, x)
def sigmoid(x):
    return 1/(1+np.exp(-x))
def tanh(x):
    return np.tanh(x)
def bce(y, y_hat):
    return np.mean(-(y*np.log(y_hat) + (1-y)*np.log(1-y_hat)))
def mse(y, y_hat):
    return np.mean((y - y_hat)**2)
```

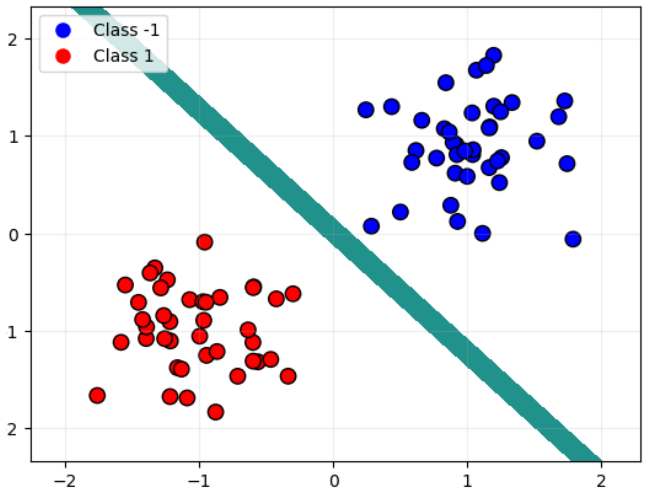
```
def accuracy(y, y_hat, t=0.5):
    y_hat = np.where(y_hat<t, 0, 1)
    acc = np.sum(y == y_hat) / len(y)
    return acc
class Neuron:

    def __init__(self, in_features, threshold, af=None, loss_fn=mse,
n_iter=100, eta=0.1, verbose=True):
        self.in_features = in_features
        # weight & bias
        self.w = np.random.randn(in_features, 1)
        self.threshold = threshold
        self.af = af
        self.loss_fn = loss_fn
        self.loss_hist = []
        self.w_grad = None
        self.n_iter = n_iter
        self.eta = eta
        self.verbose = verbose

    def predict(self, x):
        # x: [n_samples, in_features]
        y_hat = x @ self.w + self.threshold
        y_hat = y_hat if self.af is None else self.af(y_hat)
        return y_hat

    def decision_function(self, x):
        # x: [n_samples, in_features]
        y_hat = x @ self.w + self.threshold
        return y_hat

    def fit(self, x, y):
        for i in range(self.n_iter):
            y_hat = self.predict(x)
            loss = self.loss_fn(y, y_hat)
            self.loss_hist.append(loss)
            self.gradient(x, y, y_hat)
            self.gradient_descent()
            if self.verbose & (i % 10 == 0):
                print(f'Iter={i}, Loss={loss:.4}')

    def gradient(self, x, y, y_hat):
        self.w_grad = (x.T @ (y_hat - y)) / len(y)

    def gradient_descent(self):
```

```
        self.w -= self.eta * self.threshold

    def __repr__(self):
        af_name = self.af.__name__ if self.af is not None else None
        loss_fn_name = self.loss_fn.__name__ if self.loss_fn is not None
else None
        return f'Neuron({self.in_features}, {self.threshold}, {af_name},
{loss_fn_name}, {self.n_iter}, {self.eta}, {self.verbose})'

    def parameters(self):
        return {'w': self.w, 'threshold': self.threshold}
```

در ادامه دوباره مجموعه داده را به دوقسمت ۲۰ درصد تست و ۸۰ درصد آموزش تقسیم بندی کرده و به سراغ آموزش نورون می‌رویم که این کار را با ۵۰۰ دوره آموزش انجام دادم.

```
[ ] # Splitting the dataset into the Training set and Test set (80/20 split)
    x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=93, stratify=y, shuffle=True)

    # Display the dimensions of the training and testing sets
    print(f'Dimensions of the training features: {x_train.shape}')
    print(f'Dimensions of the training target: {y_train.shape}')
    print(f'Dimensions of the testing features: {x_test.shape}')
    print(f'Dimensions of the testing target: {y_test.shape}')

    Dimensions of the training features: (320, 2)
    Dimensions of the training target: (320,)
    Dimensions of the testing features: (80, 2)
    Dimensions of the testing target: (80,)

    neuron = Neuron(in_features=2, threshold=0.1, af=sigmoid, loss_fn=bce, n_iter=500, eta=0.1, verbose=True)
    neuron.fit(x_train, y_train[:, None])
    print(f'Neuron specification: {neuron}')
    print(f'Neuron parameters: {neuron.parameters()}')
```

دوباره مثل قبل مرز تصمیم بین داده ها رو دسته تست را رسم میکنیم:

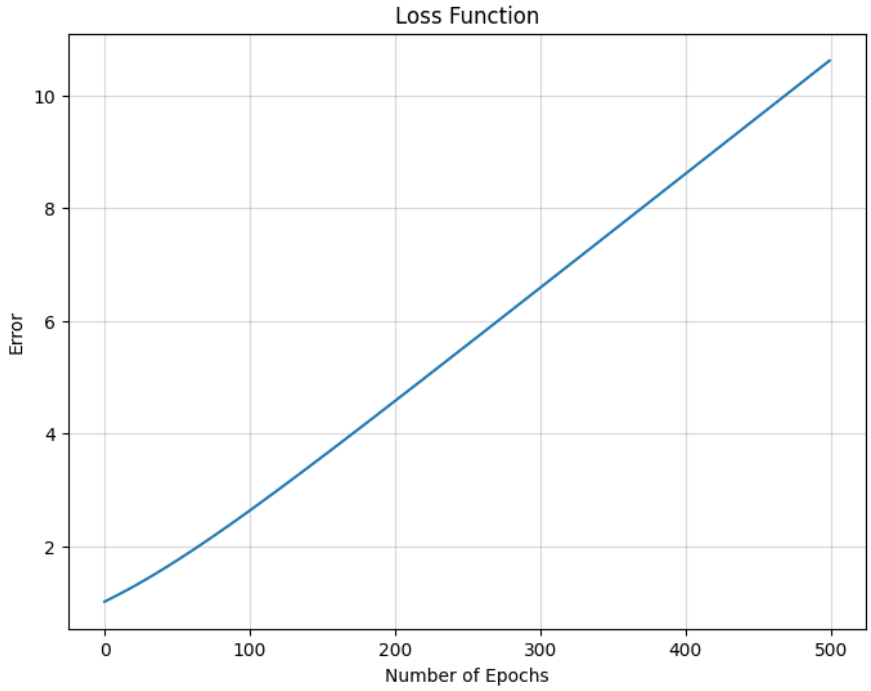


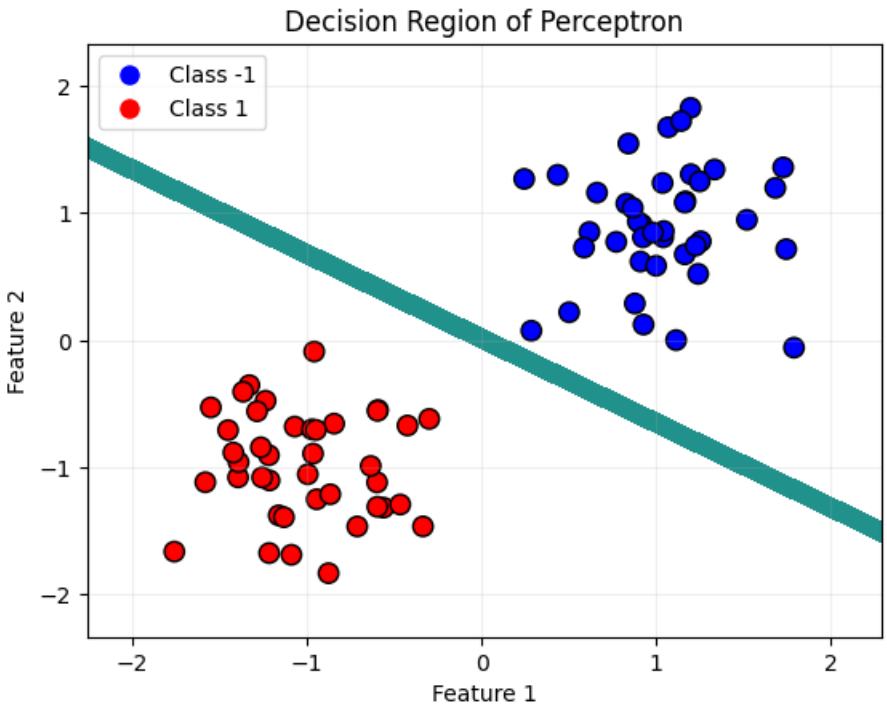
حال ترشهولد و بایاس را تغییر میدهیم تا نتایج خروجی را بر روی نمودار خطا و مرز تصمیم ببینیم:

در ابتدا ترشهولد را از نیم به ۰.۱ کاهش دادم:

```
[ ]  neuron = Neuron(in_features=2, threshold=-0.1, af=sigmoid, loss_fn=bce, n_iter=500, eta=0.1, verbose=True)
     neuron.fit(x_train, y_train[:, None])
     print(f'Neuron specification: {neuron}')
     print(f'Neuron parameters: {neuron.parameters()}')
```

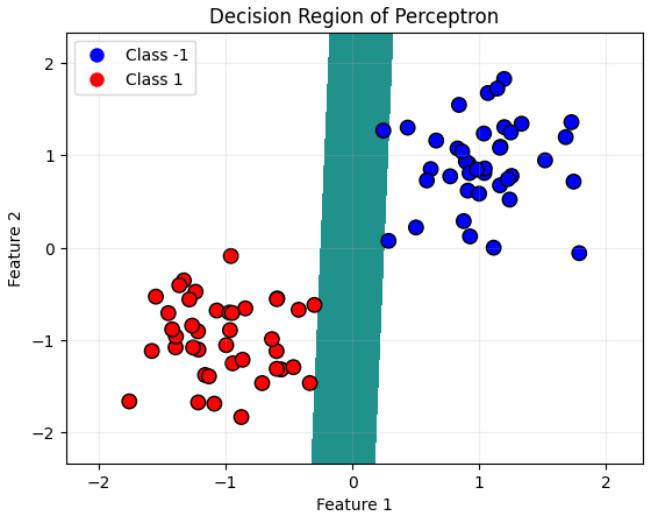
نورون را آموزش داده و نتایج را با دستورات قبلی مشاهده میکنیم:

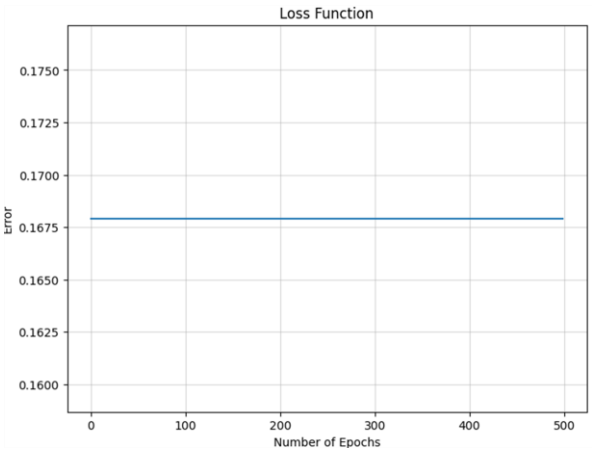




حال بایاس را طبق خواسته سوال حذف میکنیم و آستانه را برابر با صفر قرار میدهیم:

```
[ ]  neuron = Neuron(in_features=2, threshold=0, af=sigmoid, loss_fn=bce, n_iter=500, eta=0.1, verbose=True)
     neuron.fit(x_train, y_train[:, None])
     print(f'Neuron specification: {neuron}')
     print(f'Neuron parameters: {neuron.parameters()}')
```





مشاهده کردیم که تغییر دادن آستانه مستقیما روی مرز تصمیم گیری پرسپترون در طبقه بندی دارد یعنی حد تصمیم گیری بین کلاس های یک مجموعه داده با آستانه تنظیم میشود که اینکار باید با شناخت کامل دیتا ست و یا از روی سعی و خطا انجام گیرد.

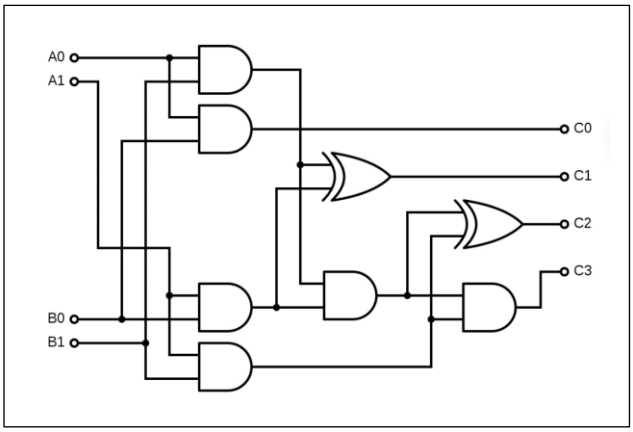
بایاس نیز به رسپترون کمک میکند تا بیشترین انعطاف را در تصمیم گیری نسبت به مبدا داشته باشد و حذف بایاس ممکن است باعث محدودیت مدل در یادگیری در برخی بازه ها شود.

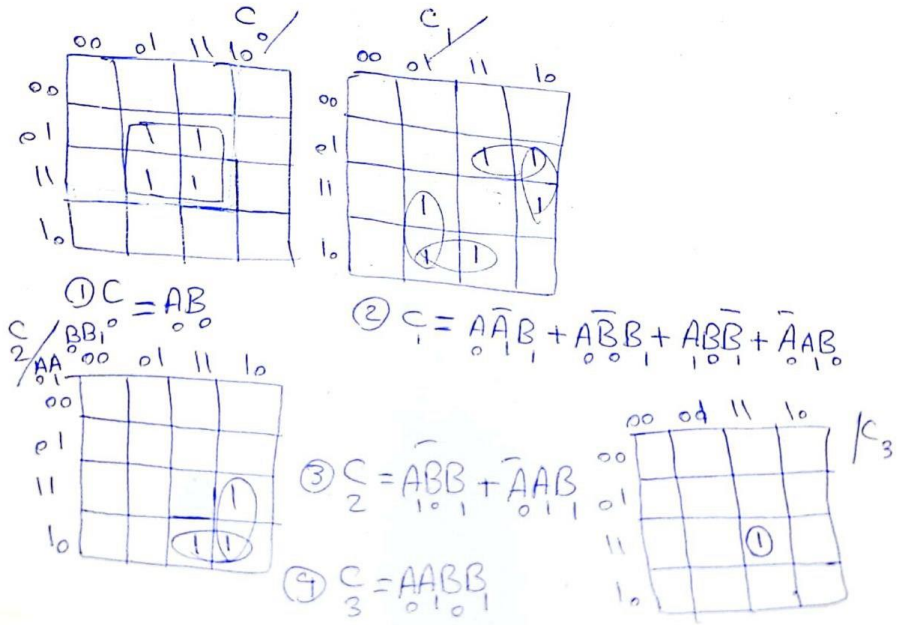
سوال دوم)

ضرب کننده باینری:

Table 1. Truth Table

INPUT A		INPUT B		OUTPUT			
A_1	A_0	B_1	B_0	C_3	C_2	C_1	C_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	0	0	0	0	0
0	1	0	1	0	0	0	1
0	1	1	0	0	0	1	0
0	1	1	1	0	0	1	1
1	0	0	0	0	0	0	0
1	0	0	1	0	0	1	0
1	0	1	0	0	1	0	0
1	0	1	1	0	1	1	0
1	1	0	0	0	0	0	0
1	1	0	1	0	0	1	1
1	1	1	0	0	1	1	0
1	1	1	1	1	0	0	1





حال به سراغ ساختن نورون میرویم:

```python
class McCulloch_Pitts_neuron():

  def __init__(self , weights , threshold):
    self.weights = weights    #define weights
    self.threshold = threshold    #define threshold

  def model(self , x):
    #define model with threshold
    if self.weights @ x >= self.threshold:
        return 1
    else:
        return 0
```

انجام ضرب داخلی و مقایسه آن با ترشهولد:

if self.weights @ x >= self.threshold:
return 1
else: return 0

این کد یک ماشین حالت محدود (DFA) با استفاده از نورون‌های McCulloch-Pitts پیاده‌سازی کرده است.
در زیر گزارش کد قرار دارد:

گزارش کد:

۱. تعریف کلاس McCulloch_Pitts_neuron:

این کلاس یک نورون McCulloch-Pitts را تعریف می‌کند.

دارای متغیرهای weights (وزن‌ها) و threshold (آستانه) است.

تابعmodel نورون، با گرفتن یک برد ورودی، فعالیت نورون را محاسبه می‌کند.

۲. تعریف تابع DFA:

این تابع یک ماشین حالت محدود (DFA) را پیاده‌سازی می‌کند.

وزن‌ها وترشهولدها برای چهار نورون در این تابع تعریف شده‌اند.

از چهار نورون ساخته شده با ورودی و وضعیت مشخص، خروجی ۳ بیتی را محاسبه و برمی‌گرداند.

۳. مثال استفاده:

در اینجا یک مثال استفاده از تابع DFA انجام شده است.

یک وضعیت و یک ورودی به تابع داده شده و خروجی ماشین حالت محدود چاپ می‌شود.

۴. تولید وضعیت‌ها و ورودی‌ها:

از کتابخانه itertools برای تولید تمام ترکیبات وضعیت‌ها و ورودی‌ها استفاده شده است.

وضعیت‌ها از مجموعه {۰, ۱} ایجاد می‌شوند.

تمام ترکیبات وضعیت‌ها و ورودی‌ها با استفاده از تابع itertools.product تولید می‌شوند.

۵. اجرای DFA برای تمام ترکیبات وضعیت‌ها و ورودی‌ها:

یک حلقه forبرای اجرای ماشین حالت محدود بر روی تمام ترکیبات وضعیت‌ها و ورودی‌ها ایجاد شده است.

خروجی ماشین حالت محدود بر اساس وضعیت و ورودی ورودی‌های مختلف محاسبه و چاپ می‌شود.

۶. تغییر خروجی بر اساس شرایط خاص:

```
# inputs
state_b = [0, 1]
state = list(itertools.product(state_b, state_b))
input_data = [1, 0]
state2 = list(itertools.product(input_data, input_data))
X = list(itertools.product(state, state2))

for i in X:
    res = DFA(i[0], i[1])

    # Modify the output based on specific conditions
    if i == ((0, 1), (1, 0)):
        res[2] = 1
    elif i == ((1, 1), (1, 1)):
        res[2] = 0
```

در اینجا، خروجی بر اساس شرایط خاصی تغییر می‌کند.

به عنوان مثال، اگر وضعیت فعلی و ورودی معینی باشد، خروجی یکی از نورون‌ها تغییر می‌کند.

حال به سراغ حل مسئله با کمترین تعداد نورون می‌رویم:

```
def binary_multiplier(input1, input2):
    neur1 = McCulloch_Pitts_neuron([1, 1, 1, 1], 4)
    neur2 = McCulloch_Pitts_neuron([2, -1, 2, -1], 3)
    neur3 = McCulloch_Pitts_neuron([3, 3], 3)
    neur4 = McCulloch_Pitts_neuron([1, 1], 2)

    M3 = neur1.model(np.array([input2[0], input2[1], input1[0], input1[1]]))
    M2 = neur2.model(np.array([input2[0], input2[1], input1[0], input1[1]]))
    M1_1 = neur2.model(np.array([input2[1], input2[0], input1[0],
input1[1]]))
    M1_0 = neur2.model(np.array([input2[0], input2[1], input1[1],
input1[0]]))
    M1 = neur3.model(np.array([M1_1, M1_0]))
```

```
    M0 = neur4.model(np.array([input2[1], input1[1]]))

    # 3 bit output
    return [M3, M2, M1, M0]
```

```
# inputs
input = [1, 0]
X1 = list(itertools.product(input, input))
X =list(itertools.product(X1, X1))

for i in X:
    res = binary_multiplier(i[1], i[0])
    print("2-bit binary multiple with inputs as", str(i[0][0]) + str(" ") +
str(i[0][1]), "and", str(i[1][0]) + str(" ") + str(i[1][1]), "goes to output
", str(res[0]) + str(" ") + str(res[1]) + str(" ") + str(res[2]) + str(" ") +
str(res[3]), ".")
```

نتایج:

```
2-bit binary multiple with inputs as 1 1 and 1 1 goes to output  1 0 0 1 .
2-bit binary multiple with inputs as 1 1 and 1 0 goes to output  0 1 1 0 .
2-bit binary multiple with inputs as 1 1 and 0 1 goes to output  0 0 1 1 .
2-bit binary multiple with inputs as 1 1 and 0 0 goes to output  0 0 0 0 .
2-bit binary multiple with inputs as 1 0 and 1 1 goes to output  0 1 1 0 .
2-bit binary multiple with inputs as 1 0 and 1 0 goes to output  0 1 0 0 .
2-bit binary multiple with inputs as 1 0 and 0 1 goes to output  0 0 1 0 .
2-bit binary multiple with inputs as 1 0 and 0 0 goes to output  0 0 0 0 .
2-bit binary multiple with inputs as 0 1 and 1 1 goes to output  0 0 1 1 .
2-bit binary multiple with inputs as 0 1 and 1 0 goes to output  0 0 1 0 .
2-bit binary multiple with inputs as 0 1 and 0 1 goes to output  0 0 0 1 .
2-bit binary multiple with inputs as 0 1 and 0 0 goes to output  0 0 0 0 .
2-bit binary multiple with inputs as 0 0 and 1 1 goes to output  0 0 0 0 .
2-bit binary multiple with inputs as 0 0 and 1 0 goes to output  0 0 0 0 .
2-bit binary multiple with inputs as 0 0 and 0 1 goes to output  0 0 0 0 .
2-bit binary multiple with inputs as 0 0 and 0 0 goes to output  0 0 0 0 .
```

سوال سوم)

ابتدا یک تابع به نام convertImageToBinary را تعریف کرده و با گرفتن مسیر فایل یک تصویر به عنوان ورود، تصویر را بر اساس شدت پیکسل به یک نمایش دودویی تبدیل می‌کنیم. این کد از ماژول Imaging در پایتون، به خصوص کلاس‌های Image و ImageDraw استفاده می‌کند.

کد با وارد کردن ماژول‌های مورد نیاز شروع می‌شود:

```
from PIL import Image, ImageDraw
```

تابع convertImageToBinary با یک پارامتر به نام path (مسیر فایل تصویر) تعریف می‌شود.

این تابع یک لیست را به عنوان نمایش دودویی تصویر برمی‌گرداند.

تصویر مشخص شده توسط مسیر ورودی با Image.open(path) باز می‌شود.

```
draw = ImageDraw.Draw(image)
```

عرض و ارتفاع تصویر به پیکسل‌ها تعیین می‌شود:

```
width = image.size[۰]
```

```
height = image.size[۱]
```

مقادیر پیکسل و آستانه‌ای برای شدت:

کد با حلقه تو در تو از تمام پیکسل‌های تصویر عبور می‌کند.

مقادیر پیکسل (RGB) استخراج می‌شوند و مجموع شدت کلی هر پیکسل محاسبه می‌شود.

یک آستانه شدتی تعیین می‌شود تا بر اساس آن تصمیم گیری شود که یک پیکسل باید سفید یا سیاه باشد. اگر شدت کلی بیشتر از آستانه مشخصی باشد، پیکسل سفید در نظر گرفته می‌شود؛ در غیر این صورت، سیاه.

رنگ پیکسل بر اساس تصمیم گیری تنظیم و نمایش دودویی نمایش تصویر به‌روزرسانی می‌شود.

کد رنگ هر پیکسل را بر اساس آستانه تعیین کرده و لیست نمایش دودویی را به‌روزرسانی می‌کند:

(۱-)binary_representation.append

(۱)binary_representation.append

تابع نمایش دودویی نهایی تصویر به صورت یک لیست به عنوان خروجی باز می‌گرداند:

return binary_representation

در کل، این کد شدت پیکسل‌ها را به‌کار گرفته و تصویر را به نمایش دودویی تبدیل می‌کند، جایی که پیکسل‌های سفید با -۱ و پیکسل‌های سیاه با ۱ نمایش داده می‌شوند. نمایش دودویی به صورت یک لیست بازمی‌گردانده می‌شود.

اضافه کردن نویز به تصاویر:

```
from PIL import Image, ImageDraw
import random

def generateNoisyImages():
    # List of image file paths
    image_paths = [
        "/content/1.jpg",
        "/content/2.jpg",
        "/content/3.jpg",
        "/content/4.jpg",
        "/content/5.jpg"
    ]

    for i, image_path in enumerate(image_paths, start=1):
        noisy_image_path = f"/content/noisy{i}.jpg"
        getNoisyBinaryImage(image_path, noisy_image_path)
        print(f"Noisy image for {image_path} generated and saved as
{noisy_image_path}")

def getNoisyBinaryImage(input_path, output_path):

    # Open the input image.
    image = Image.open(input_path)

    # Create a drawing tool for manipulating the image.
    draw = ImageDraw.Draw(image)

    # Determine the image's width and height in pixels.
    width = image.size[0]
    height = image.size[1]

    # Load pixel values for the image.
```

```
    pix = image.load()

    # Define a factor for introducing noise.
    noise_factor = 100

    # Loop through all pixels in the image.
    for i in range(width):
        for j in range(height):
            # Generate a random noise value within the specified factor.
            rand = random.randint(-noise_factor, noise_factor)

            # Add the noise to the Red, Green, and Blue (RGB) values of the
pixel.
            red = pix[i, j][0] + rand
            green = pix[i, j][1] + rand
            blue = pix[i, j][2] + rand

            # Ensure that RGB values stay within the valid range (0-255).
            if red < 0:
                red = 0
            if green < 0:
                green = 0
            if blue < 0:
                blue = 0
            if red > 255:
                red = 255
            if green > 255:
                green = 255
            if blue > 255:
                blue = 255

            # Set the pixel color accordingly.
            draw.point((i, j), (red, green, blue))

    # Save the noisy image as a file.
    image.save(output_path, "JPEG")

    # Clean up the drawing tool.
    del draw

# Generate noisy images and save them
generateNoisyImages()
```

توضیحات:

کد با وارد کردن ماژول‌های لازم، Imageو ImageDraw از کتابخانه PIL و random شروع می‌شود.

تعریف توابع:

generateNoisyImages:

بر روی لیستی از مسیرهای فایل تصاویر اصلی حلقه زده و نسخه‌های نویزی را ایجاد و ذخیره می‌کند. اطلاعات در مورد تصاویر تولید شده چاپ می‌شود.

getNoisyBinaryImage:

به تصویر ورودی نویز افزوده و نسخه‌ی نویزی را ذخیره می‌کند.

تولید تصاویر نویزی:

تابع generateNoisyImages صدا زده می‌شود تا نسخه‌های نویزی از تصاویر مشخص شده تولید و ذخیره شوند.

getNoisyBinaryImage:

تصویر ورودی را باز می‌کند و ابزار نقاشی برای دستکاری آن ایجاد می‌کند.

بر روی تمام پیکسل‌ها حلقه زده، نویز رندوم به مقادیر RGB هر پیکسل افزوده می‌شود و اطمینان حاصل می‌شود که این مقادیر داخل محدوده معتبر باقی مانده باشند.

تصویر نویزی را به عنوان یک فایل JPEG ذخیره می‌کند.

نتیجه این کد، مجموعه‌ای از تصاویر نویزی است که با تصاویر اصلی متناظر، هرکدام حاوی نویز تصادفی افزوده شده‌اند.

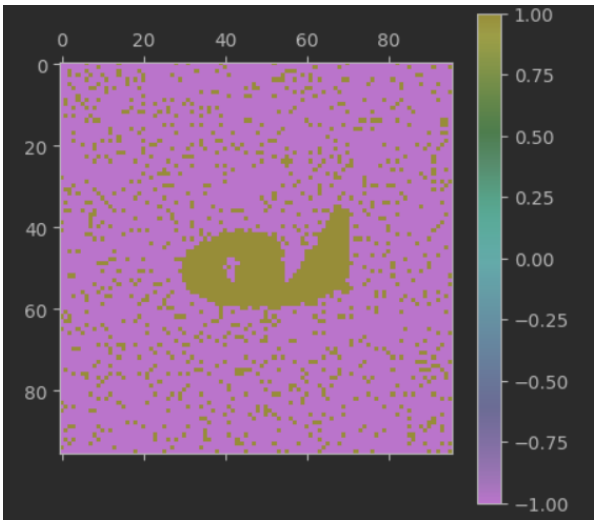
```
Noisy image for /content/1.jpg generated and saved as /content/noisy1.jpg
Noisy image for /content/2.jpg generated and saved as /content/noisy2.jpg
Noisy image for /content/3.jpg generated and saved as /content/noisy3.jpg
Noisy image for /content/4.jpg generated and saved as /content/noisy4.jpg
Noisy image for /content/5.jpg generated and saved as /content/noisy5.jpg
```

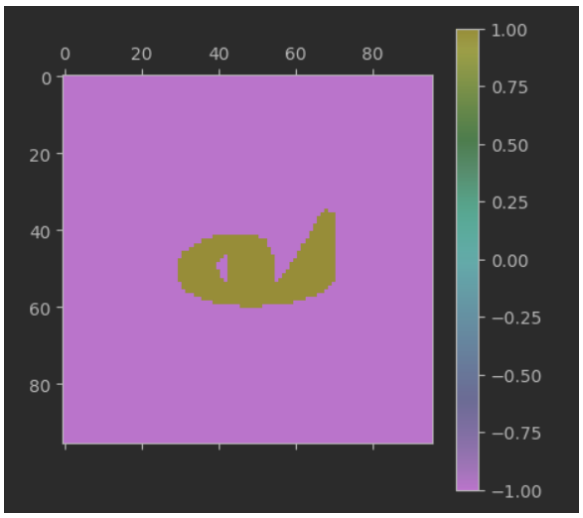
قسمت دوم

طراحی شبکه عصبی همینگ یا هاپفیلد:

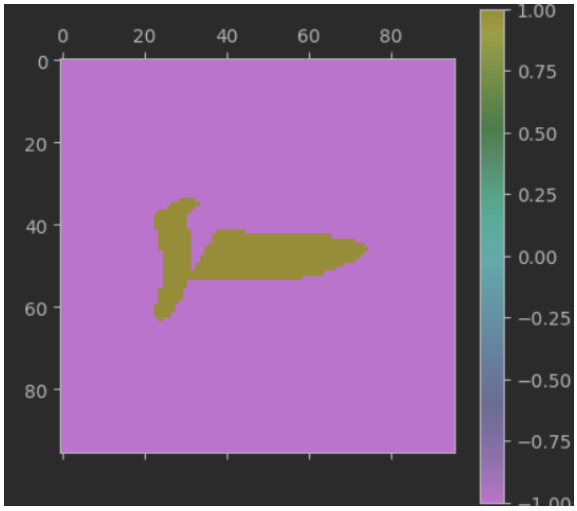
یک شبکه همینگ را برای تشخیص الگوها در تصاویر دودویی پیاده سازی شده است. با استفاده از توابع مختلف، تصویر ورودی به یک ماتریس تبدیل میشود. سپس وزنها و ماتریس اتصال سیناپسی تنظیم میشوند.

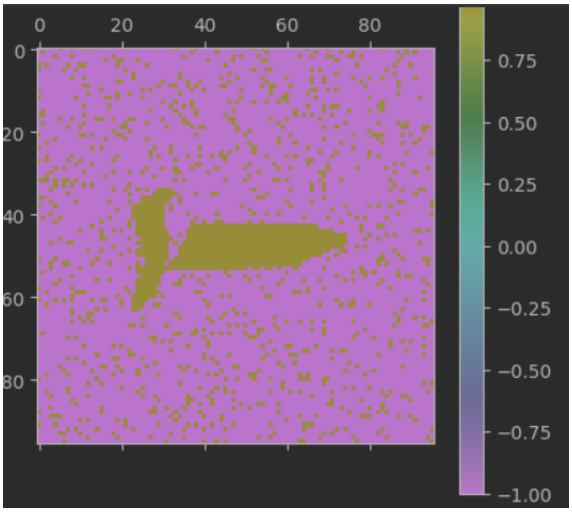
شبکه با اعمال فعالسازی و آموزش به مرور خروجی تولید میکند تا زمانی که فاصله بین خروجیهای متوالی کمتر از یک حداکثر مشخص eMAX باشد. در نهایت، کلاس با بیشترین خروجی مشخص شده و تصویر متناظر با این کلاس نمایش داده میشود.



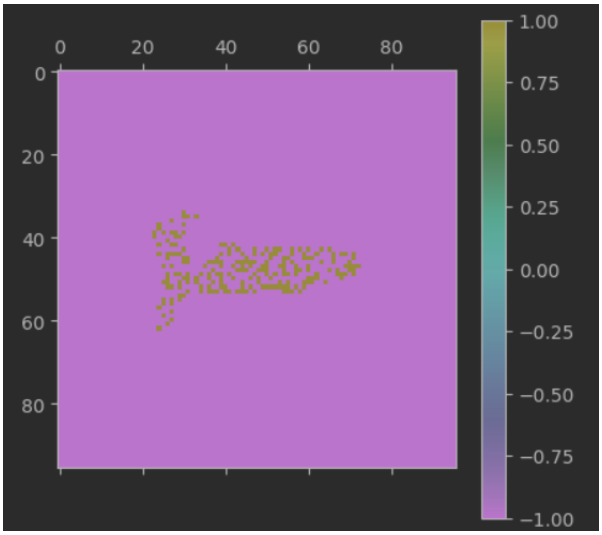


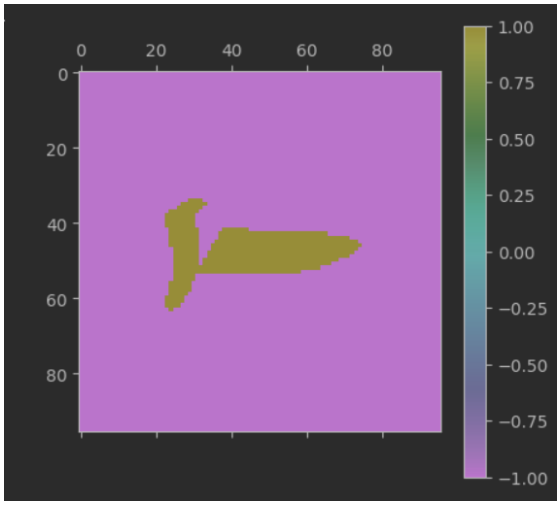
حالا کمی با ضرایب نویز بازی میکنیم و آن را تغییر میدهیم:





و باز نیز تغییر میدهیم(ادامه افزایش)





نویز تا جایی زیاد شد که شبکه به سختی آن را تشخیص داده است ومیتوان در ادامه تا قدری زیادتر کرد که شبکه نتواند آن را تشخیص دهد.

میزان Point Missing یا تعداد نقاط گمشده بیش از حد میتواند به اختلال در عملکرد شبکه یا الگوریتم هایی که از تصاویر مشابه به عنوان ورودی استفاده میکنند، منجر شود.

این اختلالات ممکن است به دلیل از دست رفتن اطلاعات مهم در نقاطی که اضافه شدهاند، یا تغییرات ناخواسته در محتوای تصویر ایجاد شوند.

راه حلهای ممکن برای مقابله با این اختلالات عبارتند از:

Point Missing میزان معقول۱۴۶

-انتخاب یک حد معقول برای تعداد نقاط گمشده به نحوی که تاثیر بر عملکرد شبکه کم باشد. این مقدار باید با توجه به خصوصیات و نیازهای مساله تنظیم شود.

-برای ارزیابی عملکرد شبکه، از تصاویر کنترل شده (تصاویر بدون نویز یا تغییرات) نیزمیتوان استفاده کرد و نتایج را با تصاویر نویزدار مقایسه کرد.

-کنترل دقت نویز افزوده شده به تصویر. مثلاً مقدارnoise_factor را به یک حد معقول تنظیم کرد تا نویز اضافه شده به حد مفید باقی بماند.

-میتوان از روشهای تصویربرداری دقیقتری برای اضافه کردن نویز و تولید نقاط گمشده استفاده کرد تا از از دست رفتن اطلاعات حیاتی جلوگیری شود .

-میتوان عملکرد شبکه را با مقدار مختلف نقاط گمشده ارزیابی کرد و تاثیر آن بر دقت و عملکرد کلی شبکه را مشاهده کرد

سوال چهارم)

دیتاست داده شده را بارگذاری میکنیم و سپس آن را مشاهده میکنیم.

NaN

In 4
```
data.isnull().sum()
```

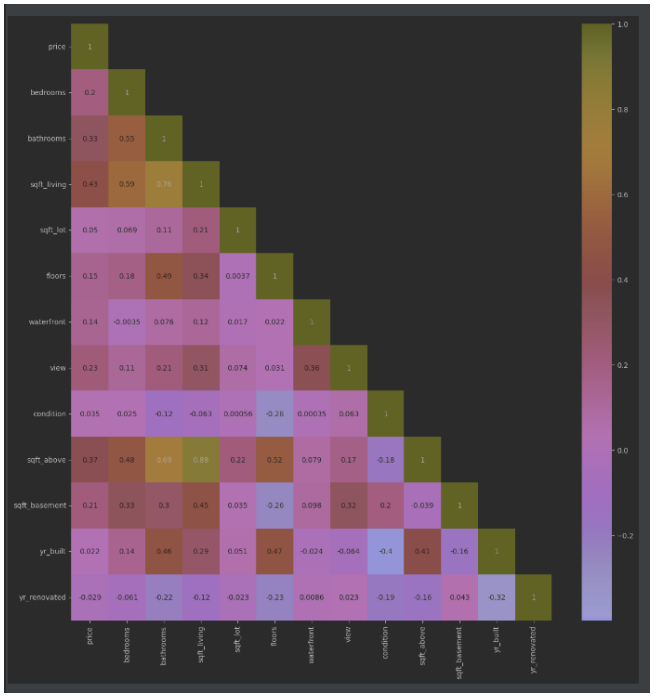
Out 4
```
date           0
price          0
bedrooms       0
bathrooms      0
sqft_living    0
sqft_lot       0
floors         0
waterfront     0
view           0
condition      0
sqft_above     0
```

In 7
```
# Remove rows with any null values
data.dropna(inplace=True)
# Select columns with numerical data types
num = data.select_dtypes(exclude=['object']).columns
num
```

Out 7
```
Index(['price', 'bedrooms', 'bathrooms', 'sqft_living', 'sqft_lot', 'floors',
       'waterfront', 'view', 'condition', 'sqft_above', 'sqft_basement',
       'yr_built', 'yr_renovated'],
      dtype='object')
```

قسمت دوم:

رسم ماتریس هم بستگی:



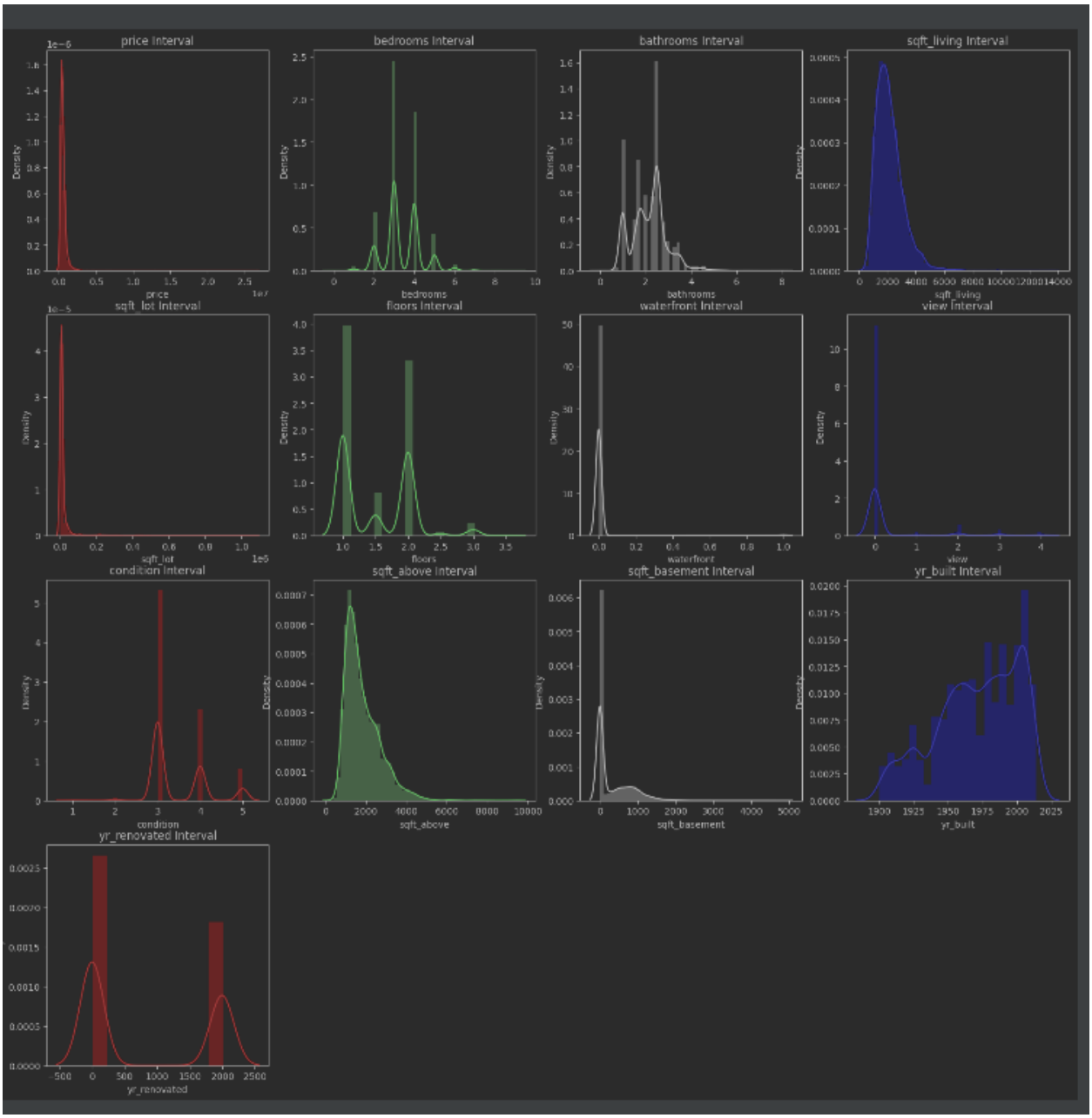
sqft_living بیشترین هم بستگی با قیمت را دارد.

```
# Calculate the correlation between columns and 'price', then sort them in descending order
correlation_matrix = data.corr()['price'].sort_values(ascending=False)
correlation_matrix
```

```
price            1.000000
sqft_living      0.430410
sqft_above       0.367570
bathrooms        0.327110
view             0.228504
sqft_basement    0.210427
bedrooms         0.200336
floors           0.151461
waterfront       0.135648
sqft_lot         0.050451
condition        0.034915
```

```
# Create a 4x4 grid of subplots for various numerical variables
plt.figure(figsize=(20, 20))

plt.subplot(4,4,1)
sns.distplot(data['price'], color="red").set_title('price Interval')

plt.subplot(4,4,2)
sns.distplot(data['bedrooms'], color="green").set_title('bedrooms Interval')

plt.subplot(4,4,3)
sns.distplot(data['bathrooms'], color="black").set_title('bathrooms
Interval')

plt.subplot(4,4,4)
sns.distplot(data['sqft_living'], color="blue").set_title('sqft_living
Interval')

plt.subplot(4,4,5)
sns.distplot(data['sqft_lot'], color="red").set_title('sqft_lot Interval')

plt.subplot(4,4,6)
sns.distplot(data['floors'], color="green").set_title('floors Interval')

plt.subplot(4,4,7)
sns.distplot(data['waterfront'], color="black").set_title('waterfront
Interval')

plt.subplot(4,4,8)
sns.distplot(data['view'], color="blue").set_title('view Interval')

plt.subplot(4,4,9)
sns.distplot(data['condition'], color="red").set_title('condition Interval')

plt.subplot(4,4,10)
sns.distplot(data['sqft_above'], color="green").set_title('sqft_above
Interval')

plt.subplot(4,4,11)
sns.distplot(data['sqft_basement'], color="black").set_title('sqft_basement
Interval')

plt.subplot(4,4,12)
sns.distplot(data['yr_built'], color="blue").set_title('yr_built Interval')

plt.subplot(4,4,13)
sns.distplot(data['yr_renovated'], color="red").set_title('yr_renovated
Interval')
```

نتیجه قسمت سوم:



قسمت چهارم:

```
# Extract 'year', 'month', and 'day' from the 'date' column
data['year'] = pd.to_datetime(data['date']).dt.year
data['month'] = pd.to_datetime(data['date']).dt.month

# Show the DataFrame with the separate 'year' and 'month' columns
data = data[['year', 'month'] + [col for col in data.columns if col not in
```

```
['year', 'month']]]

# Drop the specified columns from the DataFrame
data = data.drop(['date'], axis=1)
data
# Drop year columns from the DataFrame
data = data.drop(['year'], axis=1)
data
# List of specified categorical columns
dummy = ['city']
# Convert categorical columns to numerical using one-hot encoding
df2 = pd.get_dummies(data, columns=dummy, drop_first=True)

# Display the first few rows of the modified DataFrame
df2.head()
# Initialize LabelEncoder
l1 = LabelEncoder()

# Convert object-type columns to numerical using Label Encoding
for i in df2.columns:
    if df2[i].dtype == 'object':
        df2[i] = l1.fit_transform(df2[i])

df2
```

قسمت پنجم)

spliting

```
x = df2.drop(["price"], axis=1)  # features
y = df2["price"]                 # Output data
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=83, shuffle=True)
# Print the shapes of the datasets
print("X Train Scaler : ", x_train.shape) # Print shape of x_train
print("X Test Scaler : ",  x_test.shape)  # Print shape of x_test
print("Y Train Scaler : ", y_train.shape) # Print shape of y_train
print("Y Test Scaler : ",  y_test.shape)  # Print shape of y_test
```

```
X Train Scaler :  (3680, 59)
X Test Scaler :  (920, 59)
Y Train Scaler :  (3680,)
Y Test Scaler :  (920,)
```

normalizing

```
# Initialize Min-Max Scaler
scaler_1 = MinMaxScaler()

# Normalize the training input data
x_train = scaler_1.fit_transform(x_train)

# Normalize the test input data
x_test = scaler_1.transform(x_test)
```

```
# Convert y_train and y_test type to DataFrame
y_train = pd.DataFrame(y_train)
y_test = pd.DataFrame(y_test)

scaler_2 = MinMaxScaler()

# Normalize outputs
y_train = scaler_2.fit_transform(y_train)
y_test = scaler_2.transform(y_test)
```

MLP

```
model_3 = Sequential()

# Add the first hidden layer with 10 neurons and ReLU activation function
model_3.add(Dense(10, activation='relu', input_shape=(x_train.shape[1],)))

# Add the second hidden layer with 10 neurons and ReLU activation function
model_3.add(Dense(10, activation='relu'))

# Add the third hidden layer with 10 neurons and ReLU activation function
model_3.add(Dense(10, activation='relu'))

# Add an output layer with 1 neuron and linear activation function
model_3.add(Dense(1, activation='linear'))

model_3.summary()

```

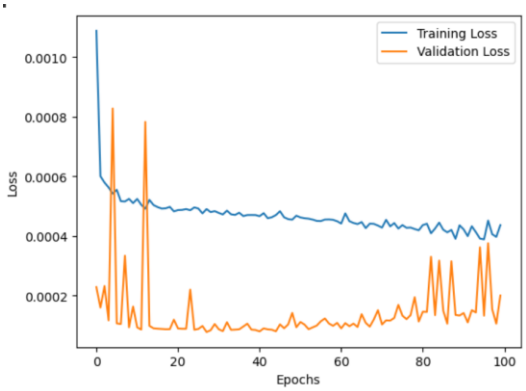
```
29/29 [==============================] - 0s 2ms/step - loss: 3.3302e-04
```

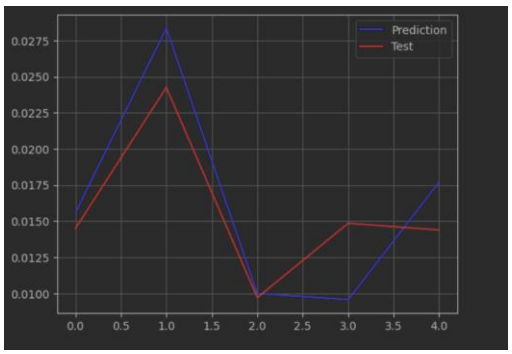
```python
model_3.compile(optimizer='adam', loss='mse')

# Split the data into training and validation sets
x_train1, x_val, y_train1, y_val = train_test_split(x_train, y_train, test_size=0.2, random_state=83, shuffle=True)

history = model_3.fit(x_train1, y_train1, validation_data=(x_val, y_val), epochs=100, batch_size=10, verbose=0)
```

```
92/92 [==============================] - 0s 1ms/step
23/23 [==============================] - 0s 2ms/step
29/29 [==============================] - 0s 2ms/step
Test R2score: 0.13447945896278812
Train R2score: 0.6811936879255166
Validation R2score: -0.16379525732449607
```


```
92/92 [==============================] - 0s 1ms/step
23/23 [==============================] - 0s 2ms/step
29/29 [==============================] - 0s 2ms/step
Test R2score: 0.13447945896278812
Train R2score: 0.6811936879255166
Validation R2score: -0.16379525732449607
```


نمودار های اتلاف و R2_score





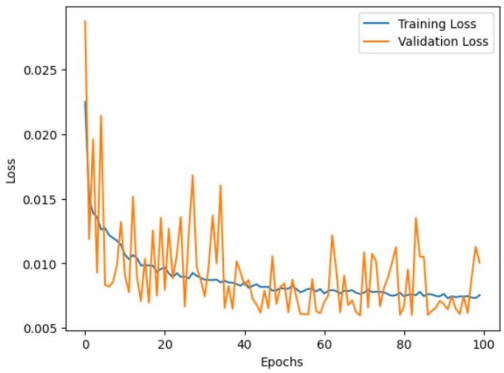
R2_score:0.306270

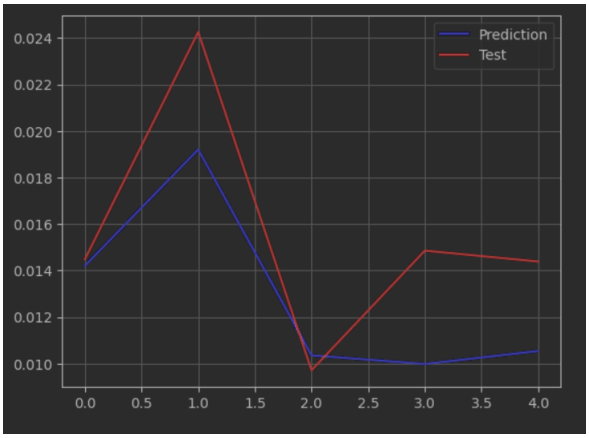
استفاده از تابع بهینه ساز:

بهینه ساز را sgdو تابع اتلاف را mean absolute errorقرار می دهیم:

SGD MAE

```
# Compile model with stochastic gradient descent optimizer and mean absolute error loss
model_3.compile(optimizer = 'sgd',loss = 'mae')

# Split the data into training and validation sets
x_train1, x_val, y_train1, y_val = train_test_split(x_train, y_train, test_size=0.2, random_state=83, shuffle=True)

history = model_3.fit(x_train1, y_train1, validation_data=(x_val, y_val), epochs=100, batch_size=10, verbose=0)

# Evaluate the model
loss = model_3.evaluate(x_test , y_test)
```





R2_score:0.01024

تفاوت:

خطای اعتبارسنجی بیشتر شده است و R2_Score نیز کاهش پیدا کرده است.

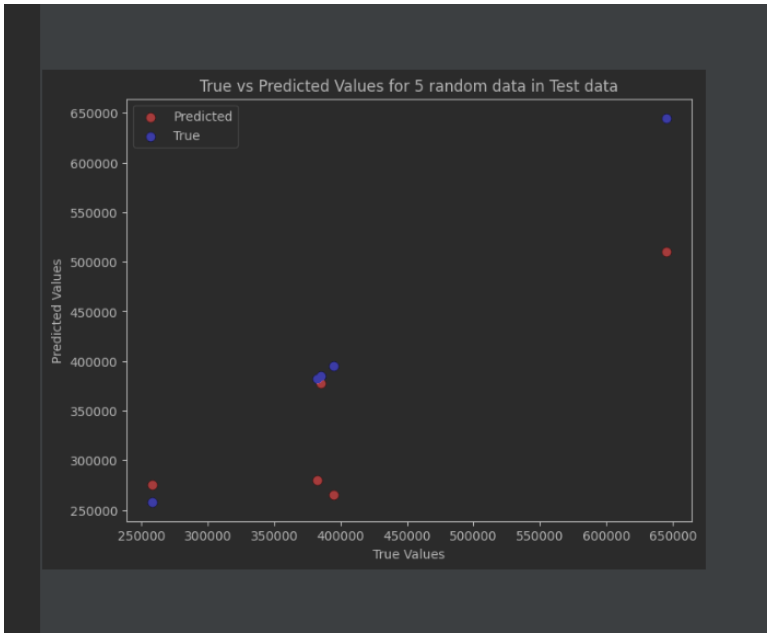
استفاده از یک بهینه ساز مناسب دیتاست کمک کننده است.

بهینه سازها مسئول به روزرسانی وزنها در هر مرحله از آموزش هستند. بهینه سازهای مختلف مثل ، SGD ، Adamو ،RMSpropخصوصیات و الگوریتم های مختلفی دارند که بر اساس مشخصه داده و مسئله مورد استفاده متفاوت هستند.

تابع اتلاف مشخص میکند که مدل چقدر از مقدار واقعی فاصله دارد. انتخاب صحیح تابع اتلاف بر اساس نوع مسئله موجود مهم است. برای مسائل رگرسیون معمولاً از MSEمیانگین مربعات خطا و برای مسائل طبقه بندی از توابعی مانند Cross-Entropyاستفاده میشود.

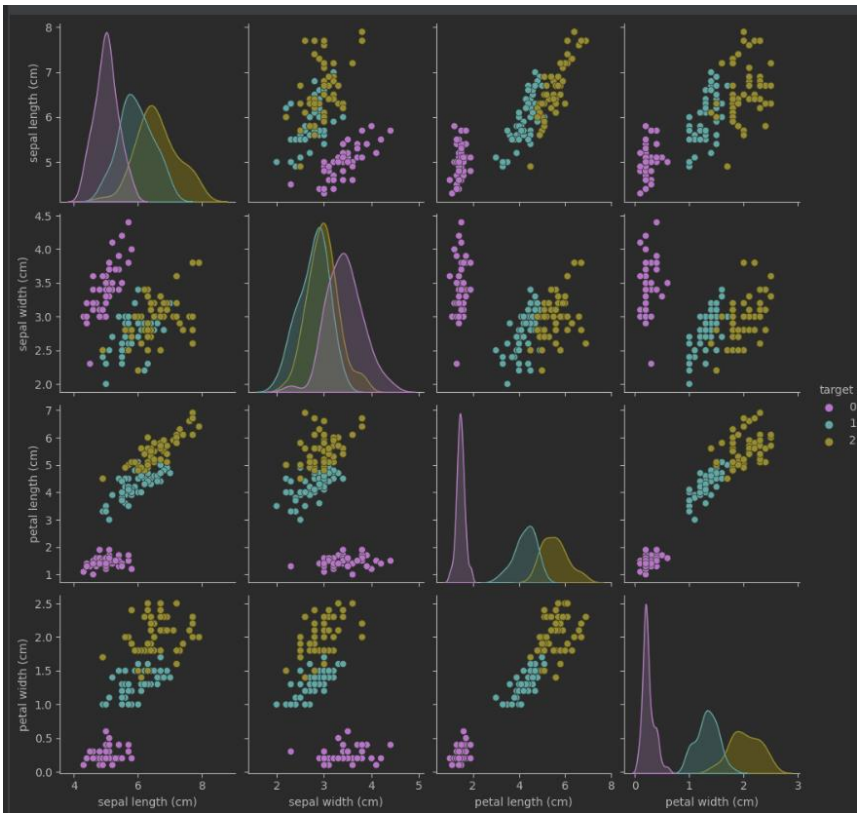
قسمت هفتم)

مقادیر پیشبینی شده و واقعی برای پنج نمونه تصادفی از داده های آزمون را در یک نمودار مقایسه نشان میدهیم. این کار به بررسی تطابق یا عدم تطابق مقادیر پیشبینی با واقعی کمک میکند.

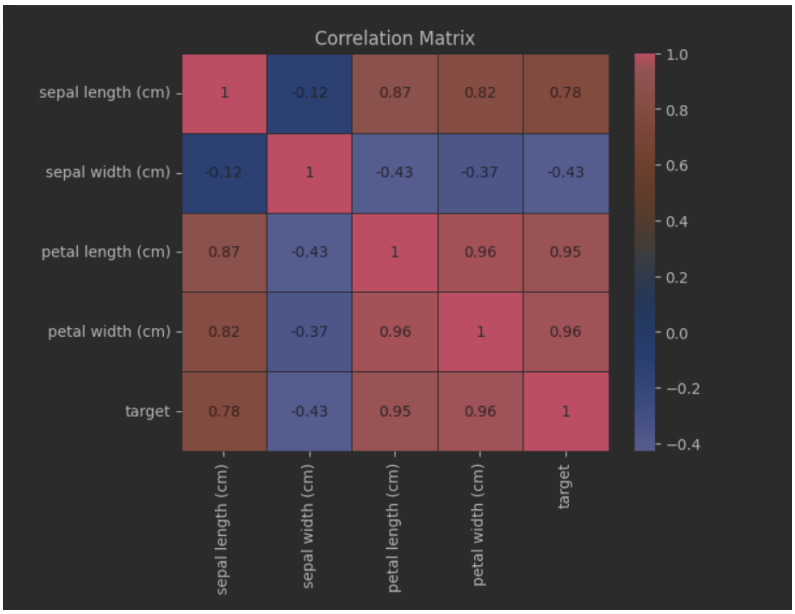


سوال پنجم)

بررسی مجموع داده بعد از بارگزاری:



رسم ماتریس هم بستگی:



```
# Split the data into training and evaluation sets
X_train, X_eval, y_train, y_eval = train_test_split(data, target,
test_size=0.2, random_state=93)
```

قسمت دوم:

```
# Logistic Regression
# Logistic Regression
class LogisticRegression:
    def __init__(self, learning_rate=0.01, epochs=1000):
        self.learning_rate = learning_rate
        self.epochs = epochs

    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    def train(self, X, y):
        self.weights = np.zeros(X.shape[1])

        for epoch in range(self.epochs):
            z = np.dot(X, self.weights)
            predictions = self.sigmoid(z)
            error = y - predictions

            gradient = np.dot(X.T, error)
            self.weights += self.learning_rate * gradient

    def predict(self, X):
        z = np.dot(X, self.weights)
        predictions = self.sigmoid(z)
        return np.round(predictions)
```

Mlp:

```
# Multi-Layer Perceptron (MLP)
class MLP:
    def __init__(self, input_size, hidden_size, output_size,
learning_rate=0.01, epochs=1000):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.learning_rate = learning_rate
        self.epochs = epochs

        self.weights_input_hidden = np.random.rand(self.input_size,
self.hidden_size)
        self.weights_hidden_output = np.random.rand(self.hidden_size,
self.output_size)
```

```
    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    def softmax(self, x):
        exp_values = np.exp(x - np.max(x, axis=1, keepdims=True))
        return exp_values / np.sum(exp_values, axis=1, keepdims=True)

    def train(self, X, y):
        for epoch in range(self.epochs):
            # Forward pass
            hidden_layer_input = np.dot(X, self.weights_input_hidden)
            hidden_layer_output = self.sigmoid(hidden_layer_input)

            output_layer_input = np.dot(hidden_layer_output,
self.weights_hidden_output)
            output_layer_output = self.softmax(output_layer_input)

            # Backward pass
            output_error = y - output_layer_output
            output_delta = output_error

            hidden_layer_error =
output_delta.dot(self.weights_hidden_output.T)
            hidden_layer_delta = hidden_layer_error * (hidden_layer_output *
(1 - hidden_layer_output))

            # Update weights
            self.weights_hidden_output += self.learning_rate *
hidden_layer_output.T.dot(output_delta)
            self.weights_input_hidden += self.learning_rate *
X.T.dot(hidden_layer_delta)

    def predict(self, X):
        hidden_layer_input = np.dot(X, self.weights_input_hidden)
        hidden_layer_output = self.sigmoid(hidden_layer_input)

        output_layer_input = np.dot(hidden_layer_output,
self.weights_hidden_output)
        output_layer_output = self.softmax(output_layer_input)

        return np.argmax(output_layer_output, axis=1)
```

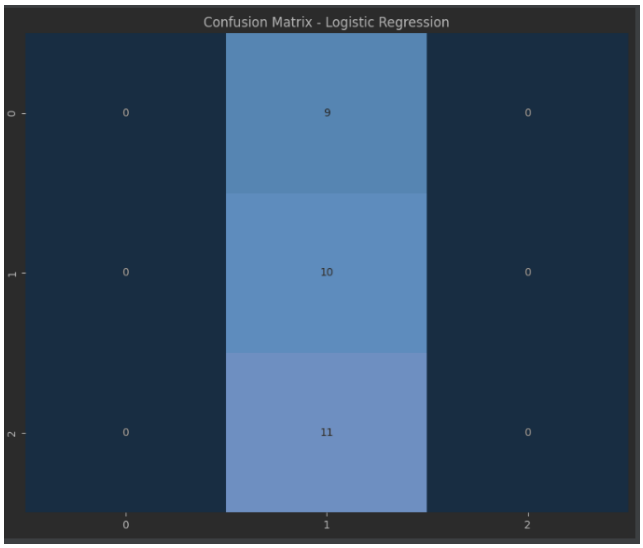
RBF:

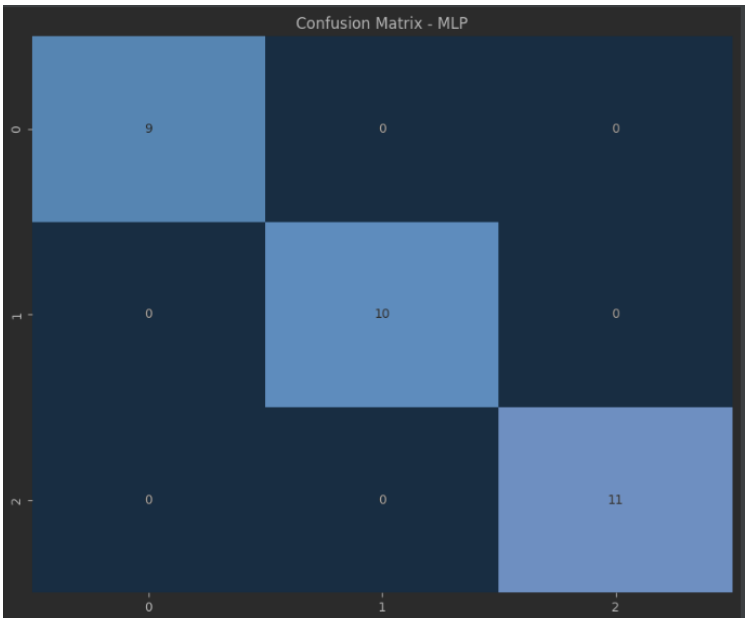
```
class RBFNN:
    def __init__(self, num_centers, learning_rate=0.01, epochs=1000):
        self.num_centers = num_centers
        self.learning_rate = learning_rate
        self.epochs = epochs

    def gaussian_rbf(self, x, c, sigma):
        return np.exp(-np.linalg.norm(x - c) / (2 * sigma**2))
```

```
    def train(self, X, y):
        self.centers = X[np.random.choice(X.shape[0], self.num_centers,
replace=False)]
        self.sigma = np.std(X)

        self.weights = np.random.rand(self.num_centers)

        for epoch in range(self.epochs):
            for i in range(X.shape[0]):
                phi = np.array([self.gaussian_rbf(X[i], c, self.sigma) for c
in self.centers])
                prediction = np.dot(phi, self.weights)
                error = y[i] - prediction

                # Update weights
                self.weights += self.learning_rate * error * phi

    def predict(self, X):
        predictions = []
        for i in range(X.shape[0]):
            phi = np.array([self.gaussian_rbf(X[i], c, self.sigma) for c in
self.centers])
            prediction = np.dot(phi, self.weights)
            predictions.append(prediction)

        return np.round(predictions)
```

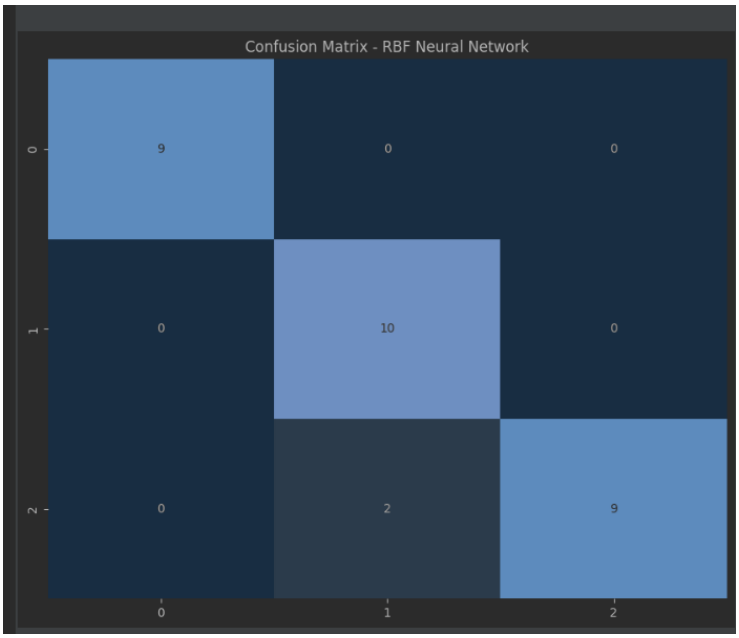
بررسی نتایج و خروجی هر یک از شبکه ها:

```
Logistic Regression Metrics:
Accuracy: 0.3333
Precision: 0.1111
Recall: 0.3333
F1 Score: 0.1667
```






```
MLP Metrics:
Accuracy: 1.0000
Precision: 1.0000
Recall: 1.0000
F1 Score: 1.0000
```




```
RBF Neural Network Metrics:
Accuracy: 0.9333
Precision: 0.9444
Recall: 0.9333
F1 Score: 0.9330
```

در میان این ۳ روش، لاجیستیک رگرسیون عملکرد ضعیف تری داشته است.

این اظهارنظر که رگرسیون لجستیک بهتر است یا مولتی‌لایه‌ای (MLP) و توابع پایه شعاعی (RBF) به موقعیت، مجموعه داده ، بستگی دارد. هر الگوریتم یادگیری ماشین نقاط قوت و ضعف خود را دارد و عملکرد آن‌ها می‌تواند بر اساس ویژگی‌های داده و پیچیدگی روابط زیرساختی متغیر باشد.

در زیر چند دلیل آورده شده که ممکن است رگرسیون لجستیک به عنوان کمتر موثر نسبت به MLP و RBF در نظر گرفته شود:

.1پیچیدگی مدل:

رگرسیون لجستیک یک مدل خطی است، به این معنا که تنها می‌تواند روابط خطی بین ویژگی‌های ورودی و خروجی را مدل کند. اگر روابط زیرساختی داده شما بیشتر پیچیده و غیرخطی باشند، شبکه‌های MLP و RBF که قابلیت گرفتن الگوهای پیچیده را دارند، ممکن است عملکرد بهتری داشته باشند.

.2رابطه‌های غیرخطی:

- اگر روابط داده شما بسیار غیرخطی باشند، شبکه‌های MLP و RBF که می‌توانند نقشه‌های غیرخطی پیچیده را یاد بگیرند و مدل کنند، ممکن است از رگرسیون لجستیک عملکرد بهتری داشته باشند.

.3نمایش ویژگی:

- رگرسیون لجستیک بر اساس ویژگی‌های دست‌ساخته تکیه می‌کند و عملکرد آن ممکن است محدود باشد اگر ویژگی‌های مرتبط به صورت صریح فراهم نشوند. به عنوان مقابل، MLPقادر است به صورت خودکار نمایش‌های ویژگی سلسله‌مراتبی از داده را یاد بگیرد و الگوهای پیچیده‌تری را ضبط کند.

.4ظرفیت برای نمایش:

- شبکه‌های MLP و RBF به دلیل قابلیت یادگیری نگاشت‌ها و ویژگی‌های سلسله‌مراتبی، ظرفیت بیشتری برای نمایش دارند. این ظرفیت افزوده می‌تواند در مواقعی که روابط پیچیده‌تری وجود دارد، مفید باشد.

5.اورفیتینگ :

- رگرسیون لجستیک تمایل کمتری به اورفیتینگ نسبت به مدل‌های پیچیده مانند MLP با تعداد پارامترهای زیاد دارد. با این حال، در برخی موارد ممکن است نیاز به مدل پیچیده‌تری باشد تا الگوهای زیرساختی داده را ضبط کند.

6. اندازه داده:

- برای مجموعه داده‌های کوچک، رگرسیون لجستیک به دلیل سادگی و کاهش خطر اورفیتینگ، ممکن است عملکرد قابل قبولی داشته باشد. با افزایش اندازه مجموعه داده، قدرت اعتباری شبکه‌های MLP و RBF ممکن است مفیدتر باشد.

احیاناً انتخاب یک مدل یادگیری ماشین بر اساس خصوصیات داده، اندازه مجموعه داده، و نیازهای ویژه مسئله اهمیت دارد. در برخی موارد، رگرسیون لجستیک ممکن است عملکرد خوبی داشته باشد، به ویژه زمانی که روابط داده اغلب خطی هستند و مجموعه داده زیاده‌رو نیست.