Dataset1-Regression_output_9

October 7, 2021

1 Dataset 1 - Regression

1.1 Import Libraries

[1]:

```
import train_test
import ABC_train_test
import regressionDataset
import network
import statsModel
import performanceMetrics
import dataset
import sanityChecks
import torch
import matplotlib.pyplot as plt
import seaborn as sns
from scipy.stats import norm
from torch.utils.data import Dataset,DataLoader
from torch import nn
import warnings
warnings.filterwarnings('ignore')
```

1.2 Parameters

General Parameters

1. Number of Samples

Discriminator Parameters

1. Size : number of hidden nodes

ABC-Generator parameters are as mentioned below: 1. mean : 1 ($\beta \sim N(\beta^*, \sigma)$ where β^* are coefficients of statistical model) or 1 ($\beta \sim N(0, \sigma)$ 2. std : $\sigma = 1, 0.1, 0.01$ (standard deviation)

[2]:

```
n_features = 10
sample_size = 100
#Discriminator Parameters
hidden_nodes = 25
#ABC Generator Parameters
mean = 1
```

```
variance = 0.001
```

1.3 Dataset

Generate a random regression problem

$Y = 1 + \beta_1 x_1 + \beta_2 x_2 + \beta_2 x_3 + ... + \beta_n x_n + N(0, \sigma)$ where $\sigma = 0.1$

[3]:
```
X,Y = regressionDataset.regression_data(sample_size,n_features)
```

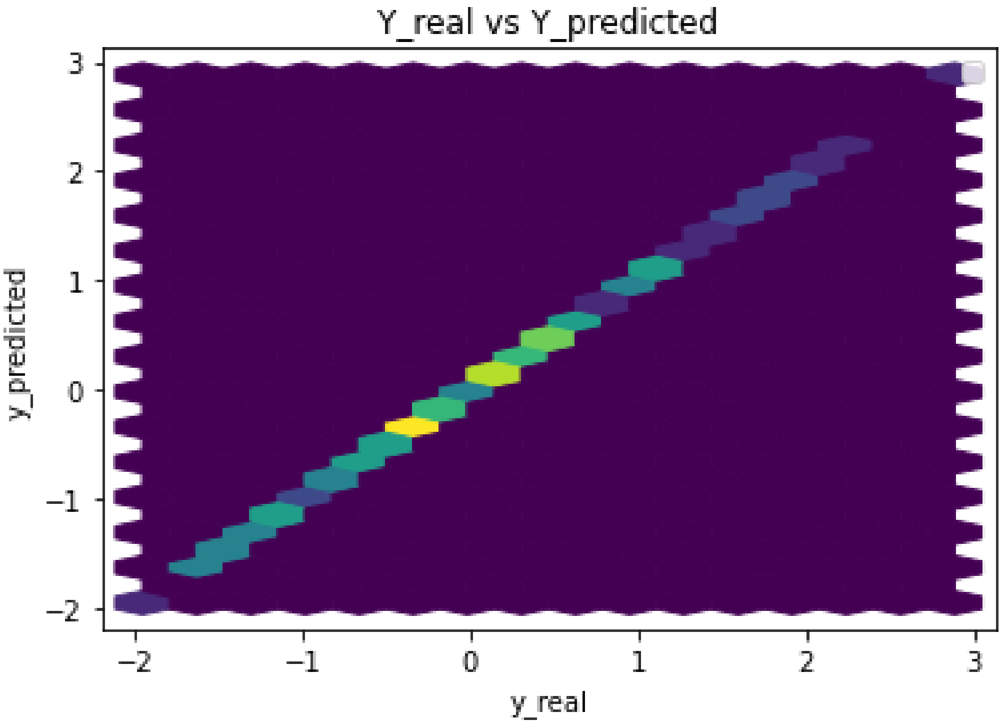
```
         X1        X2        X3        X4        X5        X6        X7  \
0 -0.215512  1.407389 -1.049263 -0.645717  0.319768  1.185369  0.220074
1  0.879960 -0.965730  0.713889  1.651659 -0.271871  0.868559  2.156089
2 -0.409349  1.653947  1.970338  0.226063 -0.040017  0.451635 -1.888333
3  0.504310  0.720850  1.239964  0.593469  0.076297  0.084789 -1.321251
4  1.268494  0.332000 -0.117844  0.389860  0.718036  1.958537 -0.892524

         X8        X9       X10           Y
0 -0.838251 -1.105253  0.501583 -143.799994
1 -0.835638 -2.275275 -0.264268   48.526112
2 -0.025118  0.008285  1.545805  259.985600
3 -1.456781  0.260325  1.051693  123.718269
4  0.029950  0.049869 -0.866056  219.299087
```

1.4 Stats Model

[4]:
```
[coeff,y_pred] = statsModel.statsModel(X,Y)
```

```
No handles with labels found to put in legend.

                            OLS Regression Results
==============================================================================
Dep. Variable:                      Y   R-squared:                       1.000
Model:                            OLS   Adj. R-squared:                  1.000
Method:                 Least Squares   F-statistic:                 4.719e+07
Date:                Thu, 07 Oct 2021   Prob (F-statistic):          1.17e-294
Time:                        07:42:35   Log-Likelihood:                 632.29
No. Observations:                 100   AIC:                            -1243.
Df Residuals:                      89   BIC:                            -1214.
Df Model:                          10
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const               0    4.6e-05          0      1.000   -9.15e-05    9.15e-05
x1             0.3547   4.78e-05   7422.215      0.000       0.355       0.355
x2             0.2638   4.76e-05   5544.259      0.000       0.264       0.264
x3             0.3900   4.77e-05   8176.351      0.000       0.390       0.390
x4             0.3695   4.78e-05   7723.600      0.000       0.369       0.370
x5             0.3024   4.69e-05   6441.701      0.000       0.302       0.302
```

```
x6             0.2334   5.02e-05   4650.847      0.000       0.233       0.233
x7             0.2007   4.97e-05   4035.451      0.000       0.201       0.201
x8             0.4178   4.87e-05   8578.080      0.000       0.418       0.418
x9             0.3952    4.9e-05   8072.177      0.000       0.395       0.395
x10            0.2092   4.78e-05   4380.172      0.000       0.209       0.209
==============================================================================
Omnibus:                        0.452   Durbin-Watson:                   1.940
Prob(Omnibus):                  0.798   Jarque-Bera (JB):                0.168
Skew:                           0.080   Prob(JB):                        0.920
Kurtosis:                       3.121   Cond. No.                         1.70
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
Parameters:  const    0.000000
x1       0.354743
x2       0.263786
x3       0.389957
x4       0.369482
x5       0.302398
x6       0.233359
x7       0.200748
x8       0.417792
x9       0.395233
x10      0.209245
dtype: float64
```



```
Performance Metrics
Mean Squared Error: 1.8858428749319536e-07
Mean Absolute Error: 0.00035209811714746445
Manhattan distance: 0.035209811714746445
Euclidean distance: 0.004342629243824476
```

2 Generator and Discriminator Networks

GAN Generator

[5]:
```
class Generator(nn.Module):

  def __init__(self,n_input):
    super().__init__()
    self.output = nn.Linear(n_input,1)

  def forward(self, x):
    x = self.output(x)
    return x
```

GAN Discriminator

[6]:
```
class Discriminator(nn.Module):
  def __init__(self,n_input,n_hidden):

    super().__init__()
    self.hidden = nn.Linear(n_input,n_hidden)
    self.output = nn.Linear(n_hidden,1)
    self.relu = nn.ReLU()

  def forward(self, x):
    x = self.hidden(x)
    x = self.relu(x)
    x = self.output(x)
    return x
```

ABC Generator

The ABC generator is defined as follows:

$Y = 1 + \beta_1 x_1 + \beta_2 x_2 + \beta_2 x_3 + ... + \beta_n x_n + N(0, \sigma)$ where $\sigma = 0.1$

$\beta_i \sim N(0, \sigma^*)$ when $\mu = 0$ else

$\beta_i \sim N(\beta_i^*, \sigma^*)$ where $\beta_i^* s$ are coefficients obtained from stats model

Parameters : μ and σ^*

σ^* takes the values 0.01,0.1 and 1

[7]:
```
def ABC_pre_generator(x_batch,coeff,variance,mean,device):

  coeff_len = len(coeff)

  if mean == 0:
    weights = np.random.normal(0,variance,size=(coeff_len,1))
    weights = torch.from_numpy(weights).reshape(coeff_len,1)
  else:
    weights = []
    for i in range(coeff_len):
      weights.append(np.random.normal(coeff[i],variance))
    weights = torch.tensor(weights).reshape(coeff_len,1)

  y_abc =  torch.matmul(x_batch,weights.float())
  gen_input = torch.cat((x_batch,y_abc),dim = 1).to(device)
  return gen_input
```

3 GAN Model

[8]:
```
real_dataset = dataset.CustomDataset(X,Y)
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

[9]:

```
generator = Generator(n_features+2)
discriminator = Discriminator(n_features+2,hidden_nodes)

criterion = torch.nn.BCEWithLogitsLoss()
gen_opt = torch.optim.Adam(generator.parameters(), lr=0.01, betas=(0.5, 0.999))
disc_opt = torch.optim.Adam(discriminator.parameters(), lr=0.01, betas=(0.5, 0.
 ↪999))
```

[10]:

```
print(generator)
print(discriminator)
```

```
Generator(
  (output): Linear(in_features=12, out_features=1, bias=True)
)
Discriminator(
  (hidden): Linear(in_features=12, out_features=25, bias=True)
  (output): Linear(in_features=25, out_features=1, bias=True)
  (relu): ReLU()
)
```

[11]:

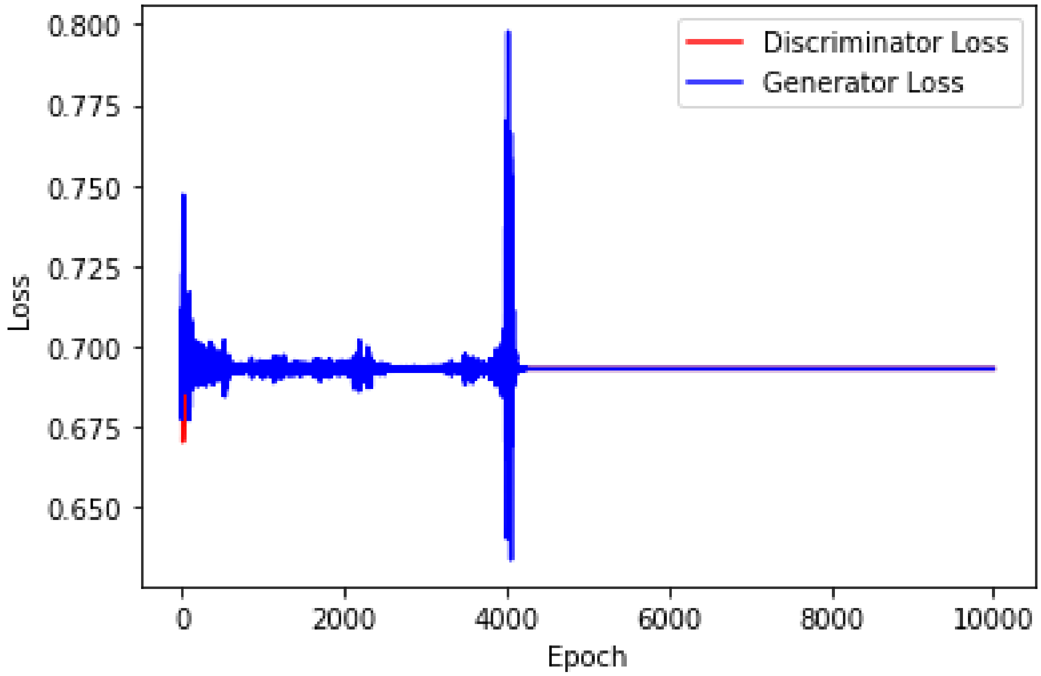
```
n_epochs = 5000
batch_size = sample_size//2
```

[12]:

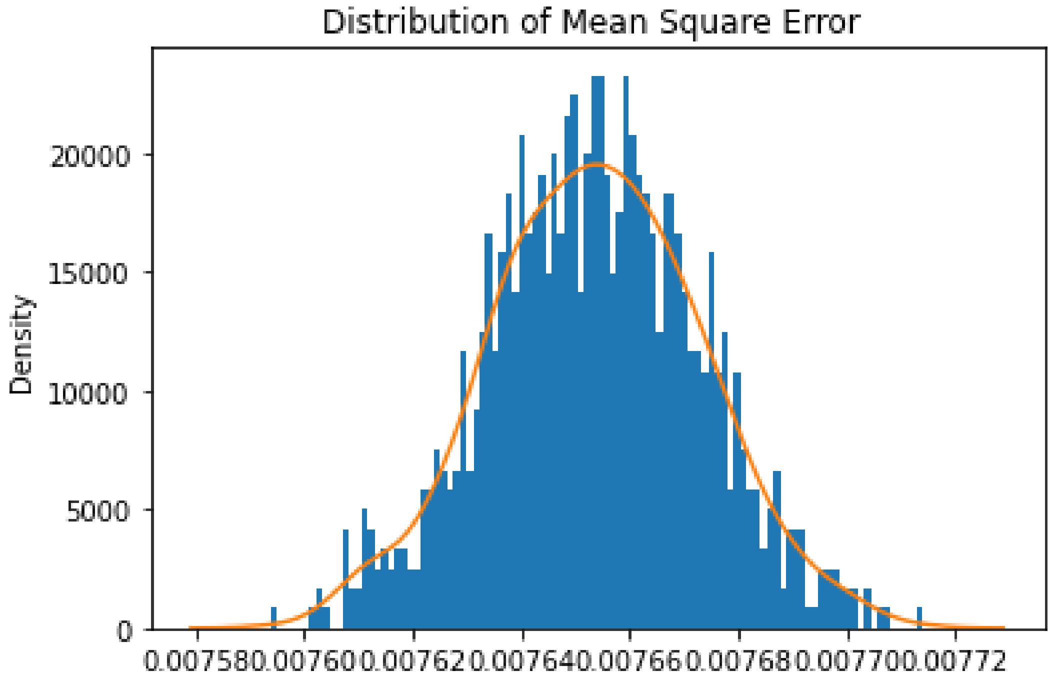
```
# Parameters
sample_size = 1000000
std = 1
mean = 0.01
```

[13]:

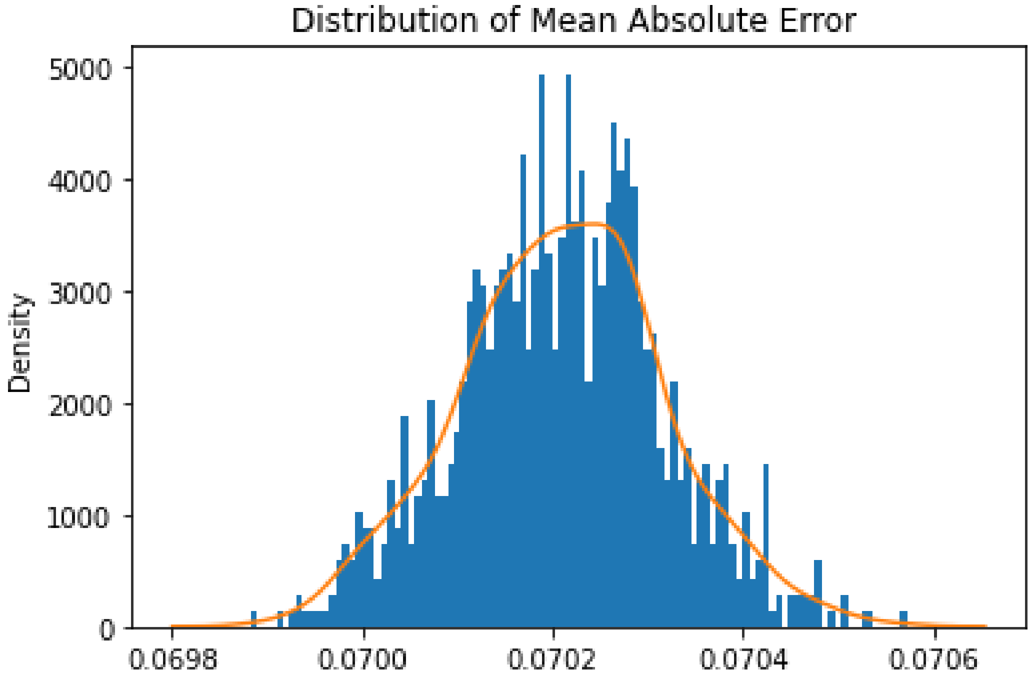
```
train_test.
 ↪training_GAN(discriminator,generator,disc_opt,gen_opt,real_dataset,batch_size,␣
 ↪n_epochs,criterion,device)
```



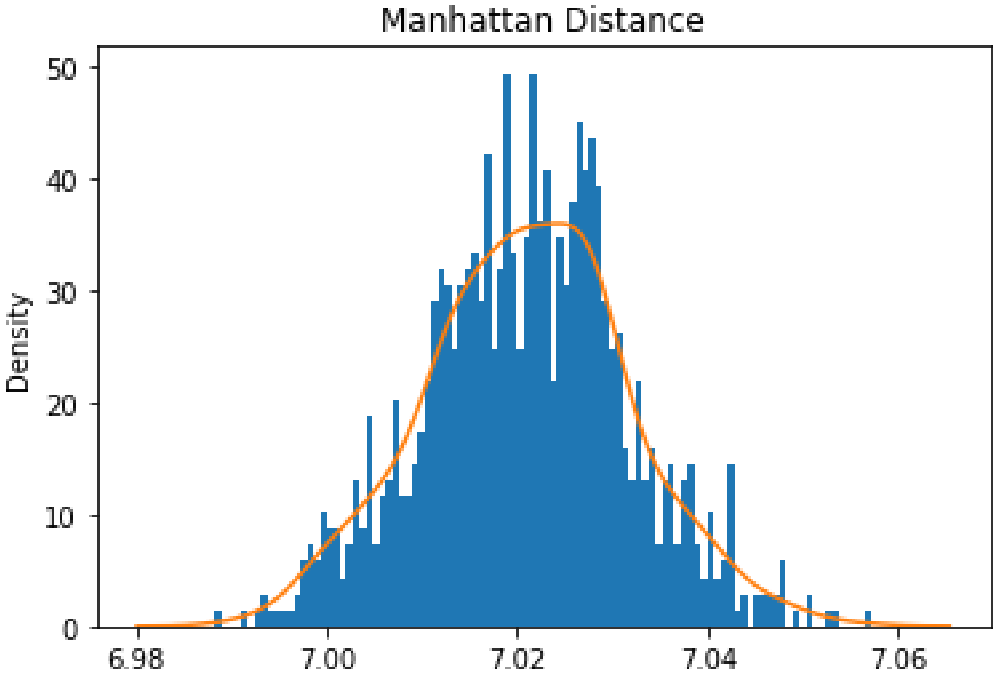
[14]:
```
train_test.test_generator(generator,real_dataset,device)
```



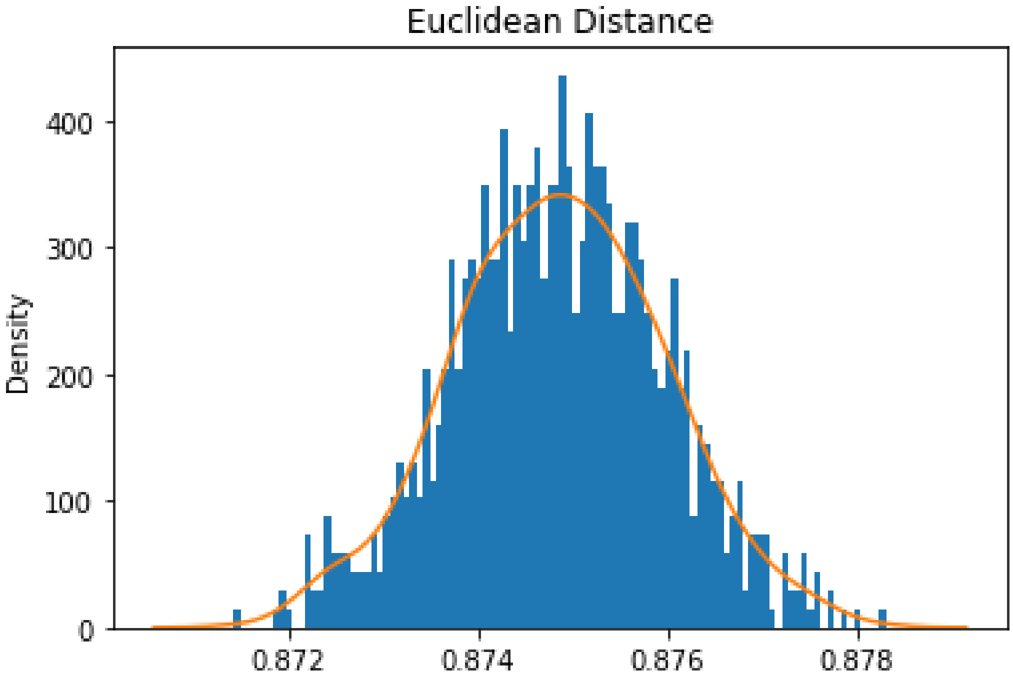
```
Mean Square Error: 0.007653671203333728
```



```
Mean Absolute Error: 0.07020997065138072
```



```
Mean Manhattan Distance: 7.020997065138072
```



```
Mean Euclidean Distance: 7.020997065138072
```

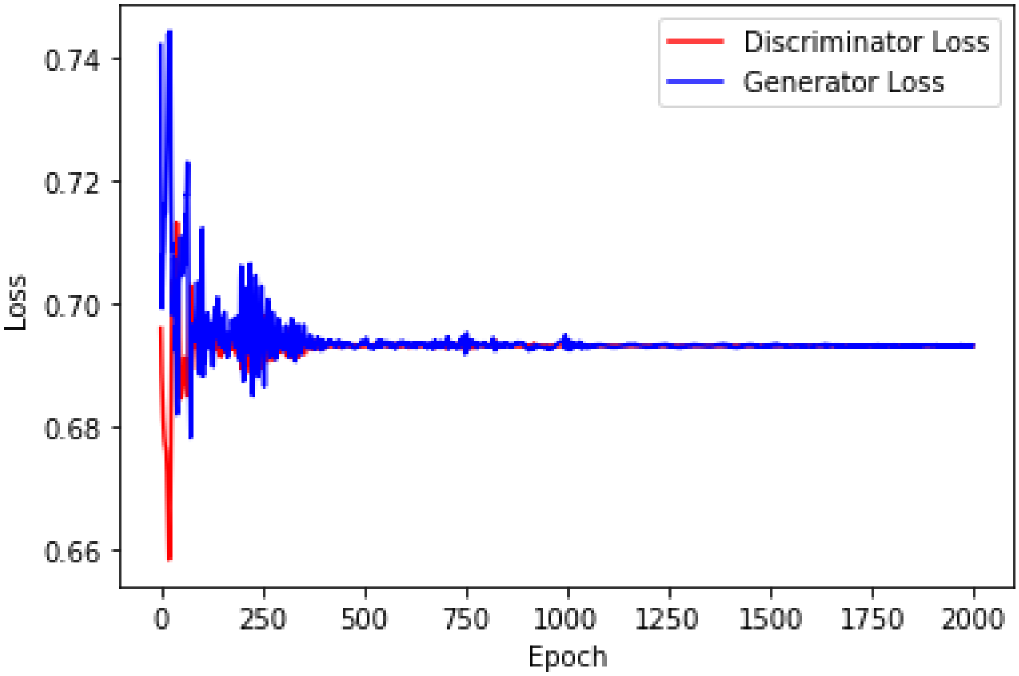
4 ABC GAN Model

Training the network

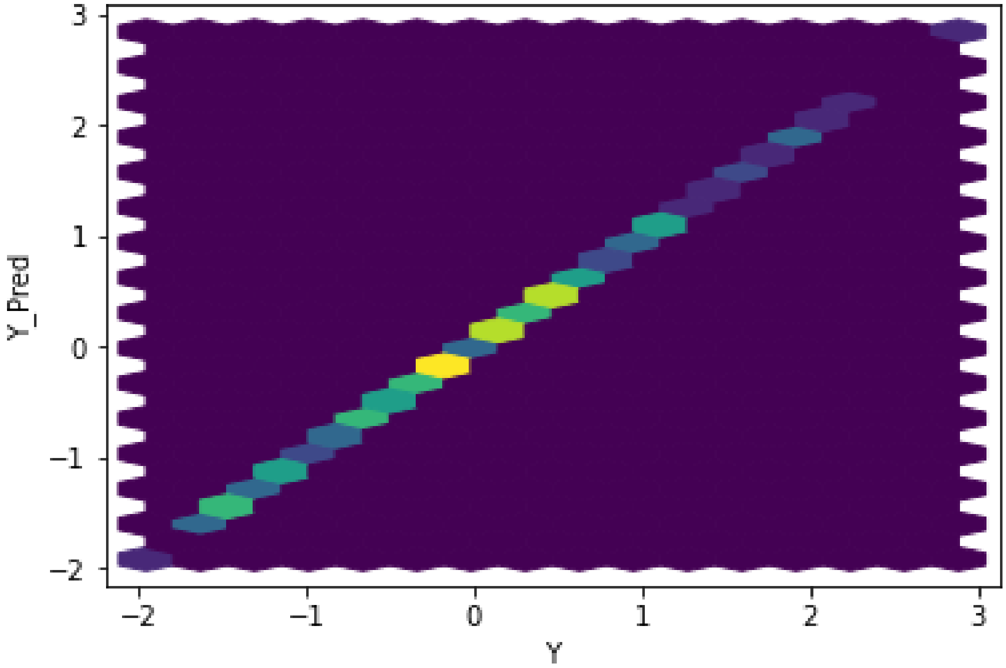
[15]:
```
gen = Generator(n_features+2)
disc = Discriminator(n_features+2,hidden_nodes)

criterion = torch.nn.BCEWithLogitsLoss()
gen_opt = torch.optim.Adam(gen.parameters(), lr=0.01, betas=(0.5, 0.999))
disc_opt = torch.optim.Adam(disc.parameters(), lr=0.01, betas=(0.5, 0.999))
```

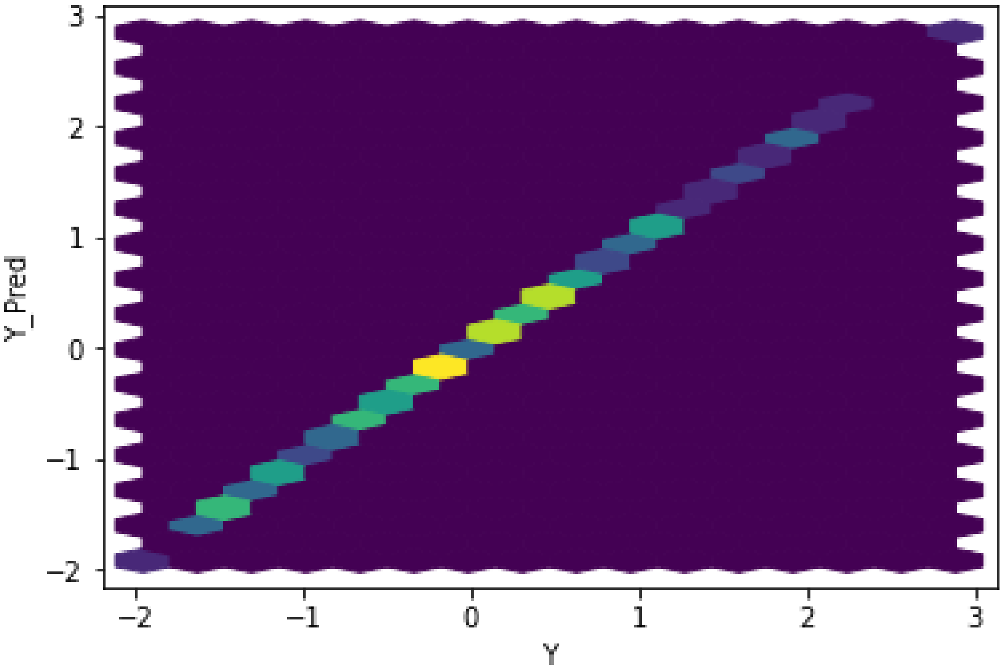
[16]:
```
n_epoch_abc = 2000
batch_size = sample_size//2
```

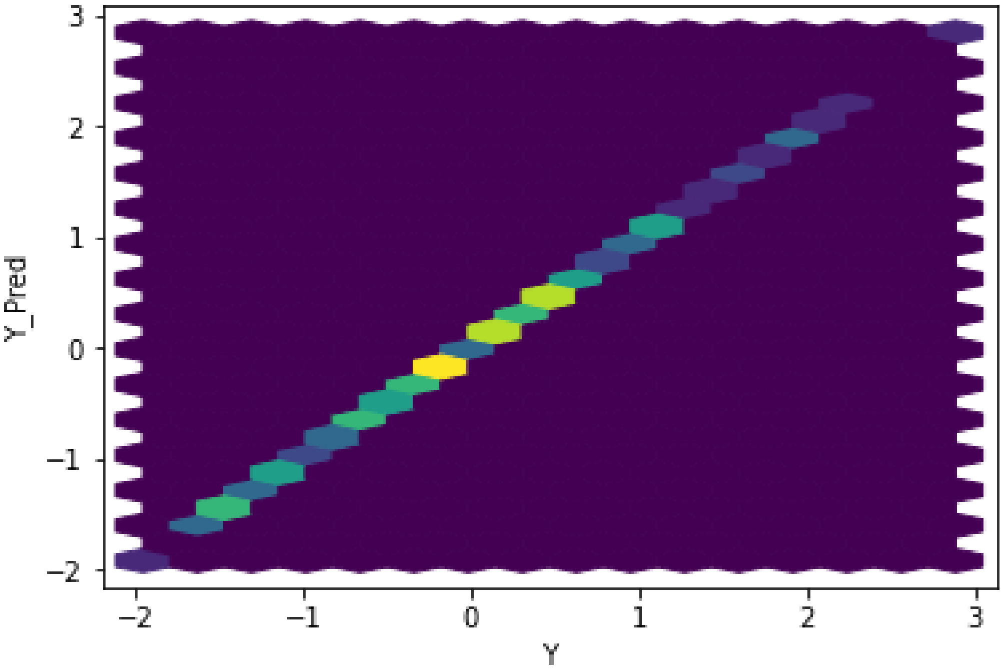
[17]:
```
ABC_train_test.training_GAN(disc, gen,disc_opt,gen_opt,real_dataset,␣
↪batch_size, n_epoch_abc,criterion,coeff,mean,variance,device)
```

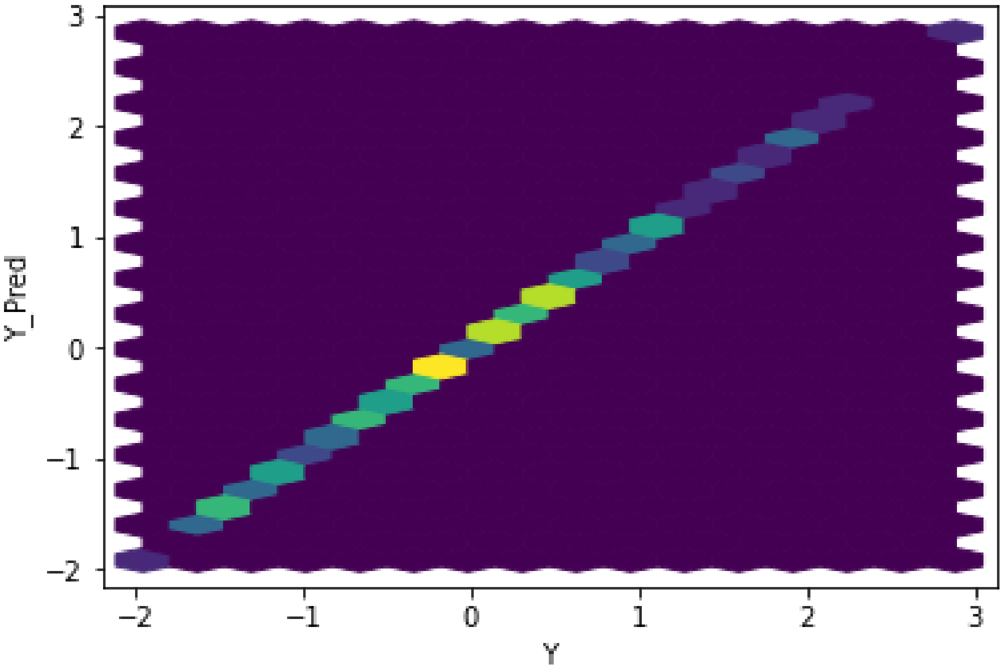


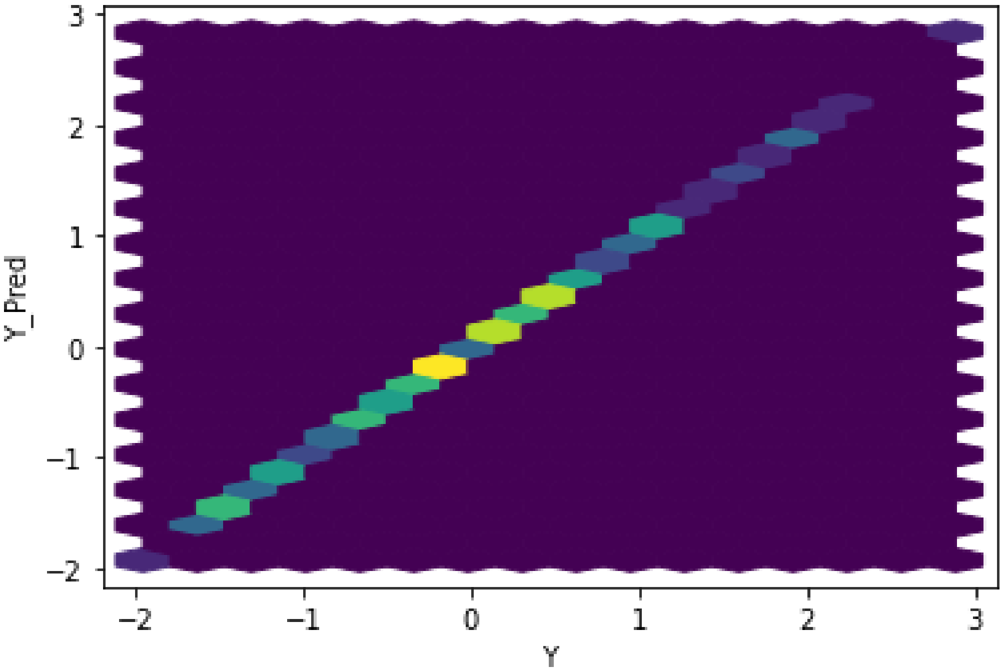
```
[18]: ABC_train_test.test_generator(gen,real_dataset,coeff,mean,variance,device)
```

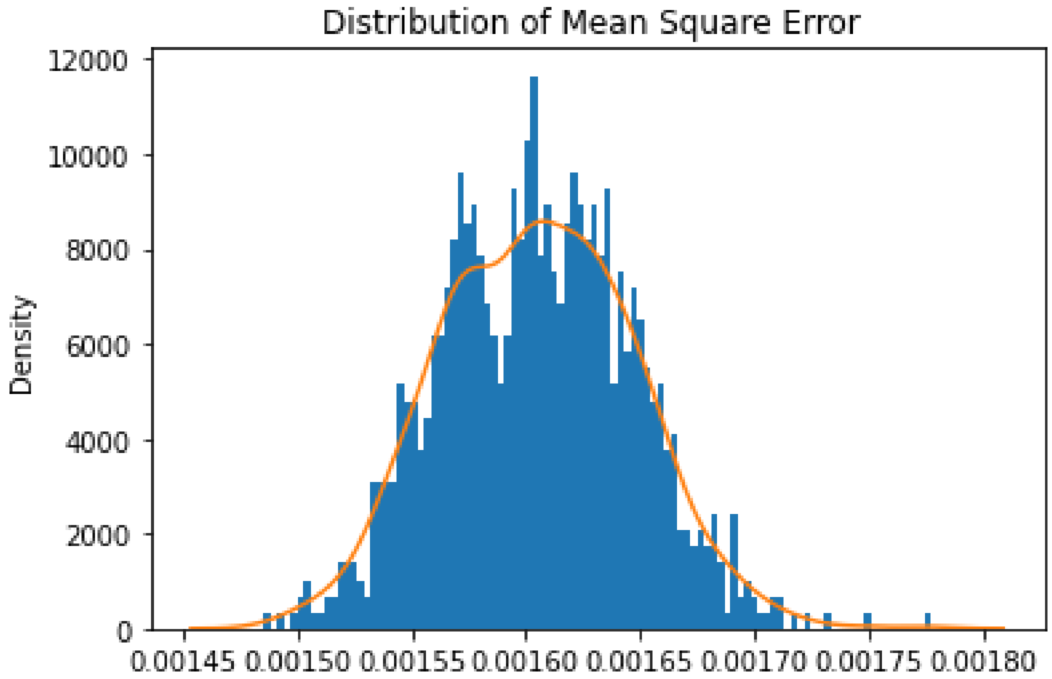



3
2
1
0
−1
−2
Y_Pred
−2
−1
0
1
2
3
Y

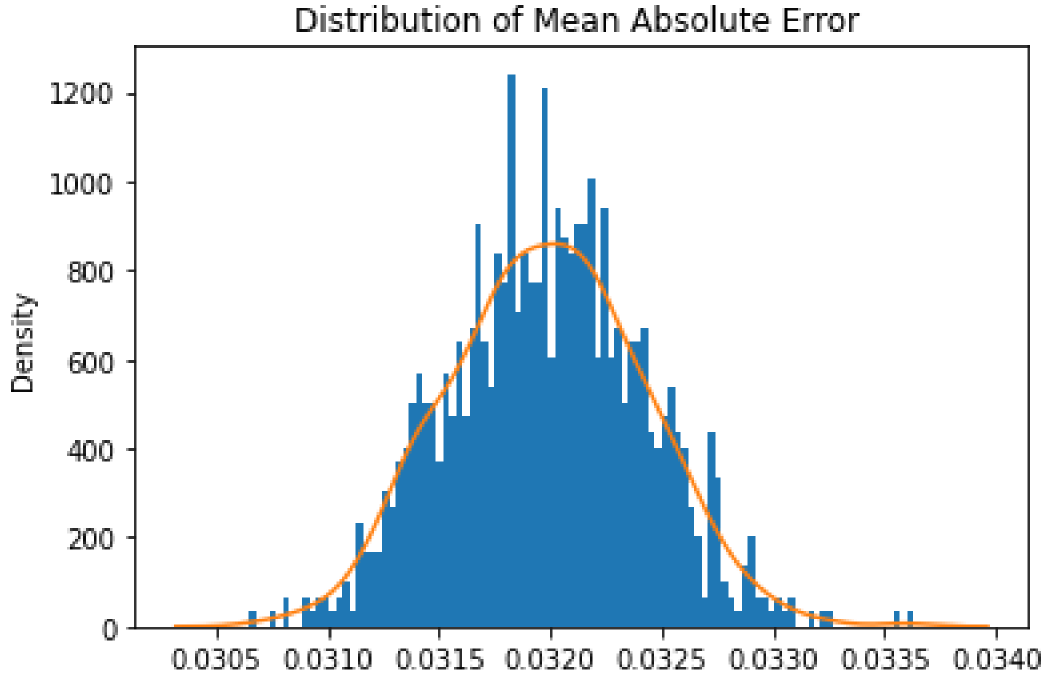

3
2
1
0
−1
−2
Y_Pred
−2
−1
0
1
2
3
Y


3
2
1
0
−1
−2
Y_Pred
−2
−1
0
1
2
3
Y

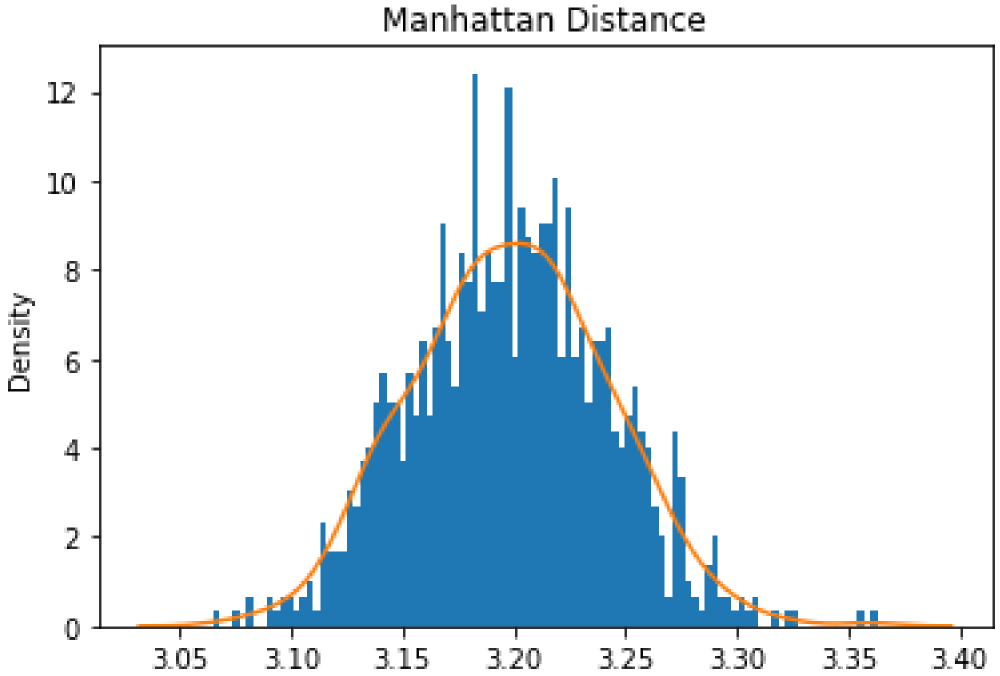

3
2
1
0
−1
−2
Y_Pred
−2
−1
0
1
2
3
Y



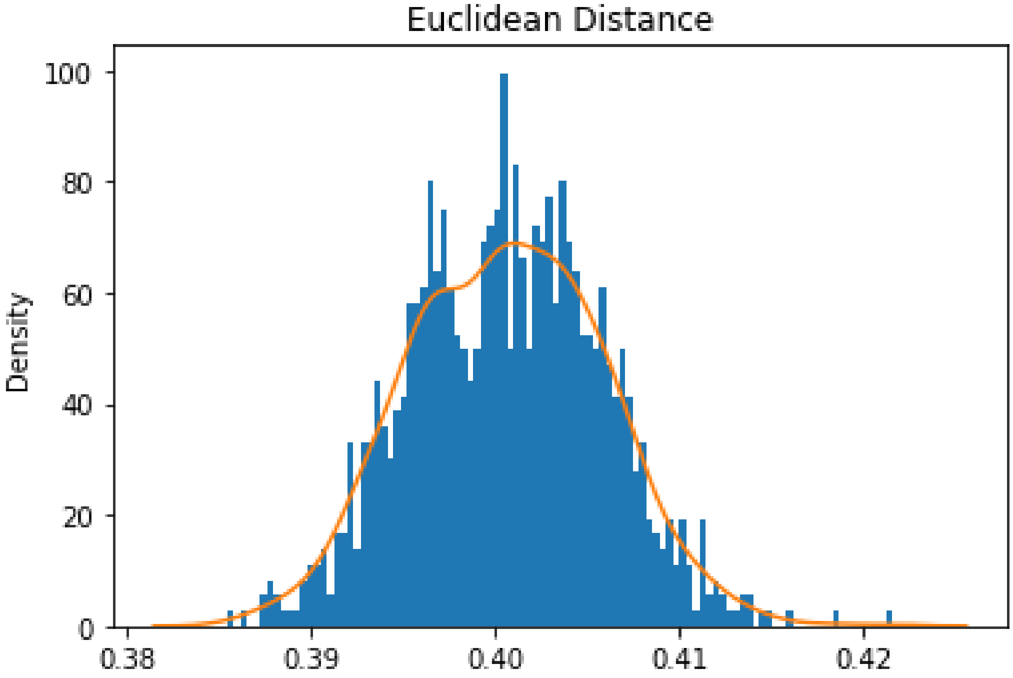
```
Mean Square Error: 0.0016051433010028833
```



```
Mean Absolute Error: 0.03198334513429552
Mean Manhattan Distance: 3.198334513429552
```

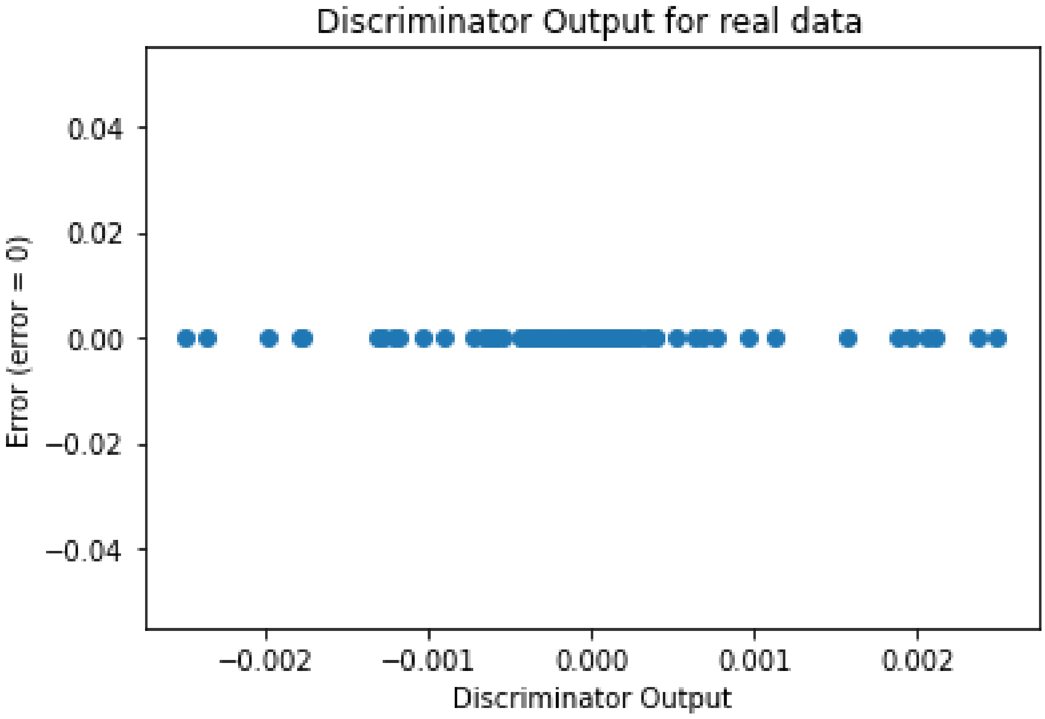


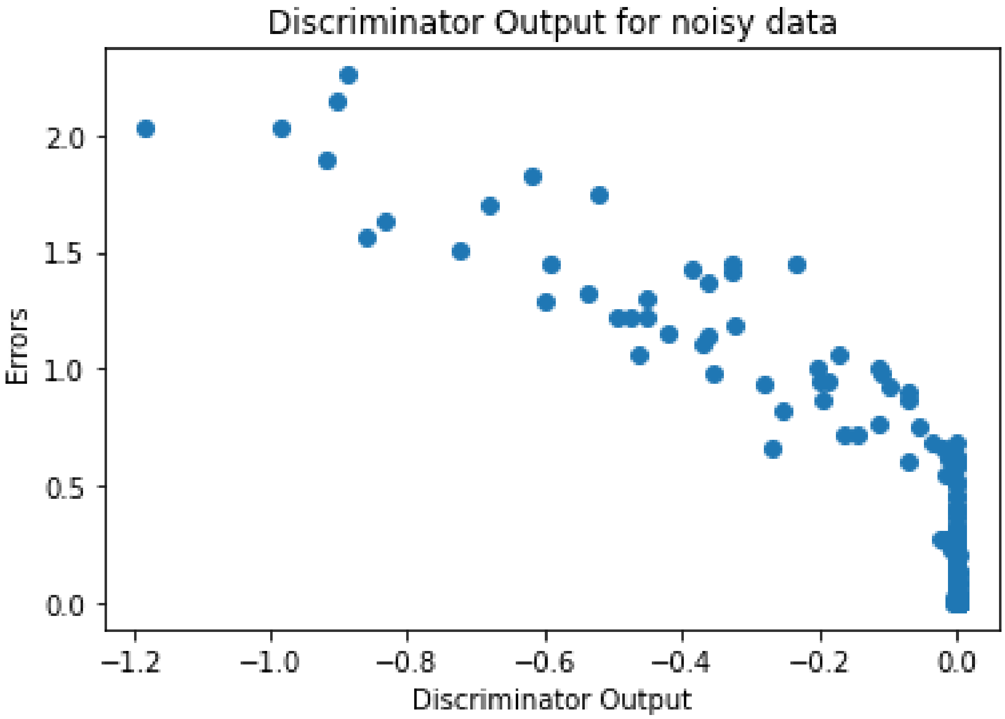
```
Mean Euclidean Distance: 0.4006078289315038
```



Sanity Checks

```
[19]: sanityChecks.discProbVsError(real_dataset,disc,device)
```





4.1 Visualization of trained GAN generator

```
[20]: for name, param in gen.named_parameters():
          print(name,param)
```

```
output.weight Parameter containing:
tensor([[-0.1273,  0.1445,  0.1260,  0.1917,  0.1734,  0.1449,  0.1160,  0.1020,
          0.1939,  0.2032,  0.1170,  0.5045]], requires_grad=True)
output.bias Parameter containing:
tensor([0.1310], requires_grad=True)
```