گزارش کار

شناسایی آماری الگو

تمرین ۱

آرمین خیاطی

9931153

سبحان نامی

9831767

1398/9/17

هدف

در این تمرین قصد داشتیم که در بخش اول رگرسیون خطی را با استفاده از یکی از دو روش Closed Form Solution و Gradient Descent محاسبه کنیم و سپس خطای مدل ارایه شده را پس از آموزش دادن مدل، روی داده های Train و Test با استفاده از روش Mean Square Error محاسبه کنیم. در بخش دوم نیز با طراحی یک مدل مبتنی بر Logistic Regression بر روی داده های دیتاست Iris میخواهیم به پیش بینی و دسته بندی این داده ها بپردازیم.

بخش اول

Closed Form Linear Regression

در طراحی و پیاده سازی الگوریتم Linear Regression به روش Closed Form اگر ورودی های X ما به سایز n باشند و یک متغیر هدف نیز داشته باشیم آنوقت میتوان معادله یک مدل رگرسیون خطی را به صورت زیر نمایش داد.

$$Y_i = W_0 + \ W_1 X_{i1} + \cdots$$

که Y_i بیانگر متغیر وابسته ، X_{i1} نشانگر متغیر های مستقل و W_1 و W_0 وزن و یا پارامتر های رگرسیون هستند. با توجه به معادله بالا ما میتوانیم پارامتر های رگرسیون را با استفاده از فرمول زیر محاسبه کنیم.

$$W = \begin{bmatrix} W_0 \\ W_1 \\ . \\ . \\ W_n \end{bmatrix} = (X'X)^{-1}X'Y$$

در فرمول بالا X' ترانهاده ماتریس X و $(X'X)^{-1}$ وارون نتیجه $X'X$ است. بعد از بدست آوردن پارامتر ها برای انجام پیش بینی بر روی ورودی های جدید از فرمول زیر استفاده میشود.

$$W'X'$$

پیاده سازی

در این تمرین از زبان پایتون و کتاب خانه های Numpy، Pandas و Matplotlib برای خواندن فایل ها، انجام پیش پردازش ها و محاسبات و رسم نمودار ها استفاده شده است. کد این بخش از تمرین در فایل Clossed_Form_Regression.ipynb موجود است. در ادامه به تشریح توابع نوشته شده که مربوط به بخش اصلی الگوریتم هستند میپردازیم.

load_data

```
def load_data():
  train_data = pd.read_csv('/content/Data-Train.csv')
  test_data = pd.read_csv('/content/Data-Test.csv',)
  print(len(train_data), " Train Data Loaded")
  print(len(test_data), " Test Data Loaded")
  return train_data, test_data
```

این تابع دو فایل CSV داده های Train و Test را بوسیله کتاب خانه Pandas درون برنامه وارد و بر میگرداند.

Normalization

```
def normalization(data):
  data[:,0] = (data[:,0]) / data[:,0].max()
  return data
```

این تابع داده ها را بین صفر و یک نرمال میکند.

prepare_data

```python
def prepare_data():
  train , test = load_data()
  train = train.to_numpy(dtype='float64')
  test = test.to_numpy(dtype='float64')
  train = normalization(train)
  test = normalization(test)
  x = train[: , 0:-1]
  y = train[: , -1]
  x_ = test[: , 0:-1]
  y_ = test[: , -1]
  return x,y,x_,y_
```

این تابع متغیر های مستقل و غیر مستقل را از برای داده های Train و Test جدا کرده و بر میگرداند.

learn_model

```python
def learn_model(x, y):
  xxt = np.dot(x.T, x)
  xxt_inv = np.linalg.inv(xxt)
  xty = np.dot(x.T, y)
  theta = np.dot(xxt_inv, xty)
  print("Theta = ", theta)
  return theta
```

این تابع فرمول توضیح داده شده در بالا برای رگرسیون خطی و یادگیری پارامتر، پیاده سازی میکند.

Predict

```python
def predict(x, model):
  return np.dot(model.T, x.T).flatten()
```

این تابع نیز با استفاده از پارامتر های بدست آمده در مرحله یادگیری، برای ورودی های که به تابع داده میشود پیشبینی میکند. ورودی model همان پارامتر های بدست آمده در مرحله یادگیری و ورودی X همان داده هایی هستند که باید برای آن ها خروجی پیش بینی شود.

Mse

```
def mse(y_, y):
  diff = np.subtract(y_, y)
  ms = np.power(diff, 2, dtype='float64')
  return np.sum(ms) / len(y)
```

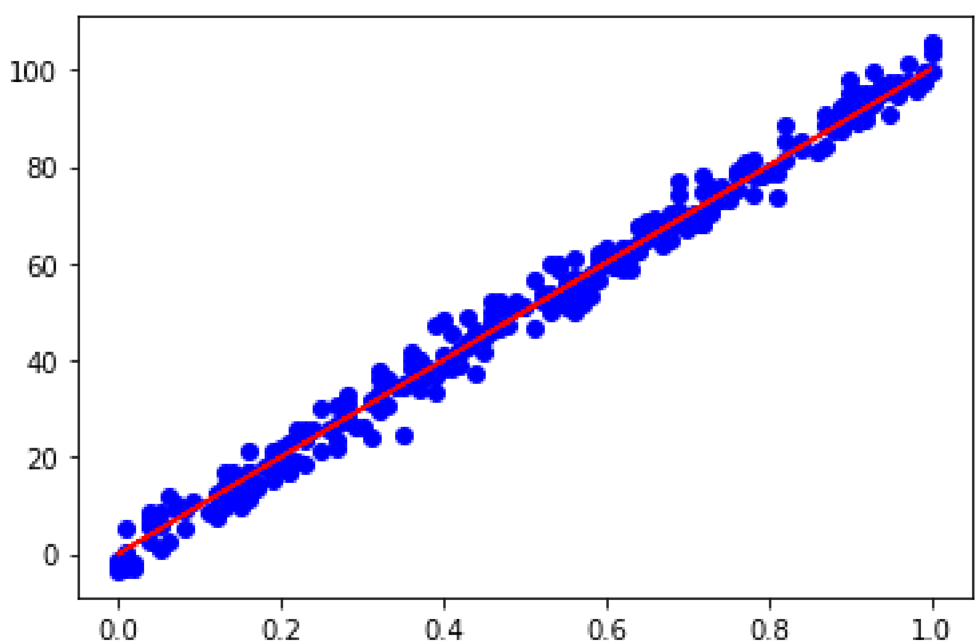
این تابع برای محاسبه Mean Square Error بر روی مقادیر پیش بینی شده و مقادیر واقعی استفاده میشود.

نتیجه

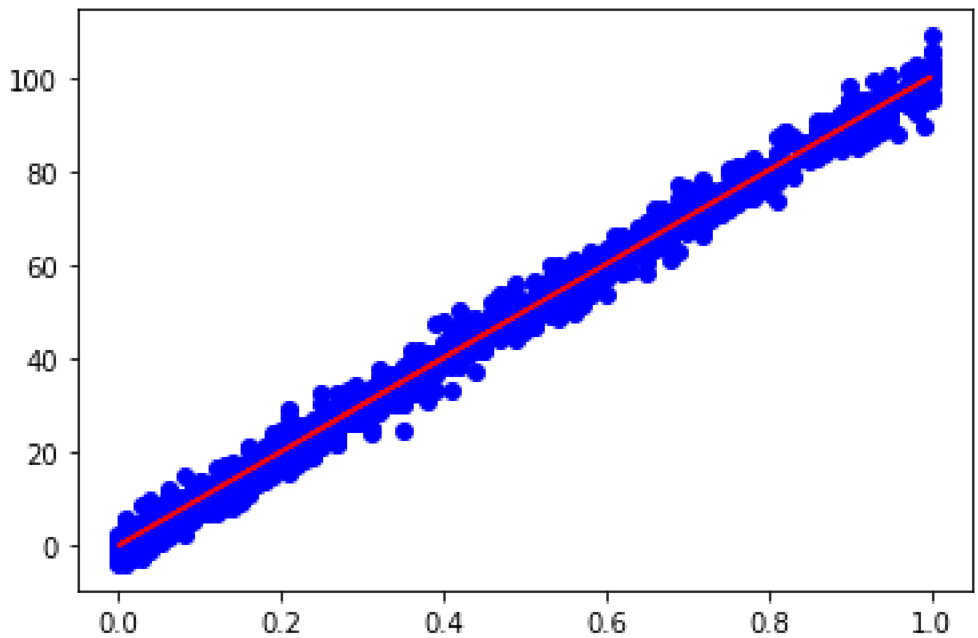
پس از اجرای مدل بر روی داده های Train نمودار ها و نتایج زیر حاصل گردید.

Test	Train	
9.332616851081493	8.339910252067114	**Mean Square Error**

نمودار زیر خط رگرسیون روی داده های Test می باشد.



این نمودار نیز خط رگرسیون روی داده های Train می باشد.



وزن یاد گرفته شده نیز مقدار زیر میباشد.

100.15481913

بخش دوم

Gradient Descent Linear Regression

سوالی که اینجا مطرح میشود این است که چرا از Gradient Descent استفاده کنیم وقتی فرمول Closed Form وجود دارد؟ این سوال دو جواب دارد.

1. برای بعضی مسائل رگرسیون غیر خطی، راه حل Closed Form وجود ندارد.
2. حتی برای رگرسیون خطی هم بعضی مواقع استفاده از Closed Form مناسب نیست زیرا ممکن است ورودی ها خیلی بزرگ باشند و یا ماتریس Sparse باشد. برای همین انجام محاسبات در این راه حل می تواند گران باشد.

الگوریتم Gradient Descent بطور کلی یک الگوریتم بهینه سازی تلقی میشود و هدف آن حداقل کردن یک تابع هزینه است. از لحاظ محاسباتی نیز ارزان است.

دو نمونه یا بهتر است بگوییم سه نمونه Gradient Descent داریم.

- Batch Gradient Descent : در این روش در هر epoch از کل دیتا ها بصورت یکجا برای محاسبه پارامتر ها استفاده میشود.
- Stochastic Gradient Descent : این روش برای داده های حجیم مناسب است که آوردن همه آن ها بصورت یکجا در رم و انجام محاسبات منطقی نیست. برای همین در هر epoch، محاسبات بر روی یک نمونه از داده انجام شده و پارامتر ها آپدیت میشوند.
- Mini Batch Gradient Descent : اگر رم کافی برای آوردن بخشی از دیتا ها در هر epoch در رم دارید میتوانید از این روش استفاده کنید.

کد نوشته شده در این بخش از هر سه روش قابل بهره گیری هست. در اینجا تابع هزینه ما تابع Mean Square Error هست و گرادیان ما مشتق این تابع بر حسب هر w در ماتریس W است که فرمول آن را در زیر میبینیم.

$$\text{Cost} = J(w) = \frac{1}{2m}\sum_{i=1}^{m}\left(h(w)^{(i)} - y^{(i)}\right)^2$$

$$\text{Gradient} = \frac{\partial J(w)}{\partial w_j} = \frac{1}{m}\sum_{i=1}^{m}\left(h(w)^{(i)} - y^{(i)}\right).X_j^{(i)}$$

برای آپدیت کردن هر وزن نیز از فرمول زیر استفاده میشود.

$$w_j = w_j - lr.\left(\frac{1}{m}\sum_{i=1}^{m}\left(h(w)^{(i)} - y^{(i)}\right).X_j^{(i)}\right)$$

متغیر lr همان نرخ یاد گیری یا Learning Rate می باشد که عددی بین صفر و یک انتخاب میکنیم. بعد از ساخت مدل و آموزش آن برای انجام پیش بینی روی داده های جدید از فرمول زیر استفاده میکنیم.

$$\hat{Y} = XW$$

پیاده سازی

در این تمرین از زبان پایتون و کتاب خانه های Numpy، Pandas و Matplotlib برای خواندن فایل ها، انجام پیش پردازش ها و محاسبات و رسم نمودار ها استفاده شده است. کد این بخش از تمرین در فایل Stochastic_GD_LR.ipynb موجود است. در ادامه به تشریح توابع نوشته شده که مربوط به بخش اصلی الگوریتم هستند میپردازیم.

load_data

```
def load_data():
  train_data = pd.read_csv('/content/Data-Train.csv')
  test_data = pd.read_csv('/content/Data-Test.csv',)
  print(len(train_data), " Train Data Loaded")
  print(len(test_data), " Test Data Loaded")
  return train_data, test_data
```

این تابع دو فایل CSV داده های Train و Test را بوسیله کتاب خانه Pandas درون برنامه وارد و بر میگرداند.

Normalization

```
def normalization(data):
  data[:,0] = (data[:,0]) / data[:,0].max()
  return data
```

این تابع داده ها را بین صفر و یک نرمال میکند.

prepare_data

```
def prepare_data():
  train , test = load_data()
  train = train.to_numpy(dtype='float64')
  test = test.to_numpy(dtype='float64')
  train = normalization(train)
  test = normalization(test)
  x = train[: , 0:-1]
  y = train[: , -1]
  x_ = test[: , 0:-1]
  y_ = test[: , -1]
  return x,y,x_,y_
```

این تابع متغیر های مستقل و غیر مستقل را از برای داده های Train و Test جدا کرده و بر میگرداند.

batching_data

```
def batching_data(x, y, batch_size=1):
  batches = []
  size= x.shape[0]
  indexes = list(range(size))
  np.random.shuffle(indexes)
  for i in range(0, size, batch_size):
    j = i + batch_size
    j = j if j < size else size
    batch = indexes[i:j]
    batches.append((x.take(batch, axis=0), y.take(batch)))
  return batches
```

در این تابع از کل دیتا ها بصورت shuffle شده دسته هایی از داده به سایز مقدار batch_size ساخته میشود.

Init_Weights

```
def init_weights(size):
  return np.random.rand(size)
```

این تابع یک وزن اولیه رندم برای ما تولید میکند.

Compute_loss

```
def compute_loss(x, y, y_, theta):
  error = y_ - y
  loss = (np.dot(x.T, error))/(len(x))
  return loss
```

این تابع نیز برای محاسبه Cost طبق همان فرمول ارائه شده استفاده می شود.

Mse

```
def mse(y_, y):
  diff = np.subtract(y_, y)
  ms = np.power(diff, 2, dtype='float64')
  return np.sum(ms) / len(y)
```

این تابع برای محاسبه Mean Square Error بر روی مقادیر پیش بینی شده و مقادیر واقعی استفاده میشود.

Predict

```
def predict(x, theta):
  return np.dot(x, theta)
```

این تابع برای پیش بینی داده های جدید بر اساس پارامتر بدست آمده انجام میشود.

Train

```
def train(x_test, y_test, epochs, batches, weights):
  for i in range(epochs):
    for batch in batches:
      x, y = batch
      y_ = predict(x, weights)
      loss= compute_loss(x, y, y_, weights)
      loss_log.append(loss)
      weights = weights - lr * loss
    y_ = predict(x_test, weights)
    _mse = mse(y_, y_test)
    mse_log.append(_mse)
    theta_log.append(weights)
    if i % 100 == 0:
        print(f'Epoch: {i} | MSE: {_mse}')
  return weights
```

در این تابع بر اساس فرمول های شرح داده شده، مدل آموزش داده میشود.

پارامتر ها

برای آموزش ما چند پارامتر را برای تنظیمات مدل تعیین کرده ایم.

- Batch_size : این مقدار برای تقسیم داده به همین مقدار، تعداد دسته استفاده میشود که ما برای Stochastic مقدار یک داده ایم.
- Epochs : این مقدار تعیین کننده تعداد تکرار و دفعات اجرای الگوریتم می باشد که ما 401 انتخاب کرده ایم.
- Lr : این همان نرخ یاد گیری است که ما 0.01 انتخاب کرده ایم.

نتیجه

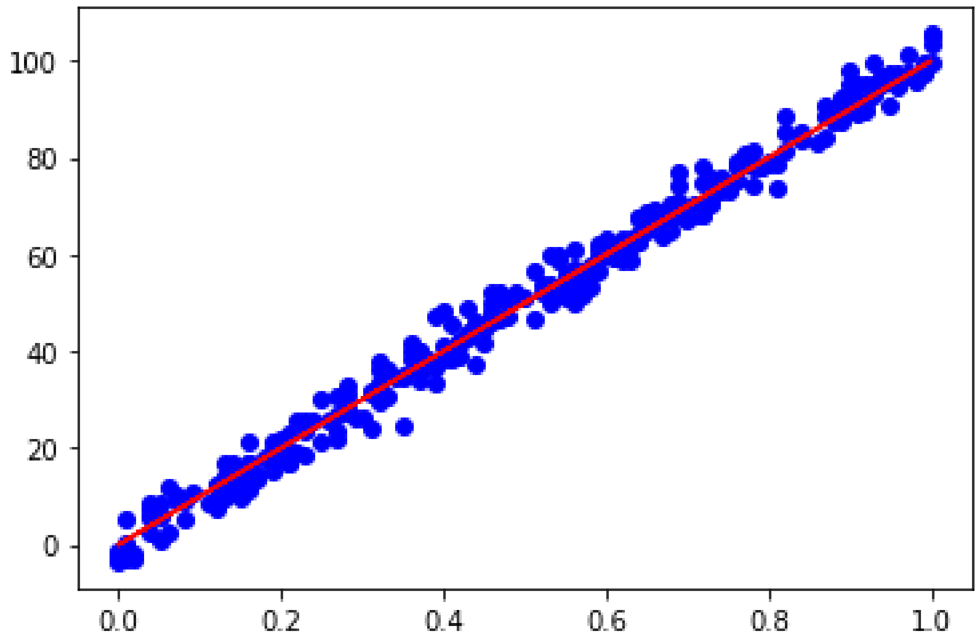
نتیجه ای که میتوان از این تمرین گرفت این است که مقدار نرخ یاد گیری به شدت میتواند در یادگیری مدل تاثیر گذار باشد و اگر مقدار بالایی در نظر گرفته شود باعث میشود مدل Underfit شود یعنی به خوبی داده ها را یاد نگیرد. نتایج بدست آمده با مقدار پارامتر های توضیح داده شده به شرح زیر است.

مقدار MSE در هر Epoch بر روی داده های تست و در نهایت مقدار آن روی داده های Train :

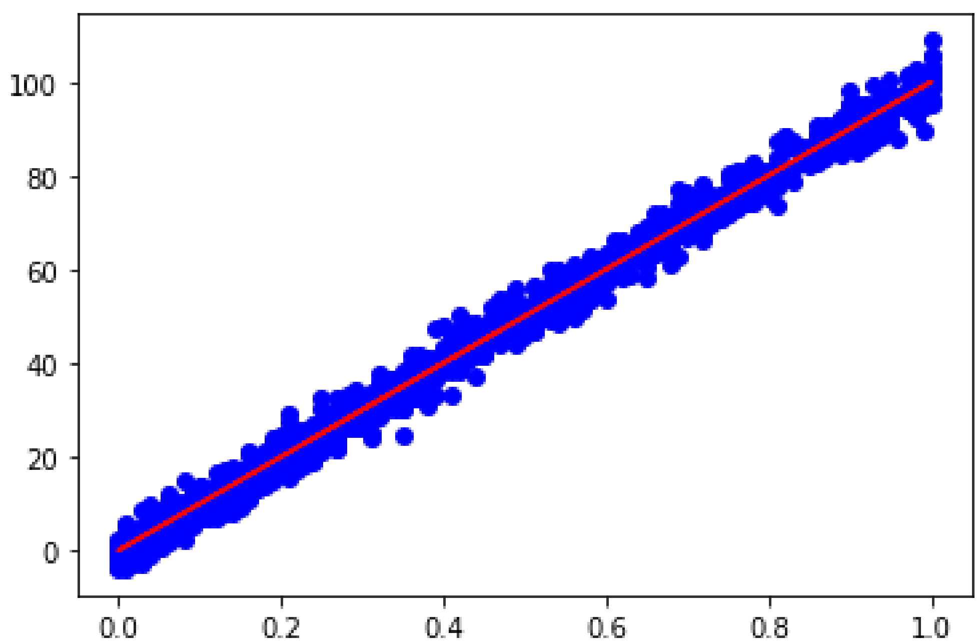
```
Epoch: 0   | MSE Test: 15.129582669515761
Epoch: 100 | MSE Test: 9.390972122080267
Epoch: 200 | MSE Test: 9.390972122080267
Epoch: 300 | MSE Test: 9.390972122080267
Epoch: 400 | MSE Test: 9.390972122080267

      MSE Train  8.339981046860567
```

نمودار خط رگرسیون روی داده های Test:



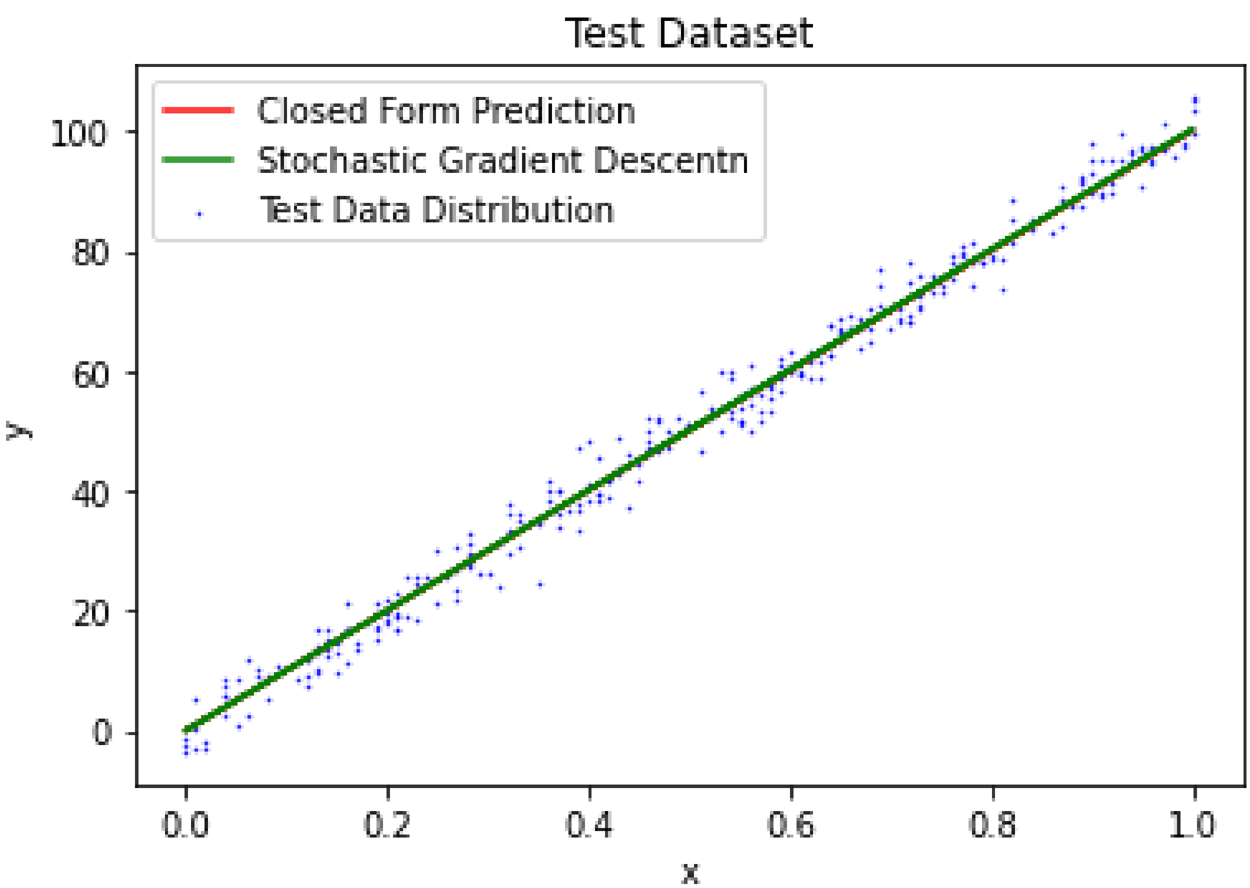
نمودار خط رگرسیون روی داده های Train:



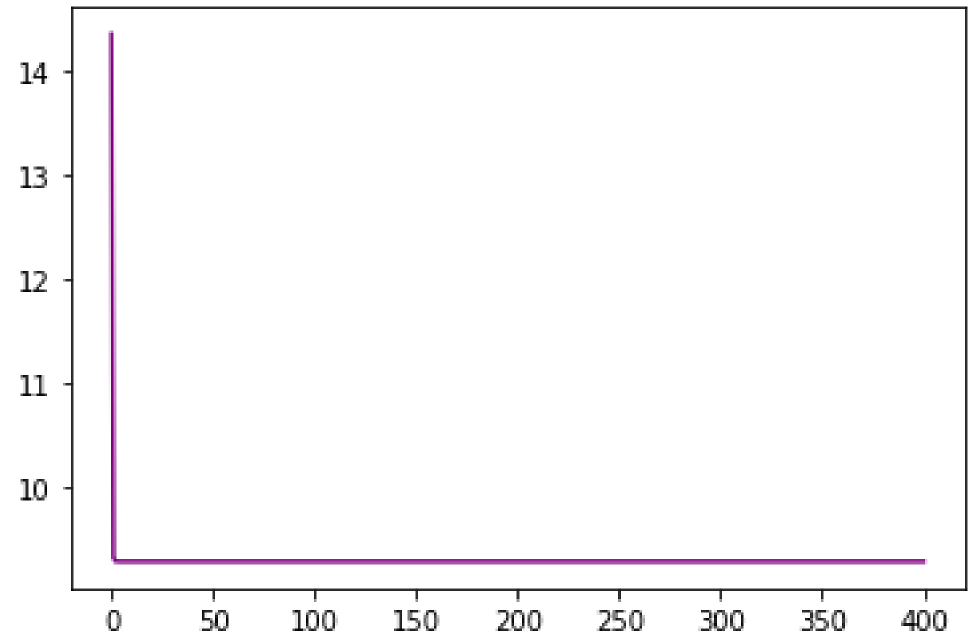
وزن یاد گرفته شده نیز مقدار زیر می باشد.

```
[0.18501241]
```

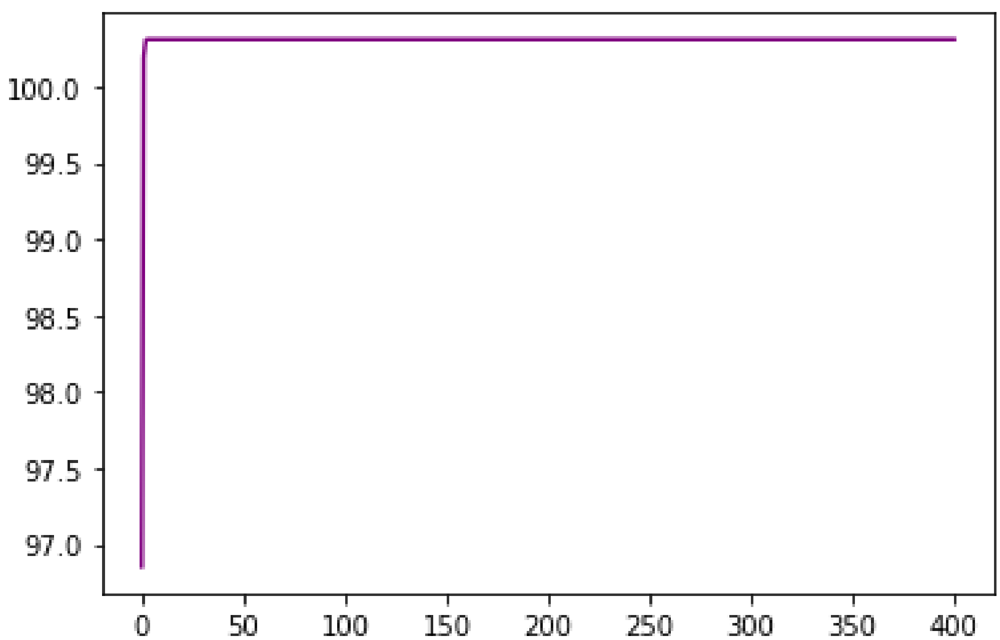
نمودار توزیع داده های تست و خط رگرسیون هر الگوریتم:



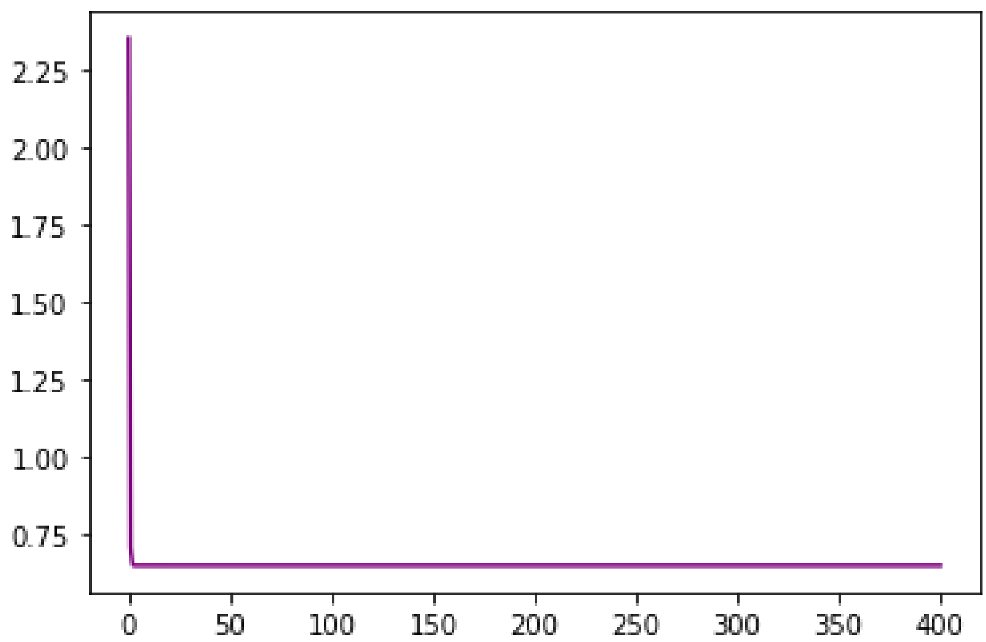
نمودار همگرایی MSE :



نمودار همگرایی وزن :



نمودار هم گرایی تابع هزینه:



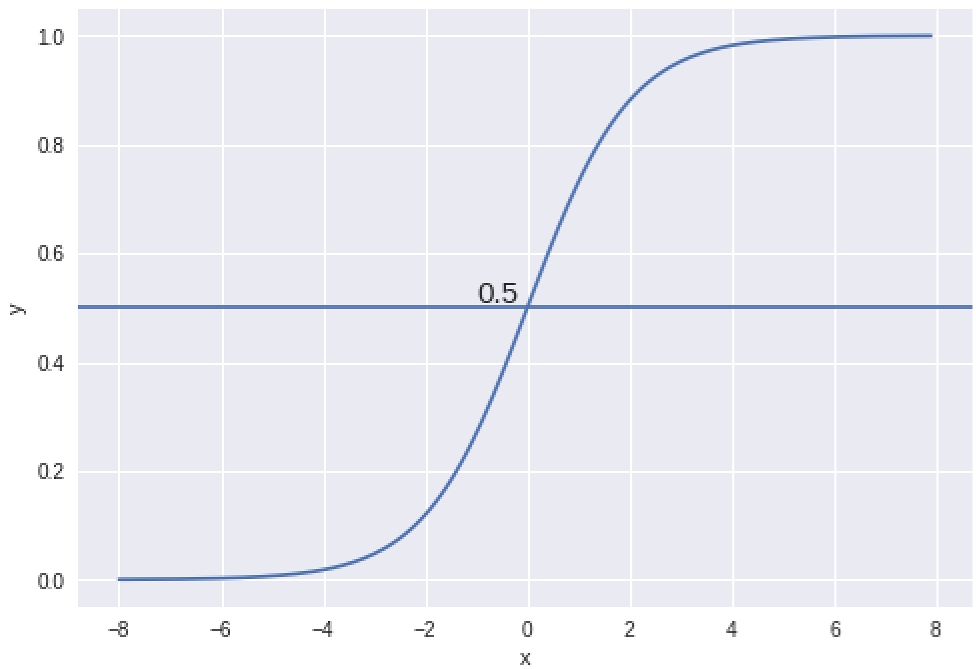
حداقل مقدار هزینه بدست آمده:

```
0.6516939975954287
```

بخش سوم

Logistic Regression

بر خلاف دو الگوریتم قبلی که متغیر وابسته یعنی Y میتوانست هر مقداری بصورت نا محدود در یک فضای پیوسته داشته باشد، در اینجا این متغیر باید چند مقدار محدود و گسسته یا به اصطلاح Categorical داشته باشد. اگر متغیر وابسته تنها دو مقدار داشته باشد، به آن Binary Logistic Regression میگوند. اگر فرمول رگرسیون خطی را به خاطر داشته باشید خروجی آن جمع وزن دار ورودی ها بود. اما Logistic Regression یک حالت کلی تر از رگرسیون خطی است یعنی خروجی آن بعد از محاسبه جمع وزن دار ورودی ها به یک تابع برای نگاشت آن به صفر و یک داده میشود و مقدار نگاشت شده به خروجی میرود. به تابعی که هر مقدار عدد حقیقی را به صفر و یک نگاشت میکند تابع فعال ساز یا Activation Function میگوند. هر تابع فعال سازی که این خصوصیت را داشته باشد قابل استفاده است اما ما در اینجا از تابع سیگموید استفاده میکنیم که نمودار آن را در زیر گذاشته ام.



همانطور که میبینید مقدار این تابع همیشه بین صفر و یک و در X = 0 برابر با 0.5 است. بنابراین میتوان سر حد احتمال برای هر دسته را 0.5 در نظر گرفت یعنی اگر احتمال ورودی ها کمتر از نیم شد مربوط به کلاس صفر و اگر بیشتر از نیم شد مربوط به کلاس یک است. اگر به خاطر داشته باشید در رگرسیون خطی مقدار $\hat{Y}$ برابر با XW بود.

$$\hat{Y} = XW$$

اگر تابع سیگموید را روی مقدار بدست آمده اعمال کنیم خروجی Logistic Regression بدست می آید.

$$\hat{Y} = \sigma(XW)$$

فرمول تابع سیگموید نیز در زیر نوشته شده است.

$$\sigma(t) = \frac{1}{1 + e^{-t}}$$

اگر این دو تابع را در هم ادغام کنیم به تابع زیر میرسیم.

$$h(x) = \hat{Y} = 1 = \frac{1}{1 + e^{-XW}}$$

$$h(x) = \begin{cases} > 0.5, & if\ XW > 0 \\ < 0.5, & if\ XW < 0 \end{cases}$$

اگر حاصل این جمع وزن دار بیشتر از صفر باشه کلاس مربوط به آن یک و اگر کمتر از صفر باشد کلاس آن صفر است.

مانند رگرسیون خطی نیز ما باید یک تابع هزینه معرفی کرده و سعی بر کمینه کردن آن داشته باشیم. تابع هزینه مورد نظر برای یک نمونه از دیتا بصورت زیر معرفی میشود.

$$cost = \begin{cases} -\log\big(h(x)\big), & if\ y = 1 \\ -\log\big(1 - h(x)\big), & if\ y = 0 \end{cases}$$

برای $-\log\big(h(x)\big)$ هر چه *h(x)* به سمت یک میل کند نمودار آن نیز به سمت صفر و هر چه به سمت صفر میل کند نمودار آن به سمت بی نهایت میرود. همینطور برای $-\log\big(1 - h(x)\big)$ نیز هر چه *h(x)* به سمت یک میل کند نمودار آن به سمت صفر و هر چه به سمت صفر میل کند نمودار آن بی نهایت میشود. اگر هر دو معادله بالا را با هم ترکیب کنیم به معادله زیر میرسیم.

$$cost(h(x), y) = -ylog\big(h(x)\big) - (1 - y)\, log\big(1 - h(x)\big)$$

اگر بخواهیم هزینه را برای تمام دیتا هایمان بدست بیاوریم باید برای تک تک آن ها این فرمول را حساب کرده و میانگین آن ها را بدست بیاوریم.

$$J(w) = -\frac{1}{m}\sum_{i=1}^{m}[y^i log\left(h\left(x^i\right)\right) + \left(1 - y^i\right) log\left(1 - h\left(x^i\right)\right)]$$

در اینجا *m* تعداد داده ها می باشد.

برای کمینه کردن این معادله از *Gradient Descent* کمک میگیریم که فرمولی شبیه آن چه در رگرسیون خطی دیدید دارد.

$$\frac{\partial J(w)}{\partial w_i} = \frac{1}{m}\sum_{i=1}^{m}\left(h\left(x^i\right) - y^i\right)x_j^i$$

برای آپدیت کردن هر وزن نیز از فرمول زیر استفاده میشود.

$$\mathrm{w}_j = w_j - lr.\left(\frac{1}{m}\sum_{i=1}^{m}\left(h\left(x^i\right) - y^i\right)x_j^i\right)$$

از آنجایی که تنها دو مقدار در هر نمونه از ورودی وجود دارد، بعد از اجرای الگوریتم و یاد گرفتن پارامتر ها معادله خطی ما بصورت زیر است.

$$h(X) = w_0 + x_1 w_1 + x_2 w_2$$

اگر بخواهیم بحث را خلاصه کنیم معادله خط تصمیم گیری یا همان *Decision Boundary* از فرمول زیر بدست می آید.

$$x_2 = -\frac{w_0 + x_1 w_1}{w_2}$$

این همه توضیحات لازم و بسیار مختصر درباره این الگوریتم بود. برای این تمرین ما از دیتاست *IRIS* استفاده خواهیم کرد که دو تا از چهار مقدار در هر نمونه از این دیتاست را حذف خواهیم کرد یعنی ماتریس ویژگی های ما دو ستونه است و یکی از دسته ها یا همان *Category* ها را نیز جهت استفاده از این دیتاست در الگوریتم *Binary Logistic Regression* حذف خواهیم کرد. ستون یک تا چهار ستون ویژگی هاست که ما ستون های 3 و 4 را حذف میکنیم و ستون پنجم ستون کلاس هاست که ما ردیف های مربوط به کلاس *Iris-versicolor* را

حذف خواهیم کرد. و باقی دو کلاس را به صفر و یک مپ میکنیم زیرا مقدار ستون کلاس بصورت رشته است و ما با عدد کار میکنیم. در زیر نمایی کلی به همراه توضیحاتی درباره هر ویژگی ارائه کرده ایم.

ردیف	ویژگی	توضیحات
1	*sepal length*	طول کاسبرگ
2	*sepal width*	عرض کاسبرگ
3	*petal length*	طول گلبرگ
4	*petal width*	عرض گلبرگ
5	*class*	نوع گیاه • *Iris-setosa* • *Iris-versicolor* • *Iris-virginica*

پیاده سازی

در این تمرین از زبان پایتون و کتاب خانه های Numpy، Pandas، Sklearn و Matplotlib برای خواندن فایل ها، انجام پیش پردازش ها و محاسبات و رسم نمودار ها استفاده شده است. کد این بخش از تمرین در فایل Logistic_Regress.ipynb موجود است. در ادامه به تشریح توابع نوشته شده که مربوط به بخش اصلی الگوریتم هستند میپردازیم.

load_data

```
def load_data():
  data = pd.read_csv('/content/iris.data',names=['s_lenght', 's_width', 'p_length', 'p_width', 'iris_class'])
  data.drop(['p_length', 'p_width'], axis=1, inplace=True)
  data = data[data.iris_class != 'Iris-versicolor']
  data["iris_class"].replace({"Iris-setosa": 0., "Iris-virginica": 1.}, inplace=True)
  X_train, X_test, y_train, y_test = train_test_split(data.iloc[ : , 0:-1], data.iloc[ : , -1], stratify=data['iris_class'], test_size=0.2, random_state=2020)
  return X_train, X_test, y_train, y_test
```

این تابع Iris.Data را بوسیله کتاب خانه Pandas درون برنامه وارد و پس از حذف دو ستون و حذف کلاس Iris_versicolor، داده ها را به دسته های Train و Test به کمک کتاب خانه Sklearn و با نرخ 80 به 20 بر میگرداند.

Normalization

```
def normalization(data):
  data = (data) / data.max()
  return data
```

این تابع داده ها را بین صفر و یک نرمال میکند.

init_weights

```
def init_weights(size):
  return np.array(np.random.rand(X_train.shape[1]), dtype='float64')
```

این تابع وزن های با مقادیر رندم برای شروع الگوریتم، تولید میکند.

Sigmoid

```
def sigmoid(x):
    y = 1 / (1 + np.exp(-x))
    return y
```

تابع سیگمویدی که توضیح داده شد را در اینجا پیاده سازی کرده ایم.

compute_loss

```
def compute_loss(y_, y, size):
  return -1/size * np.sum(y * np.log(y_)) + (1 - y) * np.log(1-y_)
```

این تابع نیز برای محاسبه Cost طبق همان فرمول ارائه شده استفاده می شود.

update_weight

```
def update_weight(x, y, y_, lr, size):
   return lr * (1/size * np.dot(x.T, (y_ - y)))
```

این تابع برای محاسبه بخش $lr.\left(\frac{1}{m}\sum_{i=1}^{m}\left(h\left(x^i\right)-y^i\right)x_j^i\right)$ مربوط به معادله آپدیت هزینه استفاده میشود.

update_bias

```
def update_bias(y, y_, lr, size):
  return lr * (1/size * np.sum(y_ - y))
```

بایاس نیز مانند وزن ها آپدیت میکنیم.

Mse

```
def mse(y_, y):
  diff = np.subtract(y_, y)
  ms = np.power(diff, 2, dtype='float64')
  return np.sum(ms) / len(y)
```

این تابع برای محاسبه Mean Square Error بر روی مقادیر پیش بینی شده و مقادیر واقعی استفاده میشود.

Predict

```
def predict(model, x):
    return sigmoid(np.dot(x, model[1:]) + model[0])
```

این تابع برای پیشبینی روی مقادیر جدید استفاده میشود و مطابق با فرمول Decision Boundary پیاده سازی شده است.

Decision_boundary

```
def decision_boundary(x, weights, bias):
  return - (bias + np.dot(weight[0], x)) / weight[1]
```

این تابع معادله خط Decision Boundary را پیاده سازی میکند.

Train

```
def train(x, y, lr, epochs, weight, bias):
    size = x.shape[0]
    for i in range(epochs):
        weighted_sum = np.dot(x, weight) + bias
        y_ = sigmoid(weighted_sum)
        loss = compute_loss(y_, y, size)
        loss_log.append(loss)
        weight = weight - update_weight(x, y, y_, lr, size)
        bias = bias - update_bias(y, y_, lr, size)
    return (weight, bias)
```

در این تابع بر اساس فرمول های شرح داده شده، مدل آموزش داده میشود.

پارامتر ها

برای آموزش ما چند پارامتر را برای تنظیمات مدل تعیین کرده ایم.

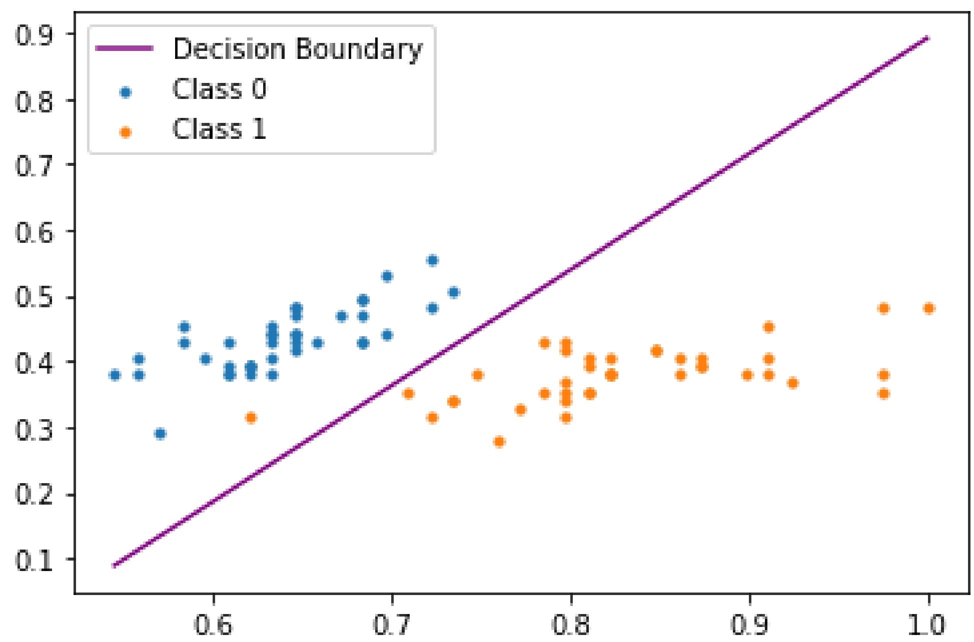
- Epochs : این مقدار تعیین کننده تعداد تکرار و دفعات اجرای الگوریتم می باشد که ما 10000 انتخاب کرده ایم.
- Lr : این همان نرخ یاد گیری است که ما 0.02 انتخاب کرده ایم.

نتیجه

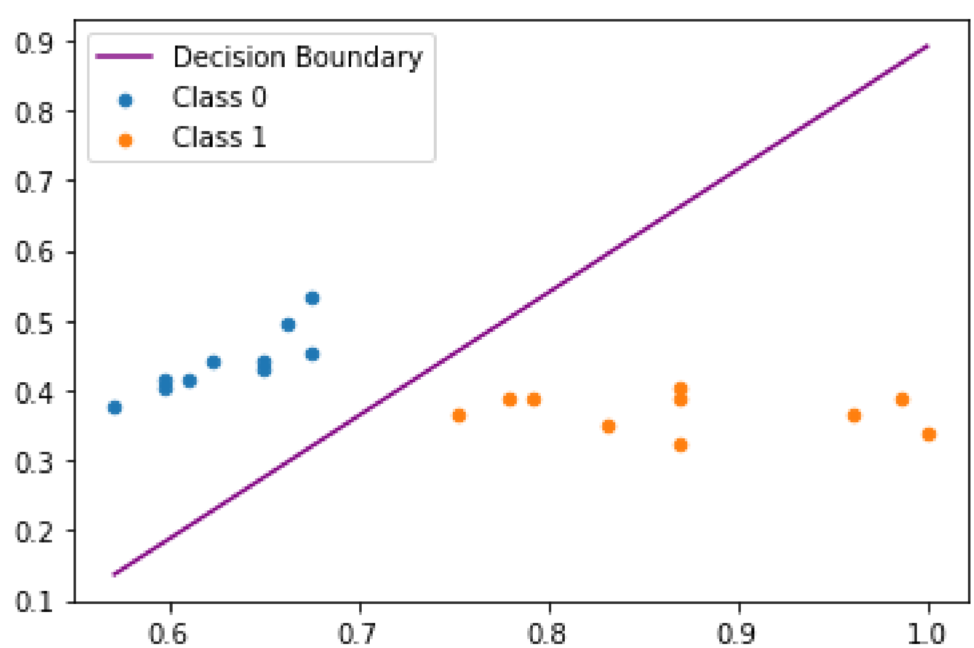
بعد از اجرای الگوریتم بر روی داده ها به نکته ای دست یافتم. اگر داده ها را بین صفر و یک نرمال کنیم مدل در یادگیری داده ها دقیق نمیشود و خطای MSE زیادی دارد. اما اگر نرمال سازی انجام نشود مدل به خوبی داده ها را یاد میگیرد و MSE کمتری دارد. مقدار نرخ یادگیری و تعداد epoch ها نیز بسیار در همگرا شدن مدل مهم است.

نتیجه با نرمال سازی:

نمودار خط دسته بندی و توزیع داده های Train:



نمودار خط دسته بندی و توزیع داده های Test:



خطای MSE روی داده های Test و Train:

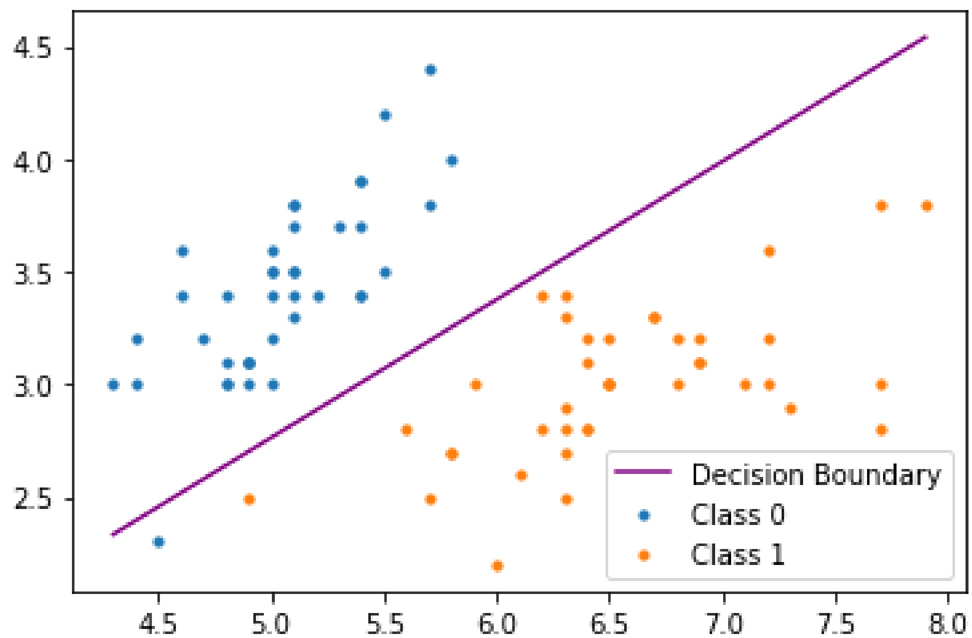
```
MSE Train =  0.1263218557144464
MSE Test =  0.10536628536297152
```

وزن های یاد گرفته شده به ترتیب از چپ : بایاس یا همان w_0، w_1 و w_2

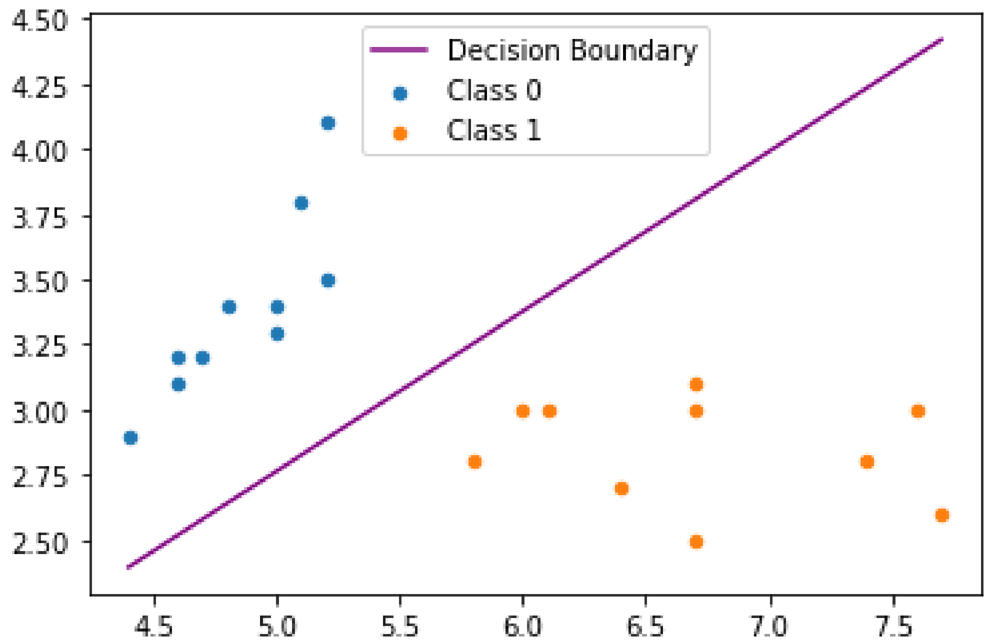
```
[-1.41063747,  3.79707562, -6.25419372]
```

نتیجه بدون نرمال سازی:

نمودار خط دسته بندی و توزیع داده های Train:



نمودار خط دسته بندی و توزیع داده های Test:



خطای MSE روی داده های Test و Train:

```
MSE Train =  0.012268323617948985
MSE Test =  0.0010944553649248978
```

وزن های یاد گرفته شده به ترتیب از چپ : بایاس یا همان w_0، w_1 و w_2

```
[-1.29405614,  3.79664167, -6.28995596]
```