parth.pandey13103347@gmail.com_10_C

June 14, 2020

0.1 Task-C: Regression outlier effect.

Objective:Visualization best fit linear regression line for different scenarios

0.1.1 Imports

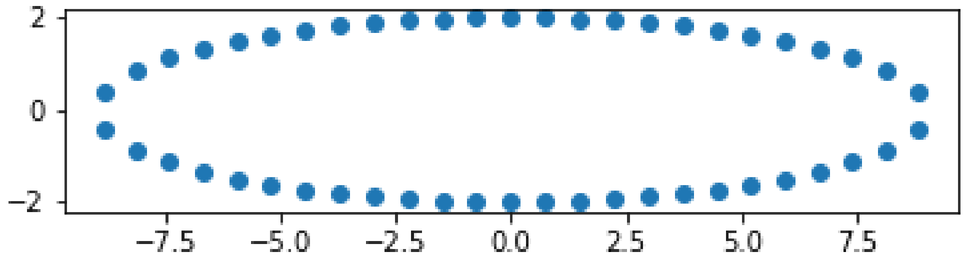
[12]:
```
# you should not import any other packages
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings("ignore")
import numpy as np
from sklearn.linear_model import SGDRegressor
```

0.1.2 Data Creation Functions

[13]:
```
import numpy as np
import scipy as sp
import scipy.optimize

def angles_in_ellipse(num,a,b):
    assert(num > 0)
    assert(a < b)
    angles = 2 * np.pi * np.arange(num) / num
    if a != b:
        e = (1.0 - a ** 2.0 / b ** 2.0) ** 0.5
        tot_size = sp.special.ellipeinc(2.0 * np.pi, e)
        arc_size = tot_size / num
        arcs = np.arange(num) * arc_size
        res = sp.optimize.root(
            lambda x: (sp.special.ellipeinc(x, e) - arcs), angles)
        angles = res.x
    return angles
```

[14]:
```
a = 2
b = 9
n = 50

phi = angles_in_ellipse(n, a, b)
```

```
e = (1.0 - a ** 2.0 / b ** 2.0) ** 0.5
arcs = sp.special.ellipeinc(phi, e)

fig = plt.figure()
ax = fig.gca()
ax.axes.set_aspect('equal')
ax.scatter(b * np.sin(phi), a * np.cos(phi))
plt.show()
```



0.1.3 Generating Data

[15]:
```
x_train= b * np.sin(phi)
y_train= a * np.cos(phi)
```

0.1.4 Steps to achieve Objective

0.1.5 Observation on checking the documentation of loss functions of Linear Models.

- On checking out the https://scikit-learn.org/stable/modules/linear_model.html

  It was brought into light that the regression implementation with regularization is called **ridge regression**

- **"This classifier is sometimes referred to as a Least Squares Support Vector Machines with a linear kernel."**

  Here the documentation says that Linear SVM is pretty similar to the current implementation.

0.1.6 Formula Used in the Self Implementation of Linear Regression

Linear Regression Loss Function Used

- Loss Function

  $\min_w \|X.w - y\|_2^2 + \alpha\|w\|_2^2$

- Gradient for w

  $\delta L/\delta w_{w_{old}} = 2 * X * ((w.X) + b - y) + 2\alpha w$

- Gradient for b

  $\delta L/\delta b_{b_{old}} = 2 * ((w.X) + b - y)$

Defining Functions for the above formula

[16]:
```
# Weight Intialization
def init_weights(x_train):
    w = np.zeros_like(x_train)
    b = 0
    return w,b

# gradient function
# Loss = Square Loss
def gradient_dw(x, y, w, b, param):
    return 2 * x * (np.dot(w,x) + b - y ) + 2 * param * w

def gradient_db(x, y, w, b, param):
    return 2 * (np.dot(w,x) + b - y )

def loss(y_true,y_pred):
    return np.sum((y_true- y_pred)**2)

def draw_line(coef,intercept, mi, ma, ax):
    # for the separating hyper plane ax+by+c=0, the weights are [a, b] and the
 ↪intercept is c
    # to draw the hyper plane we are creating two points
    # 1. ((b*min-c)/a, min) i.e ax+by+c=0 ==> ax = (-by-c) ==> x = (-by-c)/a
 ↪here in place of y we are keeping the minimum value of y
    # 2. ((b*max-c)/a, max) i.e ax+by+c=0 ==> ax = (-by-c) ==> x = (-by-c)/a
 ↪here in place of y we are keeping the maximum value of y
    points=np.array([(mi, (coef*mi + intercept)),(ma, (coef*ma + intercept))])
    ax.plot(points[:,0], points[:,1])
```

Defining train function

[18]:
```
def train(x_train, y_train, epochs, param, eta):
    w,b = init_weights(x_train[0])
    epoch_loss = 0
    for epoch in range(1,epochs+1):
        for x, y in zip(x_train, y_train):
            w = w - (eta * gradient_dw(x,y,w,b,param))
            b = b - (eta * gradient_db(x,y,w,b,param))
        y_pred = []
        for x in x_train:
            y_pred.append(np.dot(w,x) + b)
        old_loss = epoch_loss
```

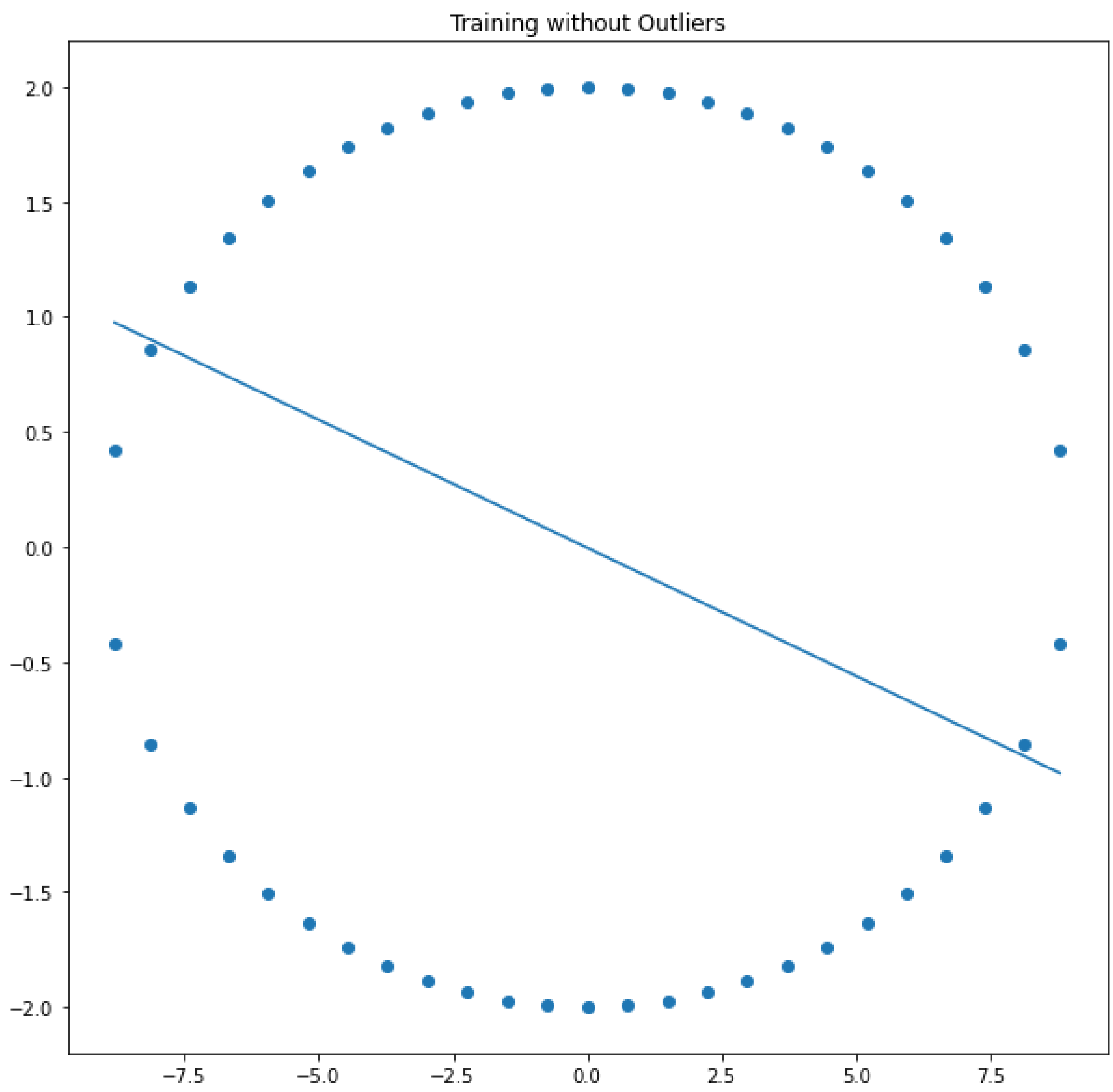
```
        epoch_loss = loss(y_train, y_pred)
        if epoch_loss==old_loss:
            break
    return w,b
```

0.1.7 Linear Regression Implementation

Training with Original Data

[19]:
```
# training on original data
_, ax = plt.subplots(1,1,figsize=(10,10))
w, b = train(x_train, y_train, 1000, 0.0001, 0.001)
ax.scatter(x_train, y_train)
draw_line(w,b,np.min(x_train),np.max(x_train),ax)
ax.set_title('Training without Outliers')
```

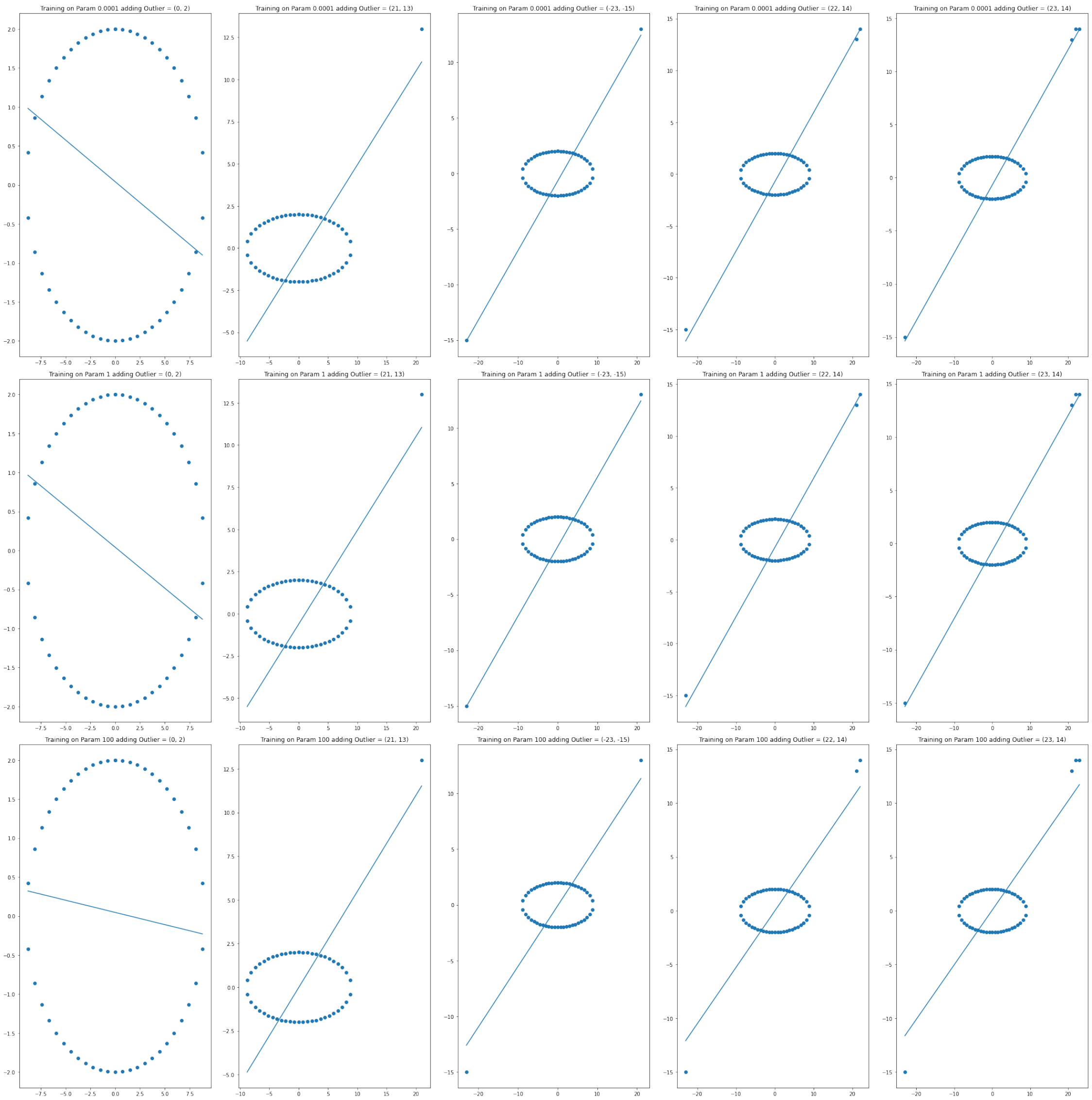
[19]:
```
Text(0.5, 1.0, 'Training without Outliers')
```



Training with outliers

[20]:
```
outliers = [(0,2),(21, 13), (-23, -15), (22,14), (23, 14)]
hyperparams = [0.0001, 1, 100]
```

[21]:
```
_, ax = plt.subplots(3,5,figsize=(30,30))
for i, param in enumerate(hyperparams):
    x_train_temp = x_train
    y_train_temp = y_train
    for j, ele in enumerate(outliers):
        x_train_temp = np.append(x_train_temp, ele[0])
        y_train_temp = np.append(y_train_temp, ele[1])
        w,b = train(x_train_temp, y_train_temp, 1000, param, 0.001)
        ax[i][j].scatter(x_train_temp, y_train_temp)
        draw_line(w,b,np.min(x_train_temp),np.max(x_train_temp),ax[i][j])
        ax[i][j].set_title('Training on Param {} adding Outlier = {}'.
 ↪format(param, ele))
plt.tight_layout()
plt.show()
```



0.1.8 Observations

- Single addition of outlier changed the angle of the hyperplane drastically
- Applying high regularization penalty did not have much benefit
- Linear Regression is sensitive to outliers.