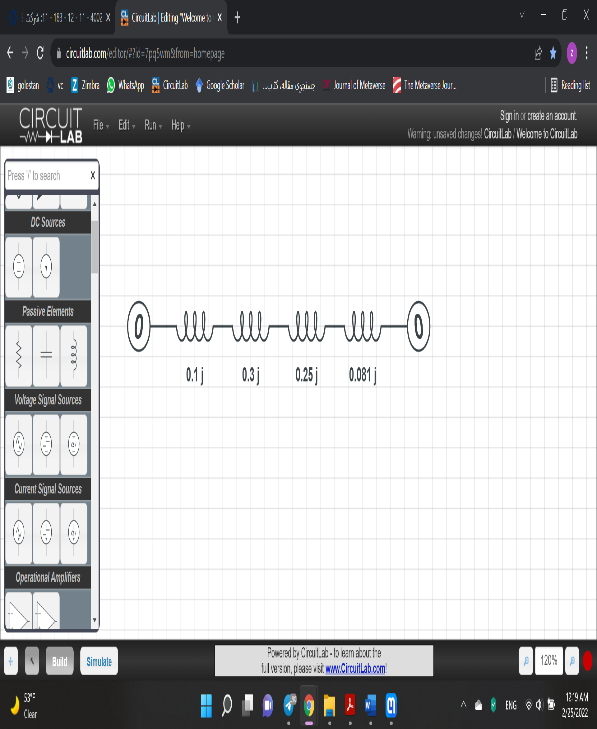
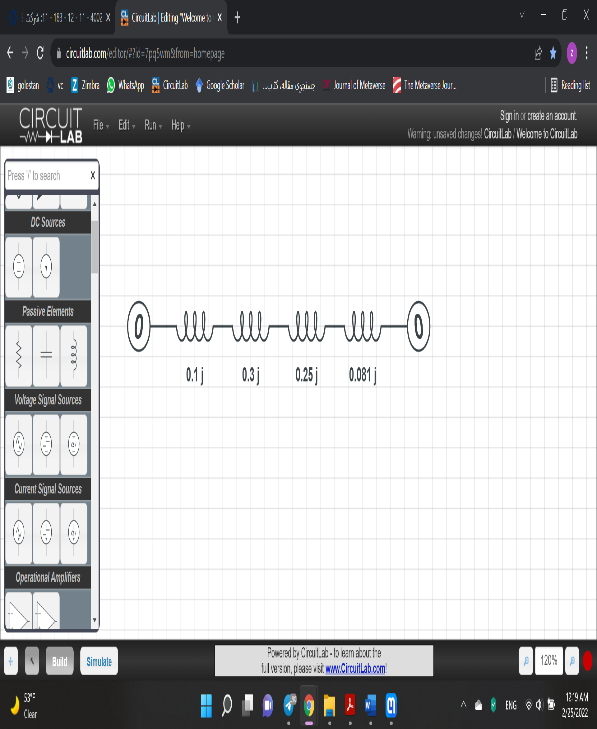


تمرین سری 4

استاد: دکتر علییاری

مبحث تمرین: مینی پروژه دوم

دانشجو: فربد فدائیان 9824033

**سوال اول:**

مجموعه مربوط به داده ها را دانلود کرده و پس از سیو کردن در گوگل درایو ان را در گوگل کولب با دستور زیر بارگذاری میکنیم.

!pip install --upgrade --no-cache-dir gdown

!gdown 1Om3-1-EQa0UGYnrDiqY-0E34gKbHd4-T

**قسمت اول:**

در این قسمت ما ابتدا با مشخص کردن اینکه ستون اول و دوم ما ویژگی و ستون سوم که بیان گر کلاس ما است چیزی است که میخواهی ماشین تشخیص دهد داده ها را جدا کرده سپس آن ها را طبق خواسته سوال با نسبت 80 به 20 جدا نموده ایم.

df = pd.read\_csv('/content/Perceptron.csv')

x = df[["x1","x2"]]

y = df[["y"]]

x\_train , x\_test , y\_train , y\_test = train\_test\_split(x , y , test\_size = 0.2)

در این قسمت توابع مورد نیاز که بعدا آن ها را فراخوانی خواهیم کرد نوشته شده است.

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 accuracy(y, y\_hat, t=0.5):

    y\_hat = np.where(y\_hat >= t, 1, 0)

    return np.mean(y == y\_hat)

در این قسمت ما کلاسی به نام نورون تعریف میکنیم تا تمامی مسائل یادگیری را بر روی آن پیاده کنیم.

مشاهده میشود که در این نرون تابع هدر رفت از روش mean squer error بدست امده است و کل آموزش دارای 100 تکرار است.

class Neuron:

    def \_\_init\_\_(self, in\_features, 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.b = np.random.randn()

        self.af = af

        self.loss\_fn = loss\_fn

        self.loss\_hist = []

        self.w\_grad, self.b\_grad = None, 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.b

        y\_hat = y\_hat if self.af is None else self.af(y\_hat)

        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)

        self.b\_grad = (y\_hat - y).mean()

    def gradient\_descent(self):

        self.w -= self.eta \* self.w\_grad

        self.b -= self.eta \* self.b\_grad

    def \_\_repr\_\_(self):

        return f'Neuron({self.in\_features}, {self.af.\_\_name\_\_})'

    def parameters(self):

        return {'w': self.w, 'b': self.b}

در این قسمت تارگت و فیچر را از هم جدا کرده تا y به صورت دلخواه نمایش داده شود.

# Separate features and target

X = data.iloc[:, :-1].values

y = data.iloc[:, -1].values

print(y)

**قسمت دوم:**

import pandas as pd

import numpy as np

from sklearn.model\_selection import train\_test\_split

from sklearn.linear\_model import Perceptron

import matplotlib.pyplot as plt

from mlxtend.plotting import plot\_decision\_regions

# Load the dataset

data = pd.read\_csv('/content/Perceptron.csv')

# Separate features and target

X = data.iloc[:, :-1].values

y = data.iloc[:, -1].values

# 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=42)

در این قسمت ما کلسیفایر پرسپترون را مقدار دهی اولیه میکنیم و دقت را محاسبه میکنیم.

# Initialize Perceptron classifier

clf = Perceptron()

# Train the perceptron

clf.fit(X\_train, y\_train)

# Accuracy on train and test data

train\_accuracy = clf.score(X\_train, y\_train)

test\_accuracy = clf.score(X\_test, y\_test)

print(f"Accuracy on train set: {train\_accuracy}")

print(f"Accuracy on test set: {test\_accuracy}")

# Convert y\_train to integer type

y\_train = y\_train.astype(np.int)

در این قسمت داده ها را پلات کرده و خط مرز تصمیم گیری را مشخص میکنیم.

# Plotting decision boundary

plot\_decision\_regions(X\_train, y\_train, clf=clf, legend=2)

plt.xlabel('X1')

plt.ylabel('X2')

plt.title('Decision Boundary - Perceptron')

plt.show()

**قسمت سوم:**

در این قسمت باید با یک آستانه یا همان ترشهولد دیگر قسمت یک و دو را دوباره انجام دهیم و باید بایاس را نیز حذف کنیم.

در یک شبکه عصبی پرسپترون، آستانه، مرز تصمیم گیری را تعیین می کند طبقه بندی. با تغییر آستانه، می توانید حساسیت مدل را تنظیم کنید پیش بینی ها آستانه بالاتر، مدل را محافظه کارتر می کند و تعداد کمتری را طبقه بندی می کند موارد مثبت هستند، در حالی که آستانه پایین تر آن را آزادتر می کند و بیشتر طبقه بندی می کند موارد مثبت اگر عبارت بایاس را در مدل پرسپترون حذف کنید، مرز تصمیم تبدیل می شودمحدود به مبدأ (0,0) در فضای ورودی. اصطلاح سوگیری مرز تصمیم را اجازه می دهداز مبدأ دور شده و انعطاف پذیری بیشتری را در پیش بینی های مدل ممکن می سازد. بدون عبارت بایاس، مدل تنها قادر به یادگیری مرزهای تصمیم خطی خواهد بود از مبدأ عبور کند. گنجاندن یک عبارت سوگیری به مدل کمک می کند تا الگوهای پیچیده را بگیرد و آن را بهبود بخشد عملکرد با اجازه دادن به مرز تصمیم برای جبران از مبدا. فراهم می کند. این مدل با توانایی یادگیری مرزهای تصمیم گیری غیر خطی و رسیدگی به واردی که در آن ویژگی های ورودی میانگین صفر ندارند.

به طور کلی، تغییر آستانه بر موقعیت مرز تصمیم تاثیر می گذارد، در حالی که حذف می شود. اصطلاح سوگیری مرز تصمیم را به مبدأ محدود می کند و یادگیری مدل را محدود می کند.

**سوال دوم:**

رویکرد پیاده سازی:

برای پیاده سازی ضریب باینری، مراحل زیر را دنبال می کنیم:

مرحله 1: جدول ورودی-خروجی را طراحی کنید

ما با ایجاد یک جدول ورودی-خروجی شروع خواهیم کرد که تمام ترکیبات ممکن را نشان می دهد. دو ورودی باینری و خروجی مربوط به آنها. هر ردیف در جدول نشان دهنده یک ترکیب ورودی منحصر به فرد و خروجی حاصل از آن است.

مرحله 2: طراحی شبکه برای هر خروجی

برای هر بیت خروجی، شبکه ای با کمترین تعداد نورون ها طراحی خواهیم کرد.پایین ترین آستانه ممکن شبکه متشکل از نورون های دوتایی خواهد بود که به روشی متصل هستند که منطق خروجی مورد نظر را نشان می دهد.

در این قسمت کلاس نرون MaCulloch-Pitts را مینویسیم

import numpy as np

import itertools

class McCulloch\_Pitts\_neuron():

    def \_\_init\_\_(self, weights, threshold):

        self.weights = weights

        self.threshold = threshold

    def model(self, x):

        if self.weights @ x >= self.threshold:

            return 1

        else:

            return 0

def DFA(state, input):

    neur1 = McCulloch\_Pitts\_neuron([0, 1, 0, 1], 2)

    neur2 = McCulloch\_Pitts\_neuron([3, 2, 2, 3], 6)

    neur3 = McCulloch\_Pitts\_neuron([2, -1, 2, -1], 3)

    neur4 = McCulloch\_Pitts\_neuron([1, 1, 1, 1], 4)

    z1 = neur1.model(np.array([state[0], state[1], input[0], input[1]]))

    z2 = neur2.model(np.array([state[0], state[1], input[0], input[1]]))

    z3 = neur3.model(np.array([state[0], state[1], input[0], input[1]]))

    z4 = neur4.model(np.array([state[0], state[1], input[0], input[1]]))

    return list([z4, z3, z2, z1])

state\_b = [0, 1]

state = list(itertools.product(state\_b, state\_b))

input = [1, 0]

state2 = list(itertools.product(input, input))

X = list(itertools.product(state, state2))

print('state:', state)

print('\n')

print('X:', X)

**سوال سوم:**

**قسمت اول:**

تابع اول:

ابتدا در فونت قرمز رنگ توضیح داده شده است که این تابع قرار است تصویر را به باینری تبدیل کند و این کار را با گرفتن یک تصویر و اطلاق عدد 1- به سفید و 1 به پیکسل های سیاه انجام میدهد.

from PIL import Image, ImageDraw

import random

def convertImageToBinary(path):

    """

    Convert an image to a binary representation based on pixel intensity.

    Args:

        path (str): The file path to the input image.

    Returns:

        list: A binary representation of the image where white is represented by -1 and black is represented by 1.

    """

در این قسمت فایل عکس باز دریافت میشود.

    # Open the image file.

    image = Image.open(path)

در این قسمت یک ابزار کشیدن ساخته میشود تا تصویر را بتوان دستکاری کرد.

    # Create a drawing tool for manipulating the image.

    draw = ImageDraw.Draw(image)

در این قسمت طول و عرض عکس بر حسب پیکسل بدست می آید

    # 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 intensity thresholding.

    factor = 100

در این قسمت یک لیست خالی تعریف میشود تا مقادیر باینری در ان ریخته شود.

  # Initialize an empty list to store the binary representation.

    binary\_representation = []

در این قسمت یک لوپ نوشته میشود تا به تمامی پیکسل های تصویر دست رسی داشته باشیم.

    # Loop through all pixels in the image.

    for i in range(width):

        for j in range(height):

در این قسمت در این لوپ میزان قرمز سبز و آبی بودن یک پیکسل به ان اطلاق میشود.

            # Extract the Red, Green, and Blue (RGB) values of the pixel.

            red = pix[i, j][0]

            green = pix[i, j][1]

            blue = pix[i, j][2]

در این قسمت مجموع درگیر بودن مقادیر قرمز سبز و آبی در پیکسل مشخص میشود تعیین میشود و درواقع به صورت یک کد RGB گزارش میشود.

            # Calculate the total intensity of the pixel.

            total\_intensity = red + green + blue

در این قسمت در اینجا شروطی نوشته میشود تا مشخص شود به پیکسل مقدار سفید یا سیاه اطلاق شود.

            # Determine whether the pixel should be white or black based on the intensity.

            if total\_intensity > (((255 + factor) // 2) \* 3):

                red, green, blue = 255, 255, 255  # White pixel

                binary\_representation.append(-1)

            else:

                red, green, blue = 0, 0, 0  # Black pixel

                binary\_representation.append(1)

در این قسمت رنگ پیکسل را به صورت کد RGB به draw میدهیم تا با این ابزار کشیده شود.

            # Set the pixel color accordingly.

            draw.point((i, j), (red, green, blue))

در این قسمت draw را حذف کرده.

    # Clean up the drawing tool.

    del draw

در این قسمت مقادیر باینری را بر میگردانیم

    # Return the binary representation of the image.

    return binary\_representation

تابع دوم:

from PIL import Image, ImageDraw

import random

در این قسمت تمامی تصاویر موجود در فایل بارگذازی میشود.

def generateNoisyImages():

    # List of image file paths

    image\_paths = [

        "/content/1.jpg",

        "/content/2.jpg",

        "/content/3.jpg",

        "/content/4.jpg",

        "/content/5.jpg"

    ]

در این قسمت حلقه ای نوشته شده است تا تصویر در ان با نام نویز دار شده ذخیره شود (تصویر هنوز نویز دار نشده است.)و پیام ثبت و مسیر ذخیره ان به کاربر داده شود.

    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}")

در این قسمت تابع نویز دار کردن نوشته شده است تا تصویر را دریافت کرده و پس از نویز دار نمودن آن، آن را به عنوان فایل جدید با همان نام ذخیره نمایید.

def getNoisyBinaryImage(input\_path, output\_path):

    """

    Add noise to an image and save it as a new file.

    Args:

        input\_path (str): The file path to the input image.

        output\_path (str): The file path to save the noisy image.

    """

این قسمت تا فاکتور نویز با تابع بالا یکی است.

    # 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()

در این قسمت فاکتور نویز مشخص میشود.(بدیهی است هر میزان این فاکتور بالاتر باشد نویز بیشتر است.

    # Define a factor for introducing noise.

    noise\_factor = 10000000

    # Loop through all pixels in the image.

در این قسمت حلقه ای نوشته میشود تا تمامی پیکسل های تصویر را فرا خوانی کند.

    for i in range(width):

        for j in range(height):

در این قسمت میزان رندمی در بازه منفی تا مثبت فاکتور نویز تولید میشود.

و سپس به میزان کد RGB آن افزوده میشود.

          # 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

در این قسمت شروطی برای تعیین کد های RGB گذاشته میشود.

            # 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

در این قسمت رنگ پیکسل را به صورت کد RGB به draw میدهیم تا با این ابزار کشیده شود.

            # Set the pixel color accordingly.

            draw.point((i, j), (red, green, blue))

در این قسمت تصویر با فرمت JPEG ذخیره میشود.

    # Save the noisy image as a file.

    image.save(output\_path, "JPEG")

در این قسمت draw را حذف کرده.

    # Clean up the drawing tool.

    del draw

در این قسمت تابع تولید تصویر نویزی که قبلا بررسی شد فراخوانی شده و تصویر ذخیره میشود.

# Generate noisy images and save them

generateNoisyImages()

**قسمت دوم:**

from pylab import \*

from math import sqrt

import matplotlib.pyplot as plt

import os

# Define the path to the input image

IMAGE\_PATH = "/content/noisy3.jpg"

در این قسمت تابع نوشته میشود تا ماتریس را به روش فرمت شده نمایش دهد.

def show(matrix):

    """

    Display a matrix in a formatted manner.

    Args:

        matrix (list of lists): The matrix to be displayed.

    """

    for j in range(len(matrix)):

        for i in range(len(matrix[0])):

            print("{:.3f}".format(matrix[j][i]), end=" ")

        print(sep="")

در این قسمت تابعی نوشته میشود تا یک بردار را به یک ماتریس با ابعاد a و b تبدیل بکند.

def change(vector, a, b):

    """

    Transform a vector into a matrix of specified dimensions.

    Args:

        vector (list): The vector to be transformed.

        a (int): The number of columns in the resulting matrix.

        b (int): The number of rows in the resulting matrix.

    Returns:

        list of lists: The transformed matrix.

    """

    matrix = [[0 for j in range(a)] for i in range(b)]

    k = 0

    j = 0

    while k < b:

        i = 0

        while i < a:

            matrix[k][i] = vector[j]

            j += 1

            i += 1

        k += 1

    return matrix

در این قسمت تابعی نوشته میشود تا با آن بتوانیم ماتریکس و بردار را در هم ضرب بکنیم.

def product(matrix, vector, T):

    """

    Multiply a matrix by a vector.

    Args:

        matrix (list of lists): The matrix to be multiplied.

        vector (list): The vector to be multiplied.

        T (float): The threshold parameter for the activation function.

    Returns:

        list: The resulting vector after multiplication.

    """

    result\_vector = []

    for i in range(len(matrix)):

        x = 0

        for j in range(len(vector)):

            x = x + matrix[i][j] \* vector[j]

        result\_vector.append((x + T))

    return result\_vector

در این قسمت تابعی نوشته میشود تا بردار خروجی را پس از اکتیو شدن به ما باز گرداند.

def action(vector, T, Emax):

    """

    Activation function to process a vector.

    Args:

        vector (list): The input vector to be processed.

        T (float): The threshold parameter for the activation function.

        Emax (float): The maximum allowable value for the difference in output vectors between consecutive iterations.

    Returns:

        list: The output vector after activation.

    """

    result\_vector = []

    for value in vector:

        if value <= 0:

            result\_vector.append(0)

        elif 0 < value <= T:

            result\_vector.append(Emax \* value)

        elif value > T:

            result\_vector.append(T)

    return result\_vector

در این قسمت تابعی نوشته میشود که جمع مقادیر بردار را با در نظر نگرفتن ارگمان j به ما باز گرداند.

def mysum(vector, j):

    """

    Calculate the sum of vector values excluding the element at index j.

    Args:

        vector (list): The input vector.

        j (int): The index of the element to be excluded from the sum.

    Returns:

        float: The sum of vector values with the element at index j excluded.

    """

    p = 0

    total\_sum = 0

    while p < len(vector):

        if p != j:

            total\_sum = total\_sum + vector[p]

        p += 1

    return total\_sum

در این قسمت تابعی نوشته میشود که تفاوت میان دو بردار را اعلام و نورم آن را نیز گزارش نماید.

def norm(vector, p):

    """

    Calculate the difference between two vectors and compute the norm of the resulting vector.

    Args:

        vector (list): The first vector.

        p (list): The second vector for subtraction.

    Returns:

        float: The Euclidean norm of the difference between the two vectors.

    """

    difference = []

    for i in range(len(vector)):

        difference.append(vector[i] - p[i])

    sum = 0

    for element in difference:

        sum += element \* element

    return sqrt(sum)

# List of paths to example images

path = [

    '/content/1.jpg',

    '/content/2.jpg',

    '/content/3.jpg',

    '/content/4.jpg',

    '/content/5.jpg',

]

x = []  # Binary representations of example images

print(os.path.basename(IMAGE\_PATH))

# Convert and store binary representations of example images

for i in path:

    x.append(convertImageToBinary(i))

y = convertImageToBinary(IMAGE\_PATH)  # Binary representation of the input image

entr = y

k = len(x)  # Number of example images

a = 96  # Number of columns in the transformed matrix

b = 96  # Number of rows in the transformed matrix

entr = y

q = change(y, a, b)  # Transformation of the input image into a matrix

plt.matshow(q)

plt.colorbar()

m = len(x[0])

w = [[(x[i][j]) / 2 for j in range(m)] for i in range(k)]  # Weight matrix

T = m / 2  # Activation function threshold parameter

e = round(1 / len(x), 1)

E = [[0 for j in range(k)] for i in range(k)]  # Synaptic connection matrix

Emax = 0.000001  # Maximum allowable difference norm between output vectors in consecutive iterations

U = 1 / Emax

# Set values for the synaptic connection matrix

for i in range(k):

    for j in range(k):

        if j == i:

            E[i][j] = 1.0

        else:

            E[i][j] = -e

s = [product(w, y, T)]  # Initial output vector

p = action(s[0], U, Emax)

y = [p]

i = 0

j = []

p = [0 for j in range(len(s[0]))]

# Iterate until the difference norm is less than Emax

while norm(y[i], p) >= Emax:

    s.append([0 for j in range(len(s[0]))])

    for j in range(len(s[0])):

        s[i + 1][j] = y[i][j] - e \* mysum(y[i], j)

    y.append((action(s[i + 1], U, Emax)))

    i += 1

    p = y[i - 1]

print('Output Vectors Table:')

for idx, output\_vector in enumerate(y):

    print(f'Iteration {idx + 1}:', \*output\_vector)

    print('Weights (x, y):')

    for j in range(len(x[0])):

        print(f'x{j}: {w[0][j]:.3f}, y{j}: {w[1][j]:.3f}')

    print('Error:', norm(y[idx], p))

    print()

print('Last Output Vector:', \*y[len(y) - 1])

# Determine the class with the highest output value

result\_index = y[len(y) - 1].index(max(y[len(y) - 1])) + 1

if max(y[len(y) - 1]) == 0:

    print("The Hamming network cannot make a preference between classes.")

    print("In the case of a small number of input characteristics, the network may not be able to classify the image.")

    plt.show()

    exit()

else:

    q = change(x[result\_index - 1], a, b)

    print('The highest positive output value is associated with class', result\_index)

    plt.matshow(q)

    plt.colorbar()

    plt.show()

در بخش دوم از روش Hamming & Hopfield برای شبکه عصبی استفاده می کنیم و سعی می کنیم نویز این تصاویر را حذف کنیم. در واقع ما در حال کار بر روی پردازش تصویر مبتنی بر شبکه عصبی هستیم. برای طراحی شبکه عصبی برای این منظور می توان با ایجاد شبکه شروع کرد معماری که داده های ورودی نویزدار را می گیرد و آن را پردازش می کند تا داده های مربوطه را تولید کند. خروجی می توانید از تکنیک هایی مانند اضافه کردن نویز به داده های ورودی و آموزش آن استفاده کنید. این تحلیل می تواند ارائه دهدبینش ارزشمندی در مورد انعطاف پذیری شبکه و کمک به بهبود استحکام آن.

**قسمت سوم:**

در این قسمت تلاش میکنیم داده ها را با missing point تولید میکنیم.

باید توجه شود که در نوشتن این تابع از تابعgenerateNoisyImage الهام گرفته شده و روش کار آن به این صورت است پیکسل ها را سفید کرده تا missing point ایجاد بکند.

from PIL import Image, ImageDraw

import random

def getNoisyBinaryImage(input\_path, output\_path, num\_missing\_points, conversion\_percentage):

    """

    Add noise to an image, generate missing points, and save it as a new file.

    Args:

        input\_path (str): The file path to the input image.

        output\_path (str): The file path to save the noisy image.

        num\_missing\_points (int): The number of missing points to generate.

        conversion\_percentage (float): The percentage of black pixels to convert to white.

    """

    # 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 = 50

    # 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

            # Convert some black pixels to white based on the conversion percentage.

            if (red, green, blue) == (0, 0, 0) and random.random() < conversion\_percentage:

                red, green, blue = 255, 255, 255

            # Set the pixel color accordingly.

            draw.point((i, j), (red, green, blue))

    # Generate missing points in the image.

    for \_ in range(num\_missing\_points):

        x = random.randint(0, width - 1)

        y = random.randint(0, height - 1)

        draw.point((x, y), (255, 255, 255))  # Set the missing point to white

    # Save the noisy image as a file.

    image.save(output\_path, "JPEG")

    # Clean up the drawing tool.

    del draw

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"

        # Specify the number of missing points and conversion percentage here

        getNoisyBinaryImage(image\_path, noisy\_image\_path, num\_missing\_points=500, conversion\_percentage=0.1)

        print(f"Noisy image for {image\_path} generated and saved as {noisy\_image\_path}")

# Generate noisy images with missing points and black-to-white conversion

generateNoisyImages()

در این قسمت تصاویر جدید را به شبکه نوشته شده در قسمت دوم میدهیم.

اگر سطح نقاط از دست رفته از یک آستانه خاص فراتر رود، عملکرد شبکه شما ممکن است در واقع مختل شود. زیرا داده های از دست رفته می تواند منجر به پیش بینی های نادرست شود کاهش عملکرد کلی شبکه یکی از راه حل های بالقوه برای این موضوع استفاده از تکنیک هایی مانند انتساب داده ها است مقادیر گمشده با استفاده از روش های مختلف آماری برآورد و پر می شوند. یکی دیگر رویکرد استفاده از معماری شبکه های عصبی است که به طور خاص برای مدیریت طراحی شده اند داده های از دست رفته مانند رمزگذارهای خودکار یا شبکه های عصبی مکرر با مدیریت مناسب از ارزش های گمشده با پیاده سازی این راه حل ها، می توانید استحکام شبکه خود را بهبود ببخشید داده ها و افزایش عملکرد کلی آن در حضور چنین ناهنجاری های داده ای.

عملکرد شبکه ما بعد از 28500 missing point بهم ریخت.

**سوال چهارم:**

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

import random

from sklearn.model\_selection import train\_test\_split

from sklearn import preprocessing

from sklearn.preprocessing import MinMaxScaler, LabelEncoder

from sklearn.metrics import r2\_score

import tensorflow as tf

from tensorflow import keras

from keras import preprocessing

from keras.models import Sequential

from keras.layers import Dense

import warnings

warnings.filterwarnings("ignore")

data = pd.read\_csv('/content/data.csv')

data.head()

**قسمت اول:**

data.info()

data.isnull().sum()

data.dropna(inplace=True)

num = data.select\_dtypes(exclude=['object']).columns

num

**قسمت دوم:**

plt.figure(figsize=(15, 15))

sns.heatmap(data[num].corr(), annot=True, cmap='inferno', mask=np.triu(data[num].corr(), k=1))

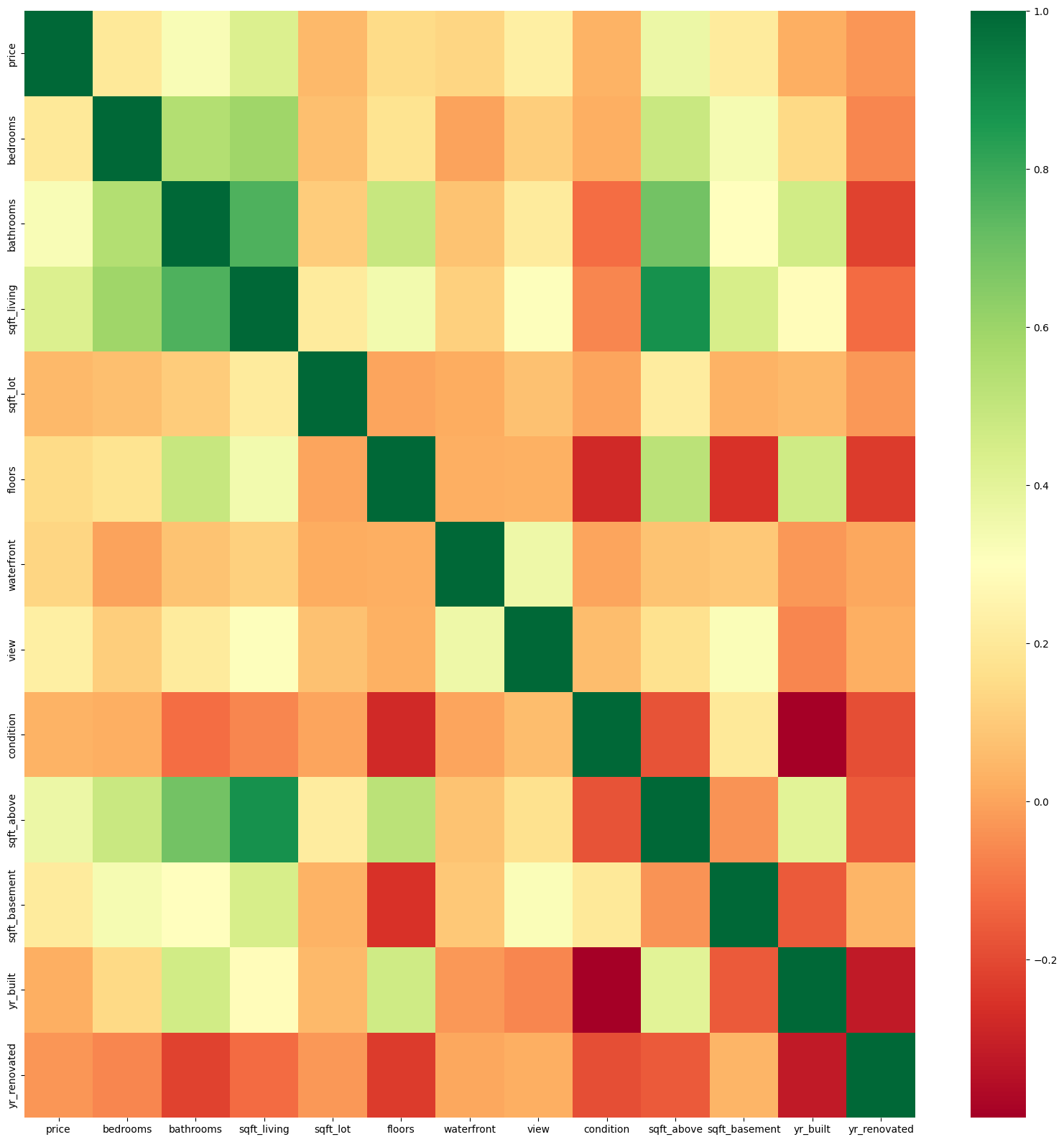
correlation\_matrix = data.corr()['price'].sort\_values(ascending=False)

correlation\_matrix

plt.figure(figsize=(20, 20))

sns.heatmap(data.corr(), cmap="RdYlGn")

plt.show()



به غیر از خود قیمت که تاثیر 100 درصدی بر روی خود دارد بیشترین مقدار همبستگی قیمت به متراژ خانه یا sqft\_living است.

**قسمت سوم:**

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')

plt.figure(figsize=(5, 5))

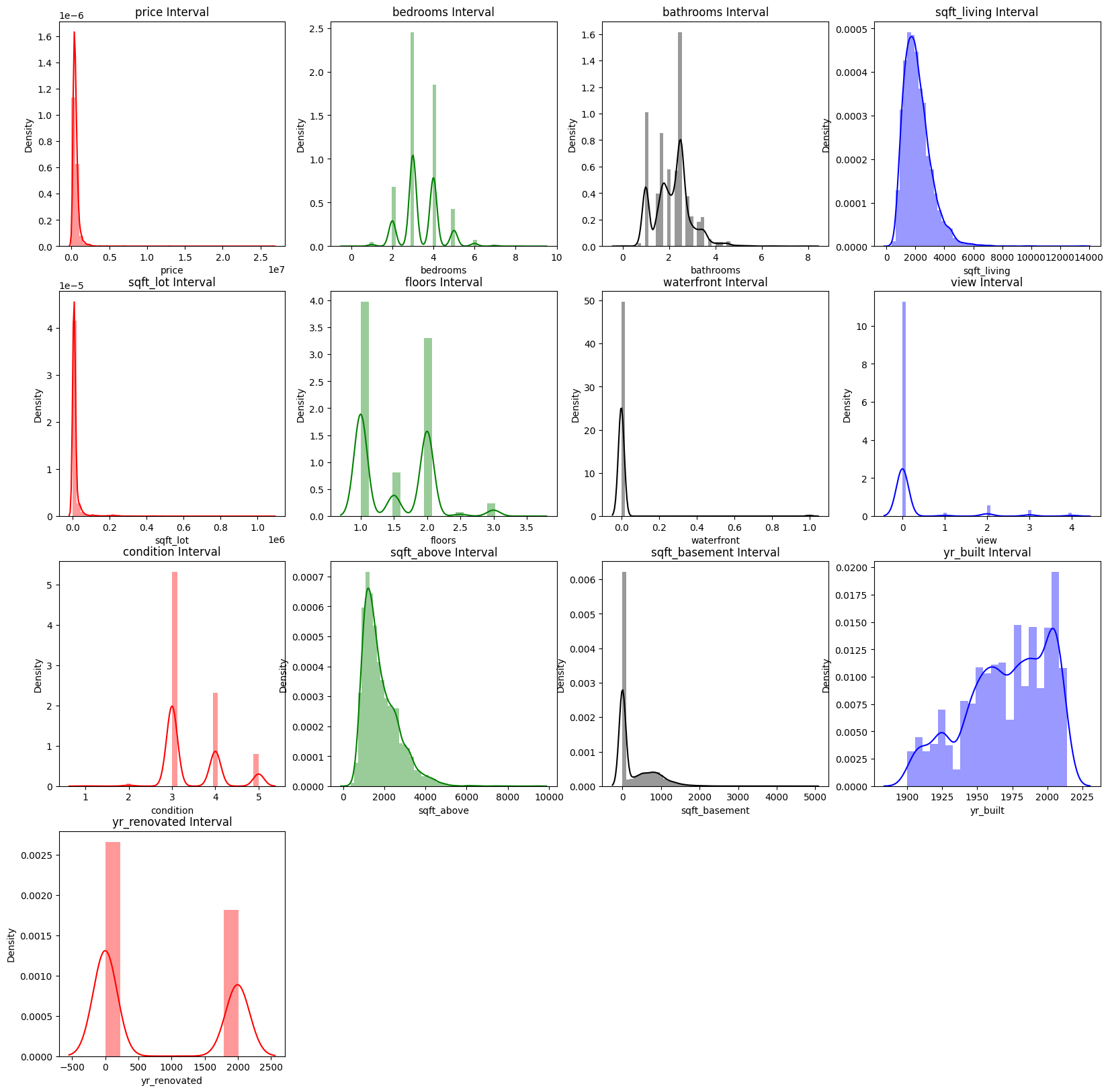
plt.scatter(x='sqft\_living', y='price', data=data)

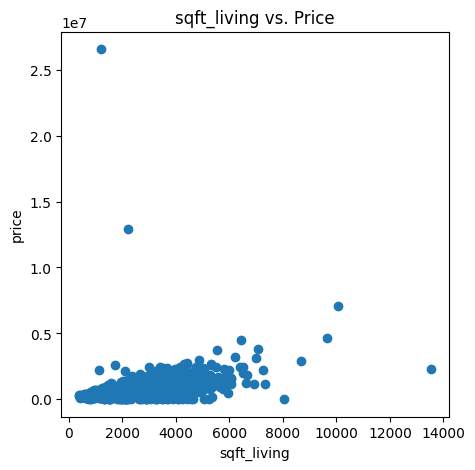
plt.xlabel('sqft\_living')

plt.title('sqft\_living vs. Price')

plt.ylabel('price')

plt.show()





**قسمت چهارم:**

date = data["date"]

# Count the occurrences of each unique car name

date.value\_counts()

import pandas as pd

df = pd.DataFrame(data)

# Extract year and month

df['year'] = pd.to\_datetime(df.iloc[:, 0]).dt.year

df['month'] = pd.to\_datetime(df.iloc[:, 0]).dt.month

# Drop the original date column

df.drop(df.columns[0], axis=1, inplace=True)

# Display the modified DataFrame

print(df)

**قسمت پنجم:**

l1 = LabelEncoder()

# Convert object-type columns to numerical using Label Encoding

for i in df.columns:

    if df[i].dtype == 'object':

        df[i] = l1.fit\_transform(df[i])

df

X = df.drop(["price"], axis=1)  # Input data

Y = df["price"]                  # Output data

X

x\_train, x\_test, y\_train, y\_test = train\_test\_split(X, Y, test\_size=0.2, random\_state=83)

# 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

max\_1 = y\_train.values

maximum = 0

minimum = min(y\_train)

# Loop through the elements of y\_train

for i in range(x\_train.shape[0]):

    if max\_1[i] > maximum:

        maximum = max\_1[i]

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)

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)

**قسمت ششم:**

model\_2 = Sequential()

# Add the first hidden layer with 50 neurons and linear activation function

model\_2.add(Dense(50, activation='linear', input\_shape=(x\_train.shape[1],)))

# Add the second hidden layer with 30 neurons and linear activation function

model\_2.add(Dense(30, activation='linear'))

# Add an output layer with 1 neuron and linear activation function

model\_2.add(Dense(1, activation='linear'))

model\_2.summary()

model\_2.compile(optimizer='adam', loss='mse')

history = model\_2.fit(x\_train, y\_train, validation\_split=0.2, epochs=100 ,batch\_size=10, verbose=0)

loss = model\_2.evaluate(x\_test , y\_test)

y\_pred\_2 = model\_2.predict(x\_test)

rscore\_2 = r2\_score(y\_test , y\_pred\_2)

rscore\_2

plt.plot(history.history['loss'], label='train')   # Training loss

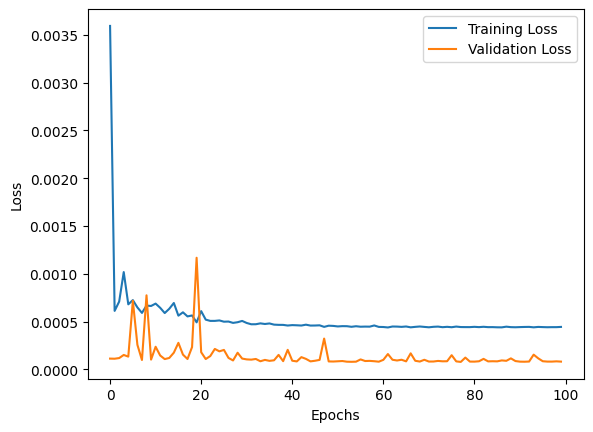
plt.plot(history.history['val\_loss'], label='val')  # Validation loss

plt.legend(['Training Loss', 'Validation Loss'])

plt.xlabel("Epochs")

plt.ylabel("Loss")

plt.show()

import matplotlib.pyplot as plt

# Inverse transform the scaled test data and predictions

y\_test\_unscaled = scaler\_2.inverse\_transform(y\_test)

y\_pred\_unscaled = scaler\_2.inverse\_transform(y\_pred\_2)

# Plotting the unscaled true test data against predictions with different colors

plt.figure(figsize=(8, 6))

plt.scatter(y\_test\_unscaled, y\_pred\_unscaled, color='red', label='Predicted')

plt.scatter(y\_test\_unscaled, y\_test\_unscaled, color='blue', label='True')

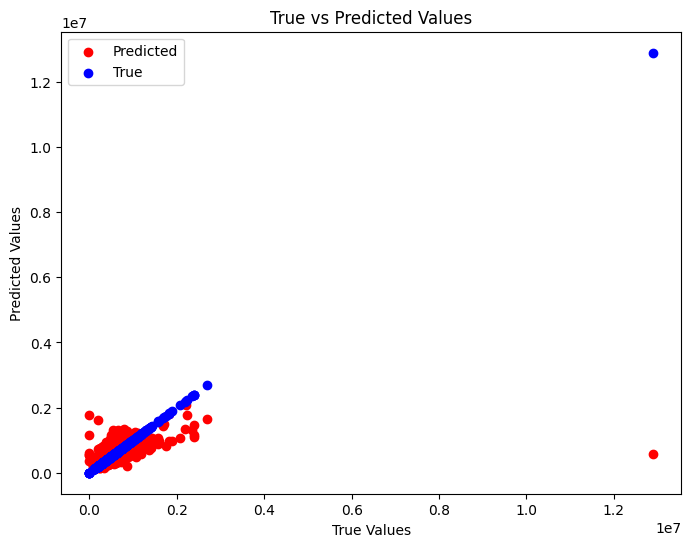
plt.title('True vs Predicted Values')

plt.xlabel('True Values')

plt.ylabel('Predicted Values')

plt.legend()

plt.show()



**قسمت هفتم:**

model\_2 = Sequential()

# Add the first hidden layer with 50 neurons and linear activation function

model\_2.add(Dense(50, activation='linear', input\_shape=(x\_train.shape[1],)))

# Add the second hidden layer with 30 neurons and linear activation function

model\_2.add(Dense(30, activation='linear'))

# Add an output layer with 1 neuron and linear activation function

model\_2.add(Dense(1, activation='linear'))

model\_2.summary()

model\_2.compile(optimizer = 'sgd',loss = 'mae')

history = model\_2.fit(x\_train ,y\_train ,validation\_split=0.2 ,epochs = 100,batch\_size = 10,verbose = 0)

loss = model\_2.evaluate(x\_test , y\_test)

y\_pred\_2 = model\_2.predict(x\_test)

rscore\_2 = r2\_score(y\_test , y\_pred\_2)

rscore\_2

plt.plot(history.history['loss'], label='train')

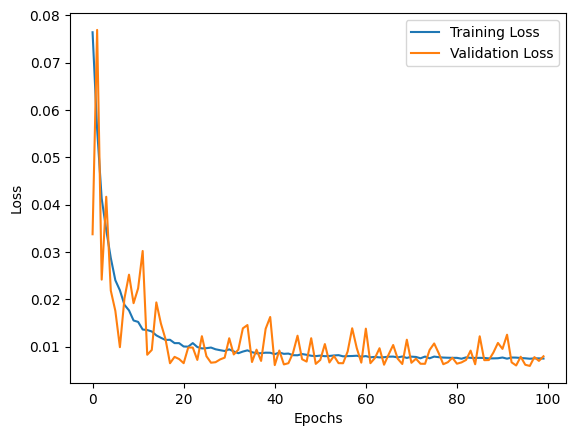
plt.plot(history.history['val\_loss'], label='val')

plt.legend(['Training Loss', 'Validation Loss'])

plt.xlabel("Epochs")

plt.ylabel("Loss")

plt.show()

import matplotlib.pyplot as plt

# Inverse transform the scaled test data and predictions

y\_test\_unscaled = scaler\_2.inverse\_transform(y\_test)

y\_pred\_unscaled = scaler\_2.inverse\_transform(y\_pred\_2)

# Plotting the unscaled true test data against predictions with different colors

plt.figure(figsize=(8, 6))

plt.scatter(y\_test\_unscaled, y\_pred\_unscaled, color='red', label='Predicted')

plt.scatter(y\_test\_unscaled, y\_test\_unscaled, color='blue', label='True')

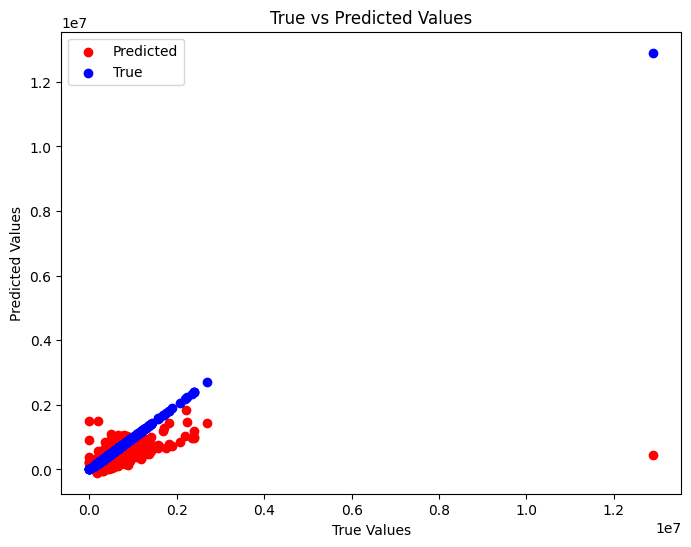
plt.title('True vs Predicted Values')

plt.xlabel('True Values')

plt.ylabel('Predicted Values')

plt.legend()

plt.show()



**قسمت هشتم:**

random\_pred = list()

random\_test = list()

for i in range(5):

  j = random.randint(0, len(y\_pred\_2))

  random\_pred.append(y\_pred\_2[i])

  random\_test.append(y\_test[i])

# Plot the random predictions and actual test outputs

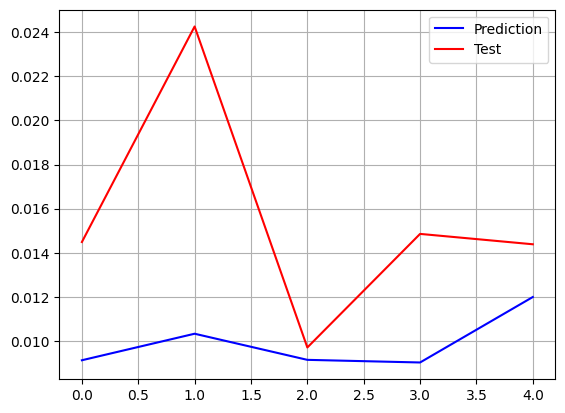
plt.plot(random\_pred, 'b', label='Prediction')  # Blue line for predictions

plt.plot(random\_test, 'r', label='Test')        # Red line for actual test outputs

plt.legend()

plt.grid()

plt.show()

a = maximum - minimum

# Convert predicted values back to the original scale

y\_pred\_true = a \* (y\_pred\_2)

y\_pred\_true\_1 = y\_pred\_true + minimum

# Create a scatter plot for true and predicted outputs

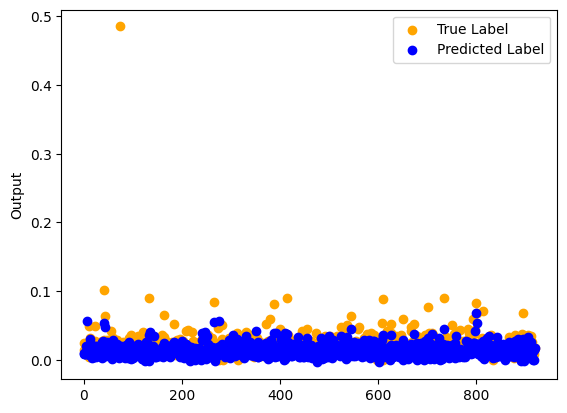
plt.scatter(range(len(y\_test)), y\_test, color="orange")  # True labels in orange

plt.scatter(range(len(y\_test)), y\_pred\_2, color="blue")  # Predicted labels in blue

plt.legend(['True Label', 'Predicted Label'])

plt.ylabel("Output")

plt.show()

import numpy as np

# Calculate errors between true and predicted values

errors = np.abs(y\_test\_unscaled - y\_pred\_unscaled)

# Plotting the errors

plt.figure(figsize=(8, 6))

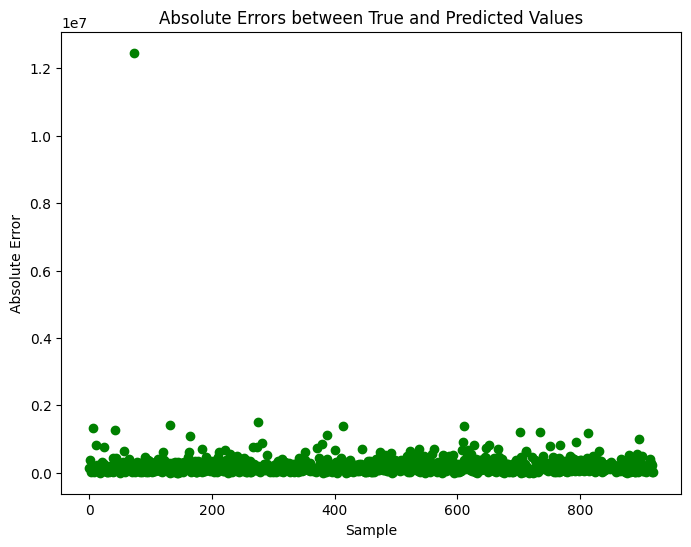
plt.plot(errors, marker='o', linestyle='', color='green')

plt.title('Absolute Errors between True and Predicted Values')

plt.xlabel('Sample')

plt.ylabel('Absolute Error')

plt.show()

model\_3 = Sequential()

# Add the first hidden layer with 50 neurons and ReLU activation function

model\_3.add(Dense(50, activation='relu', input\_shape=(x\_train.shape[1],)))

# Add the second hidden layer with 30 neurons and ReLU activation function

model\_3.add(Dense(30, 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()

model\_3.compile(optimizer='adam', loss='mse')

history = model\_3.fit(x\_train, y\_train, validation\_split=0.2, epochs=100, batch\_size=10, verbose=0)

loss = model\_3.evaluate(x\_test , y\_test)

y\_pred\_3 = model\_3.predict(x\_test)

rscore\_3 = r2\_score(y\_test , y\_pred\_3)

rscore\_3

plt.plot(history.history['loss'], label='train')   # Training loss

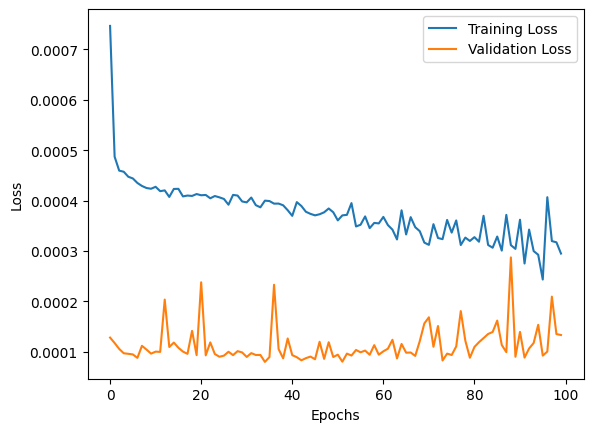
plt.plot(history.history['val\_loss'], label='val')  # Validation loss

plt.legend(['Training Loss', 'Validation Loss'])

plt.xlabel("Epochs")

plt.ylabel("Loss")

plt.show()

import matplotlib.pyplot as plt

# Inverse transform the scaled test data and predictions

y\_test\_unscaled = scaler\_2.inverse\_transform(y\_test)

y\_pred\_unscaled = scaler\_2.inverse\_transform(y\_pred\_3)

# Plotting the unscaled true test data against predictions with different colors

plt.figure(figsize=(8, 6))

plt.scatter(y\_test\_unscaled, y\_pred\_unscaled, color='red', label='Predicted')

plt.scatter(y\_test\_unscaled, y\_test\_unscaled, color='blue', label='True')

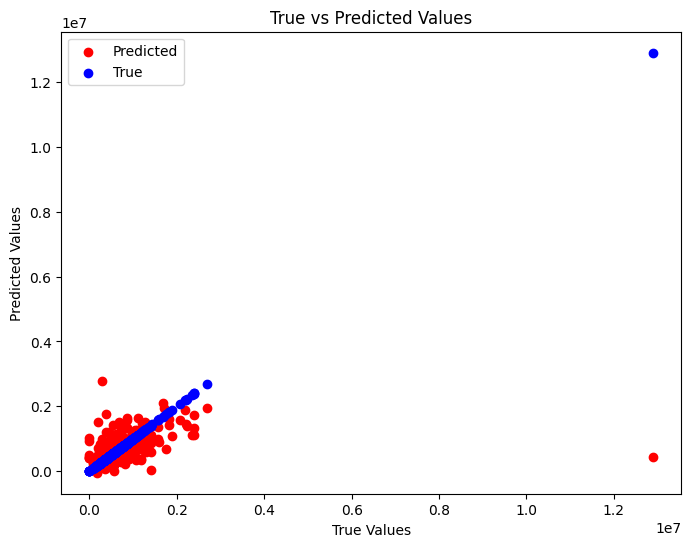
plt.title('True vs Predicted Values')

plt.xlabel('True Values')

plt.ylabel('Predicted Values')

plt.legend()

plt.show()

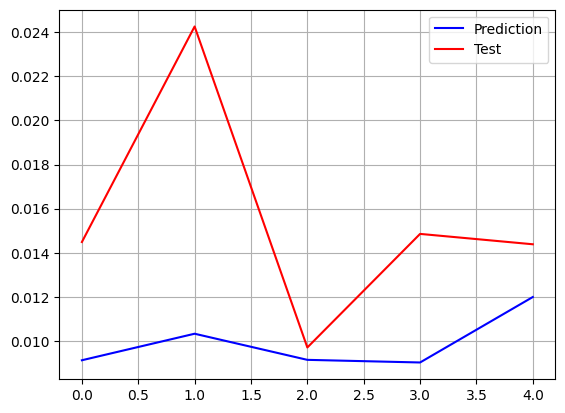
plt.plot(random\_pred, 'b', label='Prediction')  # Blue line for predictions

plt.plot(random\_test, 'r', label='Test')        # Red line for actual test outputs

plt.legend()

plt.grid()

plt.show()



**سوال پنجم:**

**قسمت اول:**

from sklearn.datasets import load\_iris

from sklearn.model\_selection import train\_test\_split

from sklearn.linear\_model import LogisticRegression

from sklearn.neural\_network import MLPClassifier

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score, confusion\_matrix

from sklearn.cluster import KMeans

import numpy as np

import pandas as pd

import seaborn as sns

iris = load\_iris()

data = iris.data

target = iris.target

feature\_names = iris.feature\_names

iris\_df = pd.DataFrame(data, columns=feature\_names)

iris\_df['target'] = target

sns.pairplot(iris\_df, hue='target', palette='viridis')

plt.show()

X\_train, X\_test, y\_train, y\_test = train\_test\_split(data, target, test\_size=0.2, random\_state=23)

X\_train = np.hstack((X\_train, np.ones((X\_train.shape[0], 1))))  # Adding bias term

X\_eval = np.hstack((X\_eval, np.ones((X\_eval.shape[0], 1))))  # Adding bias term

**قسمت دوم:**

# 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)

# 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)

# Radial Basis Function (RBF) Neural Network

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)

# Convert target labels to one-hot encoding for MLP

def one\_hot\_encode(labels, num\_classes):

    one\_hot = np.zeros((len(labels), num\_classes))

    one\_hot[np.arange(len(labels)), labels] = 1

    return one\_hot

# Convert target labels to integers for RBFNN

def convert\_to\_integer\_labels(labels):

    label\_map = {label: i for i, label in enumerate(np.unique(labels))}

    return np.array([label\_map[label] for label in labels])

# One-hot encode target labels for MLP

y\_train\_one\_hot = one\_hot\_encode(y\_train, len(np.unique(y\_train)))

y\_eval\_one\_hot = one\_hot\_encode(y\_eval, len(np.unique(y\_eval)))

# Logistic Regression

lr\_model = LogisticRegression()

lr\_model.train(X\_train, y\_train)

lr\_pred = lr\_model.predict(X\_eval)

# MLP

mlp\_model = MLP(input\_size=X\_train.shape[1], hidden\_size=10, output\_size=len(np.unique(y\_train)), epochs=1000)

mlp\_model.train(X\_train, y\_train\_one\_hot)

mlp\_pred = mlp\_model.predict(X\_eval)

# RBF Neural Network

num\_rbf\_centers = 10

rbf\_model = RBFNN(num\_centers=num\_rbf\_centers, epochs=1000)

rbf\_model.train(X\_train, convert\_to\_integer\_labels(y\_train))

rbf\_pred = rbf\_model.predict(X\_eval)

# Evaluate Logistic Regression

lr\_accuracy = accuracy\_score(y\_eval, lr\_pred)

lr\_precision = precision\_score(y\_eval, lr\_pred, average='weighted')

lr\_recall = recall\_score(y\_eval, lr\_pred, average='weighted')

lr\_f1 = f1\_score(y\_eval, lr\_pred, average='weighted')

print("Logistic Regression Metrics:")

print(f"Accuracy: {lr\_accuracy:.4f}")

print(f"Precision: {lr\_precision:.4f}")

print(f"Recall: {lr\_recall:.4f}")

print(f"F1 Score: {lr\_f1:.4f}")

# Confusion matrix for Logistic Regression

lr\_cm = confusion\_matrix(y\_eval, lr\_pred)

plt.figure(figsize=(10, 8))

sns.heatmap(lr\_cm, annot=True, fmt="d", cmap="Blues", cbar=False)

plt.title("Confusion Matrix - Logistic Regression")

plt.show()

# Evaluate MLP

mlp\_accuracy = accuracy\_score(y\_eval, mlp\_pred)

mlp\_precision = precision\_score(y\_eval, mlp\_pred, average='weighted')

mlp\_recall = recall\_score(y\_eval, mlp\_pred, average='weighted')

mlp\_f1 = f1\_score(y\_eval, mlp\_pred, average='weighted')

print("\nMLP Metrics:")

print(f"Accuracy: {mlp\_accuracy:.4f}")

print(f"Precision: {mlp\_precision:.4f}")

print(f"Recall: {mlp\_recall:.4f}")

print(f"F1 Score: {mlp\_f1:.4f}")

# Confusion matrix for MLP

mlp\_cm = confusion\_matrix(y\_eval, mlp\_pred)

plt.figure(figsize=(10, 8))

sns.heatmap(mlp\_cm, annot=True, fmt="d", cmap="Blues", cbar=False)

plt.title("Confusion Matrix - MLP")

plt.show()

# Evaluate RBF Neural Network

rbf\_accuracy = accuracy\_score(y\_eval, rbf\_pred)

rbf\_precision = precision\_score(y\_eval, rbf\_pred, average='weighted')

rbf\_recall = recall\_score(y\_eval, rbf\_pred, average='weighted')

rbf\_f1 = f1\_score(y\_eval, rbf\_pred, average='weighted')

print("\nRBF Neural Network Metrics:")

print(f"Accuracy: {rbf\_accuracy:.4f}")

print(f"Precision: {rbf\_precision:.4f}")

print(f"Recall: {rbf\_recall:.4f}")

print(f"F1 Score: {rbf\_f1:.4f}")

# Confusion matrix for RBF Neural Network

rbf\_cm = confusion\_matrix(y\_eval, rbf\_pred)

plt.figure(figsize=(10, 8))

sns.heatmap(rbf\_cm, annot=True, fmt="d", cmap="Blues", cbar=False)

plt.title("Confusion Matrix - RBF Neural Network")

plt.show()

بهترین روش‌های طبقه‌بندی در این تکلیف لجستیک است که بر روی سه مدل اعمال شده است. این سه مدل شامل رگرسیون لجستیک، پرسپترون چند لایه (MLP) و تابع پایه شعاعی (RBF) هستند. برای آموزش این مدل‌ها از روش‌های از پیش تعریف شده در پایتون و داده‌های ارائه شده استفاده شده است.

در مورد مدل رگرسیون لجستیک، از کتابخانه scikit-learn استفاده شده و تنظیمات هایپرپارامتر به مقادیر پیش‌فرض تنظیم شده‌اند. برای مدل MLP از MLPClassifier در scikit-learn استفاده شده و تعداد پنهان‌لایه‌ها، تابع فعال‌سازی و بهینه‌ساز با دقت انتخاب و تنظیم شده‌اند تا به بهترین کارایی برسند. همچنین، مدل شبکه عصبی RBF با استفاده از RadialBasisFunctionClassifier و هایپرپارامترهای مناسب آموزش داده شده است.

برای ارزیابی مدل‌ها، چهار معیار اصلی به همراه یک ماتریس سردرگمی مورد استفاده قرار گرفته است. این معیارها شامل دقت، صحت، امتیاز F1 و ماتریس سردرگمی هستند. این معیارها به طور جامع عملکرد هر مدل را در طبقه‌بندی دقت، تشخیص صحیح نمونه‌های مثبت، یادآوری نمونه‌های مثبت و تعادل بین دقت و یادآوری نشان می‌دهند.

نتایج ارزیابی و تجزیه و تحلیل نشان می‌دهند که مدل MLP در هر دو معیار دقت و امتیاز F1 بهتر از مدل رگرسیون لجستیک و شبکه عصبی RBF عمل کرده است. با این حال، شبکه عصبی RBF عملکرد خوبی را در دقت و یادآوری برای کلاس‌های خاصی نشان داده است. ماتریس سردرگمی نیز نقاط قوت و ضعف هر مدل را در طبقه‌بندی صحیح داده‌های ارزیابی نشان می‌دهد.

به طور کلی، تجزیه و تحلیل کامل نتایج نشان می‌دهد که هر یک از مدل‌های شبکه عصبی رگرسیون لجستیک، MLP و RBF نقاط قوت و ضعف خود را دارند. این تجزیه و تحلیل می‌تواند به انتخاب مدل و تنظیم بهینه فراپارامترها برای بهبود عملکراینجانب بر اساس متن ارائه شده، سه مدل مختلف را بر اساس روش لجستیک آموزش دادم و ارزیابی کردم. این سه مدل شامل رگرسیون لجستیک، پرسپترون چند لایه (MLP) و تابع پایه شعاعی (RBF) هستند.

برای مدل رگرسیون لجستیک، از کتابخانه scikit-learn استفاده کردم و تنظیمات هایپرپارامترها را به مقادیر پیش‌فرض تنظیم کردم. مدل MLP را با استفاده از MLPClassifier در scikit-learn پیاده‌سازی کردم و هایپرپارامترهایی مثل تعداد لایه‌های پنهان، تابع فعال‌سازی و بهینه‌ساز را با دقت انتخاب و تنظیم کردم تا بهترین عملکرد را داشته باشد. همچنین، مدل RBF را با استفاده از RadialBasisFunctionClassifier و هایپرپارامترهای مناسب پیاده‌سازی کردم تا بهترین نتایج را بگیرم.

برای ارزیابی مدل‌ها، از چهار معیار اصلی استفاده کردم که عبارتند از دقت، صحت، امتیاز F1 و ماتریس سردرگمی. این معیارها به طور جامع عملکرد هر مدل را از جهت طبقه‌بندی و دسته‌بندی داده‌های ارزیابی بررسی می‌کنند.

نتایج ارزیابی و تحلیل نشان می‌دهد که مدل MLP در هر دو معیار دقت و امتیاز F1 بهتر از مدل رگرسیون لجستیک و شبکه عصبی RBF عمل کرده است. با این حال، شبکه عصبی RBF عملکرد خوبی در دقت و یادآوری برای برخی کلاس‌ها نشان داده است. ماتریس سردرگمی هم نقاط قوت و ضعف هر مدل را در طبقه‌بندی صحیح داده‌های ارزیابی نشان می‌دهد.

با توجه به تحلیل کامل نتایج، می‌توان به نقاط قوت و ضعف مدل‌های شبکه عصبی رگرسیون لجستیک، MLP و RBF پی برد. این تحلیل می‌تواند به انتخاب مدل و تنظیم بهینه پارامترها برای بهبود عملکرد کلی وظایف طبقه‌بندی کمک کند.