



۱۴۰۷

دانشگاه صنعتی خواجه نصیرالدین طوسی

دانشگاه مهندسی برق _ گرایش کنترل

مینی پروژه شماره دو

یادگیری ماشین

نگارش

فاطمه امیری

۴۰۲۰۲۴۲۴

لينك گوگل كولب

لينك گيت هاب

استاد مربوطه

جناب آقای دکتر علیاری

بهار ۱۴۰۳

بِسْمِ اللّٰهِ الرَّحْمٰنِ الرَّحِيْمِ

فهرست مطالب

| | |
|----|-----------|
| ۱ | سوال یک |
| ۱ | بخش اول |
| ۴ | بخش دوم |
| ۶ | بخش سوم |
| ۱۲ | سوال دو |
| ۱۷ | بخش اول |
| ۲۲ | بخش دوم |
| ۲۳ | بخش سوم |
| ۳۴ | بخش چهارم |
| ۴۰ | سوال سه |
| ۴۰ | بخش اول |
| ۴۹ | بخش دوم |
| ۵۳ | بخش سوم |
| ۵۶ | سوال چهار |

سوال یک

بخش اول

۱. فرض کنید در یک مسئله طبقه‌بندی دو کلاسه، دو لایه انتهايی شبکه شما فعال‌ساز ReLU و سیگمويد است. چه اتفاقی می‌افتد؟

در یک مسئله دسته‌بندی دو کلاسه، انتخاب تابع‌های فعال‌سازی در لایه‌های نهایی شبکه عصبی می‌تواند به طور قابل توجهی بر عملکرد و تفسیر خروجی آن تأثیر بگذارد. تابع فعال‌ساز در شبکه عصبی وظیفه‌ی اعمال یک تبدیل غیرخطی بر خروجی هر نورون را دارد. این تبدیل به شبکه کمک می‌کند تا الگوهای پیچیده و غیرخطی را یاد بگیرد. دو تابع فعال‌ساز پرکاربرد عبارتند از:

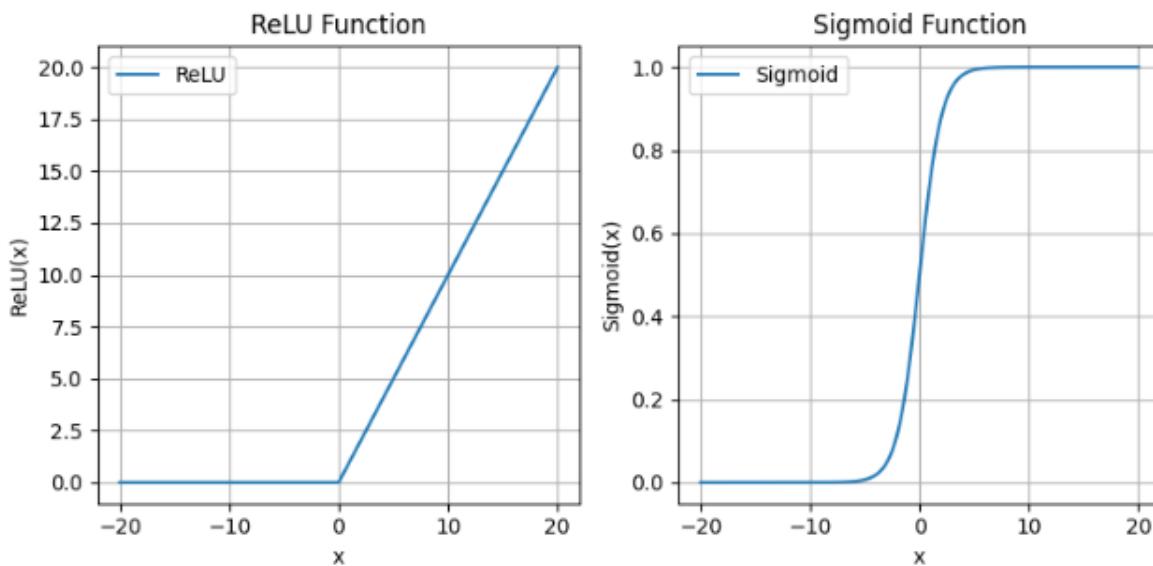
۱. ReLU : که خروجی آن اگر ورودی مثبت باشد، برابر با ورودی است و در غیر این صورت صفر است. این تابع به دلیل سادگی و کارایی در کاهش مشکل ناپدید شدن گرادیان بسیار محبوب است. این تابع اجازه نمیدهد که همه نورون‌ها در یک زمان فعال باشند، یعنی اگر یک ورودی منفی باشد، این تابع آن را صفر می‌کند و به نورون اجازه فعال‌سازی نمی‌دهد، در واقع با این کار تعداد نورون‌های فعال کاهش می‌یابد و محاسبات شبکه نیز کاهش می‌یابد.
فرمول محاسبه آن به صورت زیر است :

$$a_{ReLU}(z) = \max(0, z)$$

۲. سیگموئید : که خروجی آن یک مقدار بین ۰ و ۱ است و معمولاً برای مسائل دسته‌بندی دودویی استفاده می‌شود، زیرا می‌تواند احتمال تعلق به یک کلاس خاص را مدل‌سازی کند. (یک تابع پیوسته مشتق پذیر غیرخطی است)
فرمول محاسبه آن به صورت زیر است :

$$a_{sigmoid}(z) = \frac{1}{1 + e^{-z}}$$

با استفاده از کد پایتون ، شکل تابع های فوق به صورت زیر است :



هنگامی که داده‌ها از تابع ReLU عبور می‌کنند، مقادیر مثبت بدون تغییر باقی می‌مانند و مقادیر منفی به صفر تبدیل می‌شوند. بنابراین، خروجی این لایه شامل اعداد غیرمنفی است. پس از آن، خروجی لایه قبلی (a) در وزن‌های لایه نهایی ضرب می‌شود و مقادیر $[z^{last}]$ به تابع فعال‌سازی سیگموید وارد می‌شوند. در یک طبقه‌بندی دوکلاسه، لایه نهایی فقط یک نورون دارد و خروجی آن در بازه $[0, 1]$ خواهد بود.

در طبقه‌بندی دوکلاسه، نیازی به استفاده از دو نورون نیست زیرا اگر یک ورودی به یک کلاس تعلق نداشته باشد، به کلاس دیگر تعلق خواهد داشت. به همین دلیل از یک نورون استفاده می‌کنیم تا حجم محاسبات را کاهش دهیم. به عبارت دیگر، درصد تعلق ورودی به یک کلاس را محاسبه می‌کنیم و هر چقدر که به این کلاس تعلق نداشته باشد، به دیگری تعلق دارد.

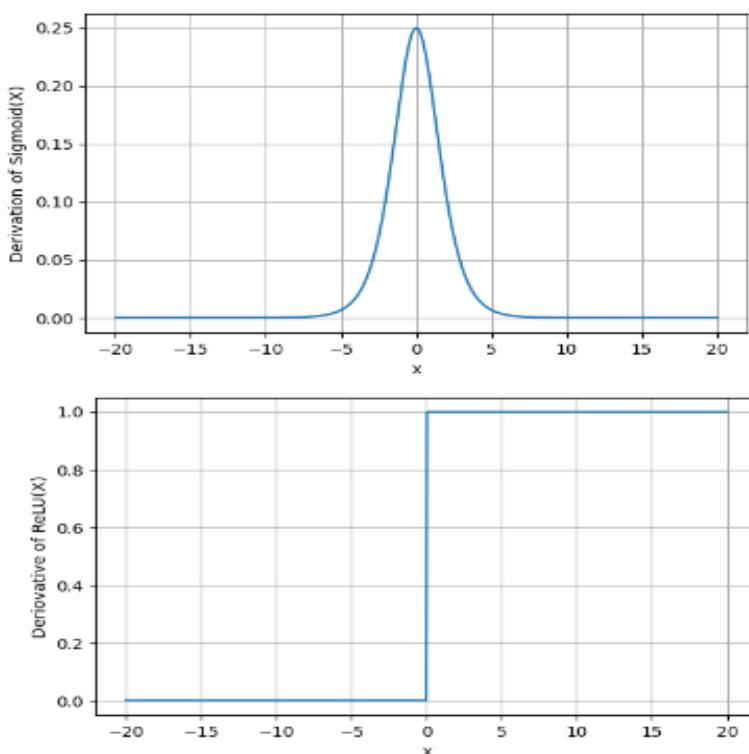
در نهایت، با مقایسه مقدار خروجی با یک آستانه یا threshold ، تعیین می‌کنیم که ورودی به کلاس ۱ تعلق دارد یا به کلاس ..

تابع سیگموید زمانی که اشباع می‌شود، در فرآیند Backward Propagation مشکلاتی ایجاد می‌کند زیرا شبیه آن در نزدیکی صفر بسیار کوچک است. این باعث می‌شود که گرادیان‌ها تقریباً ناپدید شوند و به شبکه اجازه یادگیری موثر را نمی‌دهد، به ویژه در لایه‌های عمیق‌تر. این مشکل به نام "مشکل ناپدید شدن گرادیان" شناخته می‌شود و منجر به سرعت یادگیری بسیار کند می‌شود، زیرا تغییرات وزن‌ها بسیار کم خواهد بود.

در مقابل، تابع ReLU زمانی که ورودی بزرگ‌تر از صفر باشد، خروجی آن برابر با ورودی است و شیب آن برابر با یک است. این ویژگی باعث می‌شود که گرادیان‌ها در طول Backward Propagation ثابت بمانند و ناپدید نشوند. به همین دلیل، ReLU معمولاً برای شبکه‌های عمیق ترجیح داده می‌شود زیرا باعث می‌شود که مدل به راحتی یاد بگیرد و تغییرات وزن‌ها سریع‌تر انجام شود. با این حال، ReLU نیز مشکلات خاص خود را دارد. یکی از این مشکلات " ReLU مرده" است، یعنی نورون‌هایی که خروجی آن‌ها همیشه صفر است و دیگر در فرآیند یادگیری شرکت نمی‌کنند. این اتفاق زمانی می‌افتد که ورودی به ReLU منفی باشد و باعث شود که نورون به طور دائمی غیرفعال شود.

استفاده از ترکیب این دو تابع فعال‌سازی سیگموید و ReLU می‌تواند به بهره‌برداری از مزایای هر دو کمک کند. به عنوان مثال، می‌توان از ReLU در لایه‌های مخفی استفاده کرد تا مشکلات ناپدید شدن گرادیان کاهش یابد و از سیگموئید در لایه خروجی استفاده کرد تا خروجی به صورت احتمال بین ۰ و ۱ تفسیر شود. این ترکیب معمولاً به بهبود عملکرد مدل کمک می‌کند، به ویژه در مسائل پیچیده که نیاز به یادگیری عمیق دارند. با این روش، شبکه می‌تواند از غیرخطی بودن و پراکندگی ReLU بهره‌مند شود و در عین حال یک تفسیر احتمالی از خروجی داشته باشد که توسط تابع سیگموید فراهم می‌شود.

مشتق این دو تابع نیز به صورت زیر است :



بخش دوم

۲. یک جایگزین برای ReLU در معادله ۱ آورده شده است. ضمن محاسبه گرادیان آن، حداقل یک مزیت آن نسبت به ReLU را توضیح دهید.

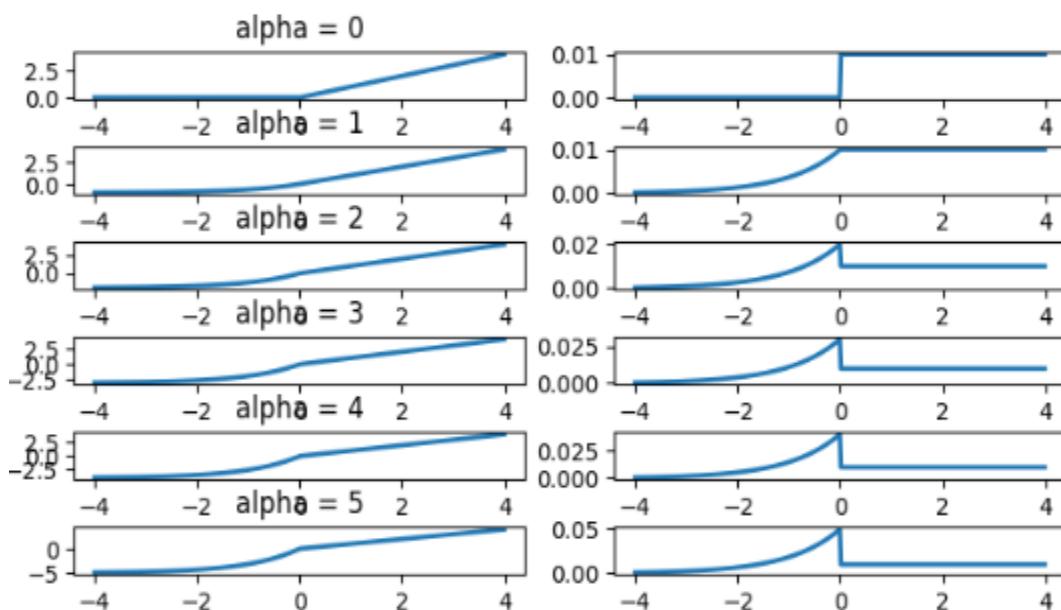
$$\text{ELU}(x) = \begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases} \quad (1)$$

در ابتدا به محاسبه گرادیان آن می پردازیم. (تابع جایگزین را f فرض می کنیم)

$$f = \begin{cases} x & ; x \geq 0 \\ \alpha(e^{-x} - 1); & x < 0 \end{cases} \Rightarrow \frac{\partial f}{\partial x} = \begin{cases} 1 & ; x \geq 0 \\ -\alpha e^{-x} & ; x < 0 \end{cases}$$

$$\stackrel{\alpha=1}{\Rightarrow} \frac{\partial f}{\partial x} = \begin{cases} 1 & ; x \geq 0 \\ -e^{-x} & ; x < 0 \end{cases}$$

برای بررسی دقیق تر مقادیر مختلفی را به ازای الfa قرار می دهیم و خروجی را بدست می آوریم. به ازای الfa از صفر تا ۵ خروجی توسط کد پایتون به صورت زیر است :



در شکل بالا تابع EReLU و گرادیان آن برای الfa های مختلف است، بدین صورت که تصاویر سمت چپ مربوط به خود تابع و تصاویر سمت راست مربوط به مشتق آن می باشند. میبینیم که اگر مقدار $\alpha = 0$ باشد انگاه تابع

ReLU بدست می آید (مانند قسمت قبل) . این تابع به ازای $x = 0$ در نقطه $\alpha = 1$ دارای مشتق می باشد.
در هر α دیگری، در نقطه $x = 0$ مشتق نداریم.

❖ برتری این تابع نسبت به ReLU عبارت است از:

۱. قابلیت خروجی دادن برای مقادیر منفی

یکی از مهم‌ترین مزایای ELU نسبت به ReLU این است که برای ورودی‌های منفی نیز خروجی‌های غیر صفر تولید می‌کند. در حالی که ReLU برای مقادیر منفی ورودی صفر تولید می‌کند، ELU با استفاده از یک تابع نمایی برای ورودی‌های منفی، خروجی‌های منفی تولید می‌کند. این ویژگی به شبکه عصبی کمک می‌کند تا اطلاعات بیشتری از ورودی‌های منفی حفظ کند و از مشکل "ReLU مرده" جلوگیری کند. مشکل "ReLU مرده" زمانی رخ می‌دهد که نورون‌های ReLU به دلیل ورودی‌های منفی یا صفر به طور دائمی غیرفعال می‌شوند.

۲. نرمی و مشتق‌پذیری بیشتر در نقطه صفر

تابع ELU در نقطه $x = 0$ پیوسته و مشتق‌پذیر است، در حالی که ReLU در نقطه $x = 0$ ناپیوستگی دارد و مشتق‌پذیر نیست. این نرمی و مشتق‌پذیری باعث می‌شود که گرادیان‌ها به طور پیوسته و هموار تغییر کنند و فرآیند آموزش شبکه عصبی پایدارتر و مؤثرتر باشد. نرمی بیشتر در نقطه $x = 0$ باعث می‌شود که ELU در بهینه‌سازی و همگرایی شبکه عصبی عملکرد بهتری داشته باشد.

۳. کاهش میانگین خروجی و تثبیت گرادیان‌ها

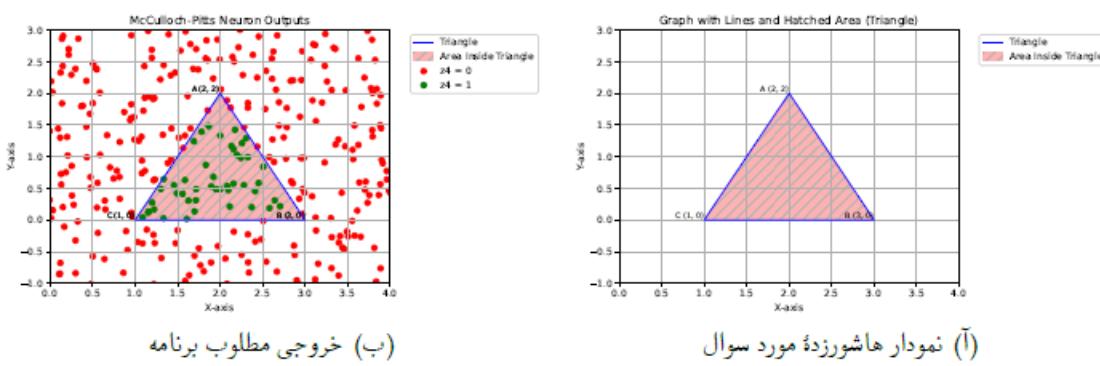
ReLU با داشتن مقادیر منفی برای ورودی‌های منفی، میانگین خروجی‌ها را به سمت صفر نزدیک می‌کند. این ویژگی می‌تواند به تثبیت و تسريع فرآیند آموزش کمک کند، زیرا باعث می‌شود که گرادیان‌ها به طور مؤثرتری جریان یابند و ناپدید شدن یا انفجار گرادیان‌ها کاهش یابد. ReLU این خاصیت را ندارد و میانگین خروجی آن مثبت است، که می‌تواند به عدم تعادل در جریان گرادیان‌ها منجر شود.

۴. سرعت همگرایی بیشتر

شبکه‌های عصبی با استفاده از ELU می‌توانند سریع‌تر همگرا شوند. به دلیل ویژگی‌های منحصر به فرد ELU مانند تولید مقادیر منفی، نرمی بیشتر، و کاهش میانگین خروجی، فرآیند بهینه‌سازی شبکه سریع‌تر و پایدارتر انجام می‌شود. این ویژگی به خصوص در شبکه‌های عمیق که مشکلات ناپدید شدن گرادیان و کندی آموزش شایع‌تر هستند، بسیار مفید است.

بخش سوم

۳. به کمک یک نورون ساده یا پرسپترون یا نورون McCulloch-Pitts^۱ شبکه‌ای طراحی کنید که بتواند ناحیه هاشورزدهٔ داخل مثلثی که در نمودار شکل ۱ نشان داده شده را از سایر نواحی تفکیک کند. پس از انجام مرحله طراحی شبکه (که می‌تواند به صورت دستی انجام شود)، برنامه‌ای که در این دفترچه کد و در کلاس برای نورون McCulloch-Pitts آموخته‌اید را به گونه‌ای توسعه دهید که ۲۰۰۰ نقطه رandom تولید کند و آن‌ها را به عنوان ورودی به شبکهٔ طراحی شده توسط شما دهد و نتایجی که خروجی «۱» تولید می‌کنند را با رنگ سبز و نتایجی که خروجی «۰» تولید می‌کنند را با رنگ قرمز نشان دهد. خروجی تولید شده توسط برنامه شما باید به صورتی که در شکل ۱ (ب) نشان داده شده است باشد (به محدودهٔ عددی محورهای x و y هم دقت کنید). اثر اضافه کردن دوتابع فعال‌ساز مختلف به فرآیند تصمیم‌گیری را هم بررسی کنید.



شکل ۱: نمودارهای مربوط به بخش «۳» سوال اول و خروجی برنامه.

برای انجام این بخش، از مدل MCP استفاده می‌کنیم. فرض می‌کنیم که سه خط وجود دارند و اگر نقطه‌ای نسبت به این خطوط شرایط خاصی را برآورده کند؛ مثلاً پایین‌تر از یک خط و بالاتر از دو خط دیگر باشد، در ناحیه هاشورخورده قرار می‌گیرد. چون در این مسئله نیاز است که یک سری شرایط به صورت درست یا نادرست بررسی شوند، از نورون MCP استفاده می‌کنیم.

در اینجا از تابع فعال ساز استفاده نمی‌کنیم و کلاس McCulloch pittis به صورت زیر تعریف می‌شود:

```
#define Muculloch pittis
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) >= 0 :
            return 1
        else :
            return 0
```

برای تعیین نقاطی که در ناحیه هاشورخورده قرار دارند، باید معادلات خطوطی که این ناحیه را ایجاد می‌کنند به صورت مناسب تعریف شوند. برای درک بهتر این موضوع، تئوری حل مسئله را توضیح می‌دهیم و سه معادله خط به دست می‌آوریم که از تقاطع این خطوط، مثلث مورد نظر ایجاد می‌شود.

سه معادله خط به صورت زیر هستند :

- 1) $y - 2 = -2(x - 2) \rightarrow y + 2x - 6 = 0$
- 2) $y - 0 = 0(x - 3) \rightarrow y = 0$
- 3) $y - 0 = 2(x - 1) \rightarrow y - 2x + 2 = 0$

اکنون برای آن که نقاط بین این خطوط قرار گیرند، باید در شرط‌های زیر صدق کنند :

$$y + 2x - 6 \leq 0 \Rightarrow -2x - y \geq -6$$

$$y \geq 0$$

$$y - 2x + 2 \leq 0 \Rightarrow 2x - y \geq 2$$

با توجه به معادلات حاصل، می‌خواهیم تعداد نورون‌های MCP مورد نیاز را محاسبه کنیم. با توجه به اینکه با استفاده از سه معادله موفق به تشخیص ناحیه مورد نظر شده‌ایم، باید سه نورون MCP را طراحی کنیم. هدف این سه نورون این است که با گرفتن ورودی‌های خود، که مختصات x و y هستند، اعلام کنند که آیا نقطه مورد نظر در ناحیه مورد نظر واقع شده یا خیر. علاوه بر این، یک نورون دیگر نیز باید طراحی شود که با دریافت خروجی سه نورون اول، تصمیم بگیرد که آیا نقطه مورد نظر درون ناحیه تعیین شده قرار دارد یا خیر.

در نتیجه، ما سه نورون داریم که ورودی‌های x و y را دریافت کرده و اگر خروجی تمامی این نورون‌ها True باشد، آنگاه با گذر از نورون نهایی(neur4) که پس از نورون‌های اولیه قرار می‌گیرد، تصمیم گیری می‌شود که آیا نقطه در ناحیه مشخص شده قرار دارد یا خیر. وزن‌ها و آستانه‌های نورون‌ها به شکل زیر محاسبه می‌شوند:

```
#The equation of the lines of the triangle
neur1 = McCulloch_Pitts_neuron([-2 , -1] , 6 )
neur2 = McCulloch_Pitts_neuron([+2 , -1] , -2 )
neur3 = McCulloch_Pitts_neuron([ 0 , +1] , 0 )
neur4 = McCulloch_Pitts_neuron([1 , 1 , 1] , -3 )
```

که در واقع اولین عضو وزن ها هستند و عضو دوم threshold ها هستند. لازم به ذکر است که ما محور x و y نداریم و منظور از x و y در معادلات بالا، همان x_1 و x_2 است و وزن ها همان ضرایب x ها هستند و هم همان بایاس عرض از مبدأ می باشد.

درواقع در کد بالا داده هایی که داخل مثلث باشند مقدار صفر می گیرند یعنی مقدار برای هر سه خط، یک می شود و از مقدار ۳ کم میشود و درنهایت صفر میشود. (neur4). وزن های این نورون به این معنی است که نقاطی که توسط نورون های قبلی تشخیص داده شده اند (نقاطی که داخل مثلث قرار دارند)، همگی به عنوان ورودی به این نورون وارد می شوند. زمانی که همه این ورودی ها یک باشند، مقدار خطی این نورون (مجموع وزن ها ضرب در ورودی ها به اضافه آستانه) برابر با $0 = -3 + (1 * 1) + (1 * 1) + (1 * 1)$ می شود.

پس یک تابع Triangle می نویسیم که به صورت زیر است :

```
#define model for dataset
def Triangle(x,y):
    #The equation of the lines of the triangle
    neur1 = McCulloch_Pitts_neuron([-2, -1], 6)
    neur2 = McCulloch_Pitts_neuron([+2, -1], -2)
    neur3 = McCulloch_Pitts_neuron([ 0, +1], 0)
    neur4 = McCulloch_Pitts_neuron([1, 1, 1], -3)

    zone1 = neur1.model(np.array([x,y]))
    zone2 = neur2.model(np.array([x,y]))
    zone3 = neur3.model(np.array([x,y]))
    zone4 = neur4.model(np.array([zone1, zone2, zone3]))

    return list([zone4])
```

در این شبکه، خروجی های سه نورون اول به عنوان ورودی به نورون چهارم می رساند. اگر خروجی همه سه نورون اول برابر ۱ باشد، به این معنی است که نقطه مورد نظر در ناحیه مورد نظر قرار دارد. اما اگر حداقل یکی از خروجی های نورون های اولیه صفر باشد، نورون نهایی مقدار ۰ را باز می گرداند، که نشان می دهد که نقطه مورد نظر در ناحیه مورد نظر قرار نمی گیرد.

اکنون ۲۰۰ نقطه رندوم را به مدل می دهیم تا تشخیص دهد و اگر نقطه داخل ناحیه بود با رنگ سبز و اگر نقطه خارج از ناحیه بود با رنگ قرمز مشخص کند.

```

num_point = 2000
x_val = np.random.uniform( 0 , 4 , num_point )      #x_asis
y_val = np.random.uniform( -1 , 3 , num_point )      #y_asis

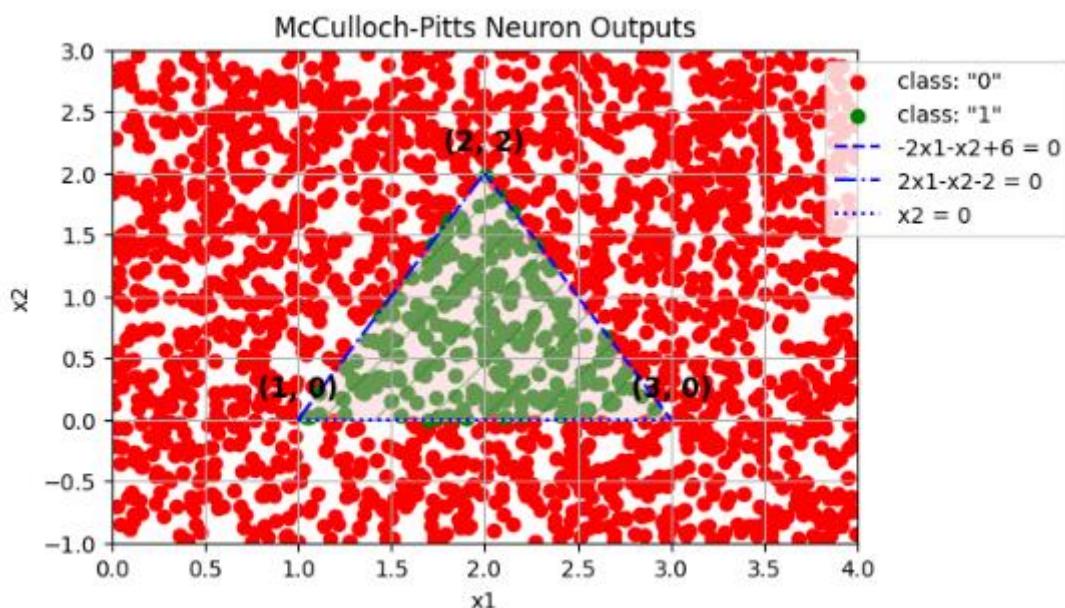
red_point = []           # outside zone
green_point = []         # inside zone

for i in range(num_point):
    flag = Triangle(x_val[i] , y_val[i])
    if flag == [0] :
        red_point.append((x_val[i] , y_val[i]))
    else:
        green_point.append((x_val[i] , y_val[i]))


# Separate x and y values for red and green points
red_x, red_y = zip(*red_point)
green_x, green_y = zip(*green_point)

```

حال خروجی به صورت زیر است :

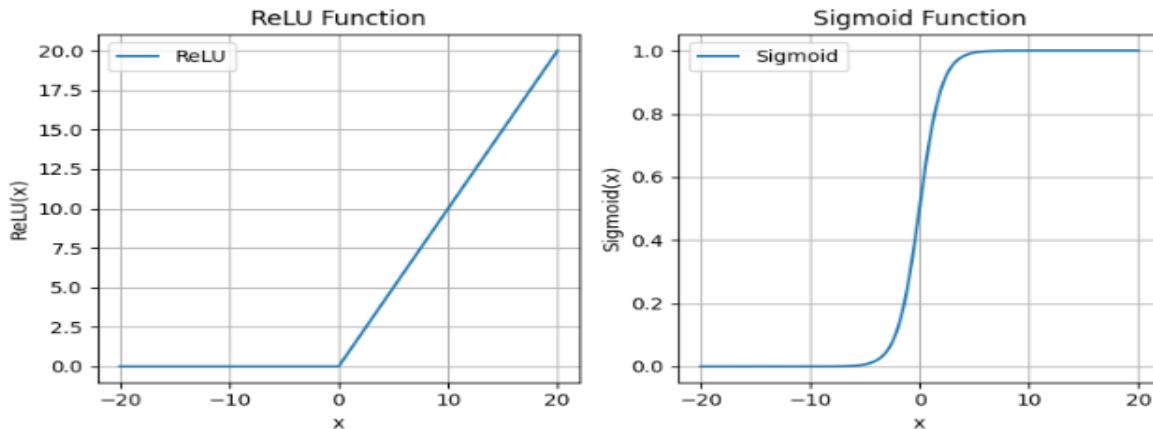


می بینیم که مدل به خوبی عمل کرده است و نقاط را به خوبی تشخیص داده است.

مدلی که در اینجا استفاده کردیم با استفاده از تابع فعال‌ساز **hard limit** بود. این تابع فعال‌ساز، برای تمامی نورون‌ها استفاده شد و با استفاده از تابع **hard limit**، به عنوان خروجی صرفاً قرار داشتن یا نداشتن نقطه در ناحیه خواسته شده معلوم می‌شود.

در این قسمت اثر اضافه کردن دو تابع فعال ساز به فرایند تصمیم‌گیری را بررسی می‌کنیم:

از توابع فعال ساز سیگموید و ReLU استفاده می‌کنیم. نمودار این‌ها به صورت زیر است.



و کد این توابع به صورت تابع های زیر است :

```
# Define sigmoid and ReLU functions
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def relu(x):
    return np.maximum(0, x)
```

در ابتدا از سیگموید استفاده می کنیم :

خروجی این تابع بین ۰ و ۱ قرار می گیرد. وزن های نورون ها طوری تنظیم می شوند که اگر خروجی به ۱ نزدیک باشد، اشاره به حضور در ناحیه مورد نظر دارد و اگر به ۰ نزدیک باشد، به خروج از ناحیه مورد نظر اشاره می دهد. به دلیل پیوستگی خروجی تابع، برای تصمیم گیری در مورد تعلق به یک کلاس، یک آستانه یا threshold تعیین می شود. برای استفاده از تابع sigmoid، آستانه را معمولاً در مقدار ۰,۵ تعیین می کنیم، به این معنی که اگر خروجی از این تابع بیشتر از ۰,۵ باشد، به یک کلاس نسبت داده می شود و اگر کمتر باشد، به کلاس دیگری نسبت داده می شود. پس به ازای تمام نقاط رندوم، بررسی می شود که کمتر از ۰,۵ است یا بیشتر، تا متوجه شویم مربوط به کلاس قرمز است یا سبز !

```
for i in range(num_point):
    flag = Triangle(x_val[i] , y_val[i])
    if flag < [0.5] :
        red_point.append((x_val[i] , y_val[i]))
    else:
        green_point.append((x_val[i] , y_val[i]))
```

یک کلاس به صورت زیر تعریف می کنیم که ازتابع فعال ساز سیگموید استفاده می کنیم:

```
class McCulloch_Pitts_neuron:  
    def __init__(self, weights, threshold):  
        self.weights = np.array(weights)  
        self.threshold = threshold  
  
    def model(self, X):  
        X = np.array(X)  
        linear_output = self.weights @ X + self.threshold  
        return self.activation(linear_output)  
  
    def activation(self, linear_output):  
        # Default to the original threshold logic without an activation function  
        return 1 if linear_output >= 0 else 0  
  
    def set_activation_function(self, activation_func):  
        self.activation = activation_func
```

متغیر tol را در اینجا در نظر می گیریم. این متغیر برای تنظیم حساسیت یا دقت مدل در تصمیم‌گیری‌هایی که بر اساس آستانه انجام می‌شود، استفاده شود. به هر یک از آستانه‌های نورون‌ها یک مقدار tol اضافه می‌شود به عبارت دیگر، مقادیر آستانه نورون‌ها از آستانه‌های اصلی کمی بیشتر هستند. این عمل باعث می‌شود که نورون‌ها به عنوان ناحیه تشخیص‌دهنده، به نقاطی که در مرزها قرار دارند، حساس‌تر باشند و دقت بیشتری در تصمیم‌گیری در مورد تعلق یا عدم تعلق نقاط به داخل مثلث داشته باشند.

در کل، با افزایش مقدار tol، حساسیت مدل در تصمیم‌گیری‌هایی که بر اساس آستانه انجام می‌شود، کمتر می‌شود، و با کاهش آن، حساسیت بیشتری خواهد داشت. این به معنی این است که با افزایش tol، مدل به تصمیم‌های مبتنی بر آستانه‌ها کمتر اعتماد می‌کند و با کاهش آن، اعتماد بیشتری دارد. هر چقدر مقدار tol به صفر نزدیک باشد، مدل به hard limit نزدیک‌تر می‌شود و هر چقدر بزرگ‌تر در نظر گرفته شود، تعداد داده‌هایی که اشتباه طبقه‌بندی شده‌اند بیشتر است.

پس تغییر آستانه‌ها با اضافه کردن tol به مدل کمک کند تا نقاطی که در مرز قرار دارند، دقیق‌تر طبقه‌بندی شوند. این کار به خصوص در نورون‌هایی که مسئول تشخیص نقاط داخل یا خارج از یک ناحیه خاص هستند، اهمیت دارد. پس تابع تشکیل مثلث را این بار با در نظر گرفتن tol به صورت زیر تعریف می‌کنیم :

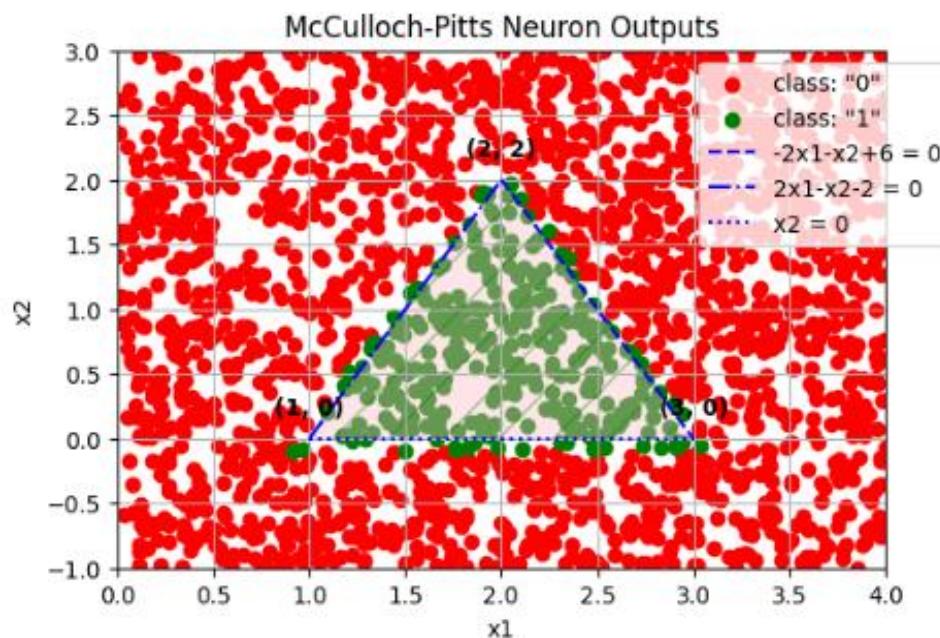
```
def Triangle(x,y):  
    neur1 = McCulloch_Pitts_neuron([-2 , -1] , 6+tol )  
    # set activation functions  
    # neur1.set_activation_function(sigmoid)  
    neur2 = McCulloch_Pitts_neuron([+2 , -1] , -2+tol )  
    # neur2.set_activation_function(sigmoid)  
    neur3 = McCulloch_Pitts_neuron([ 0 , +1] , 0+tol )  
  
    neur4 = McCulloch_Pitts_neuron([1 , 1 , 1] , -3+tol )  
    neur4.set_activation_function(sigmoid)  
  
    zone1 = neur1.model(np.array([x,y]))  
    zone2 = neur2.model(np.array([x,y]))  
    zone3 = neur3.model(np.array([x,y]))  
    zone4 = neur4.model(np.array([zone1 , zone2 , zone3]))  
  
    return list(zone4)
```

با تنظیم تابع فعال سازی به سیگموید، خروجی نورون چهارم به یک مقدار پیوسته بین 0 و 1 تبدیل می‌شود. این تغییر باعث می‌شود که نورون چهارم بتواند دقیق‌تر در تصمیم‌گیری داشته باشد. خروجی نورون چهارم اکنون یک مقدار بین 0 و 1 است که نشان‌دهنده احتمال تعلق نقطه ورودی به کلاس مورد نظر است.

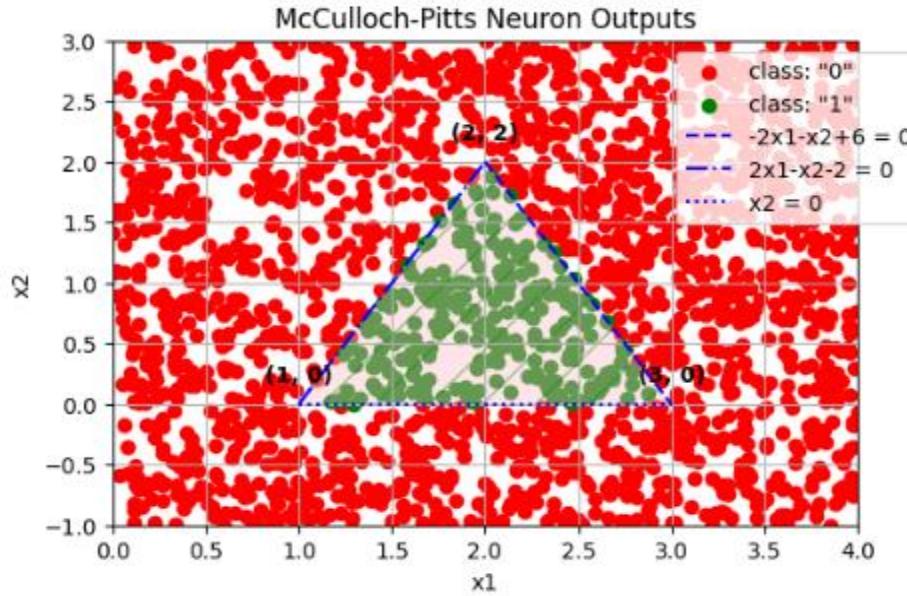
اگر خروجی نورون چهارم نزدیک به 1 باشد، به معنای حضور نقطه در مثلث است و اگر خروجی نورون چهارم نزدیک به 0 باشد، به معنای خارج بودن نقطه از ناحیه مورد نظر است.

برای تصمیم‌گیری نهایی، معمولاً یک آستانه (مثل 0.5) در نظر گرفته می‌شود. اگر خروجی بیشتر از 0.5 باشد، نقطه به یک کلاس نسبت داده می‌شود و اگر کمتر باشد، به کلاس دیگر نسبت داده می‌شود.

مشابه حالت قبل خروجی را برای 2000 داده رندوم رسم می‌کنیم و $tol = 0.1$ را برابر در نظر می‌گیریم :



می‌بینیم که تعدادی از داده‌ها اشتباه طبقه‌بندی شده‌اند، حال مقدار $tol = 0.1$ را برابر در نظر می‌گیریم و داریم :



مشاهده می شود که در این حالت طبقه بندی بهتر صورت گرفته است و نتیجه می شود که کم بودن tol عملکرد مدل را مانند حالت بدون تابع فعال ساز قسمت قبل، می کند.

اکنون از تابع ReLU استفاده می کنیم :

تابع ReLU یک تابع فعال سازی است که برای استفاده در شبکه های عصبی مصنوعی استفاده می شود و در بخش های قبل به توضیح بیشتر درباره آن پرداختیم. محدوده خروجی به این صورت است که تابع ReLU خروجی صفر را برای همه مقادیر منفی و مقدار ورودی را برای مقادیر مثبت بر می گرداند. به عبارت دیگر، اگر ورودی مثبت باشد، خروجی برابر با خود ورودی است و اگر منفی باشد، خروجی صفر است.

پس درواقع یک شرط می نویسیم که ببینیم هر نقطه رندوم مربوط به کدام کلاس است. (اگر مقدار صفر شد خارج مثلث است در غیر این صورت اگر مقدار برابر خود ورودی شد، داخل مثلث است).

```

red_point = [] # outside zone
green_point = [] # inside zone

for i in range(num_point):
    flag = Triangle(x_val[i] , y_val[i])
    if flag == [0] :
        red_point.append((x_val[i] , y_val[i]))
    else:
        green_point.append((x_val[i] , y_val[i]))

```

در تفسیر ساده‌تر، ReLU مانند یک سوئیچ عمل می‌کند. اگر مقدار ورودی مثبت باشد، خروجی همان مقدار ورودی است و اگر مقدار ورودی منفی باشد، خروجی صفر است.

مانند قسمت قبل یک کلاس برای تعریف MCP داریم : که از تابع فعال ساز هم استفاده می‌شود.

```
▶ class McCulloch_Pitts_neuron:
    def __init__(self, weights, threshold):
        self.weights = np.array(weights)
        self.threshold = threshold

    def model(self, X):
        X = np.array(X)
        linear_output = self.weights @ X + self.threshold
        return self.activation(linear_output)

    def activation(self, linear_output):
        # Default to the original threshold logic without an activation function
        return 1 if linear_output >= 0 else 0

    def set_activation_function(self, activation_func):
        self.activation = activation_func
```

سپس تابعی برای محاسبه نقاط داخل مثلث می‌نویسیم :

```
def Triangle(x,y):
    neur1 = McCulloch_Pitts_neuron([-2 , -1] , 6+tol )
    # Set activation functions
    # neur1.set_activation_function(relu)
    neur2 = McCulloch_Pitts_neuron([+2 , -1] , -2+tol )
    # neur2.set_activation_function(relu)
    neur3 = McCulloch_Pitts_neuron([ 0 , +1] , 0+tol )
    # neur3.set_activation_function(relu)

    neur4 = McCulloch_Pitts_neuron([1 , 1 , 1] , -3+tol )
    neur4.set_activation_function(relu)

    zone1 = neur1.model(np.array([x,y]))
    zone2 = neur2.model(np.array([x,y]))
    zone3 = neur3.model(np.array([x,y]))
    zone4 = neur4.model(np.array([zone1 , zone2 , zone3]))

    return list([zone4])
#-----
```

در کد بالا نورون چهارم برای تعیین نهایی اینکه نقطه در داخل یا خارج از مثلث قرار دارد استفاده می‌شود. با استفاده از دستور neur4.set_activation_function(relu) تابع فعال‌سازی این نورون به تابع ReLU تغییر داده می‌شود.

به طور پیش‌فرض، نورون‌های McCulloch_Pitts از یک تابع آستانه ساده (خطی) استفاده می‌کنند که خروجی را به ۰ یا ۱ محدود می‌کند.

با تنظیم تابع فعال سازی به ReLU ، نورون چهارم دیگر از این تابع پیش فرض استفاده نمی کند. در عوض، تابع ReLU اعمال می شود که خروجی آن برای مقادیر منفی صفر و برای مقادیر مثبت همان مقدار ورودی خواهد بود. وقتی ورودی های neur4 zone3، zone2، zone1 را داده می شوند، نورون چهارم مقدار خطی خود را محاسبه می کند (مجموع وزن ها ضربدر ورودی ها به علاوه آستانه). سپس، تابع ReLU این مقدار خطی را تبدیل می کند: اگر این مقدار خطی منفی باشد، خروجی نورون چهارم صفر خواهد بود؛ اگر مثبت باشد، خروجی همان مقدار خطی خواهد بود.

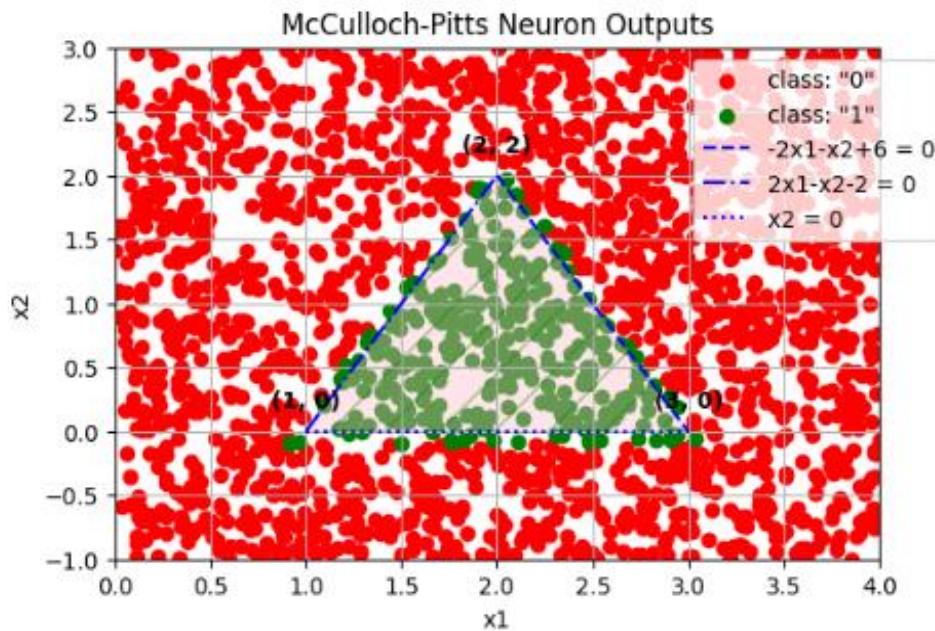
پس با تنظیم تابع فعال سازی neur4 به ReLU ، عملکرد نورون چهارم تغییر می کند تا مقادیر منفی را به صفر تبدیل کند و مقادیر مثبت را به همان شکل حفظ کند. این تغییر می تواند تأثیر مهمی در تصمیم گیری نهايی داشته باشد، به خصوص زمانی که ترکیب ورودی ها از نورون های قبلی (که تعیین می کنند نقطه در داخل یا خارج مثلث است) مورد نظر است.

دقیقاً مانند حالت قبل در اینجا هم باید از یک tol یا تولرانس استفاده کنیم که تنظیم مقدار آن دست ماست و هر چه به صفر نزدیک تر باشد طبقه بنده بهتر صورت می گیرد و مدل به hard limit شبیه تر می شود.

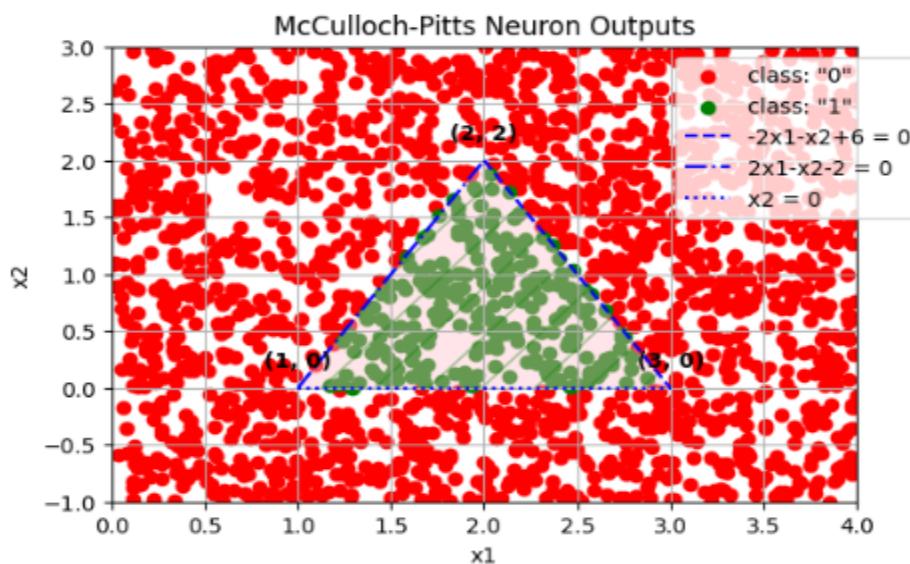
به هر آستانه ای که برای نورون ها تعیین شده است، مقدار tol اضافه می کنیم. به عنوان مثال، آستانه نورون اول به جای ۶ برابر با ۱،۰۰۰۱ شده است (به ازای $\text{tol}=0.0001$). این کار به منظور تنظیم دقیق تر مرزهای تصمیم گیری و بهبود دقت مدل انجام شده باشد. با اضافه کردن tol به آستانه، مقدار خطی خروجی نورون ها به مقدار بسیار کمی تغییر می کند. این تغییر می تواند به عنوان یک تولرانس کوچک در تصمیم گیری ها عمل کند.

به عنوان مثال، اگر مقدار خطی خروجی یک نورون بسیار نزدیک به آستانه باشد، اضافه کردن tol می تواند تعیین کند که خروجی نهايی نورون ۰ یا ۱ باشد.

برای دو تولرانس مختلف چک می کنیم. ابتدا tol را برابر ۰،۱ در نظر می گیریم :



می بینیم که تعدادی از داده ها به خوبی طبقه بندی نشده اند، پس مقداری نزدیک تر به صفر در نظر می گیریم،
اکنون tol را برابر ۱۰۰۰۰ در نظر می گیریم :



می بینیم که در این حالت طبقه بندی مدل خیلی بهتر شده است و عملکردی مشابه حالت hard limit دارد.

سوال دو

بخش اول

۱. دیتاست CWRU Bearing که در «مینیپروژه شماره یک» با آن آشنا شدید را به خاطر آورید. علاوه بر دو کلاسی که در آن مینیپروژه در نظر گرفتید، با مراجعه به صفحه داده‌های عیب در حالت ۱۲k OR007@6_X_B007^ و X_B007@6 اضافه کنید. با انجام این کار یک کلاس داده سالم و سه کلاس از داده‌های دارای سه عیب متفاوت خواهد داشت. در مورد این که هر فایل مربوط به چه نوع عیبی است به صورت کوتاه توضیح دهید.

سپس در ادامه، تمام کارهایی که در بخش «۲» سوال دوم «مینیپروژه یک» برای استخراج ویژگی و آماده‌سازی دیتا انجام داده بودید را روی دیتاست جدید خود پیاده‌سازی کنید. در قسمت تقسیم‌بندی داده‌ها، یک بخش برای «اعتبارسنجی» به بخش‌های «آموزش» و «آزمون» اضافه کنید و توضیح دهید که کاربرد این بخش چیست.

داده‌های جمع‌آوری‌شده برای یاتاقان‌های سالم و معیوب در سمت رانش (fan end) و سمت فن (drive end) با سرعت نمونه‌برداری ۱۲،۰۰۰ و ۴۸،۰۰۰ نمونه در ثانیه (برای آزمایش‌های سمت رانش) و ۱۲،۰۰۰ نمونه در ثانیه (برای آزمایش‌های سمت فن) بوده است. فایل‌های داده در فرمت Matlab شامل داده‌های ارتعاشات یاتاقان‌های سمت رانش و سمت فن و همچنین سرعت چرخش موتور هستند. متغیرهای داخل فایل‌ها شامل داده‌های شتاب‌سنج سمت رانش (DE)، داده‌های شتاب‌سنج سمت فن (FE)، داده‌های شتاب‌سنج پایه (BA)، داده‌های سری زمانی (time) و دور در دقیقه (RPM) می‌باشند.

داده‌های معیوب به سه دسته‌بندی اصلی تقسیم می‌شوند: عیب شیار داخلی (Inner race)، عیب ساقمه (Ball) و عیب شیار خارجی (Outer race). نام‌گذاری داده‌ها برای هر نوع عیب به ترتیب به صورت IR007_0 برای عیب شیار داخلی، B007_0 برای عیب ساقمه، و OR007@6_0 برای عیب شیار خارجی است. عیب‌های مربوط به شیار خارجی دارای سه موقعیت مختلف (مرکز، عمود، و مقابل) هستند که در این تمرین، حالت مرکز انتخاب شده است.

پس در کل ۴ حالت داریم یک حالت دیتاهای سالم و سه حالت عیب به صورت فوق. داده‌ها را دانلود می‌کنیم و در ابتدا مانند تمرین قبل، پیش می‌رویم.

برای دانلود دیتاست‌های مربوط به هر کلاس، از دستور زیر استفاده می‌شود و ۴ دیتاست را دانلود می‌کنیم.

```
pip install --upgrade --no-cach-dir gdown
gdown 1MeHr0QxBWnG_VAHgRxS0pgrmjIebR47m
gdown 1NcFe97vF2n4HYHHy3hC9a_Y2hKfhUAVD
gdown 15Yf81TkjYk04UF7PzgQH8IPdKDFZSVHP
gdown 1WjSJzjrA_izOw110CNgAVkfJYqdtudoEV
```

پس از دانلود داده‌ها، فایل‌ها با استفاده از دستور `scipy loadmat` از کتابخانه `scipy` بارگذاری می‌شوند

```
! pip install scipy
import scipy.io as sio
from scipy.io import loadmat

data_n = sio.loadmat('/content/drive/MyDrive/MachineLearning/HW2/97.mat')
data_f1 = sio.loadmat('/content/drive/MyDrive/MachineLearning/HW2/105.mat')
data_f2 = sio.loadmat('/content/drive/MyDrive/MachineLearning/HW2/118.mat')
data_f3 = sio.loadmat('/content/drive/MyDrive/MachineLearning/HW2/130.mat')
```

. مشابه تمرین قبلی، داده‌ها از بخش‌های مختلفی تشکیل شده‌اند که در زیر قابل مشاهده است :

```
[ ] #columne of normal and faults
columns_n = list(data_n.keys())
columns_f1 = list(data_f1.keys())
columns_f2 = list(data_f2.keys())
columns_f3 = list(data_f3.keys())
print (columns_n)
print (columns_f1)
print (columns_f2)
print (columns_f3)

→ [ '__header__', '__version__', '__globals__', 'X097_DE_time', 'X097_FE_time', 'X097RPM']
[ '__header__', '__version__', '__globals__', 'X105_DE_time', 'X105_FE_time', 'X105_BA_time', 'X105RPM']
[ '__header__', '__version__', '__globals__', 'X118_DE_time', 'X118_FE_time', 'X118_BA_time', 'X118RPM']
[ '__header__', '__version__', '__globals__', 'X130_DE_time', 'X130_FE_time', 'X130_BA_time', 'X130RPM']
```

برای ادامه حل مسئله، مدل DE-time که در تمامی فایل‌ها مشترک است، انتخاب می‌کنیم و سپس همانند تمرین سری اول، ۳۰۰ نمونه به اندازه ۳۰۰ از داده‌های اصلی جدا می‌کنیم.

من در ابتدا برای تمام بخش‌های ۴ دیتاست ماتریس‌های 300×300 جدا کردم و سپس دیتای DE-time را برای ادامه کار انتخاب کردم. (درواقع ۱۵ کلاس از ۴ کلاس زیر را انتخاب کردم.)

```
' Class "X097_DE_time"'s matrix has the shape of (300, 300)
Class "X097_FE_time"'s matrix has the shape of (300, 300)
Class "X097RPM"'s matrix has the shape of (300, 300)
Class "X105_DE_time"'s matrix has the shape of (300, 300)
Class "X105_FE_time"'s matrix has the shape of (300, 300)
Class "X105_BA_time"'s matrix has the shape of (300, 300)
Class "X105RPM"'s matrix has the shape of (300, 300)
Class "X118_DE_time"'s matrix has the shape of (300, 300)
Class "X118_FE_time"'s matrix has the shape of (300, 300)
Class "X118_BA_time"'s matrix has the shape of (300, 300)
Class "X118RPM"'s matrix has the shape of (300, 300)
Class "X130_DE_time"'s matrix has the shape of (300, 300)
Class "X130_FE_time"'s matrix has the shape of (300, 300)
Class "X130_BA_time"'s matrix has the shape of (300, 300)
Class "X130RPM"'s matrix has the shape of (300, 300)
```

پس ۴ دسته داده دارم که به صورت زیر هستند:

```
[ ] normal_mat = matrices['X097_DE_time']
fault1_mat = matrices['X105_DE_time']
fault2_mat = matrices['X118_DE_time']
fault3_mat = matrices['X130_DE_time']
```

اکنون ۱۰ ویژگی مشابه تمرین قبل از این دیتاست استخراج می کنم:

```
▶ class Features:
    def __init__(self,matrix):
        self.matrix = matrix
        self._extract()

    def _extract(self):
        self.features = {
            'standard deviation': stats.tstd(self.matrix, axis=1),
            'peak': np.max(self.matrix, axis=1),
            'skewness': stats.skew(self.matrix, axis=1),
            'square root mean': np.square(np.mean(np.sqrt(np.abs(self.matrix))), axis=1),
            'kurtosis': stats.kurtosis(self.matrix, axis=1),
            'crest factor': np.max(self.matrix, axis=1) / np.sqrt(np.mean(np.square(self.matrix), axis=1)),
            'clearance factor': np.max(self.matrix, axis=1) / np.square(np.mean(np.sqrt(np.abs(self.matrix))), axis=1),
            'mean': np.mean(self.matrix, axis=1),
            'absolute mean': np.mean(np.abs(self.matrix), axis=1),
            'root mean square': np.sqrt(np.mean(np.square(self.matrix), axis=1)),
        }

    def __getitem__(self,key):
        return self.features[key]
```

```
normal_features = Features(normal_mat)
fault1_features = Features(fault1_mat)
fault2_features = Features(fault2_mat)
fault3_features = Features(fault3_mat)
```

اکنون به داده ها لیبل میزنیم، داده های سالم با لیبل صفر و داده های معیوب، با توجه به نوع عیب لیبل های ۱ و ۲ و ۳ دارند. این لیبل ها برابر مقدار ۷ می شود و فیچر هایی که استخراج شد همان X ها میشود.

داده های هر ۴ کلاس را به صورت سطری زیر هم مینویسیم پس ۱۲۰۰ نمونه با ۱۱ ستون داریم که ۱۰ ستون فیچر ها هستند و یک ستون لیبل ها ، مانند زیر :

| | standard deviation | peak | skewness | square root mean | kurtosis | crest factor | clearance factor | mean | absolute mean | root mean square | label |
|------|--------------------|----------|-----------|------------------|-----------|--------------|------------------|----------|---------------|------------------|-------|
| 0 | 0.083577 | 0.217794 | -0.345679 | 0.062231 | -0.398710 | 2.566209 | 3.499770 | 0.015528 | 0.070864 | 0.084870 | 0 |
| 1 | 0.083564 | 0.217794 | -0.338346 | 0.062037 | -0.400868 | 2.567792 | 3.510731 | 0.015308 | 0.070730 | 0.084818 | 0 |
| 2 | 0.083485 | 0.217794 | -0.328538 | 0.061793 | -0.398773 | 2.572172 | 3.524582 | 0.014936 | 0.070510 | 0.084673 | 0 |
| 3 | 0.083353 | 0.217794 | -0.320434 | 0.061443 | -0.391665 | 2.578067 | 3.544672 | 0.014568 | 0.070214 | 0.084480 | 0 |
| 4 | 0.083315 | 0.217794 | -0.316926 | 0.061286 | -0.389052 | 2.579855 | 3.553729 | 0.014445 | 0.070091 | 0.084421 | 0 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 1195 | 0.708669 | 2.578252 | 0.120438 | 0.309686 | 3.839738 | 3.637641 | 8.325377 | 0.042632 | 0.435893 | 0.708770 | 3 |
| 1196 | 0.700343 | 2.578252 | 0.088689 | 0.305191 | 4.002725 | 3.682642 | 8.447984 | 0.036161 | 0.429608 | 0.700109 | 3 |
| 1197 | 0.684690 | 2.531958 | -0.039789 | 0.300245 | 4.024920 | 3.701085 | 8.432960 | 0.027787 | 0.421235 | 0.684112 | 3 |
| 1198 | 0.675964 | 2.531958 | 0.003276 | 0.295611 | 4.192825 | 3.747207 | 8.565161 | 0.033991 | 0.415083 | 0.675692 | 3 |
| 1199 | 0.666968 | 2.531958 | 0.053539 | 0.291491 | 4.365575 | 3.795705 | 8.686214 | 0.040040 | 0.409033 | 0.667059 | 3 |

1200 rows × 11 columns

اما مشکل این است که به ترتیب قرار می‌گیرند یعنی ۳۰۰ ستون اول برای داده‌های نرمال است و .. ، برای حل این مشکل داده‌ها را shuffle می‌کنیم.

برای تقسیم بندی داده‌ها، داده‌ها را به تست و اموزش و اعتبار سنجی تقسیم می‌کنیم و نسبت این‌ها به صورت زیر است :

```
Size of Train data is:  
X --> (720, 10)  
y --> (720,)  
Size of Validation data is:  
X --> (240, 10)  
y --> (240,)  
Size of Test data is:  
X --> (240, 10)  
y --> (240,)
```

❖ کاربرد بخش اعتبار سنجی

مجموعه‌داده اعتبار سنجی (Validation Set) در یادگیری ماشین نوش بسیار مهمی ایفا می‌کند و برای چندین هدف مهم استفاده می‌شود. این اهداف عبارتند از:

۱. انتخاب مدل (Model Selection)

انتخاب مدل به معنای انتخاب بهترین مدل از بین چندین مدل مختلف است. با استفاده از مجموعه‌داده اعتبار سنجی، می‌توان عملکرد هر مدل را بررسی کرد و مدلی را که بهترین عملکرد را دارد، انتخاب کرد. این فرآیند به اطمینان از انتخاب مدلی که به خوبی به داده‌های جدید تعمیم می‌یابد، کمک می‌کند.

۲. تنظیم کردن (Hyperparameter Tuning)

Hyperparameters پارامترهایی هستند که در فرآیند آموزش مدل تنظیم می‌شوند و تاثیر زیادی بر عملکرد نهایی مدل دارند. این پارامترها شامل عواملی مانند نرخ یادگیری (learning rate)، تعداد لایه‌ها در یک شبکه عصبی و تعداد نورونها در هر لایه هستند. با استفاده از مجموعه‌داده اعتبارسنجی، می‌توان این پارامترها را تنظیم کرد تا مدل بهترین عملکرد ممکن را داشته باشد.

۳. جلوگیری از Overfitting

Overfitting زمانی رخ می‌دهد که مدل به خوبی بر روی داده‌های آموزشی عملکرد دارد، اما بر روی داده‌های جدید (که قبل‌اً ندیده) عملکرد خوبی ندارد. مجموعه‌داده اعتبارسنجی می‌تواند به شناسایی زمانی که مدل شروع به overfit شدن می‌کند کمک کند. این کار با ارزیابی مداوم عملکرد مدل بر روی داده‌های اعتبارسنجی و تطبیق مدل به گونه‌ای که بتواند به خوبی تعمیم دهد، انجام می‌شود.

۴. Early Stopping در شبکه‌های عصبی و یادگیری عمیق

در یادگیری عمیق و شبکه‌های عصبی، یکی از روش‌های جلوگیری از overfitting، استفاده از تکنیکی به نام Early Stopping است. در این روش، فرآیند آموزش مدل زمانی متوقف می‌شود که عملکرد مدل بر روی مجموعه‌داده اعتبارسنجی شروع به بدتر شدن می‌کند. این نشان می‌دهد که مدل در حال overfit شدن بر روی داده‌های آموزشی است و ادامه آموزش ممکن است به تعمیم‌پذیری مدل آسیب برساند.

بعد از جداسازی داده‌ها، مرحله بعدی تبدیل آنها به فرم معمولی است، که از طریق دستور StandardScaler انجام می‌شود. در این مرحله، ابتدا با توجه به داده‌های آموزش، پارامترهای scaler را تنظیم می‌کنیم و سپس این تبدیل را بر روی تمامی داده‌ها اعمال می‌کنیم.

```
from sklearn.preprocessing import StandardScaler

# Fit the scaler on training data and transform both training and test data
scaler = StandardScaler()
scaler.fit(x_train)

x_train_scaled = scaler.transform(x_train)
x_val_scaled = scaler.transform(x_val)
x_test_scaled = scaler.transform(x_test)
```

بخش دوم

۲. یک مدل Multi-Layer Perceptron (MLP) ساده با ۲ لایه پنهان یا بیشتر بسازید. بخشی از داده‌های آموزش را برای اعتبارسنجی کنار بگذارید و با انتخاب بهینه‌ساز و تابع اتلاف مناسب، مدل را آموزش دهید. نمودارهای اتلاف و Accuracy مربوط به آموزش و اعتبارسنجی را رسم و نتیجه را تحلیل کنید. نتیجهٔ تست مدل روی داده‌ای آزمون را با استفادهٔ ماتریس درهم‌ریختگی و `classification_report` نشان داده و نتایج به صورت دقیق تحلیل کنید.

حالات اول ()

در این جا، برای تشکیل مدل MLP از کتابخانه SKlearn استفاده می‌کنیم. یک مدل سه لایه با هایپرپارامترهایی به صورت زیر تنظیم می‌کنیم

MLPClassifier:

```
▶ mlp = MLPClassifier(hidden_layer_sizes=(4, 3, 3), random_state=24, max_iter=100, solver='adam', activation='relu', tol=0.0001, learning_rate_init=0.001, verbose=True, batch_size=10)

# Fit the model
mlp.fit(x_train_scaled, y_train)

# Evaluate on validation set
val_score = mlp.score(x_val_scaled, y_val)
print("Validation Accuracy: {:.2f}%".format(val_score * 100))
print('-----')
print((accuracy_score(mlp.predict(x_test_scaled), y_test)))
```

با توجه به یک حلقه `for` که برای هر لایه مقادیر یک تا ۵ نورون قرار می‌داد و در نتیجه بهترین دقت را میداد، بهترین تعداد نورون برای هر لایه به ترتیب به صورت زیر است که در مدل بالا لحاظ شده است.

```
→ Best parameters: {'hidden_layer_sizes': (4, 3, 3)}
Validation Accuracy: 1.0
```

با استفاده از کد زیر نمودار loss را برای داده‌های آموزش و اعتبارسنجی رسم می‌کنیم:

```

▶ mlp.fit(x_train_scaled, y_train)
# Evaluate on validation set
val_score = mlp.score(x_val_scaled, y_val)

# Initialize lists to store loss
train_loss = []
val_loss = []

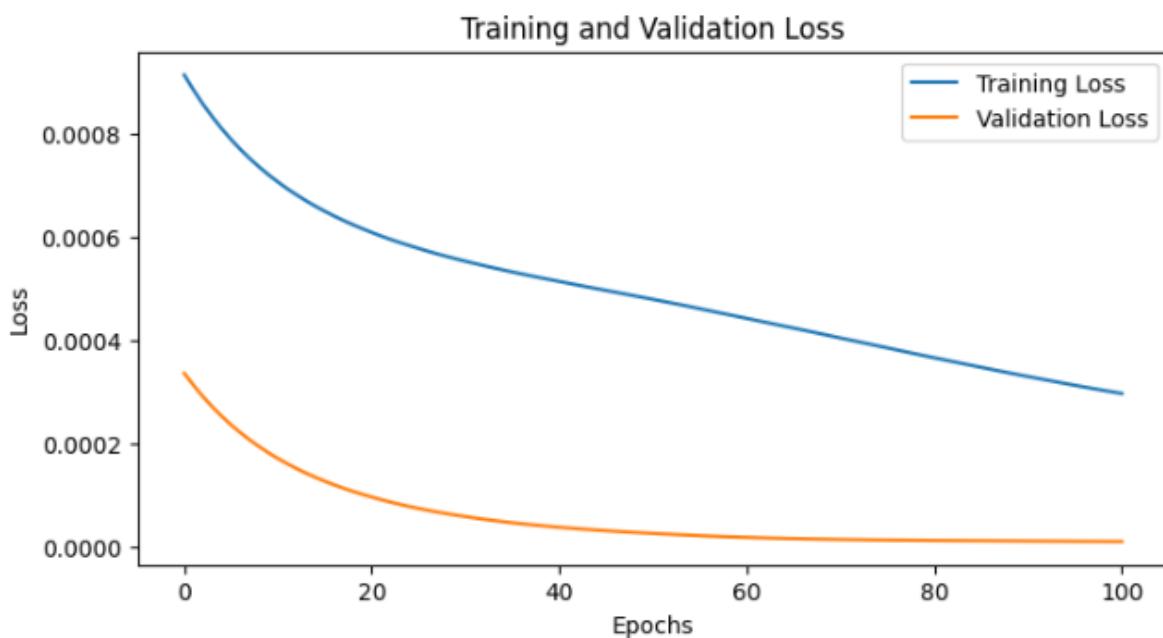
# Train the model
for epoch in range(mlp.max_iter+1):
    mlp.max_iter = epoch
    mlp.partial_fit(x_train_scaled, y_train, classes=np.unique(y_train))

    # Calculate training loss
    train_loss.append(mlp.loss_)

    # Calculate validation loss
    y_val_pred_proba = mlp.predict_proba(x_val_scaled)
    val_loss_epoch = -np.mean(np.log(y_val_pred_proba[np.arange(len(y_val)), y_val])) # Using log loss (cross-entropy)
    val_loss.append(val_loss_epoch)

```

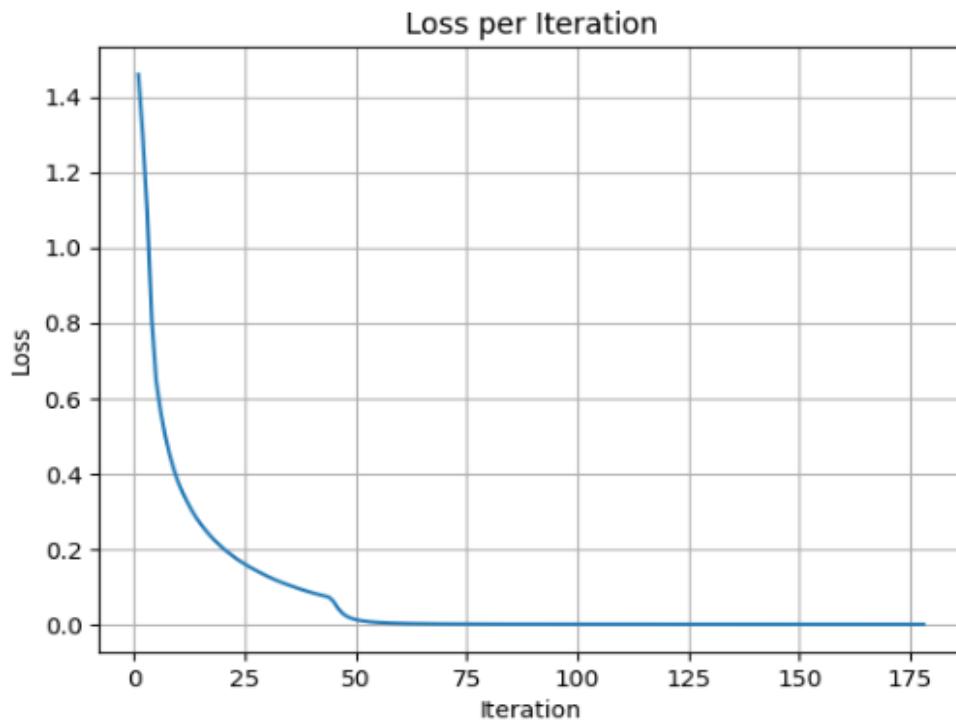
برای هر ایپاک از یک تا ۱۰۰ رسم میشود. داخل خودتابع `mlpclassifier` مقدار `mlp.loss_` توسط `loss` برای داده های آموزش محاسبه میشود اما این مقدار `loss` برای هر دوره، آخرین مقدار `loss` آن است که طبیعتا بهترین مقدار آن است.



اگر بخواهیم مقدار `loss` را برای داده های آموزش در هر ایتریشن را محاسبه کنیم (همان مقادیری که با ران کردن مدل `mlp` دیده میشوند)، نمودار زیر بدست می اید:

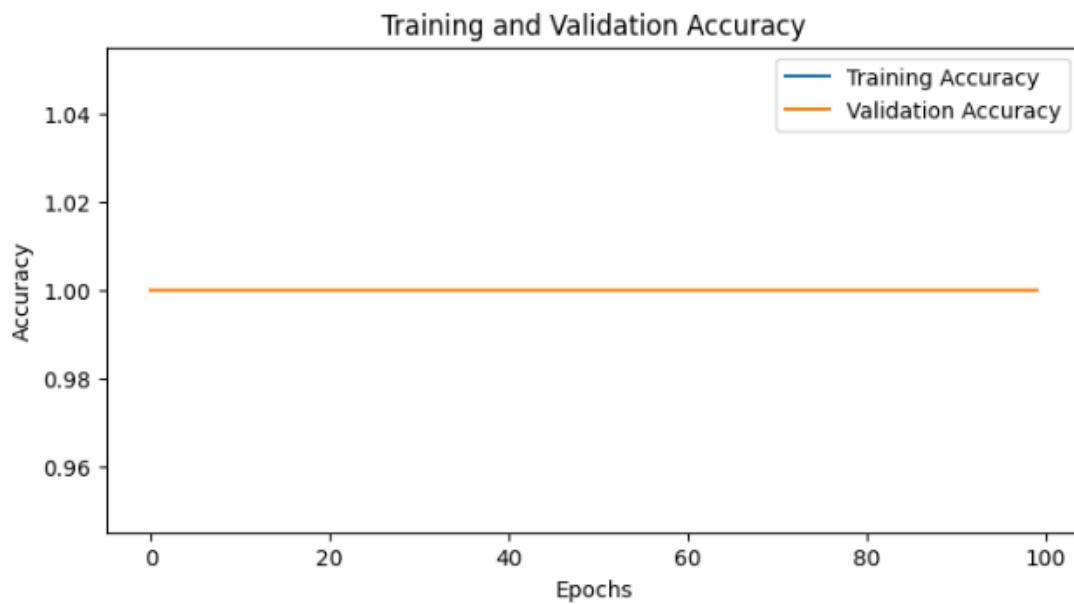
از دستور روبه رو استفاده میشود.

```
losses = mlp.loss_curve_
```



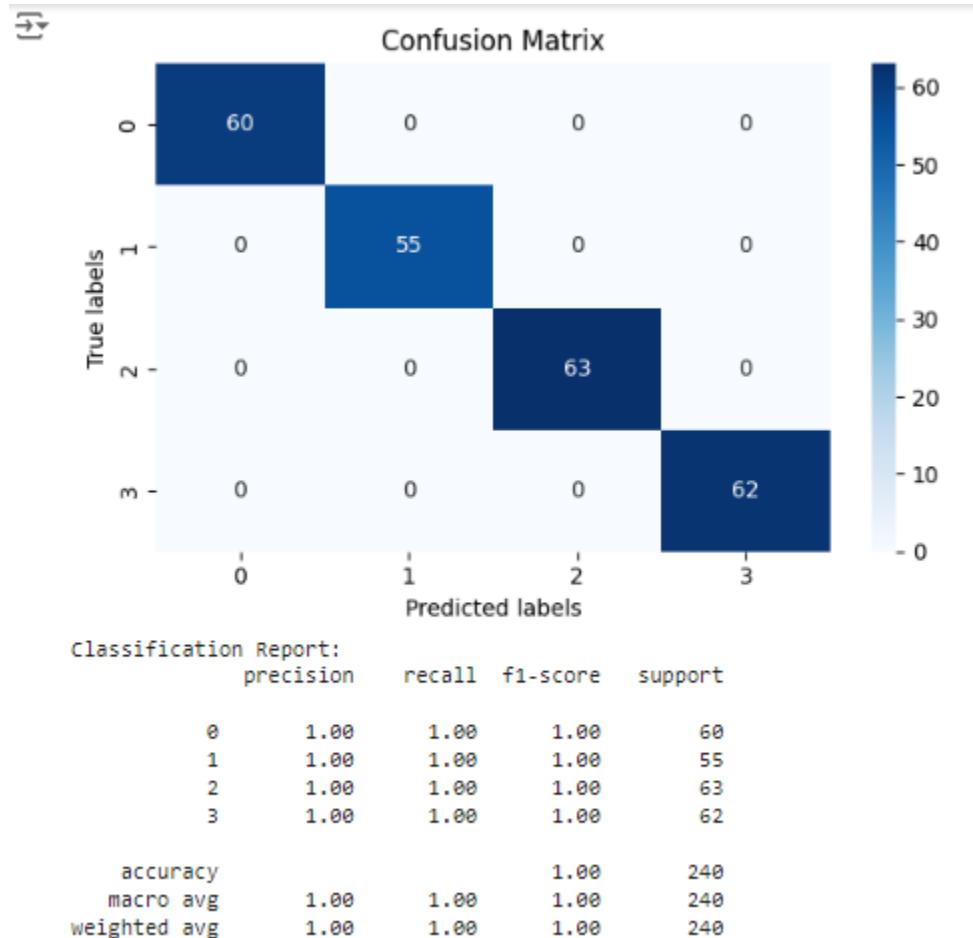
پس علت تفاوت این نمودار با حالت قبل در این است که حالت قبلی، loss های اخرين را در نظر می گيرد.

نمودار های دقیق هم به صورت زیر بدست می ایند:



که دقیق یک را از همان ابتدا نشان می دهد.

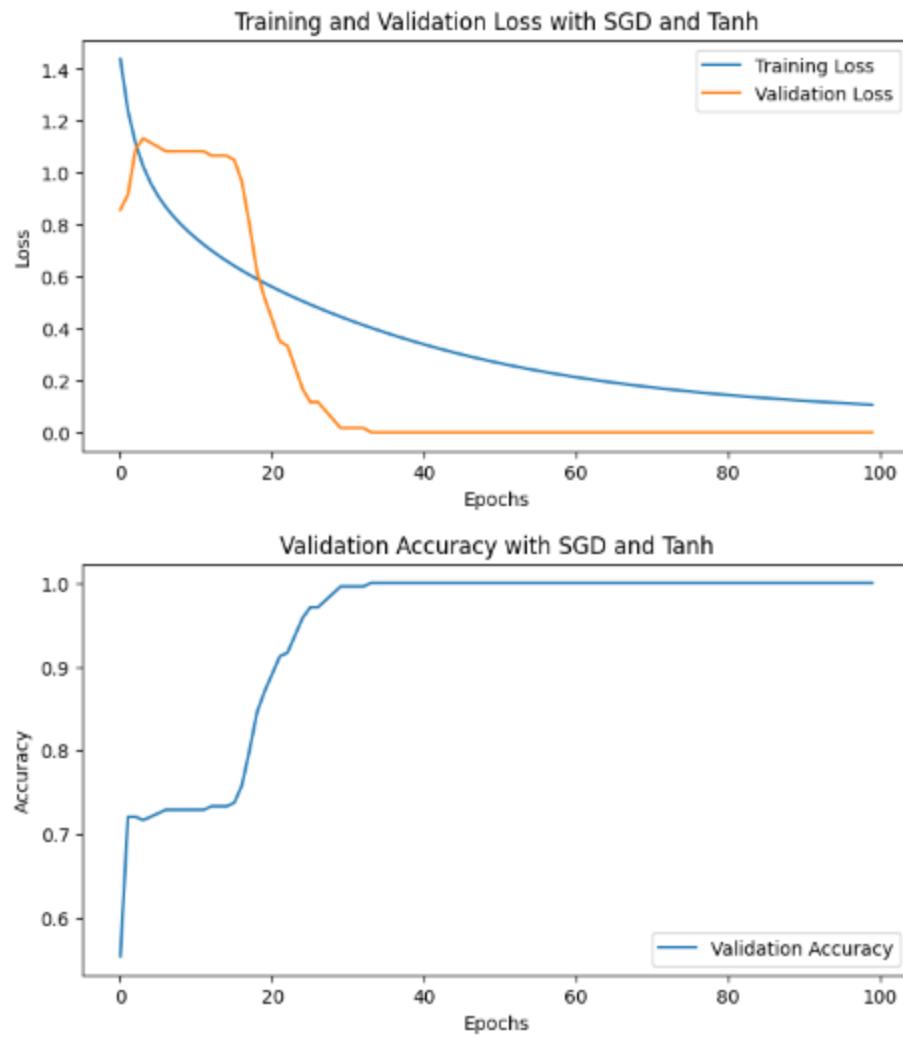
ماتریس درهم ریختگی و classification report هم به صورت زیر هستند که نشان می دهند دقیق مدل ۱۰۰ درصد است و تمام داده ها به خوبی طبقه بندی شده اند.



هایپرپارامتر هارا تغییر می دهیم و داریم :

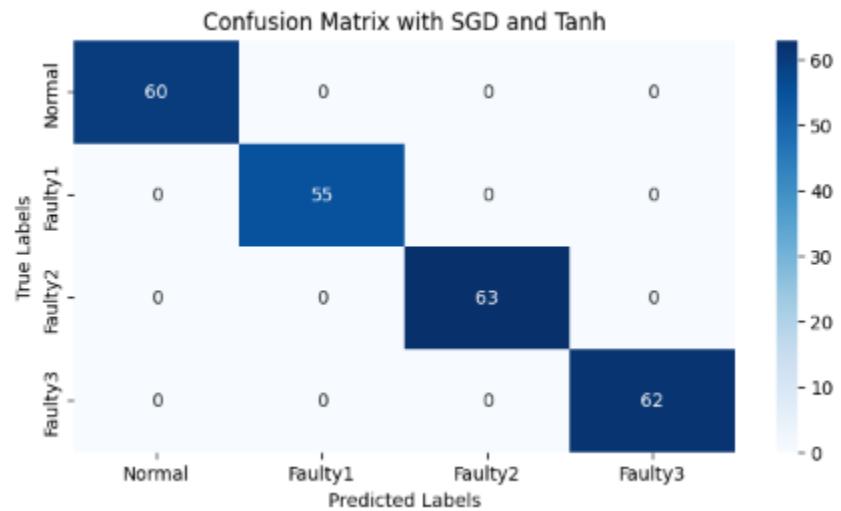
در حالت اول از activation='tanh', solver='sgd' استفاده می کنیم و نمودار ها به صورت زیر بدست می

آیند:



و عملکرد مدل هم به صورت زیر است:

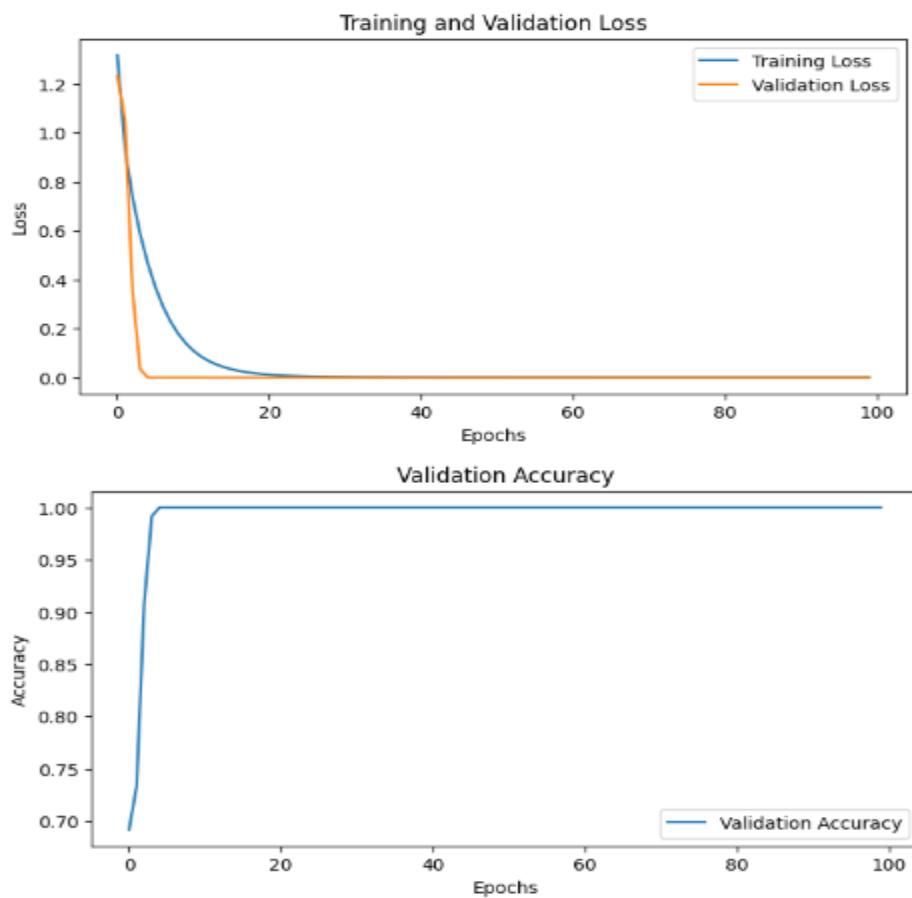
| Classification Report for Test Data with SGD and Tanh: | | | | |
|--|-----------|--------|----------|---------|
| | precision | recall | f1-score | support |
| 0 | 1.00 | 1.00 | 1.00 | 60 |
| 1 | 1.00 | 1.00 | 1.00 | 55 |
| 2 | 1.00 | 1.00 | 1.00 | 63 |
| 3 | 1.00 | 1.00 | 1.00 | 62 |
| accuracy | | | 1.00 | 240 |
| macro avg | 1.00 | 1.00 | 1.00 | 240 |
| weighted avg | 1.00 | 1.00 | 1.00 | 240 |



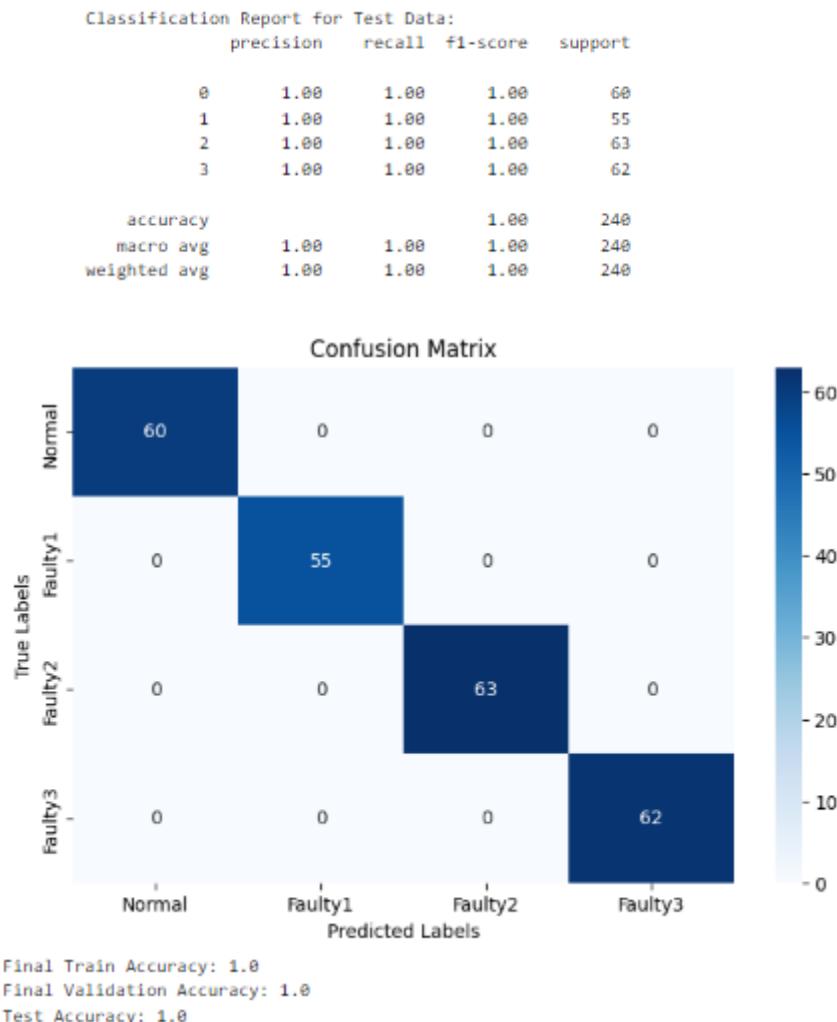
که میبینیم دقیقت در این حالت برابر 100 درصد شده است.

در حالت دوم از `activation='tanh', solver='adam'` استفاده می کنیم:

نمودار ها به صورت زیر هستند :



و عملکرد مدل به صورت زیر است که میبینیم دقیقی برابر با ۱۰۰ درصد بدست آمده است.



در این حالت هم رسم نمودارها سخت تر بود و باید از چند حلقه استفاده می کردیم و هم امکان این که برای هر لایه تابع فعال ساز جدا تعریف کنیم و... وجود نداشت. پس به سراغ بررسی حالت دوم رفتیم.

حالت دوم (

از Keras Sequential API استفاده کردیم.

یک شبکه عصبی ۳ لایه با تعداد لایه های ۵ و ۴ و ۴ انتخاب کردیم البته با توجه به اینکه ۴ کلاس داریم، لزومی ندارد حتما لایه آخر ۴ نرون داشته باشد و با یک عدد دو بیتی هم می توان ۴ حالت ساخت. در لایه های hidden از فعال ساز `relu` و لایه آخر از `softmax` استفاده شد (به دلیل چند کلاسه بودن داده ها)

```

from keras.models import Sequential
from keras.layers import Dense
import tensorflow as tf

# Set the random seed for reproducibility
random.seed(24)
np.random.seed(24)
tf.random.set_seed(24)

model_k = Sequential([
    Dense(units=5, activation='relu'),
    Dense(units=4, activation='relu'),
    Dense(units=4, activation='softmax'),
])
model_k.compile(
    optimizer='adam',
    loss='sparseCategoricalCrossentropy',
    metrics=['accuracy']
)

histories = model_k.fit(
    x=x_train_scaled,
    y=y_train,
    validation_data=(x_val_scaled, y_val),
    epochs=50,
    batch_size=1,
    verbose=2,
)

```

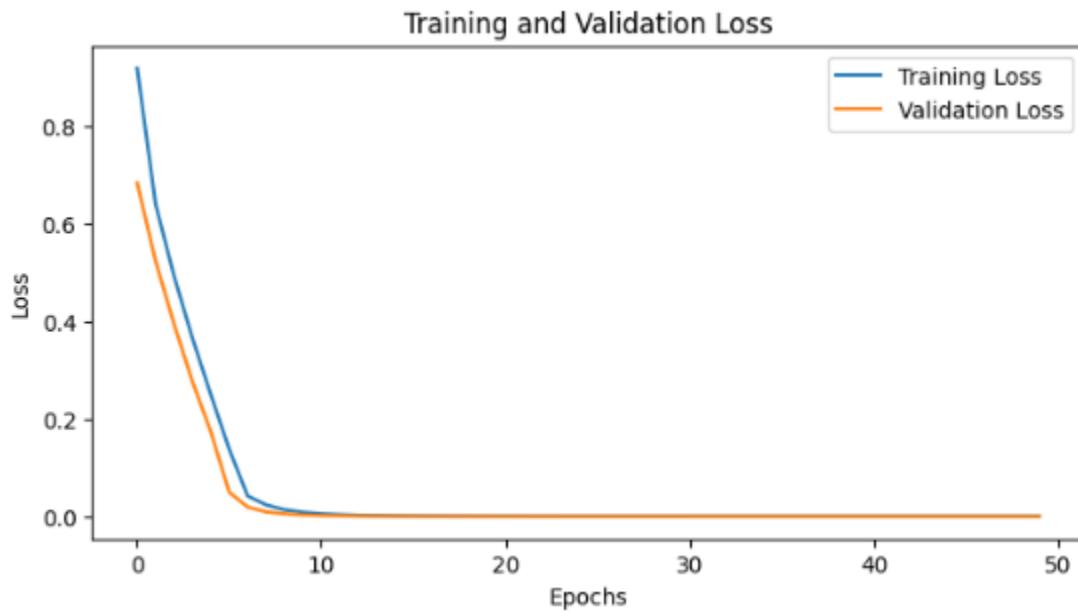
خروجی ها به عنوان مثال برای 7 ایپاک اول به صورت زیر هستند :

```

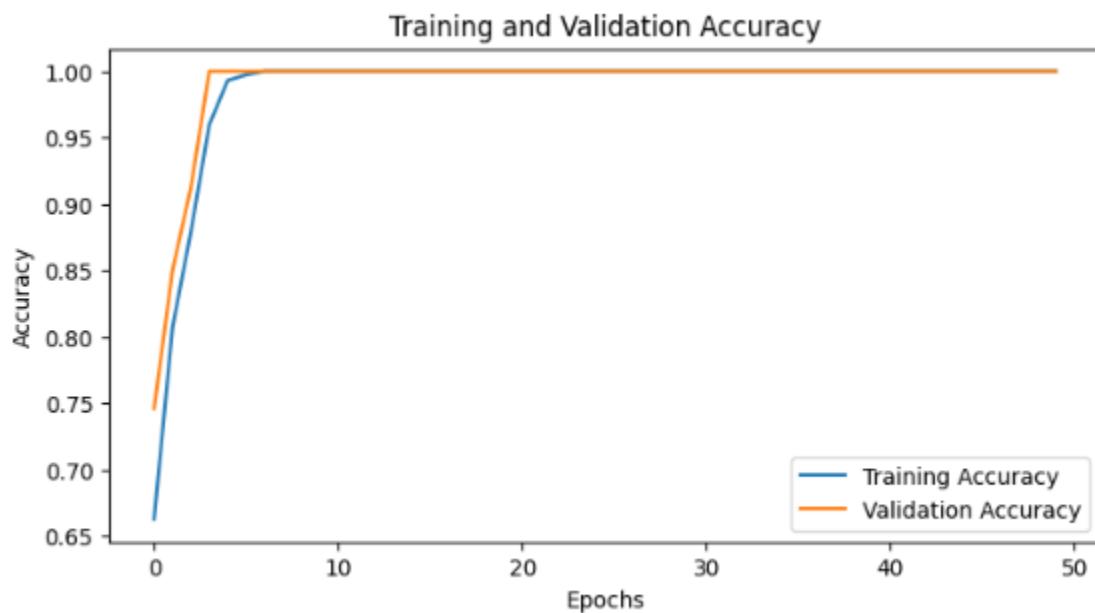
Epoch 1/50
720/720 - 3s - loss: 0.9202 - accuracy: 0.6625 - val_loss: 0.6849 - val_accuracy: 0.7458 - 3s/epoch - 4ms/step
Epoch 2/50
720/720 - 2s - loss: 0.6412 - accuracy: 0.8069 - val_loss: 0.5244 - val_accuracy: 0.8500 - 2s/epoch - 2ms/step
Epoch 3/50
720/720 - 1s - loss: 0.4939 - accuracy: 0.8792 - val_loss: 0.3951 - val_accuracy: 0.9125 - 1s/epoch - 2ms/step
Epoch 4/50
720/720 - 1s - loss: 0.3661 - accuracy: 0.9597 - val_loss: 0.2769 - val_accuracy: 1.0000 - 1s/epoch - 2ms/step
Epoch 5/50
720/720 - 1s - loss: 0.2500 - accuracy: 0.9931 - val_loss: 0.1737 - val_accuracy: 1.0000 - 1s/epoch - 2ms/step
Epoch 6/50
720/720 - 1s - loss: 0.1379 - accuracy: 0.9972 - val_loss: 0.0495 - val_accuracy: 1.0000 - 1s/epoch - 2ms/step
Epoch 7/50
720/720 - 1s - loss: 0.0418 - accuracy: 1.0000 - val_loss: 0.0191 - val_accuracy: 1.0000 - 1s/epoch - 2ms/step

```

حال با توجه به شکل، نمودار های loss و دقت را رسم می کنیم :

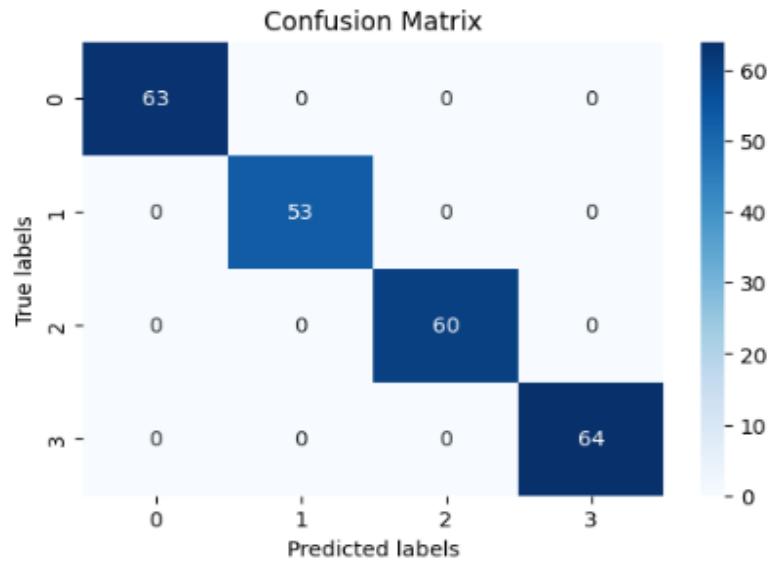


و نمودار دقت به صورت زیر است :



می بینیم که بعد از حدود ۸ ایپاک به دقت ۱۰۰ می رسیم.

ماتریس در هم ریختگی را رسم می کنیم و داریم :



همه مقادیر روی قطر اصلی هستند یعنی به درستی تشخیص داده شده اند.

حال برای بررسی دقیق از از **classification report** استفاده می کنیم:

| Classification Report: | | | | | |
|------------------------|-----------|--------|----------|---------|--|
| | precision | recall | f1-score | support | |
| 0 | 1.00 | 1.00 | 1.00 | 63 | |
| 1 | 1.00 | 1.00 | 1.00 | 53 | |
| 2 | 1.00 | 1.00 | 1.00 | 60 | |
| 3 | 1.00 | 1.00 | 1.00 | 64 | |
| accuracy | | | 1.00 | 240 | |
| macro avg | 1.00 | 1.00 | 1.00 | 240 | |
| weighted avg | 1.00 | 1.00 | 1.00 | 240 | |

میبینیم که دقت ۱۰۰ شده است. خوب است راجع به بخش های مختلف آن توضیحی داده شود:

$$\left\{ \begin{array}{l} precision = \frac{Truepositives(TP)}{TruePositives(TP)+FalsePositives(FP)} \\ Recall = \frac{Truepositives(TP)}{TruePositives(TP)+FalseNegatives(FN)} \\ F1score = \frac{2*Precision*Recall}{(Precision+Recall)} \end{array} \right.$$

بخش سوم

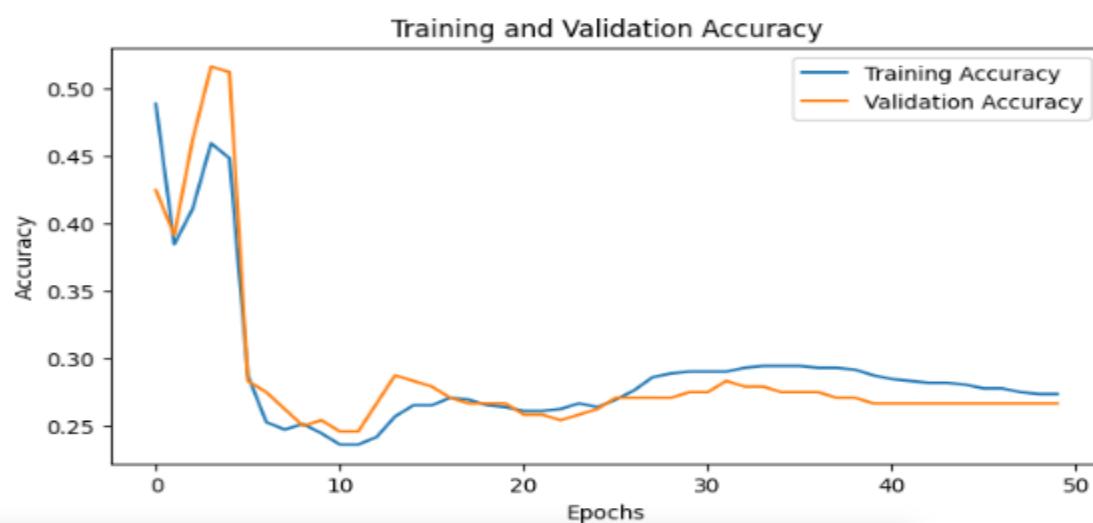
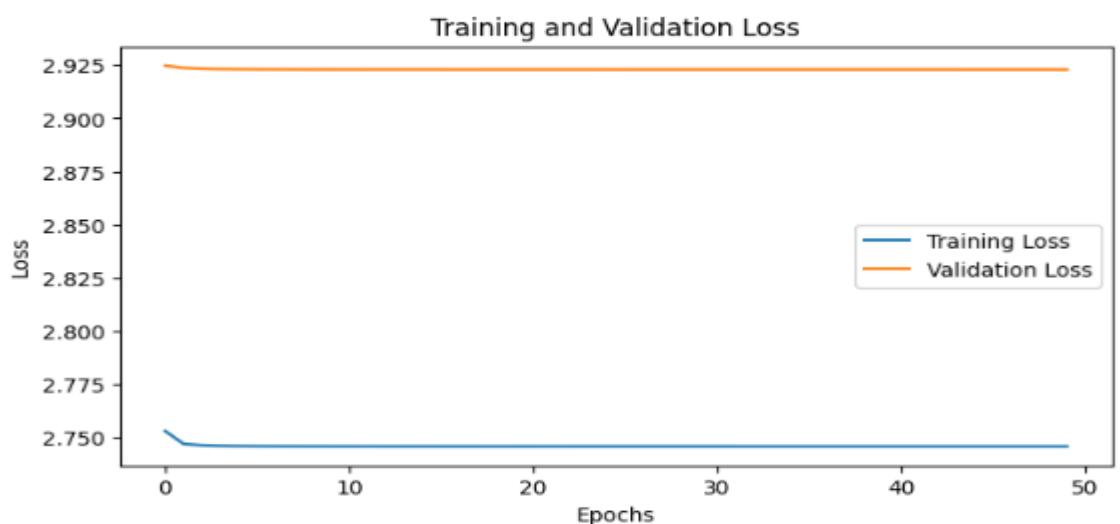
۳. فرآیند سوال قبل را با یک بهینه‌ساز و تابع اتلاف جدید انجام داده و نتایج را مقایسه و تحلیل کنید. بررسی کنید که آیا تغییر تابع اتلاف می‌تواند در نتیجه اثرگذار باشد؟

برای حالتی که از `sklearn` استفاده شده بود، در همان بخش دوم، این قسمت هم بررسی شد. در اینجا حالت دوم بررسی می‌شود.

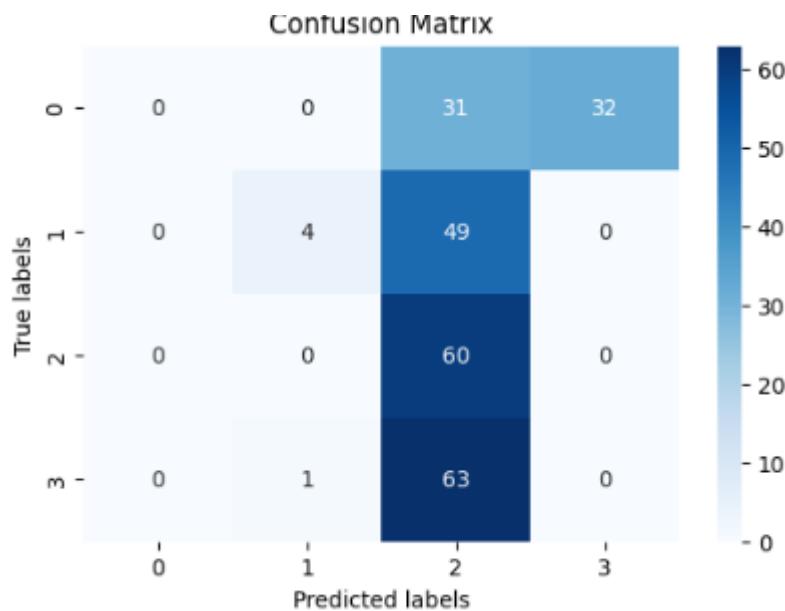
حالت اول) از توابع زیر استفاده می‌شود: (بقیه موارد همان حالت قبل است)

`optimizer=SGD(), loss='mean_squared_error'`

نمودارها را مشاهده می‌کنیم:



از شکل ها هم مشخص است که عملکرد مدل بدتر شده است اما به سراغ تحلیل دقیق تر می رویم :



| Classification Report: | | | | | |
|------------------------|-----------|--------|----------|---------|--|
| | precision | recall | f1-score | support | |
| 0 | 0.00 | 0.00 | 0.00 | 63 | |
| 1 | 0.80 | 0.08 | 0.14 | 53 | |
| 2 | 0.30 | 1.00 | 0.46 | 60 | |
| 3 | 0.00 | 0.00 | 0.00 | 64 | |
| accuracy | | | 0.27 | 240 | |
| macro avg | 0.27 | 0.27 | 0.15 | 240 | |
| weighted avg | 0.25 | 0.27 | 0.14 | 240 | |

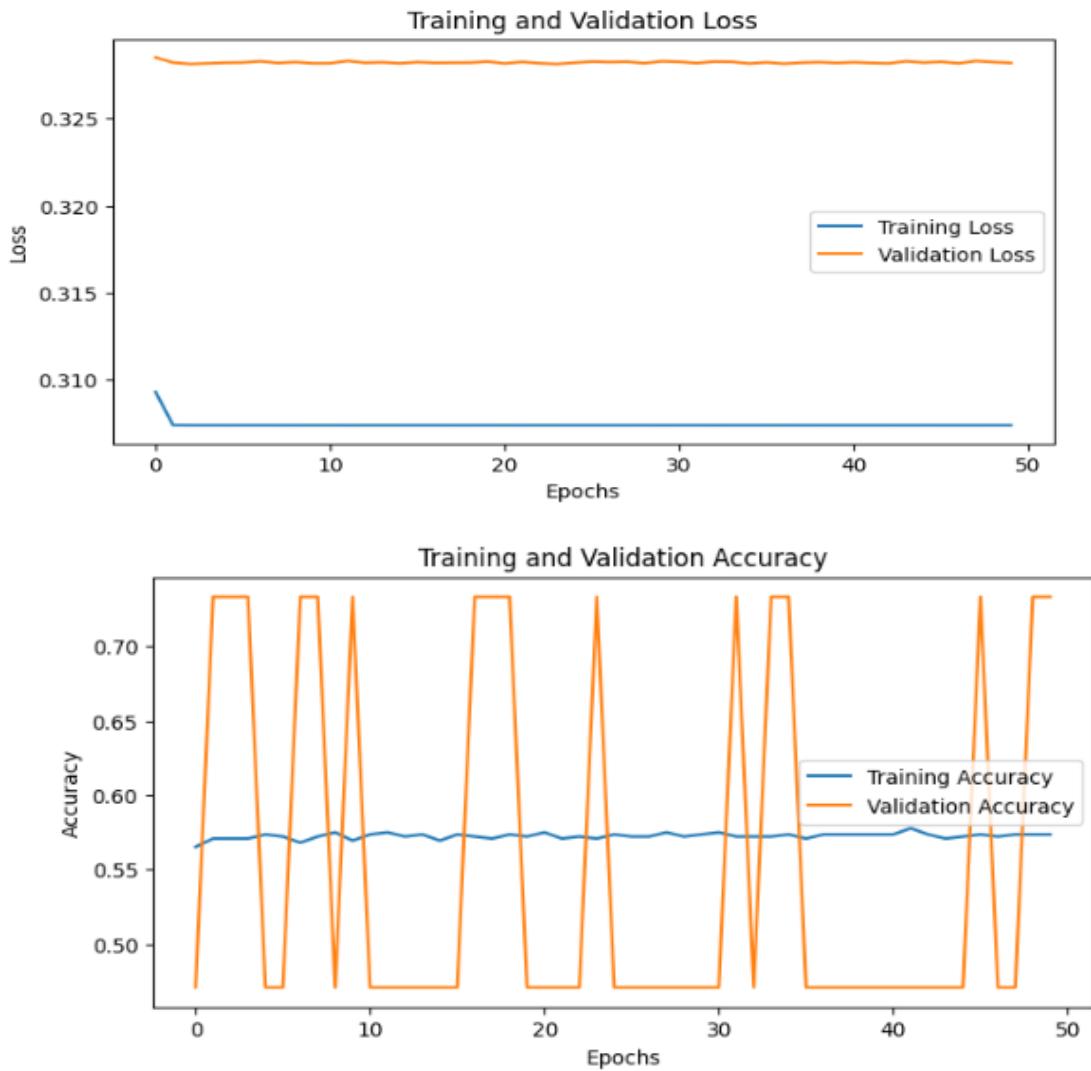
دقت حدود ۲۶ درصد بدست اومد که عملکرد خیلی بدتر شد ! (تعداد اعضای عناصر قطر اصلی هم خیلی کم هستند).

به بررسی حالت دیگر می پردازیم :

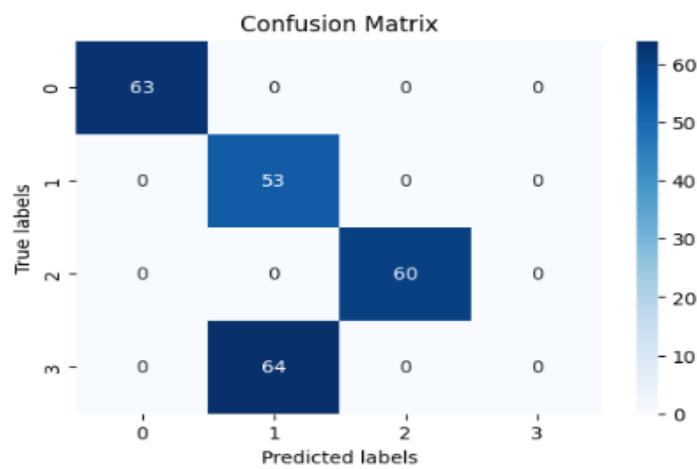
حالت دوم) از توابع زیر استفاده می شود: (بقیه موارد همان حالت قبل است)

`optimizer= RMSprop(), loss='categorical_hinge'`

خروجی ها را رسم می کنیم :



عملکرد بد مدل از شکل ها هم مشخص است اما دقیق تر میبینیم :



| Classification Report: | | | | | |
|------------------------|-----------|--------|----------|---------|--|
| | precision | recall | f1-score | support | |
| 0 | 1.00 | 1.00 | 1.00 | 63 | |
| 1 | 0.45 | 1.00 | 0.62 | 53 | |
| 2 | 1.00 | 1.00 | 1.00 | 60 | |
| 3 | 0.00 | 0.00 | 0.00 | 64 | |
| accuracy | | | 0.73 | 240 | |
| macro avg | 0.61 | 0.75 | 0.66 | 240 | |
| weighted avg | 0.61 | 0.73 | 0.65 | 240 | |

We can see that the performance of the network has become much worse!!

دقت حدود ۶۰ درصد شد و در کل عملکرد مدل طبقه بندی چند کلاسه با این پارامتر ها بسیار بدتر شدند! همان‌طور که از نتایج برمی‌آید، با عوض کردن تابع هزینه و بهینه ساز مدل، عملکرد مدل تغییر می‌کند که بر همین اساس می‌توان نتیجه گرفت که انتخاب تابع هزینه و بهینه ساز روی عملکرد مدل تاثیر دارد.

بخش چهارم

۴. در مورد K-Fold Cross-validation و Stratified K-Fold Cross-validation و مزایای هریک توضیح دهید. سپس با ذکر دلیل، یکی از این روش‌ها را انتخاب کرده و بخش «۲» این سوال را با آن پیاده‌سازی کنید و نتایج خود را تحلیل کنید.

ابتدا روش K-fold cross-validation را توضیح می‌دهیم و ازانجایی که روش validation شباهت زیادی به روش قبلی دارد، بخش متفاوت آن را توضیح می‌دهیم.

K-Fold Cross-validation

K-fold cross-validation یکی از تکنیک‌های اعتبارسنجی مدل‌های یادگیری ماشین است که برای ارزیابی عملکرد یک مدل و اطمینان از عمومی‌سازی (generalization) مناسب آن استفاده می‌شود. در این روش، کل دیتا است به K بخش مساوی تقسیم می‌شود. سپس مدل به تعداد K بار آموزش داده می‌شود. در هر بار آموزش، یکی از این بخش‌ها (fold‌ها) به عنوان مجموعه ارزیابی (یا اعتبارسنجی) در نظر گرفته می‌شود و با استفاده از دیگر بخش‌ها مدل آموزش داده می‌شود. یعنی در هر دور آموزش، $K-1$ مجموعه به عنوان داده آموزش استفاده می‌شود. این فرآیند به صورت زیر است:

۱. تقسیم دیتاست: دیتاست به K بخش مساوی ($fold$) تقسیم می‌شود.
۲. آموزش و ارزیابی مدل: برای هر یک از K بخش:
 - یکی از بخش‌ها به عنوان مجموعه ارزیابی انتخاب می‌شود.
 - باقی $K-1$ بخش به عنوان مجموعه آموزش استفاده می‌شود.
 - مدل با استفاده از داده‌های آموزش ساخته می‌شود و سپس بر روی داده‌های ارزیابی تست می‌شود.
۳. محاسبه عملکرد: خطای عملکرد مدل برای هر یک از K بخش محاسبه و ثبت می‌شود.
۴. میانگین‌گیری: در نهایت، میانگین تمامی خطاهای به دست آمده برای ارائه یک ارزیابی کلی از عملکرد مدل محاسبه می‌شود.

شکل زیر روش k -fold cross-validation را نمایش داده است. در هر دور یک $fold$ را به عنوان آزمون انتخاب کرده و دیگران را به عنوان آموزش استفاده می‌کیم.



اگر قصد داشته باشیم عملکرد مدل‌هایی با ساختارهای مختلف را مقایسه کنیم، از تمامی خطاهای به دست آمده با استفاده از این روش، میانگین می‌گیریم. این کار باعث می‌شود که ارزیابی ما دقیق‌تر و نسبت به تقسیم‌بندی‌های خاصی از داده‌ها کمتر حساس باشد. اما برای انتخاب مدل نهایی، یکی از K مدل‌های آموزش داده شده را انتخاب می‌کنیم. معمولاً مدلی که بهترین عملکرد را روی داده‌های ارزیابی داشته است، به عنوان مدل نهایی انتخاب می‌شود.

Stratified K-Fold Cross-validation

تفاوتی که این روش با روش قبلی دارد این است که در اینجا، برای هر **fold** ایجاد شده، تلاش می‌شود که پخش داده موجود، مانند پخش داده کل باشد. یعنی نسبت هر کلاس از داده در هر **fold** ایجاد شده، با نسبت پخش کلاس‌ها در کل داده یکسان است.

این روش به ویژه در مواردی که داده‌ها دارای کلاسی‌های نامتوازن (imbalanced classes) هستند، مفید است. به عنوان مثال، اگر در دیتاست اصلی ۷۰ درصد از نمونه‌ها به کلاس A و ۳۰ درصد به کلاس B تعلق داشته باشند، در هر **fold** از روش Stratified K-Fold نیز همین نسبتها حفظ می‌شوند. این کار باعث می‌شود که هر **fold** نماینده دقیقی از کل دیتاست باشد و عملکرد مدل بهتر و واقعی‌تر ارزیابی شود.

فرآیند Stratified K-Fold Cross-validation به صورت زیر است:

۱. تقسیم دیتاست با حفظ نسبت کلاس‌ها : دیتاست به K بخش مساوی تقسیم می‌شود، به طوری که نسبت هر کلاس در هر بخش با نسبت کلاس‌ها در کل دیتاست برابر است.

۲. آموزش و ارزیابی مدل: برای هر یک از K بخش:

- یکی از بخش‌ها به عنوان مجموعه ارزیابی انتخاب می‌شود.

- باقی K-1 بخش به عنوان مجموعه آموزش استفاده می‌شود.

- مدل با استفاده از داده‌های آموزش ساخته می‌شود و سپس بر روی داده‌های ارزیابی تست می‌شود.

۳. محاسبه عملکرد و میانگین‌گیری: مشابه روش K-Fold Cross-validation، خطاهای محاسبه و میانگین گرفته می‌شود.

این روش به ویژه در پروژه‌هایی که داده‌ها به صورت نامتوازن (imbalanced) هستند، دقت ارزیابی مدل را افزایش می‌دهد و از ایجاد بایاس‌های ناشی از عدم تعادل کلاس‌ها جلوگیری می‌کند.

پس در کل K-Fold Cross-validation یک روش موثر برای ارزیابی مدل‌ها است که با استفاده از تقسیم بندی چندگانه دیتاست به بخش‌های مساوی، عملکرد مدل را بر روی داده‌های مختلف بررسی می‌کند. Stratified K-Fold Cross-validation نیز با حفظ نسبت کلاس‌ها در هر بخش، بهبود دقت ارزیابی را در داده‌های نامتوازن ارائه می‌دهد.

در اینجا چون داده‌ها متعدد هستند از kfold استفاده می‌کنیم:

از کد زیر استفاده می شود:

another method:

```
✓ 0s [25] from sklearn.model_selection import KFold
      from tensorflow.keras.models import Sequential
      from tensorflow.keras.layers import Dense
      from sklearn.metrics import accuracy_score
      kf = KFold(n_splits=30, shuffle=True, random_state=24)      # Or other desired metric
                                                               # Adjust parameters as needed

      model_3 = Sequential([
          Dense(units = 5, activation='relu'),
          Dense(units = 4, activation='relu'),
          Dense(units = 4, activation='softmax'),
      ])

      model_3.compile(
          optimizer='Adam',
          loss='SparseCategoricalCrossentropy',
          metrics=['accuracy']
      )

✓ 18s total_accuracy = 0

for train_index, test_index in kf.split(x_shuffled):
    X_train, X_test = x_shuffled[train_index], x_shuffled[test_index]
    Y_train, Y_test = y_shuffled[train_index], y_shuffled[test_index]

    model_3.fit(X_train, Y_train)
    Y_pred = model_3.predict(X_test)
    accuracy = accuracy_score(Y_test, np.argmax(Y_pred, axis = 1))
    total_accuracy += accuracy
    print(f"Fold Accuracy: {accuracy:.4f}")

average_accuracy = total_accuracy / kf.n_splits
print(f"Average Accuracy: {average_accuracy:.4f}")
```

به مقدار دقت ۹۲ درصد رسیده ایم.

```
-----[=====
37/37 [=====] - 0s 3ms/step - loss: 0.4504 - accuracy: 0.9250
2/2 [=====] - 0s 10ms/step
Fold Accuracy: 0.9250
37/37 [=====] - 0s 3ms/step - loss: 0.4085 - accuracy: 0.9474
2/2 [=====] - 0s 8ms/step
Fold Accuracy: 0.9500
```

یک روش دیگر هم استفاده از کد زیر است که دقت داده های تست را در نهایت ۹۳ درصد داده است:

```

# Define a function to create and compile the model
def create_model():
    model = Sequential([
        Dense(units=5, activation='relu'),
        Dense(units=4, activation='relu'),
        Dense(units=4, activation='softmax')
    ])
    model.compile(
        optimizer=Adam(),
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy']
    )
    return model

kf = KFold(n_splits=10)

scores = []

```

```

for train_index, test_index in kf.split(x_shuffled):
    # Split data into training and testing sets
    x_train, x_test = x_shuffled[train_index], x_shuffled[test_index]
    y_train, y_test = y_shuffled[train_index], y_shuffled[test_index]

    # Further split training data into training and validation sets
    kf_inner = KFold(n_splits=5)
    inner_scores = []
    for train_inner_index, val_index in kf_inner.split(x_train):
        x_train_fold, x_val_fold = x_train[train_inner_index], x_train[val_index]
        y_train_fold, y_val_fold = y_train[train_inner_index], y_train[val_index]

        # Create model
        model = create_model()

        # Fit model with validation data
        history = model.fit(
            x_train_fold, y_train_fold,
            validation_data=(x_val_fold, y_val_fold),
            epochs=50,
            batch_size=1,
            verbose=0
        )

        # Evaluate model on validation set
        _, accuracy = model.evaluate(x_val_fold, y_val_fold, verbose=0)
        inner_scores.append(accuracy)

```

```

# Average validation accuracy
avg_val_accuracy = sum(inner_scores) / len(inner_scores)

# Create model using the entire training data
model = create_model()

# Fit model using entire training data
model.fit(x_train, y_train, epochs=50, batch_size=1, verbose=0)

# Evaluate model on test set
_, test_accuracy = model.evaluate(x_test, y_test, verbose=0)

# Append test accuracy to scores list
scores.append(test_accuracy)

# Print average test accuracy
print("Average test accuracy:", sum(scores) / len(scores))

```

→ Average test accuracy: 0.9366666674613953

سوال سه

در این سؤال از داده‌های پوشش جنگلی استفاده کردیم. این داده‌ها در کتابخانه scikit-learn موجود هستند و برای دانلود این داده‌ها از دستور fetch-covtype استفاده شد.

```
[68] import numpy as np
     import matplotlib.pyplot as plt
     import pandas as pd
     import seaborn as sns
     import warnings
     warnings.filterwarnings('ignore')

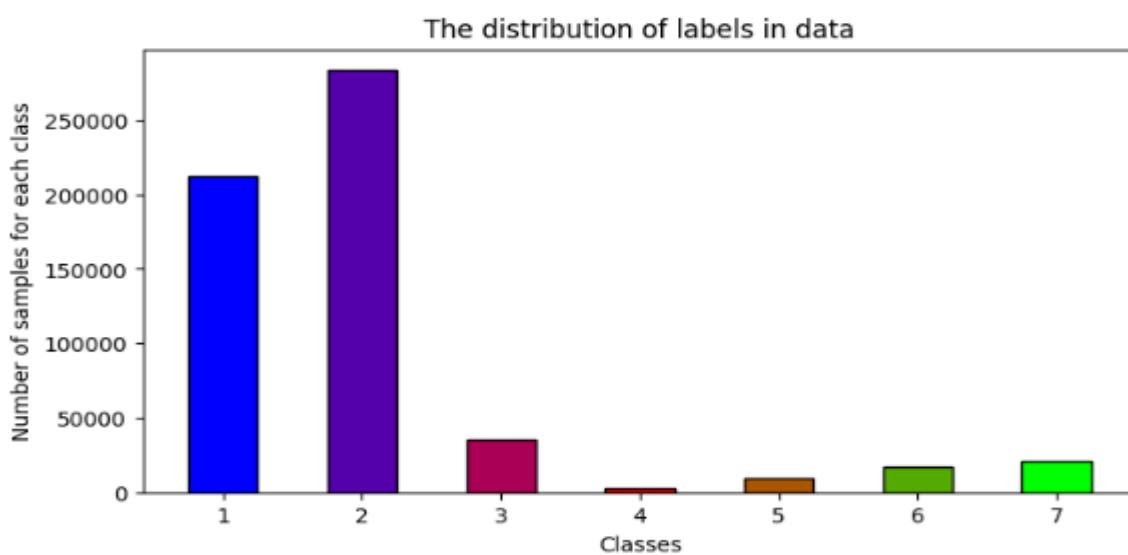
     from sklearn.datasets import fetch_covtype

[69] #download data
     dataset = fetch_covtype()
```

داده‌ها به صورت زیر هستند: که در سایت اطلاعات بیشتری از آن‌ها موجود است.

| | |
|----------------|--------|
| Classes | 7 |
| Samples total | 581012 |
| Dimensionality | 54 |
| Features | int |

داده‌ها در 7 کلاس هستند، حال باید بررسی کنیم که وضعیت پخش داده‌ها در این دیتاست، چگونه است در واقع در هر کلاس چند نمونه وجود دارد!



میبینیم که تعداد داده ها در هر کلاس خیلی پراکنده است و متعادل نیستند، این باعث می شود که مدل ما عملکرد خوبی نداشته باشد. مثلا داده ها در یک کلاس کمتر از ۵۰۰۰۰ تا و در کلاسی دیگر بیشتر از ۲۵۰۰۰۰ تا می باشند! اگر این عدم تعادل در داده های آموزشی نیز وجود داشته باشد، مدل ایجاد شده به طور قابل توجهی به سمت کلاسی که بیشترین تعداد داده را دارد متمایل خواهد شد (دارای bias می شود) و عملکرد مطلوبی نخواهد داشت. برای رفع این مشکل و متعادل کردن این دیたست، از روش Under-sampling استفاده می کنیم.

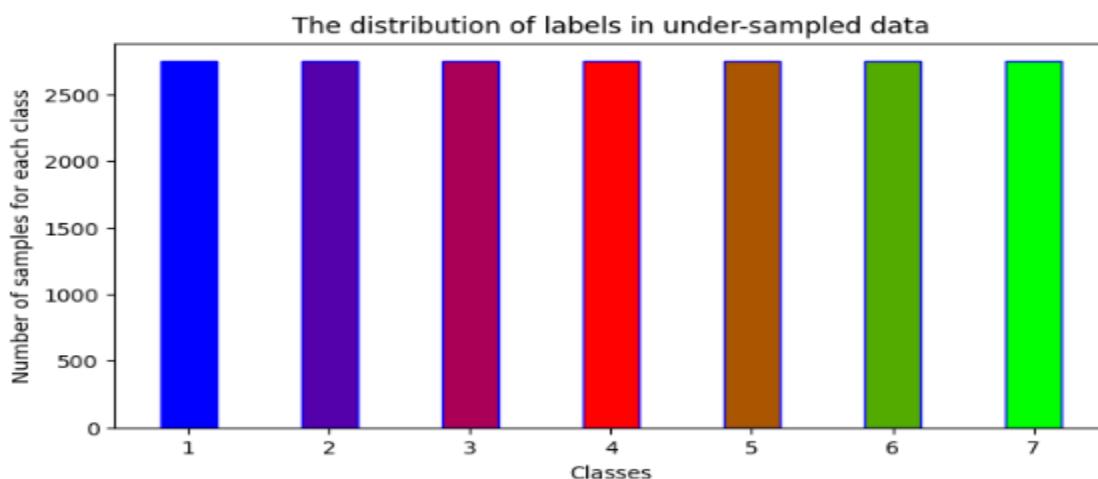
این روش شامل کاهش تعداد نمونه ها از کلاسی است که بیشترین تعداد داده را دارد تا تعداد نمونه های آن با کلاس های دیگر برابر شود. به این ترتیب، مدل ما می تواند به طور یکسانی از همه کلاس ها یاد بگیرد و تعادل بهتری در پیش بینی ها داشته باشد. البته باید توجه داشت که Under-sampling ممکن است منجر به از دست دادن اطلاعات مهم شود، بنابراین انتخاب دقیق نمونه ها برای حذف بسیار حائز اهمیت است. در برخی موارد، ترکیب روش های دیگری مانند Over-sampling یا استفاده از الگوریتم های مقاوم به عدم تعادل داده نیز می تواند موثر باشد. برای حل این مشکل از کد زیر استفاده می کنیم :

```
[76] sampler = RandomUnderSampler(random_state=24)
      x_resampled, y_resampled = sampler.fit_resample(x, y)

[77] hist, bins = np.histogram(y_resampled,bins=7)
      cmap = cm.get_cmap('brg');

      bins = np.unique(y_resampled) # For the bins to come
      plt.bar(bins, hist, width=0.4 ,edgecolor='blue',color=cmap(np.linspace(0, 1, len(hist))));
      plt.xticks(range(1,8), range(1,8));
      plt.title('The distribution of labels in under-sampled data')
      plt.ylabel('Number of samples for each class')
      plt.xlabel('Classes')
```

خروجی را مشاهده می کنیم :



پس از اعمال روش Under-sampling ، تعداد نمونه‌ها در هر کلاس یکسان شده و دیتاست متعادل می‌شود. با این حال، این روش منجر به کاهش تعداد کل داده‌ها می‌شود. به عبارت دیگر، برای متعادل کردن دیتاست، مجبور به حذف برخی از نمونه‌های کلاس‌هایی که بیشترین تعداد را دارند هستیم.

پس اکنون ۷ کلاس داریم که هر کلاس شامل حدود ۲۷۰۰ نمونه است. به سراغ ادامه حل سوال می‌رویم ...

بخش اول

۱. با استفاده از بخشی از داده‌ها، مجموعه‌داده را به دو بخش آموزش و آزمون تقسیم کنید (حداقل ۱۵ درصد از داده‌ها را برای آزمون نگه دارید). توضیح دهید که از چه روشی برای انتخاب بخشی از داده‌ها استفاده کردید. آیا روش بهتری برای این کار می‌شناسید؟

در ادامه، برنامه‌ای بنویسید که درخت تصمیمی برای طبقه‌بندی کلاس‌های این مجموعه‌داده طراحی کند. خروجی درخت تصمیم خود را با برنامه‌نویسی و یا به صورت دستی تحلیل کنید.

برای تقسیم‌بندی داده‌ها از روش نمونه‌برداری تصادفی (random sampling) استفاده می‌کنیم. تابع مورد استفاده برای این تقسیم‌بندی، train_test_split نام دارد. در این روش، داده‌های آموزشی به صورت کاملاً تصادفی انتخاب می‌شوند. در این بخش، ۱۵ درصد از داده‌ها به مجموعه آزمون اختصاص داده شده و باقی داده‌ها به عنوان مجموعه آموزش استفاده می‌شوند.

علاوه بر نمونه‌برداری تصادفی، دو استراتژی دیگر نیز برای انتخاب داده‌های آموزشی و آزمونی وجود دارد:

۱. اعتبارسنجی متقابل (Cross Validation) : در این روش، دیتاست به چندین بخش تقسیم می‌شود و مدل به تعداد بخش‌های تقسیم شده آموزش داده می‌شود. سپس بهترین مدل از میان این مدل‌ها انتخاب می‌شود. هنگامی که از این روش برای مقایسه الگوریتم‌های مختلف یا مدل‌هایی با فراپارامترهای مختلف استفاده می‌شود، خطای تمامی مدل‌ها محاسبه و میانگین گرفته می‌شود تا الگوریتم‌ها بر اساس این میانگین با هم مقایسه شوند. این روش باعث می‌شود ارزیابی مدل‌ها دقیق‌تر باشد و از همه داده‌ها به طور مؤثرتری استفاده شود.

۲. بوت‌استرپ (Bootstrap) : این روش مبتنی بر انتخاب زیرمجموعه‌هایی از دیتاست اصلی با جایگذاری است. به این معنا که از یک دیتاست اصلی، چندین زیردیتاست ایجاد می‌شود که هر کدام با قابلیت جایگذاری داده‌ها انتخاب می‌شوند. یعنی برای انتخاب داده‌ها در هر زیردیتاست، ممکن است برخی داده‌ها چندین بار انتخاب شوند

و برخی اصلاً انتخاب نشوند. این روش به ارزیابی مدل‌ها کمک می‌کند تا با تنوع بیشتری از داده‌ها مواجه شوند و برآورده دقیق‌تری از عملکرد مدل بر روی داده‌های نادیده به دست آید.

استفاده از هر کدام از این روش‌ها به هدف و نوع پروژه بستگی دارد و می‌تواند تأثیر زیادی بر عملکرد و دقت نهایی مدل داشته باشد. عموماً، این روش‌ها زمانی به کار گرفته می‌شوند که تعداد داده‌های آموزشی کم باشد، اما اگر پس از کاهش تعداد داده‌ها (under-sampling) به اندازه‌ای مناسب (مثلاً ۲۷۰۰ داده برای هر کلاس) برسیم، نیازی به استفاده از دو روش دیگر نداریم.

❖ در فرآیند آموزش مدل درخت تصمیم‌گیری، چندین روش مختلف برای آموزش مدل مقایسه شد.^۴ روش) پس در ابتدا :

```
[106] dt = DecisionTreeClassifier(random_state=24)
      dt1 = clone(dt)
      dt3 = clone(dt)
      dt2 = clone(dt)
```

این کد برای ایجاد چهار مدل درخت تصمیم استفاده می‌شود. توابع ()clone برای ایجاد نسخه‌های مستقل از یک مدل موجود استفاده می‌شود تا هر مدل مستقل از دیگری باشد و تغییرات در یک مدل تأثیری بر سایر مدل‌ها نگذارد. در اینجا، dt1، dt2، و dt3 به ترتیب سه نسخه از مدل اصلی DecisionTreeClassifier با همان تنظیمات و تصمیمات مولده تولید می‌شوند.

از میان این روش‌ها، بهترین آن‌ها برای مراحل بعدی انتخاب شدنند. این روش‌ها شامل موارد زیر بودند:

۱. آموزش داده‌هایی که تحت عملیات under-sampling قرار گرفته‌اند؛ به این معنا که تعداد نمونه‌های هر کلاس را به حدی کاهش داده‌ایم که تعادل بین کلاس‌ها برقرار شود.

```
# With under-sampled data
dt.fit(x_train_r, y_train_r)
print(f'Model score for under-sampled data is: {dt.score(x_test_r, y_test_r):0.4f}', end='\n\n')
```

۲. آموزش داده‌ها بدون هیچ کار و عملیات اضافه‌ای؛ به عبارت دیگر، استفاده از داده‌های اصلی بدون تغییر در توزیع یا تعداد آن‌ها.

```
# With normal data
dt1.fit(x_train, y_train)
print(f'Model score for normal data is: {dt1.score(x_test, y_test):0.4f}', end='\n\n')
```

۳. آموزش داده‌ها با اعمال وزن به الگوریتم؛ به این معنا که وزن‌هایی به هر کلاس اختصاص داده شده و در فرآیند آموزش مورد استفاده قرار گرفته‌اند، به‌طوری‌که تأثیر هر کلاس بر مدل متوازن باشد.

```

# With Weighted classes
weight = [
    len(y)/len(y[y==1]),
    len(y)/len(y[y==2]),
    len(y)/len(y[y==3]),
    len(y)/len(y[y==4]),
    len(y)/len(y[y==5]),
    len(y)/len(y[y==6]),
    len(y)/len(y[y==7])
]
w_train = [weight[a-1] for a in y_train]
w_test = [weight[a-1] for a in y_test]
dt2.fit(x_train, y_train, w_train)
print(f"The weight for each class is : {np.round(weight).astype('int16')}")
print(f'Model score for weighted data is: {dt2.score(x_test,y_test):0.4f}', end='\n\n')

```

۴. آموزش داده‌ها با استفاده از روش ۵-fold Cross-validation، به این معنا که داده‌ها به ۵ بخش تقسیم شده و مدل به طور متوالی بر روی ۴ بخش آموزش داده می‌شود و بر روی یک بخش دیگر آزمایش می‌شود، و این فرآیند برای ۵ بار تکرار می‌شود.

۵. آموزش داده‌ها با استفاده از روش ۵-fold Cross-validation & Stratified Cross-validation، تضمین می‌کند که توزیع کلاس‌ها در هر بخش حفظ شود.

```

# With K-fold method
kfold_score = cross_val_score(dt, X, y, cv=5)
print(f'Model score for 5-fold CV is : {np.mean(kfold_score):.4f} --> (mean value)', end='\n\n')

# Stratified K-fold model
kf = StratifiedKFold(n_splits=5, shuffle=True, random_state=24)
fold_acc = {'train':[], 'val':[]}
fold_hat = {'train':[], 'val':[]}
fold_ground = {'train':[], 'val':[]}
fold_input = {'train':[], 'val':[]}
all_models = []

# Train model
for train_index, val_index in kf.split(x_train,y_train):
    X_train_fold, X_val_fold = x_train[train_index], x_train[val_index]
    y_train_fold, y_val_fold = y_train[train_index], y_train[val_index]

    fold_input['train'].append(X_train_fold)
    fold_input['val'].append(X_val_fold)
    fold_ground['train'].append(y_train_fold)
    fold_ground['val'].append(y_val_fold)

    # Train selected folds
    model = clone(dt3)
    model.fit(X_train_fold,y_train_fold)

    # Accuracy
    train_hat = model.predict(X_train_fold)
    fold_hat['train'].append(train_hat)
    train_hat = train_hat == y_train_fold
    train_score = np.sum(train_hat.astype('float64'))/len(y_train_fold)
    fold_acc['train'].append(train_score)

    val_hat = model.predict(X_val_fold)
    fold_hat['val'].append(val_hat)
    val_hat = val_hat == y_val_fold
    val_score = np.sum(val_hat.astype('float64'))/len(y_val_fold)
    fold_acc['val'].append(val_score)

```

این مقایسه‌ها به منظور انتخاب بهترین روش آموزش برای مدل درخت تصمیم‌گیری انجام شد.

نتایج حاصل از این روش‌ها را در ادامه خواهیم دید:

```
Model score for under-sampled data is: 0.8017  
Model score for normal data is: 0.9406  
The weight for each class is : [ 3 2 16 212 61 33 28]  
Model score for weighted data is: 0.8918  
Model score for 5-fold CV is : 0.5584 --> (mean value)  
Model score for Stratified 5-fold CV is 0.9348 --> (mean value)
```

اگرچه عملکرد مدل‌ها برای کلاس‌های مختلف متفاوت است و هیچ مدلی به طور کامل از بقیه عملکرد بهتری ندارد، اما به طور کلی می‌توانیم بر اساس عملکرد کلی، حالت دوم را انتخاب کنیم چرا که دقت حدود ۹۴ درصد دارد که بهتر از بقیه حالت‌هاست. به عبارت دیگر، استفاده از داده‌های اصلی بدون هیچ وزنی، نتیجه‌ای بهتر و مطلوب‌تر را در کلیت مدل ارائه می‌دهد. بنابراین، برای ادامه محاسبات، تصمیم گرفته شده است که از تمامی داده‌ها بدون در نظر گرفتن وزن‌دهی استفاده شود. البته حالت پنجم هم عملکرد نزدیک به این مدل دارد ولی از حالت دوم استفاده می‌کنیم ...

اکنون اطلاعات اساسی درخت را استخراج می‌کنیم:

ابتدا، با استفاده از متغیر `dt1.tree`، اطلاعات مربوط به درخت را به دست می‌آوریم. سپس، سه ویژگی اصلی را استخراج می‌کنیم: ۱) **Max depth of the tree** که عمق بیشترین مسیر در درخت را نمایش می‌دهد، ۲) **Number of samples at leaves** که تعداد نمونه‌های موجود در برگ‌های درخت را نشان می‌دهد و ۳) **Impurity at leaves** که میزان اصطکاک یا **Impurity** در برگ‌های درخت را نشان می‌دهد. این اطلاعات اساسی معمولاً برای ارزیابی و فهم بهتر عملکرد مدل درخت تصمیم استفاده می‌شوند که به صورت زیر هستند:

```

tree_ob = dt1.tree_
# max depth
md = tree_ob.max_depth
print(f'Max depth of the tree is {md}', end='\n\n')

# number of samples in the leaves
ns = tree_ob.n_node_samples[-1]
print(f'There are {ns} number of samples at leaves', end='\n\n')

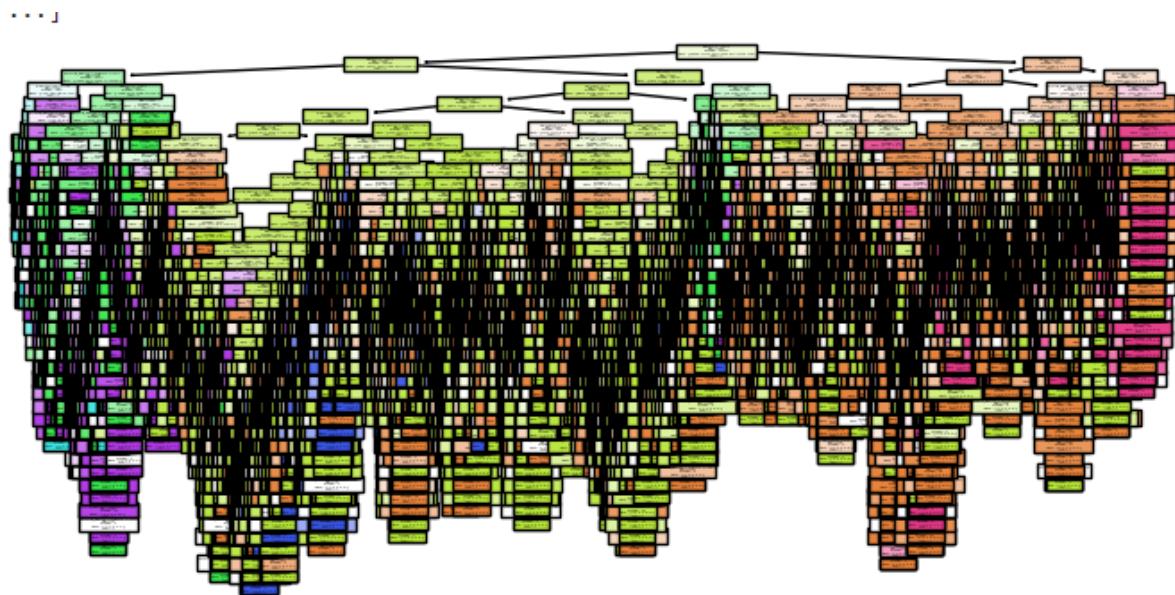
# Impurity at leaves
i_in_leaf = tree_ob.impurity[-1]
print(f'Impurity at leaves is {i_in_leaf}')


Max depth of the tree is 41
There are 1 number of samples at leaves
Impurity at leaves is 0.0

```

شکل کلی درخت به صورت زیر است:

```
plot_tree(dt1, filled=True, feature_names=dataset.feature_names, class_names=[str(i) for i in np.unique(y)])
```

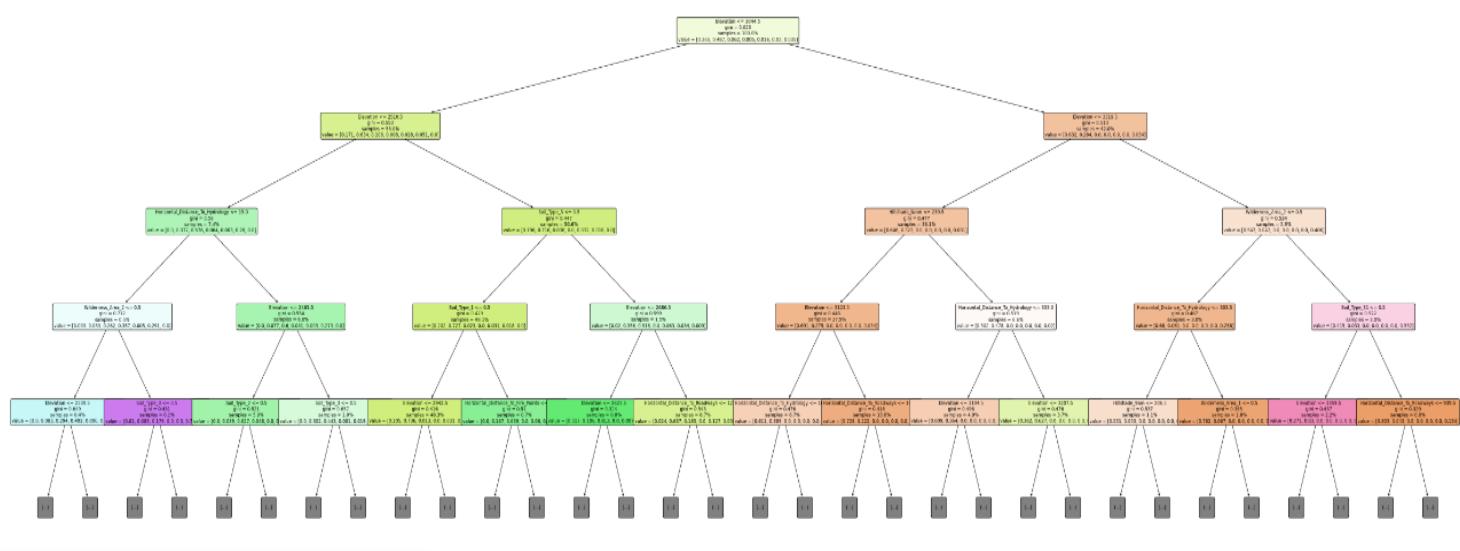


اما از کد زیر استفاده می کنیم:

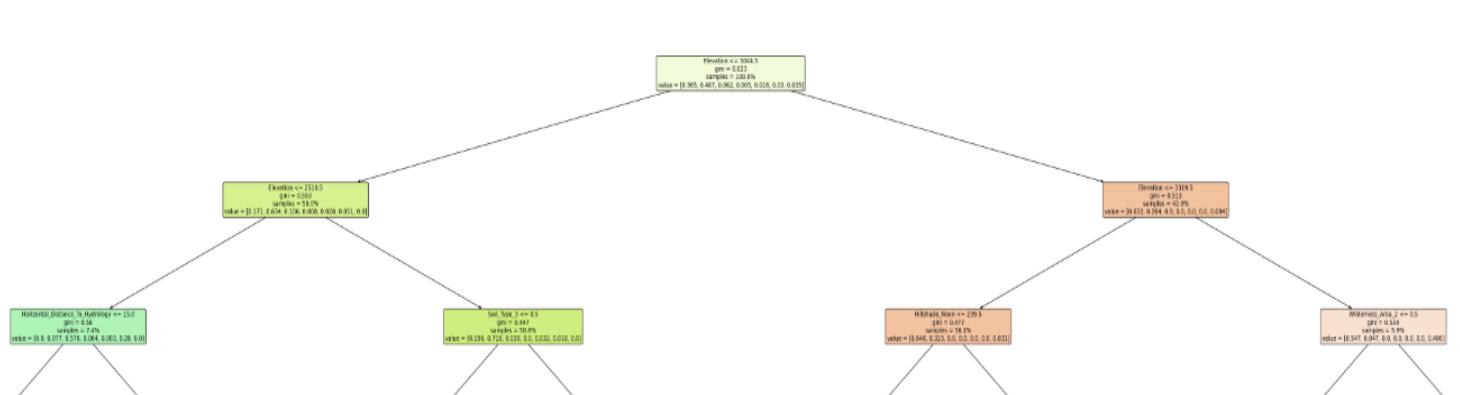
```
plt.figure(figsize=(60,20))
plot_tree(dt1, max_depth=4, filled=True , fontsize=10,feature_names=features, rounded=True, proportion=True)
```

در واقع محدود تر می کنیم و به صورت کامل رسم نمی کنیم(به علت پیچیدگی محاسبات و سخت ران شدن
مدل)

و درخت به فرم زیر می شود :



در واقع لایه اول درخت به صورت زیر است :



تقسیم بندی برای لایه دوم درخت به صورت زیر است :

در هر بلوک اطلاعاتی مانند نام ویژگی، مقدار gini، تعداد نمونه‌های موجود در آن گره به نمایش درآمده‌اند.

مانند : (اولین بلوک)

```

Elevation <= 3044.5
gini = 0.623
samples = 100.0%
value = [0.365, 0.487, 0.062, 0.005, 0.016, 0.03, 0.035]

```

که در تصویر بالا، یک آرایه با ۷ عنصر عددی است که نشان دهنده فراوانی یا احتمال هر کلاس/برچسب در این گره از درخت است. مجموع این عناصر برابر با ۱ است. به عنوان مثال، اگر برچسب های داده ها $[1, 0, 1]$ باشند، اولین دو عنصر ممکن است فراوانی کلاس ۰ و ۱ در این گره باشند. elevation=3044.5 یک معیار تصمیم گیری برای جدا کردن این گره از گره قبلی براساس یک ویژگی خاص است. برای مثال، اگر ویژگی پیشگو، درآمد باشد، بالای ۳۰۴۴.۵ دلار درآمد به سمت راست درخت و کمتر از آن به سمت چپ درخت قرار می گیرد. gini=0.623 یک شاخص ناهمگنی داده ها در این گره است که بین ۰ (کاملاً همگن) و ۰.۵ (کاملاً ناهمگن) است. یک مقدار gini بالا نشان می دهد که داده ها در این گره متنوع هستند. samples=100.0% نشان می دهد که همه نمونه های داده در این گره از درخت قرار گرفته اند.

بلوک بعدی سمت چپ:

```

Elevation <= 2510.5
gini = 0.553
samples = 58.0%
value = [0.171, 0.634, 0.106, 0.008, 0.028, 0.051, 0.0]

```

۵۸ درصد نمونه ها در این گره هستند. لایه بعدی این بلوک به صورت زیر است:

```

Horizontal_Distance_To_Hydrology <= 15.0
gini = 0.58
samples = 7.4%
value = [0.0, 0.077, 0.576, 0.064, 0.003, 0.28, 0.0]

```

```

Soil_Type_3 <= 0.5
gini = 0.447
samples = 50.6%
value = [0.196, 0.716, 0.038, 0.0, 0.032, 0.018, 0.0]

```

از اون ۵۸ درصد، در این دولایه به نسبت ۷,۴ و ۶,۵ تقسیم شدند. این روال تا انتهای ادامه می باید.

این اطلاعات برای درک چگونگی ساخت، اعتبارسنجی و ارزیابی عملکرد یک مدل درخت تصمیم در مسائل دسته بندی یا پیش بینی استفاده می شوند.

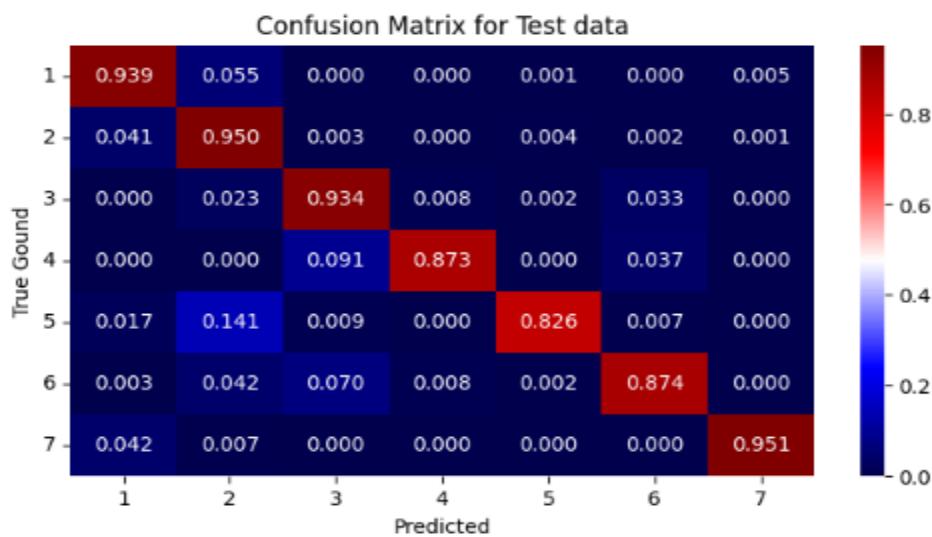
این درخت آموزش داده شده تا ۴۳ سطح عمق جدا کنندگی داشته و در هر برگ آن، تنها ۲ نمونه وجود دارد. از این اطلاعات، می توان چند نکته مهم را استخراج کرد: اولاً، به دلیل بزرگ بودن تعداد سطوح این درخت، امکان

نمایش همه گرهها و برگهای آن به صورت تصویر یا متن وجود ندارد. دوماً، این مدل به شدت دچار overfitting شده است؛ به عبارت دیگر، بیش از حد به داده‌های آموزش خود برازش شده و احتمالاً نمی‌تواند با داده‌های جدید به خوبی کار کند.

بخش دوم

۲. با استفاده از ماتریس درهم ریختگی و حداقل سه شاخصه ارزیابی مربوط به وظیفه طبقه‌بندی، عمل کرد درخت آموزش داده شده خود را روی بخش آزمون داده‌ها ارزیابی کنید و نتایج را به صورت دقیق گزارش کنید.
تأثیر مقادیر کوچک و بزرگ حداقل دو فراپارامتر را بررسی کنید. تغییر فراپارامترهای مربوط به هرس کردن چه تأثیری روی نتایج دارد و مزیت آن چیست؟

ماتریس درهم ریختگی درخت تصمیم را برای داده‌های تست به صورت زیر نمایش می‌دهیم :

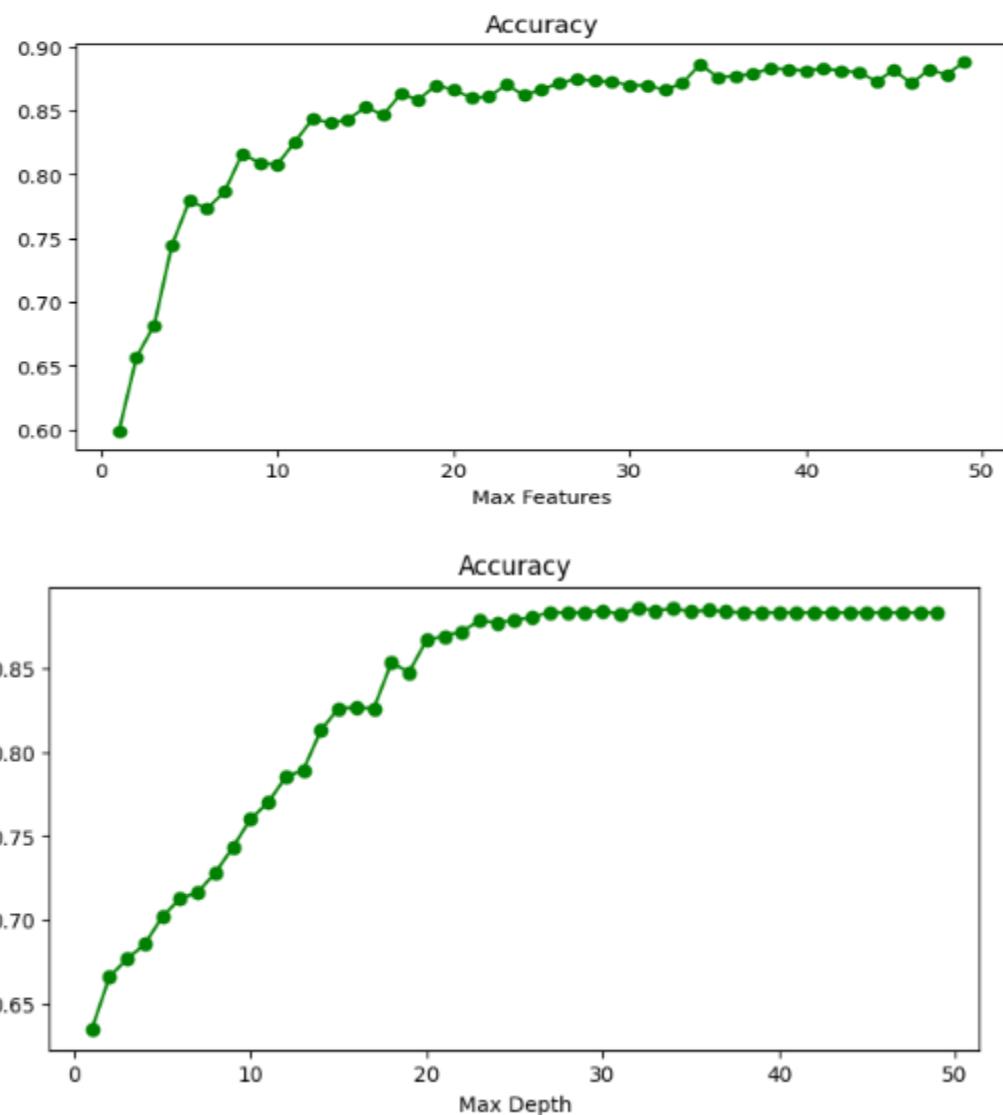


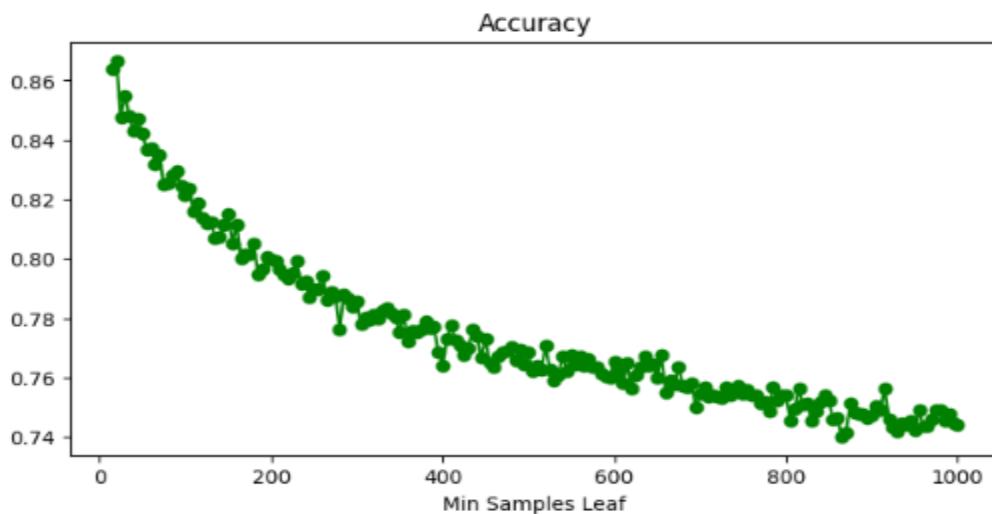
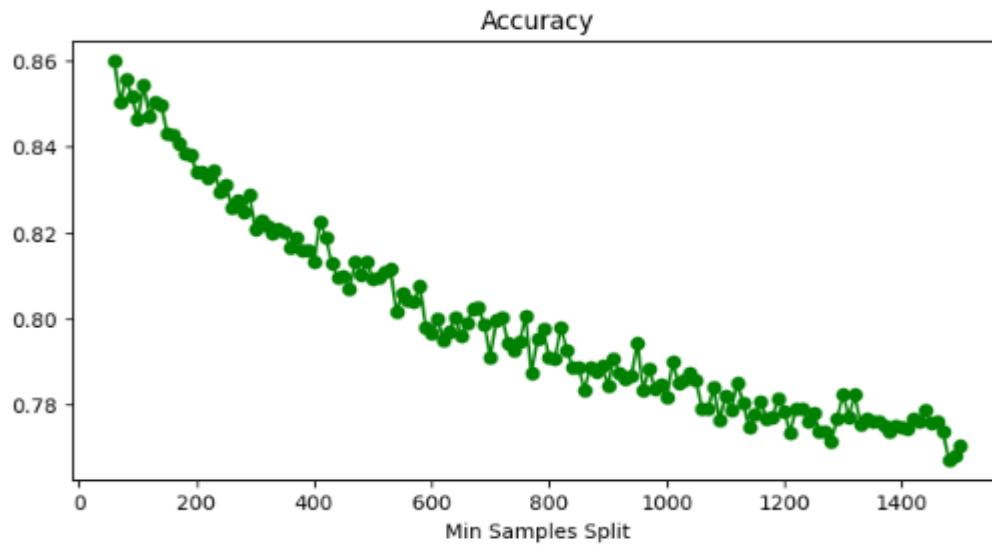
از سه معیار دیگر برای بررسی مدل استفاده می‌شود :

| 1 to 7 of 7 entries Filter | | | | | | |
|----------------------------|-------|-----------|--------|--------|--------|----------|
| index | class | precision | | recall | | f1_score |
| 0 | 1 | | 0.9411 | | 0.9412 | 0.9412 |
| 1 | 2 | | 0.9509 | | 0.9503 | 0.9508 |
| 2 | 3 | | 0.9385 | | 0.9374 | 0.9389 |
| 3 | 4 | | 0.8478 | | 0.8499 | 0.8489 |
| 4 | 5 | | 0.8351 | | 0.8605 | 0.8478 |
| 5 | 6 | | 0.8903 | | 0.8881 | 0.8882 |
| 6 | 7 | | 0.9502 | | 0.9453 | 0.9477 |

❖ بررسی هایپرپارامتر ها

در ادامه توسط کد پایتون یک آنالیز برای هایپرپارامترهای مدل Decision Tree انجام می‌دهیم. این آنالیز از متغیرهای مختلفی مانند حداقل عمق درخت `max_depth`, تعداد نمونه‌های لازم برای تقسیم یک گره `min_samples_leaf`, تعداد نمونه‌های لازم برای یک برگ `min_samples_split` و تعداد ویژگی‌های مجاز برای جستجو در هر تقسیم `max_features` را در نظر می‌گیر و به طور کلی یک حلقه برای هر پارامتر استفاده می‌کند تا مقادیر مختلف آن را بررسی کند و دقت مدل را برای هر مقدار از هر پارامتر ذخیره کند. سپس، نمودارهایی از دقت مدل بر حسب مقادیر هر پارامتر رسم می‌شود. به عنوان مثال، نمودار "Accuracy vs Max Depth" نشان می‌دهد که چطور دقت مدل تغییر می‌کند در واکنش به افزایش عمق درخت. همچنین دیگر نمودارها هم نشان‌دهنده تأثیر تغییر هر پارامتر بر دقت مدل هستند. در ادامه نتایج را مشاهده می‌کنیم :





وقتی مدل را با محدودیت‌هایی مثل حداقل عمق درخت max depth ، تعداد نمونه‌های لازم برای تقسیم یک گره min samples split ، و تعداد نمونه‌های لازم برای یک برگ min samples leaf محدود می‌کنیم، این محدودیت‌ها می‌توانند از بیش‌بازش یا *overfitting* جلوگیری کنند، یعنی مدل به جای یادگیری الگوهای واقعی در داده‌ها، الگوهای نمونه‌های تصادفی را هم یاد بگیرد.

نکته‌ای که در اینجا مطرح شده این است که حداقل عمق درخت یا max depth می‌تواند تأثیر متفاوتی بر روی دقت مدل داشته باشد. از طرفی، نمودار آن نشان می‌دهد که با افزایش عمق درخت، دقت مدل بر روی داده‌های آزمون افزایش می‌یابد، اما در عین حال، با محدود کردن دقت مدل، از بیش‌بازش جلوگیری می‌شود.

با استفاده از نمودارهایی که عملکرد مدل برای مقادیر مختلف **hyperparameter**ها را نشان می‌دهند، می‌توان انواع مختلفی از محدودیت‌ها را در نظر گرفته و با استفاده از روش‌های مثل **GridSearch**، بهترین مقادیر این **hyperparameter**ها را برای مدل انتخاب کرد. این کار معمولاً بھبود عملکرد مدل را در پیش‌بینی داده‌های جدید و ناشناخته بھبود می‌بخشد.

:GridSearch

از کد زیر استفاده می‌کنیم :

در این کد **GridSearch** با استفاده از پارامترها، مدل را با استفاده از اعتبارسنجی متقطع ($cv=3$) و بر اساس معیار دقت (`scoring='accuracy'`) بهینه می‌کند و سپس مدل بهینه شده با داده‌های آموزشی، آموزش داده می‌شود و بهترین نمره و پارامترهای بهینه یافته بر روی **داده‌های تست چاپ می‌شود**.

```
❶ main_model = DecisionTreeClassifier(max_depth=20, min_samples_split=20, min_samples_leaf=20, max_features=20, random_state=24)
grid_model = clone(main_model)

grid_param = {
    'max_depth': np.arange(30,42,3),
    'min_samples_leaf': np.arange(2,50,10),
    'max_features': np.arange(40,50,5)
}

❷ # Instantiate GridSearchCV
grid_search = GridSearchCV(grid_model, grid_param, cv=3, scoring='accuracy', verbose=1)

grid_search.fit(x_train_scaled, y_train)

print(f'The best score on Test data is :{grid_search.score(x_test_scaled,y_test):.4f}')
print(f'The best parameters are :{grid_search.best_params_}')


❸ Fitting 3 folds for each of 40 candidates, totalling 120 fits
The best score on Test data is :0.9246
The best parameters are :{'max_depth': 30, 'max_features': 45, 'min_samples_leaf': 2}
```

The best score on Test data is :0.9246

با توجه به کد بالا دقت حدود ۹۲ درصد می‌شود، به ازای هایپرپارامترهای مناسب که به صورت زیر هستند:

The best parameters are :{'max_depth': 30, 'max_features': 45, 'min_samples_leaf': 2}

بخش سوم

۳. توضیح دهید که روش‌هایی مانند جنگل تصادفی و AdaBoost چگونه می‌توانند به بهبود نتایج کمک کنند. سپس، با انتخاب یکی از این روش‌ها و استفاده از فرآپارامترهای مناسب، سعی کنید نتایج پیاده‌سازی در مراحل قبلی را ارتقاء دهید.

روش‌های ensemble learning از دسته روش‌های RandomForest و AdaBoost هستند. این روش‌ها از مجموعه‌ای از مدل‌ها برای بهبود دقت و پایداری پیش‌بینی‌ها استفاده می‌کنند. به طور کلی، ensemble learning با استفاده از ایجاد زیرمجموعه‌هایی از مجموعه داده اصلی و جایگشت داده، مدل‌های مختلف و مستقلی تولید می‌کند. این فرآیند به دست آوردن مدل‌هایی با ویژگی‌های مکمل و ترکیب آن‌ها منجر می‌شود که خاصیت‌های زیر را به همراه دارد:

کاهش واریانس (جلوگیری از overfitting)

یکی از مزایای اصلی استفاده از روش‌های ensemble learning، کاهش واریانس مدل است. در مدل‌های تک‌تک (single models)، مانند درخت تصمیم، احتمال overfitting بالاست؛ به این معنی که مدل به جای یادگیری الگوهای کلی داده، به جزئیات و نویزهای داده تمرکز می‌کند. با ترکیب چندین مدل در روش‌های ensemble، نتایج نهایی به میانگین نتایج مدل‌های مختلف نزدیک می‌شود. این امر باعث می‌شود که تأثیر نویزها و جزئیات غیرمفید کاهش یابد و در نتیجه، مدل کلی عملکرد بهتری بر روی داده‌های جدید داشته باشد.

Generalization

روش‌های ensemble با ایجاد و ترکیب مدل‌های مختلف، توانایی تعمیم‌دهی را افزایش می‌دهند. هر مدل به صورت جداگانه بخش‌های مختلفی از داده را می‌آموزد و با ترکیب این مدل‌ها، توانایی شناسایی الگوهای متنوع و پیچیده‌تر در داده‌ها فراهم می‌شود. به عبارت دیگر، ترکیب چندین مدل به کشف و پوشش دادن جوانب مختلفی از توزیع آماری موجود در داده کمک می‌کند که باعث بهبود توانایی پیش‌بینی مدل نهایی می‌شود.

تأثیر کمتر نویز روی مدل

از آنجایی که مدل‌های individual در روش‌های ensemble بر روی زیرمجموعه‌های مختلفی از داده‌ها آموزش داده می‌شوند، تأثیر نویز در داده‌های آموزشی کاهش می‌یابد. این روش‌ها با درنظر گرفتن عملکرد تمامی مدل‌ها، ناهنجاری‌ها و داده‌های نویزی را تضعیف می‌کنند. در نتیجه، مدل نهایی از پایداری و دقت بالاتری برخوردار است و بهتر می‌تواند الگوهای اصلی داده را شناسایی کند.

افزایش دقت مدل

یکی دیگر از مزایای مهم استفاده از روش‌های ensemble learning، افزایش دقت مدل است. ترکیب مدل‌های مختلف با رویکردهایی مثل میانگین‌گیری (boosting) (bagging) یا تقویت (boosting)، معمولاً منجر به مدل‌هایی با دقت بالاتر می‌شود. این افزایش دقت به دلیل ترکیب قدرت پیش‌بینی مدل‌های مختلف و جبران نقاط ضعف هر مدل توسط مدل‌های دیگر است.

انعطاف‌پذیری و تطبیق‌پذیری

روش‌های ensemble learning به دلیل توانایی استفاده از مدل‌های مختلف و ترکیب آن‌ها، انعطاف‌پذیری بالایی دارند. این روش‌ها می‌توانند با انواع مختلف داده‌ها و مسائل یادگیری تطبیق پیدا کنند و بهبود عملکرد قابل توجهی را در مسائل مختلف به همراه داشته باشند.

مقابله با مشکلات کلاس‌بندی نامتعادل

در مسائل دسته‌بندی با کلاس‌های نامتعادل، روش‌های ensemble می‌توانند با تخصیص وزن‌های مختلف به کلاس‌های مختلف یا با استفاده از تکنیک‌های خاصی مانند undersampling و oversampling، عملکرد بهتری را نشان دهند و تعادل بهتری بین کلاس‌ها برقرار کنند.

روش‌های AdaBoost و RandomForest به عنوان نمونه‌های برجسته‌ای از روش‌های ensemble نشان‌دهنده قدرت و مزایای این رویکردها هستند. AdaBoost با تقویت مدل‌های ضعیفتر و تاکید بر داده‌هایی که به درستی طبقه‌بندی نشده‌اند و RandomForest با ایجاد مجموعه‌ای از درختان تصمیم مستقل و ترکیب نتایج آن‌ها، هر دو بهبود دقت و پایداری مدل نهایی را تضمین می‌کنند. این روش‌ها بهویژه در مسائل پیچیده و داده‌های نویزی یا نامتعادل بسیار مؤثر هستند.

برای این بخش از الگوریتم جنگل تصادفی استفاده می‌کنیم و نتایج را با مدل بخش قبل مقایسه می‌نماییم.

کد این قسمت به صورت زیر است و مقدار max-depth را برابر 30 در نظر می‌گیریم.

دقت حدود $95\%-94$ درصد بدست می‌آید و می‌بینیم که عملکرد نهایی مدل جنگل تصادفی نسبت به درخت تصمیم‌گیری، بهبود یافته است.

The score of random forest model on test data is 0.9469

Random forest model

```
✓ [121] from sklearn.ensemble import RandomForestClassifier
      rf = RandomForestClassifier(max_depth=30)
      rf.fit(x_train_scaled, y_train)

→ + RandomForestClassifier
    RandomForestClassifier(max_depth=30)

✓ ➜ test_score = rf.score(x_test_scaled, y_test)
      print(f'The score of random forest model on test data is {test_score:.4f}', end='\n\n')

      print('Classification report for the random forest model is')
      hattt = rf.predict(x_test_scaled)
      cr = classification_report(y_test, hattt)
      print(cr)

→ The score of random forest model on test data is 0.9469

      Classification report for the random forest model is
      precision    recall   f1-score   support
      1          0.96     0.92     0.94     31753
      2          0.94     0.97     0.95     42625
      3          0.95     0.96     0.95     5328
      4          0.91     0.86     0.88     408
      5          0.96     0.72     0.82     1382
      6          0.92     0.90     0.91     2633
      7          0.97     0.94     0.96     3023

      accuracy                  0.95     87152
      macro avg       0.94     0.90     0.92     87152
      weighted avg    0.95     0.95     0.95     87152
```

سوال چهار

دیتاست بیماری قلبی را در نظر بگیرید. داده‌ها را به دو بخش آموزش و آزمون تقسیم کرده و ضمن انجام پیش‌پردازش‌هایی که روی آن لازم می‌دانید و با فرض گاوی بودن داده‌ها، از الگوریتم طبقه‌بندی Bayes استفاده کنید و نتایج را در قالب ماتریس درهم‌ریختگی و [classification_report](#) تحلیل کنید. تفاوت میان دو حالت Macro و Micro را در کتابخانه سایکیتلرن شرح دهید.

درنهایت، پنج داده را به صورت تصادفی از مجموعه آزمون انتخاب کنید و خروجی واقعی را با خروجی پیش‌بینی شده مقایسه کنید.

در ابتدا دیتاست بیماری قلبی را از Kaggle دانلود می‌کنیم.

درباره مجموعه داده‌ها می‌توان گفت: این مجموعه داده از سال ۱۹۸۸ تاریخ دارد و شامل چهار پایگاه داده است: کلیولند، مجارستان، سوئیس و لانگ بیچ. این مجموعه شامل ۷۶ ویژگی است که شامل ویژگی پیش‌بینی شده است. اما تمام آزمایشات منتشر شده به استفاده از یک زیرمجموعه از ۱۴ ویژگی اشاره دارند. هدف این مجموعه داده تشخیص بیماری قلبی در بیمار است. فیلد "target" به وجود بیماری قلبی در بیمار اشاره دارد. این فیلد دارای مقادیر صحیح است: ۰ = بدون بیماری و ۱ = بیماری

۱۴ ویژگی استفاده شده در مجموعه داده عبارتند از: سن، جنسیت، نوع درد سینه، فشار خون استراحتی، کلسترول سرمی، قند خون، نتایج الکتروکاردیوگرافی استراحتی، بیشینه ضربان قلب، آنژین تحریکی، زیرفشار، شب قله اس‌تی تمرين، تعداد عروق اصلی، تال ($0^+ =$ نرمال؛ ۱ = نقص ثابت؛ ۲ = نقص قابل برگشت) و ویژگی پیش‌بینی شده (بیماری قلبی).

در نمایی کلی دیتاست به صورت زیر است:

| | age | sex | cp | trestbps | chol | fbs | restecg | thalach | exang | oldpeak | slope | ca | thal | target |
|------|-----|-----|-----|----------|------|-----|---------|---------|-------|---------|-------|-----|------|--------|
| 0 | 52 | 1 | 0 | 125 | 212 | 0 | 1 | 168 | 0 | 1.0 | 2 | 2 | 3 | 0 |
| 1 | 53 | 1 | 0 | 140 | 203 | 1 | 0 | 155 | 1 | 3.1 | 0 | 0 | 3 | 0 |
| 2 | 70 | 1 | 0 | 145 | 174 | 0 | 1 | 125 | 1 | 2.6 | 0 | 0 | 3 | 0 |
| 3 | 61 | 1 | 0 | 148 | 203 | 0 | 1 | 161 | 0 | 0.0 | 2 | 1 | 3 | 0 |
| 4 | 62 | 0 | 0 | 138 | 294 | 1 | 1 | 106 | 0 | 1.9 | 1 | 3 | 2 | 0 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 1020 | 59 | 1 | 1 | 140 | 221 | 0 | 1 | 164 | 1 | 0.0 | 2 | 0 | 2 | 1 |
| 1021 | 60 | 1 | 0 | 125 | 258 | 0 | 0 | 141 | 1 | 2.8 | 1 | 1 | 3 | 0 |
| 1022 | 47 | 1 | 0 | 110 | 275 | 0 | 0 | 118 | 1 | 1.0 | 1 | 1 | 2 | 0 |
| 1023 | 50 | 0 | 0 | 110 | 254 | 0 | 0 | 159 | 0 | 0.0 | 2 | 0 | 2 | 1 |
| 1024 | 54 | 1 | 0 | 120 | 188 | 0 | 1 | 113 | 0 | 1.4 | 1 | 1 | 3 | 0 |

1025 rows × 14 columns

این دیتاست شامل ۱۰۲۵ نمونه و ۱۳ فیچر است و ستون target هم لیبل آن است. پس لیبل به عنوان ۷ و ۱۳ فیچر به عنوان X در نظر گرفته می شوند.

در گام بعد چک می کنیم بینیم تعداد داده های مربوط به هر کلاس چند تاست :

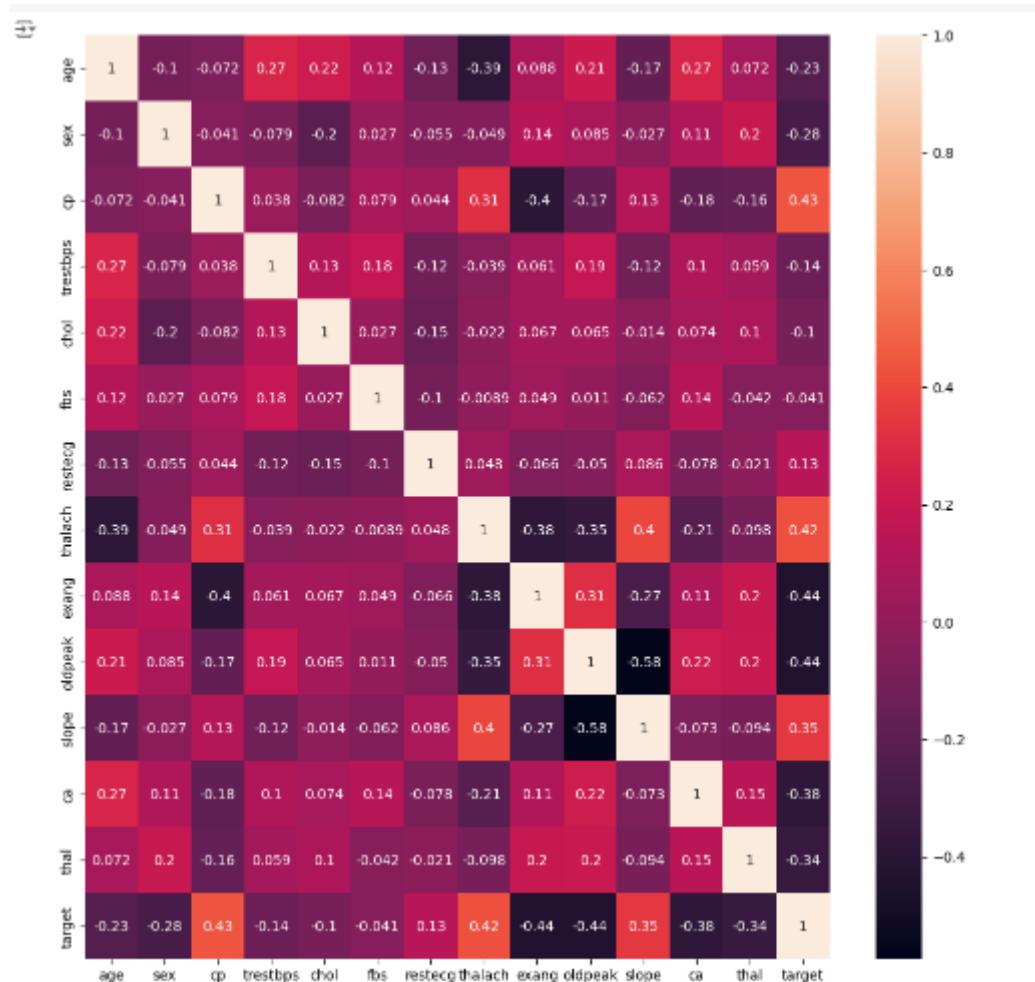
Number of data for each class:

```
[ ] number_of_ones = y.sum()
print(f'Number of class one data: {number_of_ones}')
print(f'Number of class zeros data: {len(y) - number_of_ones}')

→ Number of class one data: 526
Number of class zeros data: 499
```

که به نسبت خوبی تقسیم شده اند.

حالا هم بستگی بین این فیچر هارا بررسی می کنیم تا بینیم چه ارتباطی با هم دارند، درواقع ما لازم نیست که از هر ۱۳ فیچر برای شبکه استفاده کنیم و می تونیم از مناسب ترین فیچر ها استفاده کنیم.



به صورت بالا همبستگی فیچر ها مشخص میشود. در واقع در زیر واضح تر است :

```
r Correlation of each feature with the label:  
oldpeak    0.438441  
exang      0.438029  
cp          0.434854  
thalach    0.422895  
ca          0.382085  
slope       0.345512  
thal        0.337838  
sex         0.279501  
age         0.229324  
trestbps   0.138772  
restecg    0.134468  
chol        0.099966  
fbs         0.041164
```

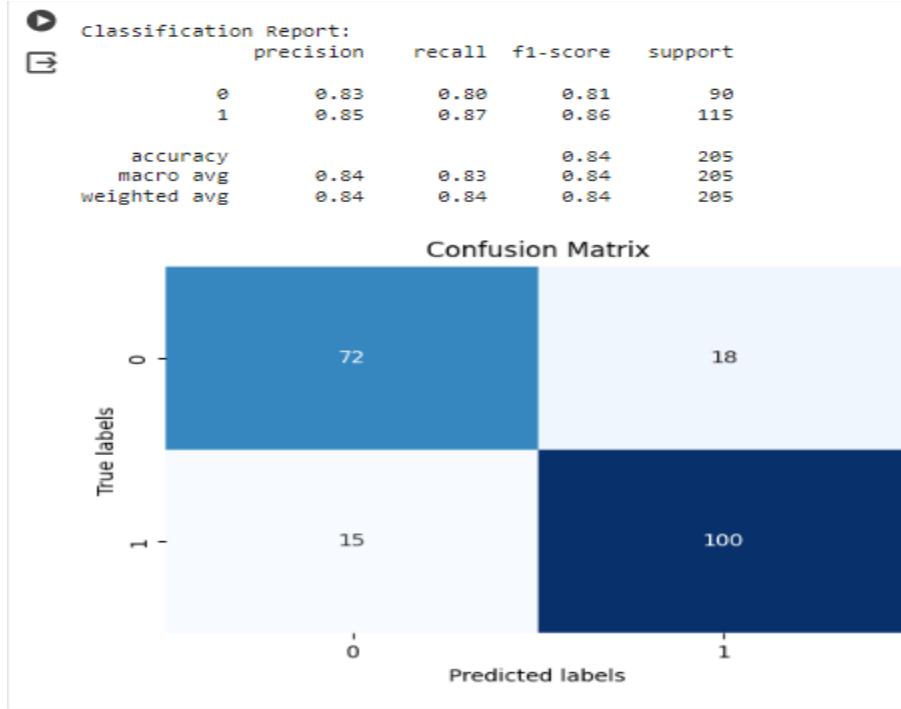
حالا در چند حالت بررسی می کنیم.

بک بار از تمام فیچر ها استفاده می کنیم :

اکنون از الگوریتم Bayes استفاده می کنیم و کد آن به صورت زیر است :

```
# Separate features and target  
X = df.drop(columns=['target'])  
y = df['target']  
  
# Splitting the dataset into the Training set and Test set  
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=24)  
  
# Standardize features by removing the mean and scaling to unit variance  
scaler = StandardScaler()  
X_train = scaler.fit_transform(X_train)  
X_test = scaler.transform(X_test)  
  
# Create and train Gaussian Naive Bayes classifier  
classifier = GaussianNB()  
classifier.fit(X_train, y_train)  
  
# Predicting the Test set results  
y_pred = classifier.predict(X_test)  
  
# Confusion Matrix  
conf_matrix = confusion_matrix(y_test, y_pred)  
print("Confusion Matrix:")  
print(conf_matrix)  
  
# Classification Report  
class_report = classification_report(y_test, y_pred, digits=2, output_dict=False, zero_division='warn')  
print("\nClassification Report:")  
print(class_report)  
  
# Plot Confusion Matrix  
plt.figure(figsize=(6, 4))  
sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues", cbar=False)  
plt.xlabel('Predicted labels')  
plt.ylabel('True labels')  
plt.title('Confusion Matrix')  
plt.show()
```

و نتایج دو فرم خواسته شده به صورت زیر است :



✓ نتایج به دست آمده را به صورت زیر تحلیل می کنیم :

ماتریس درهم ریختنگی :

- مثبت واقعی (TP) : ۷۲ (کلاس ۰ واقعی که به درستی به عنوان کلاس ۰ پیش‌بینی شده است)
- مثبت غلط (FP) : ۱۸ (کلاس ۰ واقعی که به اشتباه به عنوان کلاس ۱ پیش‌بینی شده است)
- منفی غلط (FN) : ۱۵ (کلاس ۱ واقعی که به اشتباه به عنوان کلاس ۰ پیش‌بینی شده است)
- منفی واقعی (TN) : ۱۰۰ (کلاس ۱ واقعی که به درستی به عنوان کلاس ۱ پیش‌بینی شده است)

classification_report

شامل چند بخش است که در ادامه به توضیح آن می پردازیم :

- دقت (Precision): نسبت مثبت‌های صحیحی که به درستی تشخیص داده شدند به کل مثبت‌هایی که پیش‌بینی شده‌اند است. برای کلاس ۰، دقت ۸۳٪ است که به این معنی است که ۸۳٪ از مثبت‌هایی که به عنوان کلاس ۰ پیش‌بینی شدند، واقعاً کلاس ۰ بودند. برای کلاس ۱، دقت ۸۵٪ است که نشان می‌دهد ۸۵٪ از مثبت‌هایی که به عنوان کلاس ۱ پیش‌بینی شدند، واقعاً کلاس ۱ بودند.

- بازخوانی (Recall): بازخوانی (همچنین به عنوان حساسیت شناخته می‌شود) نسبت مثبت‌های صحیحی که به درستی تشخیص داده شدند به کل مثبت‌ها در کلاس واقعی است. برای کلاس ۰، بازخوانی ۰,۸۰ است که به این معنی است که ۰٪ از کل مثبت‌های واقعی کلاس ۰ به درستی به عنوان کلاس ۰ تشخیص داده شدند. برای کلاس ۱، بازخوانی ۰,۸۷ است که نشان می‌دهد ۷٪ از کل مثبت‌های واقعی کلاس ۱ به درستی به عنوان کلاس ۱ تشخیص داده شدند.

- امتیاز F1: امتیاز F1 میانگین هندسی دقت و بازخوانی است. این امتیاز تعادلی بین دقت و بازخوانی ارائه می‌دهد. برای کلاس ۰، امتیاز ۰,۸۱ F1 است و برای کلاس ۱، ۰,۸۶ است.

$$\text{کلاس ۰: } ۰,۸۱ = \frac{(دقت * \text{بازخوانی}) + (دقت * \text{بازخوانی})}{(دقت + \text{بازخوانی})}$$

$$\text{کلاس ۱: } ۰,۸۶ = \frac{(دقت * \text{بازخوانی}) + (دقت * \text{بازخوانی})}{(دقت + \text{بازخوانی})}$$

- پشتیبانی (Support): پشتیبانی تعداد وقوع واقعی کلاس در مجموعه داده مشخص شده است. برای کلاس ۰، پشتیبانی ۹۰ و برای کلاس ۱، ۱۱۵ است.

- دقت کل (Accuracy): دقت نسبت مثبت‌های صحیحی که به درستی پیش‌بینی شده‌اند به کل نمونه‌ها است. دقت کل ۰,۸۴ است که نشان می‌دهد مدل ۴٪ از نمونه‌ها را به درستی پیش‌بینی کرده است.

- میانگین ماکرو (Macro avg): این میانگین را محاسبه می‌کند که میانگین غیروزن‌دار دقت و بازخوانی برای همه کلاس‌ها است. برای دقت، بازخوانی و امتیاز F1، این مقدار ۰,۸۴ است.

- میانگین وزنی (Weighted avg): این میانگین را محاسبه می‌کند که میانگین وزن‌دار دقت، بازخوانی و امتیاز F1 برای همه کلاس‌ها است، با وزن‌دهی توسط پشتیبانی. برای دقت، بازخوانی و امتیاز F1، این مقدار همچنین ۰,۸۴ است.

پس، با دقت، بازخوانی و امتیاز F1 مناسب برای هر دو کلاس، نشان می‌دهد که مدل به خوبی عمل کرده است.
در نتیجه می‌توان گفت که :

- ماتریس در هم ریختگی تفکیک دقیقی از تعداد مثبت‌های واقعی، منفی‌های واقعی، مثبت‌های غلط و منفی‌های غلط را ارائه می‌دهد.

- گزارش دسته‌بندی (classification_report) دقت، بازخوانی، امتیاز F1، پشتیبانی و همچنین میانگین ماکرو و وزن‌دهی شده برای همه کلاس‌ها را ارائه می‌دهد.

- دقت، بازخوانی و امتیاز F1 از گزارش دسته‌بندی می‌تواند با استفاده از مقادیر ماتریس ابعام محاسبه شود.
- دقت از گزارش دسته‌بندی همان مقدار محاسبه شده از ماتریس درهم ریختگی است.

به طور کلی، هر دو ماتریس درهم ریختگی و گزارش دسته‌بندی به مشاهده‌ی مشابهی درباره‌ی عملکرد دسته‌بندی می‌انجامند و تأیید می‌کنند که مدل با دقت، بازخوانی و امتیاز F1 خوب برای هر دو کلاس عمل کرده است.

دقت این مدل هم به صورت زیر است : حدود ۸۴ درصد

```
▶ from sklearn.metrics import accuracy_score
accuracy_score(classifier.predict(X_test), y_test)
0.8390243902439024
```

این بار تلاش می‌کنیم تا فیچر‌های مناسب را استخراج کنیم :

با استفاده از کد زیر، فیچر‌هایی که لیبل بیشتر از ۰,۲۵ هم بستگی دارند را نگه می‌داریم.

keep features with correlation > 0.25

```
▶ # Calculate the correlation of each feature with the label
correlations1 = df.corr()['target'][:-1] # Exclude the correlation of Y with itself

# Get the absolute value of correlations
abs_correlations1 = correlations1.abs()

# Sort the correlations by absolute value in descending order
sorted_correlations1 = abs_correlations1.sort_values(ascending=False)

# Print the sorted correlations
print("Correlation of each feature with the label:")
print(sorted_correlations1)

# Define a threshold for relevance, e.g., keep features with correlation > 0.25
threshold = 0.25
relevant_features1 = sorted_correlations1[sorted_correlations1 > threshold].index.tolist()

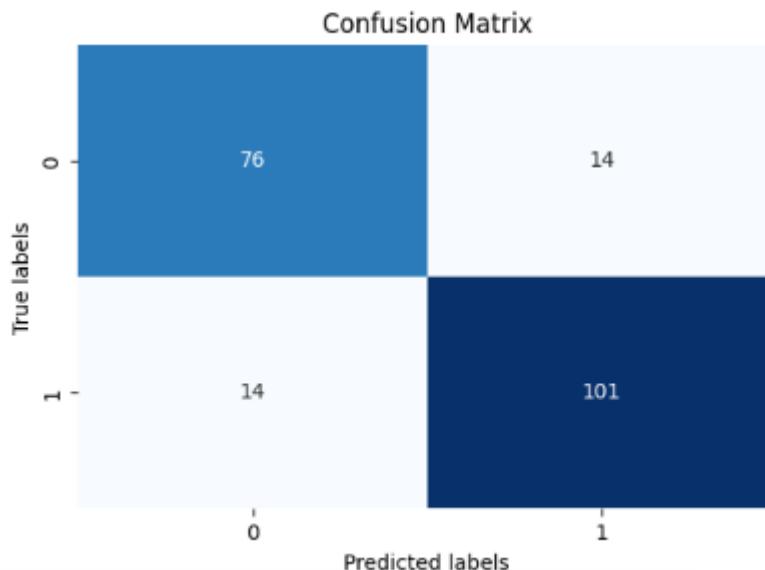
# Print the most relevant features
print("Most relevant features based on correlation:")
print(relevant_features1)
```

۸ فیچر از ۱۳ فیچر باقی می‌ماند :

```
Most relevant features based on correlation:
['oldpeak', 'exang', 'cp', 'thalach', 'ca', 'slope', 'thal', 'sex']
```

حالا این ۸ فیچر را به عنوان X به مدل می دهیم و خروجی را بررسی می کنیم :

```
Confusion Matrix:  
[[ 76  14]  
 [ 14 101]]  
  
Classification Report:  
 precision    recall   f1-score   support  
  
      0       0.84     0.84     0.84      90  
      1       0.88     0.88     0.88     115  
  
accuracy                           0.86      205  
macro avg       0.86     0.86     0.86      205  
weighted avg    0.86     0.86     0.86      205
```



دقت حدود ۸۶ درصد می شود که بهتر از حالت قبل که ۸۴ درصد بود، شد.

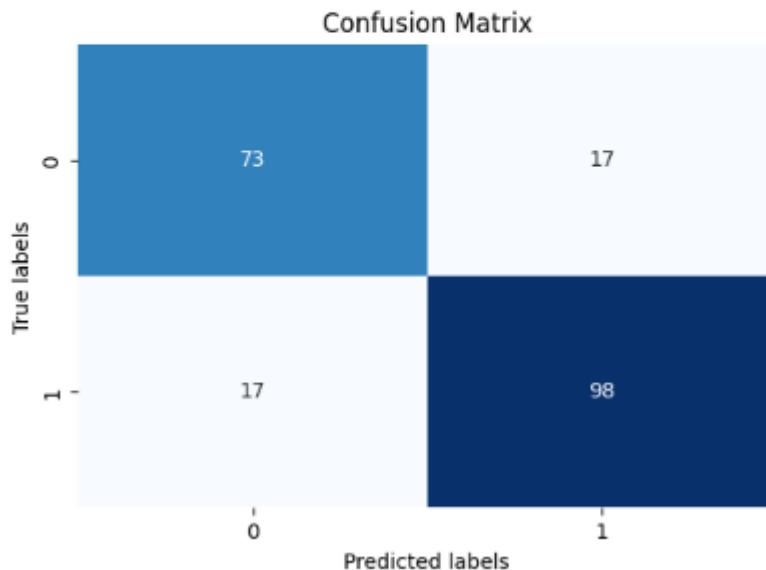
حالا مقدار `threshold` را افزایش می دهیم تا ببینیم که اگر از فیچر هایی که هم بستگی بیشتری دارند استفاده کنیم، خروجی چه تغییری می کند! مقدار هم بستگی را 0.3 در نظر می گیریم درواقع فیچر های بیشتر از 0.3 هم بستگی را نگه می داریم :

این ۷ فیچر به عنوان X در نظر گرفته می شود :

```
Most relevant features based on correlation:  
['oldpeak', 'exang', 'cp', 'thalach', 'ca', 'slope', 'thal']
```

حال مدل را در این حالت بررسی می کنیم و داریم :

| Classification Report: | | | | | |
|------------------------|-----------|--------|----------|---------|--|
| | precision | recall | f1-score | support | |
| 0 | 0.81 | 0.81 | 0.81 | 90 | |
| 1 | 0.85 | 0.85 | 0.85 | 115 | |
| accuracy | | | 0.83 | 205 | |
| macro avg | 0.83 | 0.83 | 0.83 | 205 | |
| weighted avg | 0.83 | 0.83 | 0.83 | 205 | |



می بینیم که دقت ۸۳ درصد شده و کاهش یافته است. اگر مقدار threshold را بیشتر از ۰،۳ در نظر بگیریم، دقت زیر ۸۰ درصد می شود، پس نتیجه می گیریم که همان ۸ فیچر بهترین انتخاب برای مدل است. و از همان مدل برای ادامه استفاده می شود.

✓ تفاوت حالت های میانگین گیری:

در کتابخانه scikit-learn، هنگام محاسبه معیارهایی مانند دقت، بازخوانی و امتیاز F1 برای مسائل دسته‌بندی چندکلاسه، دو حالت میانگین‌گیری وجود دارد: ماکرو و میکرو.

در ادامه توضیحی درباره تفاوت این دو حالت آمده است:

- میانگین ماکرو یا Macro Average

در میانگین ماکرو، معیارها برای هر کلاس به طور مستقل محاسبه شده و سپس میانگین آنها برای تمام کلاس‌ها گرفته می‌شود؛ این روش تمام کلاس‌ها را بدون در نظر گرفتن تراز و تعداد آنها در مجموعه داده، به صورت یکسان در نظر می‌گیرد.

میانگین ماکرو زمانی که می‌خواهیم عملکرد کلی دسته‌بند را برای همه کلاس‌ها بدون در نظر گرفتن ناهمواری‌های کلاسی ارزیابی کنیم، مفید است. این میانگین به عنوان میانگین دقت، بازخوانی یا امتیاز F1 برای هر کلاس بدون در نظر گرفتن ناهمواری کلاسی محاسبه می‌شود.

- میانگین میکرو یا Micro Average :

در میانگین میکرو، مجموع مثبت‌های واقعی، مثبت‌های غلط و منفی‌های غلط برای همه کلاس‌ها محاسبه شده و سپس دقت، بازخوانی و امتیاز F1 بر اساس این تعدادهای جهانی محاسبه می‌شوند.

این روش با در نظر گرفتن ناهمواری کلاسی عمل می‌کند زیرا هر نمونه را به طور مساوی وزن می‌دهد (بدون در نظر گرفتن کلاس). میانگین میکرو زمانی که می‌خواهیم عملکرد کلی دسته‌بند را در نظر بگیریم در حالی که ناهمواری‌های کلاسی را در نظر می‌گیریم، مفید است. این میانگین به عنوان دقت، بازخوانی یا امتیاز F1 برای تمام کلاس‌ها ترکیب شده، با در نظر گرفتن تعداد جهانی محاسبه می‌شود.

پس در کل میانگین ماکرو تمام کلاس‌ها را به یکسان در نظر می‌گیرد و زمانی که هر کلاس اهمیت یکسانی دارد، مناسب است و میانگین میکرو با در نظر گرفتن ناهمواری کلاسی عمل می‌کند و زمانی که مجموعه داده ناهموار است و می‌خواهیم وزن بیشتری به کلاس‌های با پشتیبانی بیشتر بدھیم، مناسب است.

در 'classification_report' scikit-learn، می‌توانیم با استفاده از پارامتر 'average' حالت میانگین‌گیری را انتخاب کنیم. تنظیم 'average='macro'' معیارها را با استفاده از میانگین ماکرو محاسبه می‌کند، در حالی که 'average='micro'' آنها را با استفاده از میانگین میکرو محاسبه می‌کند. اگر 'average=None' باشد، معیارها برای هر کلاس به صورت جداگانه بازگردانده می‌شوند.

✓ در گام آخر، ۵ داده را به صورت تصادفی از مجموعه آزمون انتخاب می‌کنیم و خروجی واقعی را با خروجی پیش‌بینی شده مقایسه می‌کنیم :

با استفاده از کد زیر ۵ داده‌ی رندوم از داده‌های تست انتخاب می‌کنیم :

```

❷ import random
# Set the random seed
random.seed(24)

# Randomly select 5 indices from the test set
random_indices = random.sample(range(len(x_test_new)), 5)

print("Randomly selected 5 data points:")
for idx in random_indices:
    print(f"Index: {idx}")
    print(f"Actual Output: {y_test_new.iloc[idx]}")
    print(f"Predicted Output: {y_pred1[idx]}")
    print()

```

خروجی به صورت زیر است :

```

❷ Randomly selected 5 data points:
Index: 182
Actual Output: 1
Predicted Output: 1

Index: 98
Actual Output: 1
Predicted Output: 1

Index: 149
Actual Output: 1
Predicted Output: 1

Index: 46
Actual Output: 0
Predicted Output: 0

Index: 55
Actual Output: 1
Predicted Output: 1

```

میبینیم که هر ۵ داده به خوبی طبقه بندی شده اند در حالیکه دقت مدل ۸۶ درصد است. پس تعداد بیشتری داده رندوم انتخاب می کنیم تا بهتر درک کنیم :

```
Randomly selected 5 data points:  
Index: 182  
Actual Output: 1  
Predicted Output: 1  
  
Index: 98  
Actual Output: 1  
Predicted Output: 1  
  
Index: 149  
Actual Output: 1  
Predicted Output: 1  
  
Index: 46  
Actual Output: 0  
Predicted Output: 0  
  
Index: 55  
Actual Output: 1  
Predicted Output: 1  
  
Index: 42  
Actual Output: 0  
Predicted Output: 0  
  
Index: 49  
Actual Output: 1  
Predicted Output: 1  
  
Index: 43  
Actual Output: 0  
Predicted Output: 0  
  
Index: 171  
Actual Output: 1  
Predicted Output: 0  
  
Index: 174  
Actual Output: 1  
Predicted Output: 1
```

میبینیم که داده رندوم با ایندکس ۱۷۱ به درستی طبقه بندی نشده است چون در کل دقت مدل ۸۶ درصد است و قرار نیست تمام داده ها درست طبقه بندی شوند.