Dataset1-Regression_output_3

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 -0.693473 -0.307449  0.791028  0.282455 -1.055211 -1.355088 -0.350854
1 -0.779890 -0.032196  0.631653  1.094421  1.502106 -0.128212 -0.905099
2 -0.524418 -0.148843 -0.649982  0.671701 -0.101217  0.243394  0.060774
3  0.115101 -0.855718 -1.610841 -1.186042  0.310870  0.603865  0.663096
4 -0.901162  0.493394 -0.780245  0.491633  0.701921 -1.193419  0.338657

         X8        X9       X10           Y
0 -0.667922 -3.193338 -0.401429 -441.353463
1 -0.878307 -0.116382 -0.795225  -23.568801
2  1.630716 -1.137799 -0.173657   81.535440
3 -0.904360 -1.094860  1.398108    8.516448
4  1.223406  2.945688 -1.110324   74.190219
```

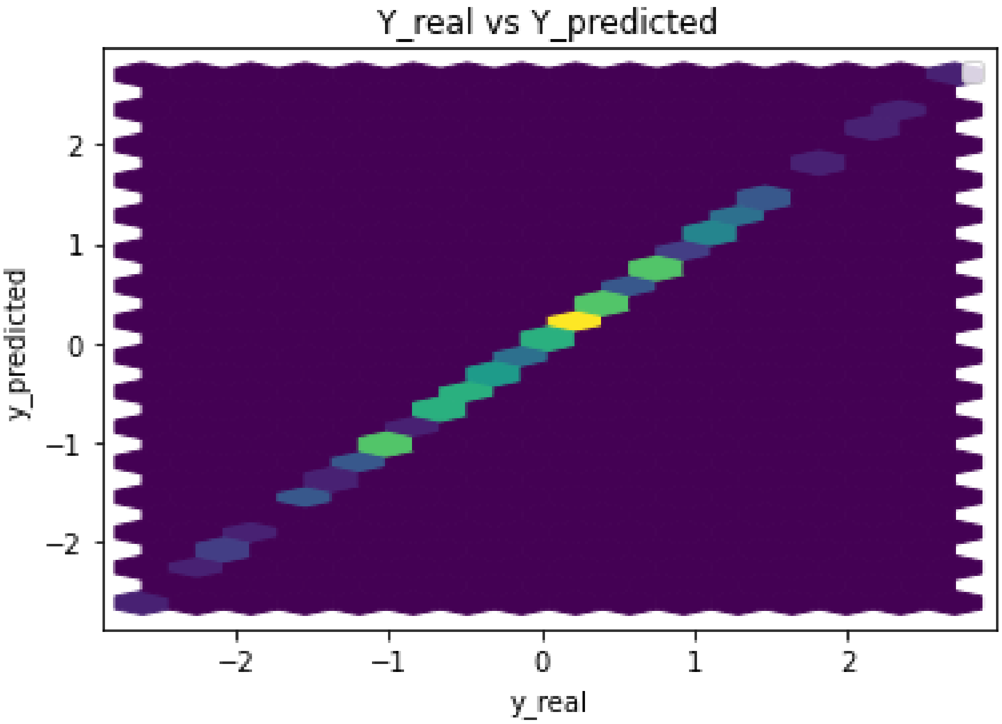
1.4 Stats Model

[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:                 3.690e+07
Date:                Thu, 07 Oct 2021   Prob (F-statistic):          6.62e-290
Time:                        07:38:08   Log-Likelihood:                 619.99
No. Observations:                 100   AIC:                            -1218.
Df Residuals:                      89   BIC:                            -1189.
Df Model:                          10
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const               0   5.21e-05          0      1.000      -0.000       0.000
x1             0.0783   5.62e-05   1393.326      0.000       0.078       0.078
x2             0.1867   5.43e-05   3440.984      0.000       0.187       0.187
x3             0.1275   5.51e-05   2316.365      0.000       0.127       0.128
x4             0.0228   5.37e-05    425.349      0.000       0.023       0.023
x5             0.4972   5.45e-05   9125.502      0.000       0.497       0.497
```

```
x6             0.4070   5.42e-05   7505.425      0.000       0.407       0.407
x7             0.0342   5.62e-05    608.894      0.000       0.034       0.034
x8             0.4806   5.43e-05   8850.908      0.000       0.480       0.481
x9             0.1968   5.42e-05   3630.169      0.000       0.197       0.197
x10            0.4621   5.25e-05   8806.925      0.000       0.462       0.462
==============================================================================
Omnibus:                        2.642   Durbin-Watson:                   1.743
Prob(Omnibus):                  0.267   Jarque-Bera (JB):                2.045
Skew:                           0.320   Prob(JB):                        0.360
Kurtosis:                       3.286   Cond. No.                         1.65
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
Parameters:  const    0.000000
x1       0.078348
x2       0.186728
x3       0.127532
x4       0.022831
x5       0.497220
x6       0.407050
x7       0.034237
x8       0.480572
x9       0.196806
x10      0.462082
dtype: float64
```



```
Performance Metrics
Mean Squared Error: 2.4117105686728074e-07
Mean Absolute Error: 0.0003886528591217828
Manhattan distance: 0.03886528591217828
Euclidean distance: 0.004910916990413101
```

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]:

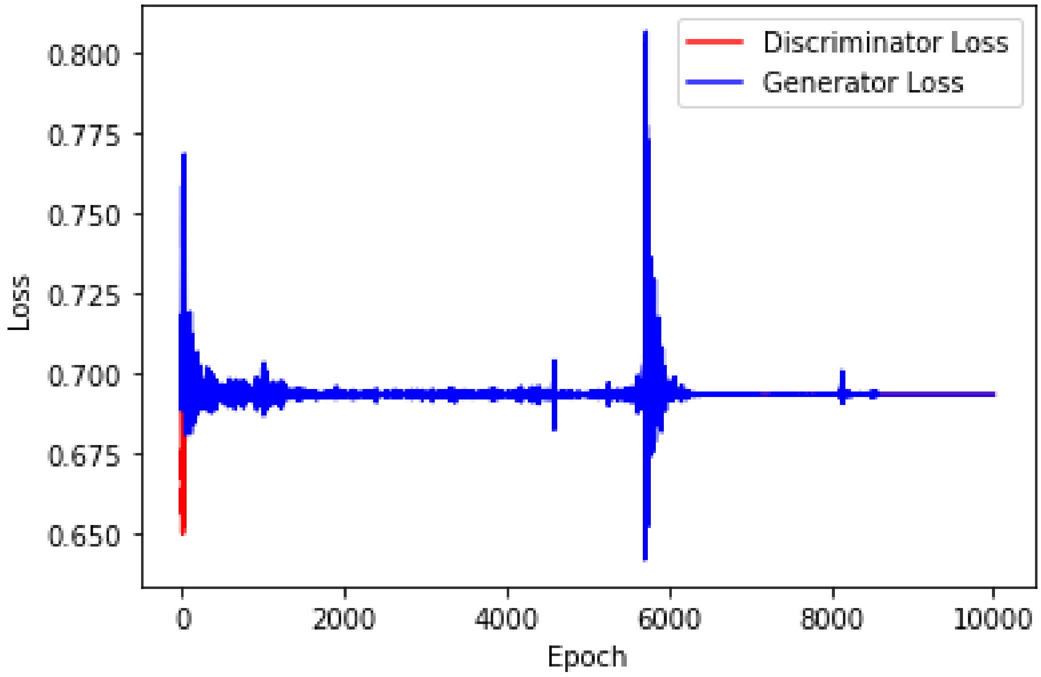
```
n_epochs = 5000
batch_size = sample_size//2
```

[12]:

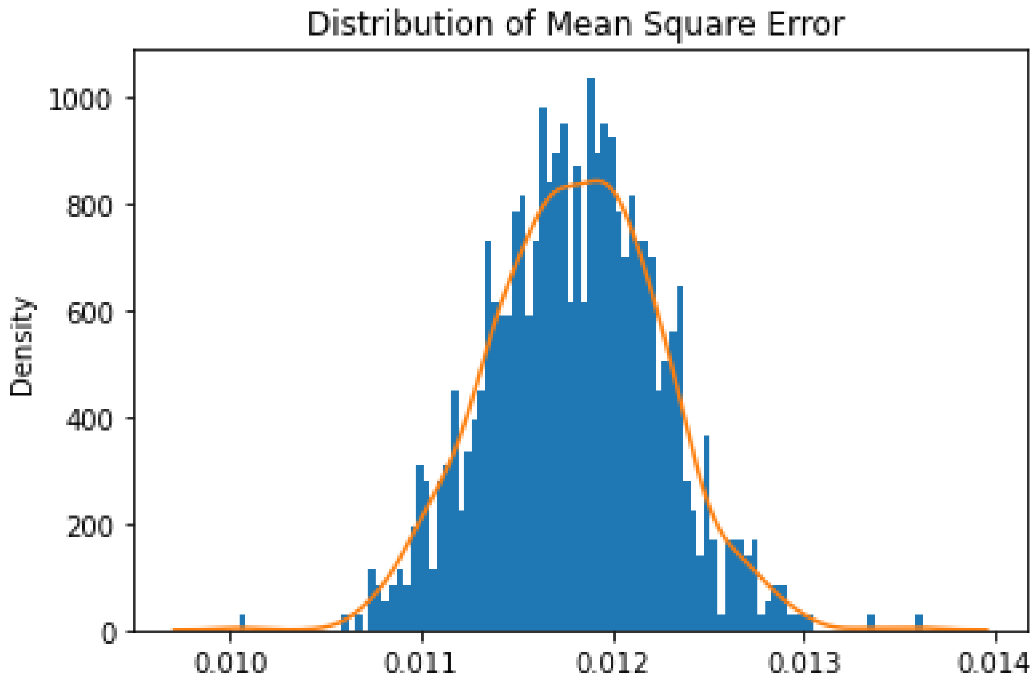
```
# Parameters
sample_size = 1000000
std = 1
mean = 1
```

[13]:

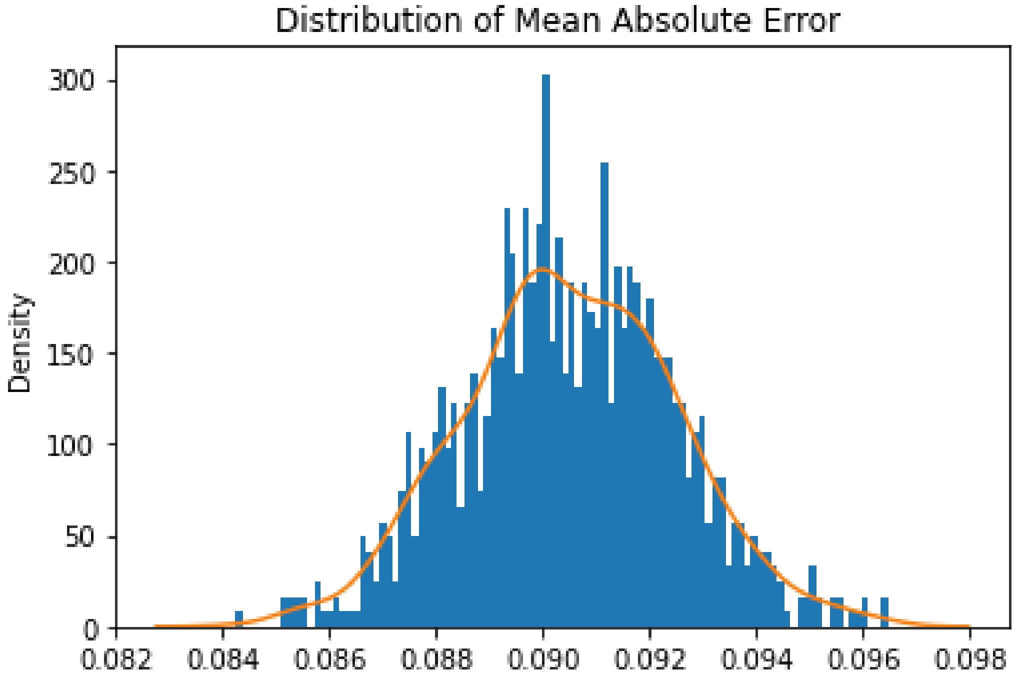
```
train_test.
 ↪training_GAN(discriminator,generator,disc_opt,gen_opt,real_dataset,batch_size,␣
 ↪n_epochs,criterion,device)
```



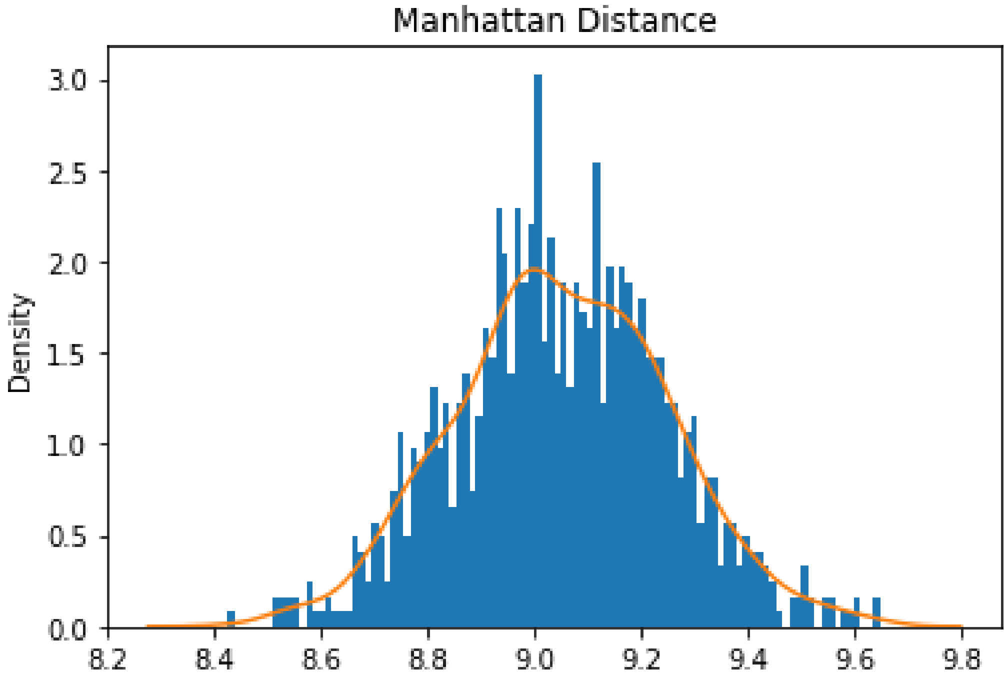
```
[14]: train_test.test_generator(generator,real_dataset,device)
```



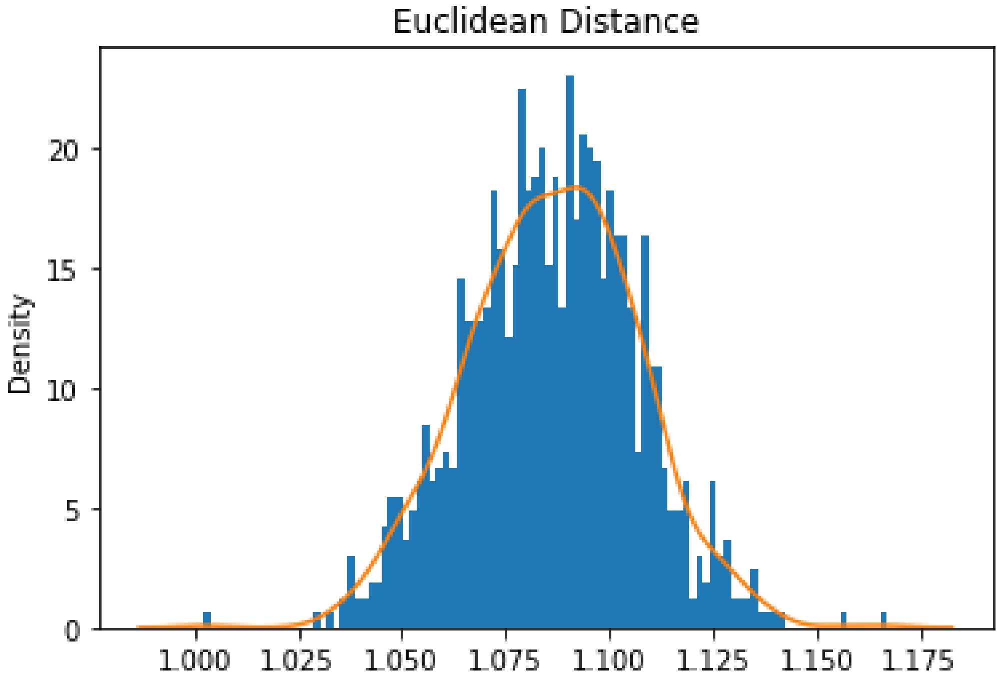
```
Mean Square Error: 0.011805442770057626
```



```
Mean Absolute Error: 0.09050252705901861
```



```
Mean Manhattan Distance: 9.050252705901862
```



```
Mean Euclidean Distance: 9.050252705901862
```

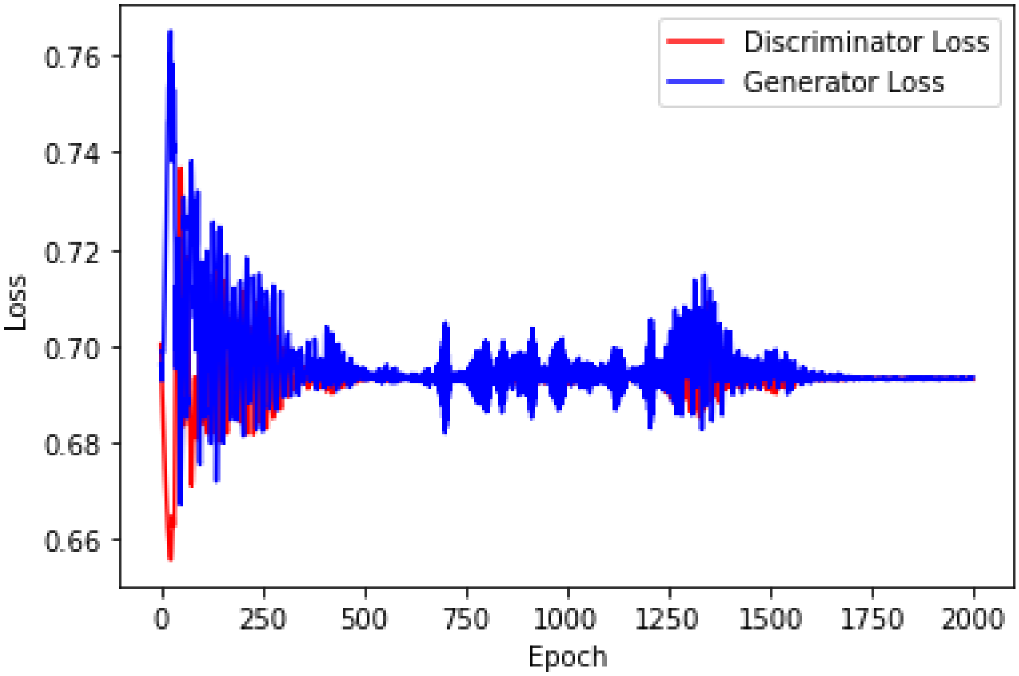
4 ABC GAN Model

Training the network

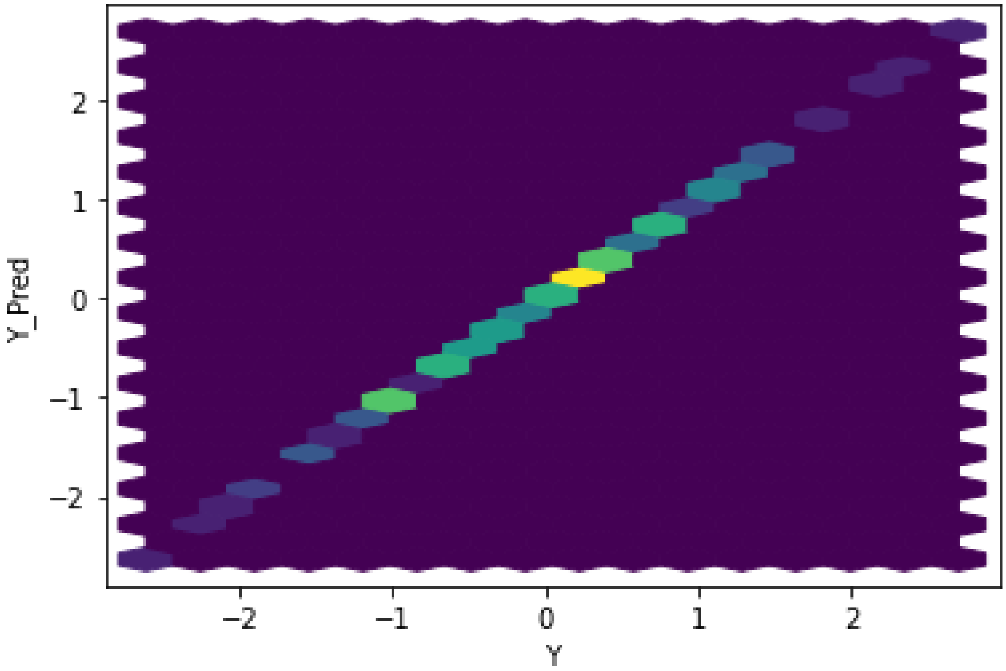
[15]:
```
gen = Generator(n_features+2)
disc = Discriminator(n_features+2,hidden_nodes)

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))
```

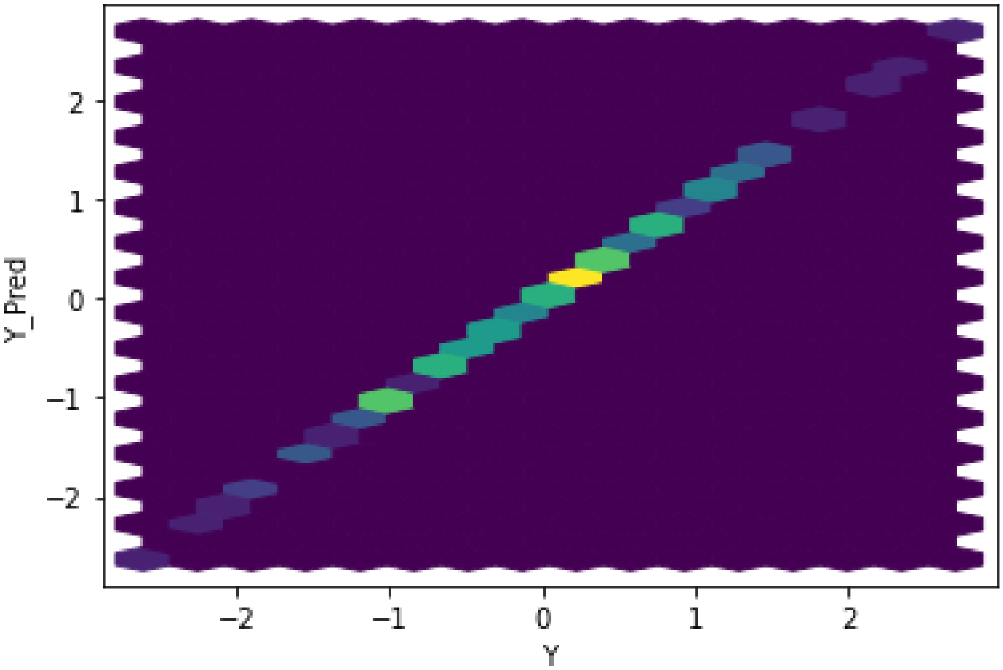
[16]:
```
n_epoch_abc = 2000
batch_size = sample_size//2
```

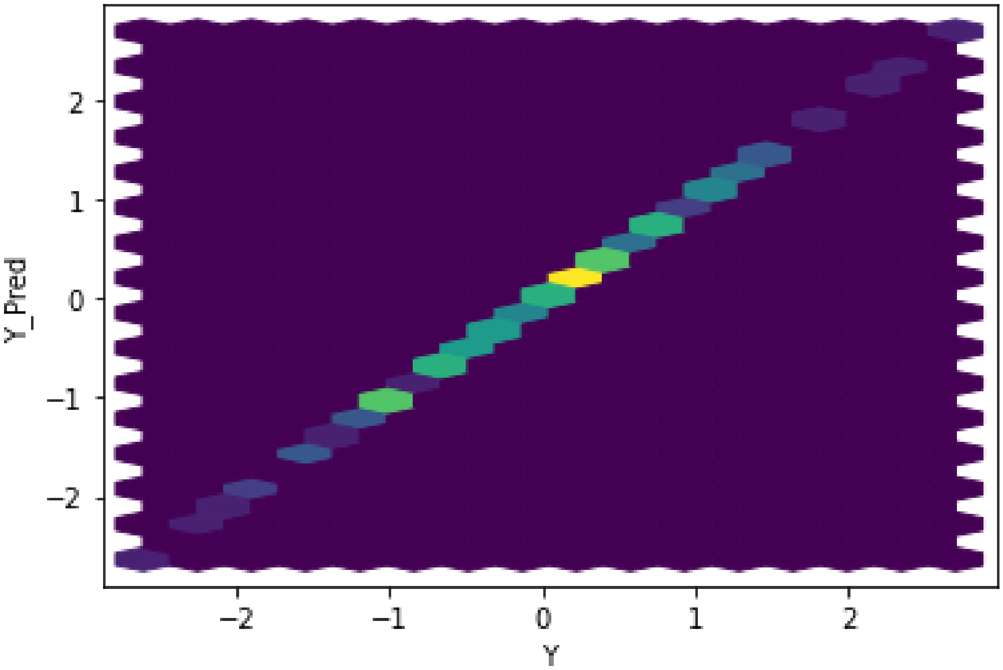
[17]:
```
ABC_train_test.training_GAN(disc, gen,disc_opt,gen_opt,real_dataset,␣
 ↪batch_size, n_epoch_abc,criterion,coeff,mean,variance,device)
```

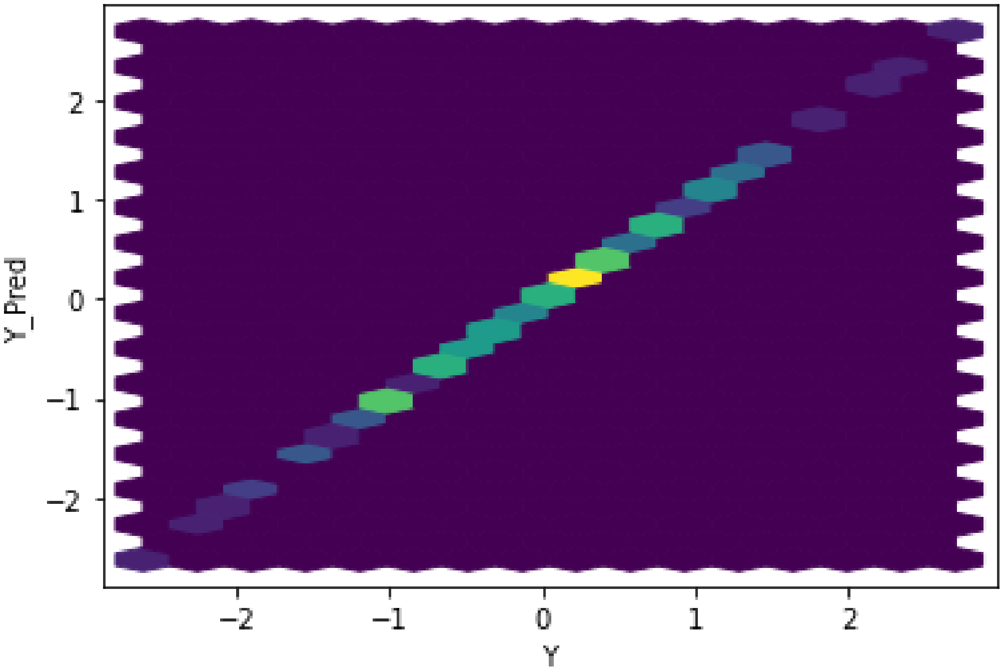


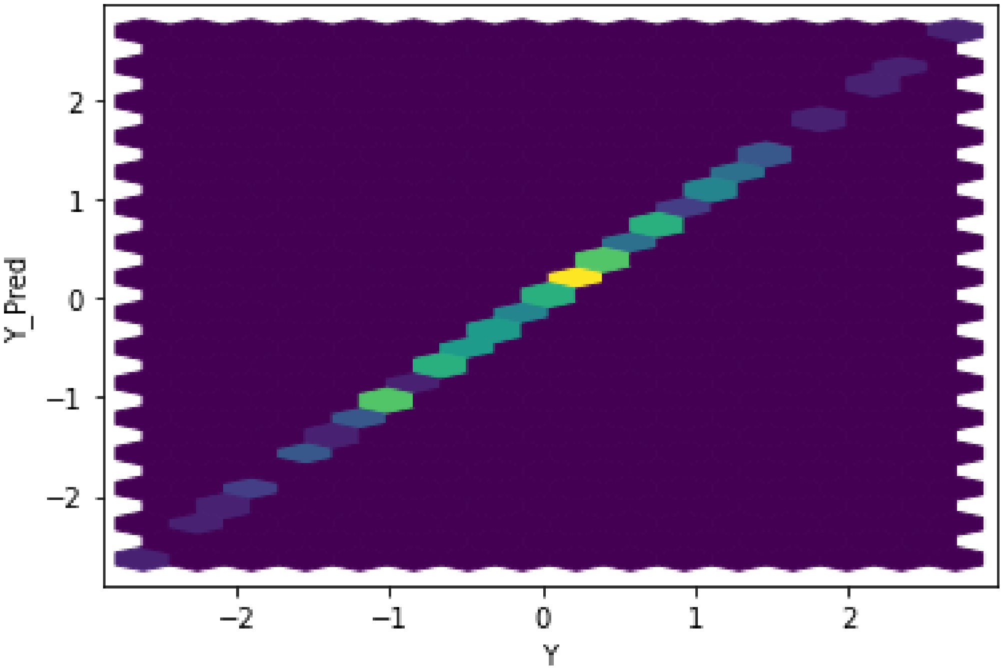
[18]:
```
ABC_train_test.test_generator(gen,real_dataset,coeff,mean,variance,device)
```

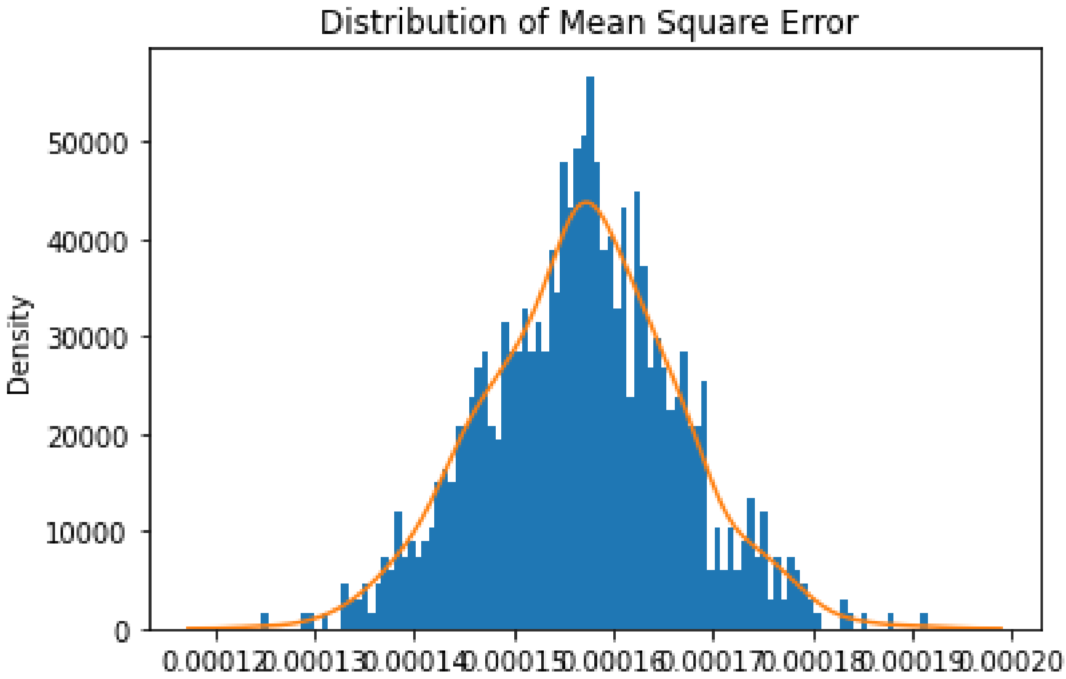



2
1
0
−1
−2
Y_Pred
−2
−1
0
1
2
Y

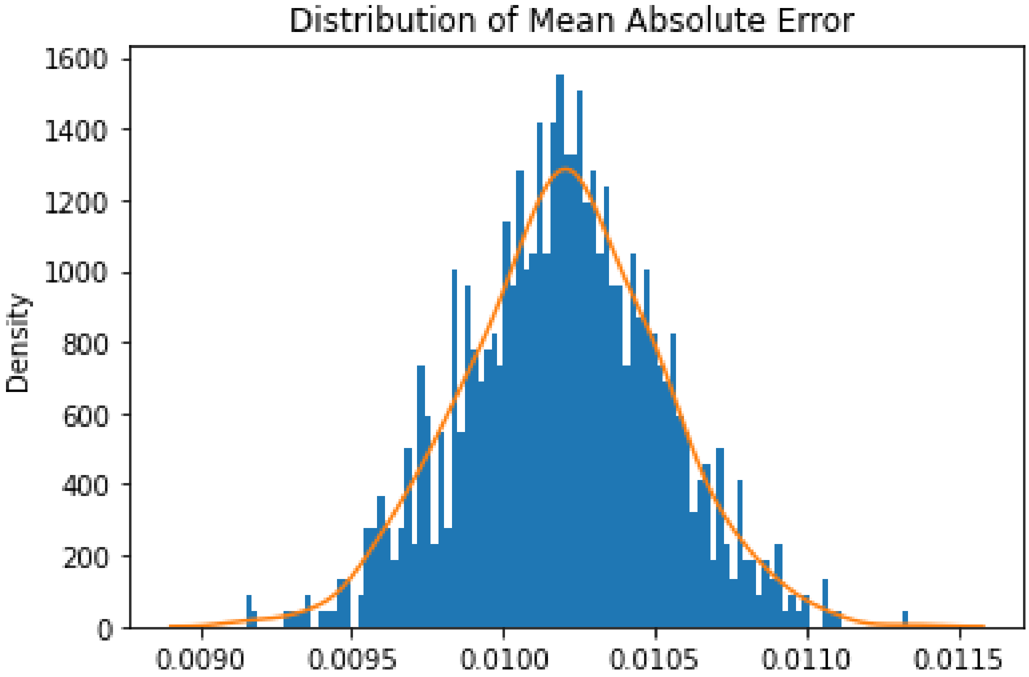

2
1
0
−1
−2
Y_Pred
−2
−1
0
1
2
Y


Y_Pred
2
1
0
−1
−2
−2
−1
0
1
2
Y

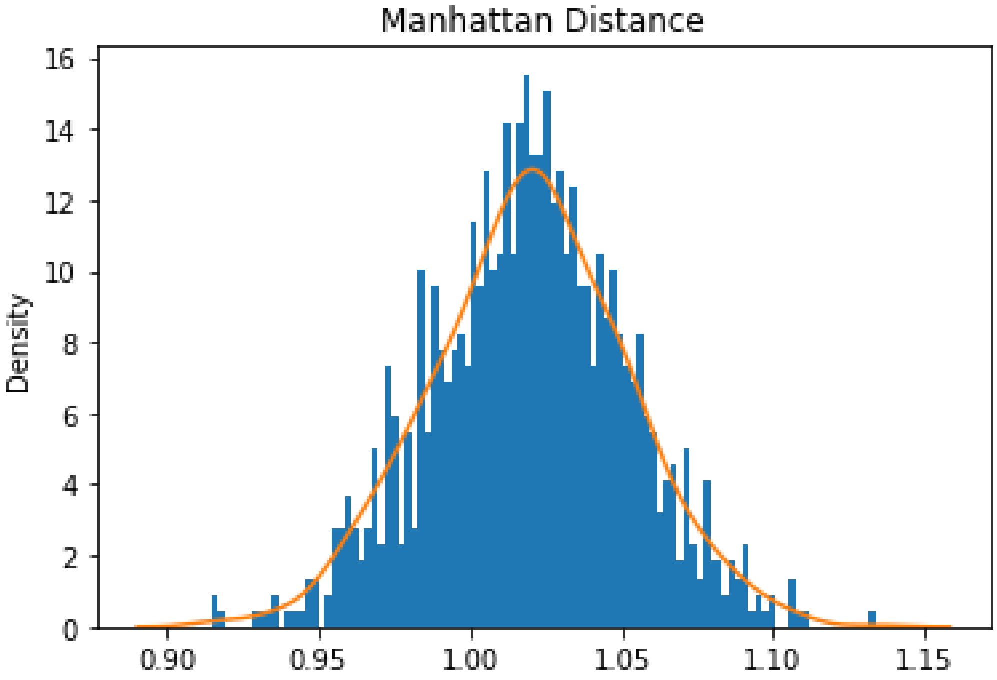

Y_Pred
2
1
0
−1
−2
−2
−1
0
1
2
Y



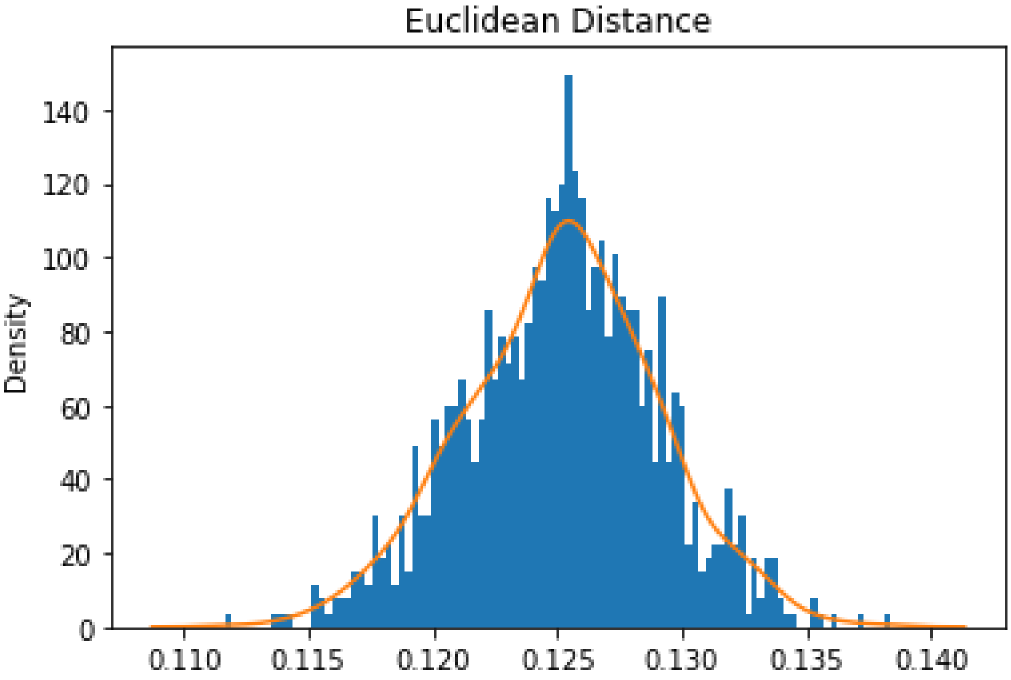
```
Mean Square Error: 0.0001568571731029724
```



```
Mean Absolute Error: 0.010196051908209092
Mean Manhattan Distance: 1.019605190820992
```

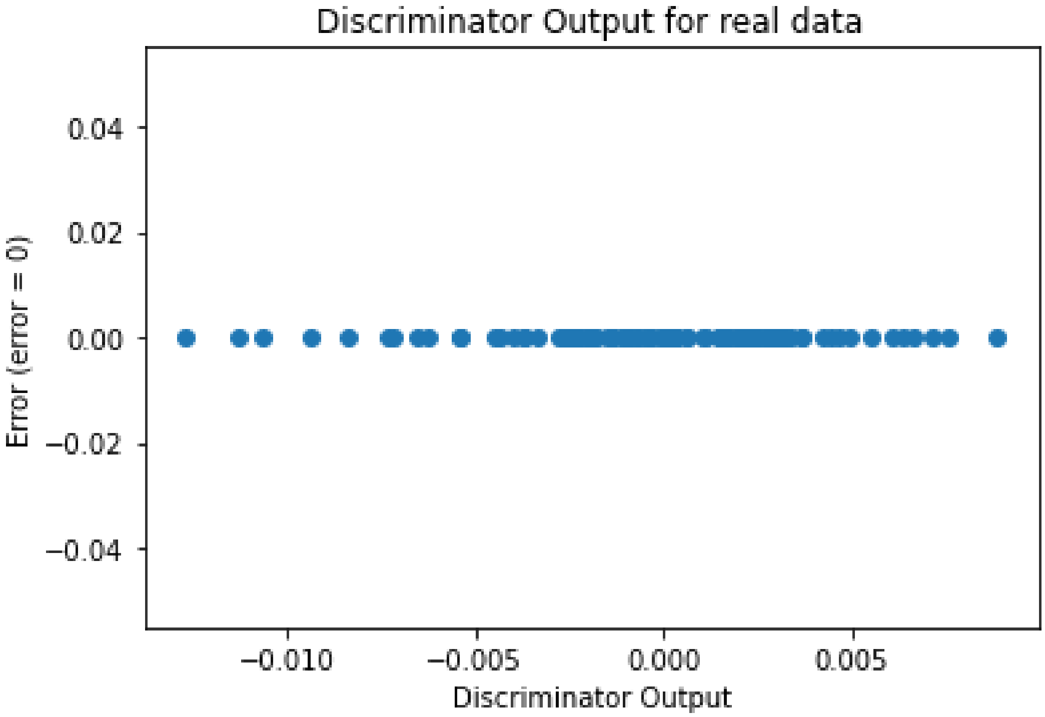


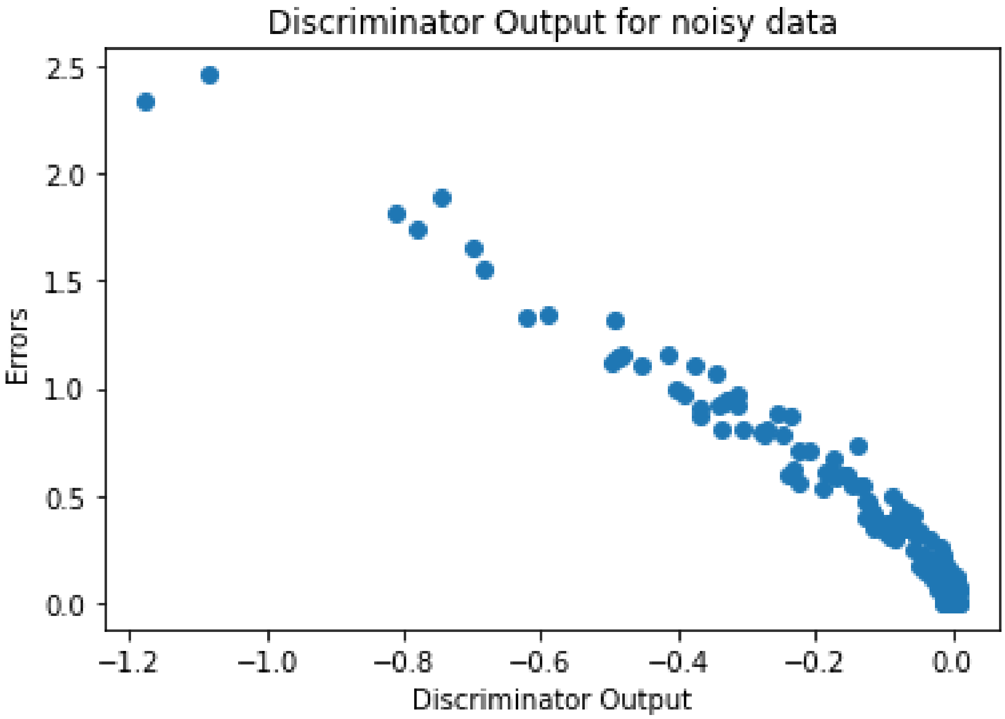
```
Mean Euclidean Distance: 0.1251822410135943
```



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([[-0.1036,  0.0498,  0.1130,  0.0752,  0.0110,  0.3089,  0.2498,  0.0264,
          0.3007,  0.1203,  0.2952,  0.3769]], requires_grad=True)
output.bias Parameter containing:
tensor([0.1028], requires_grad=True)
```