Dataset3-Boston_Housing_output_4

October 7, 2021

1 Dataset 3 - Boston Housing

1.1 Parameters

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) 3. prior: 0 (Correct) or 1 (Misspecified)

[1]:
```
#ABC_Generator
std = 1
mean = 1
prior = 0
```

[2]:
```
# Parameters
std = 1
mean = 0
```

1.2 Import Libraries and Dataset

[3]:
```
import warnings
warnings.filterwarnings('ignore')
```

[4]:
```
import statsModel
import sanityChecks
import bostonDataset
import ABC_train_test
import dataset
import train_test
import torch
from torch import nn
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from statistics import mean
import pandas as pd
from sklearn import preprocessing
from sklearn.datasets import load_boston
%matplotlib inline
```

1.2.1 Dataset

[5]:
```
X,Y = bostonDataset.boston_data()
n_features = 13
```

```
      CRIM    ZN  INDUS  CHAS    NOX     RM   AGE     DIS  RAD    TAX  \
0  0.00632  18.0   2.31   0.0  0.538  6.575  65.2  4.0900  1.0  296.0
1  0.02731   0.0   7.07   0.0  0.469  6.421  78.9  4.9671  2.0  242.0
2  0.02729   0.0   7.07   0.0  0.469  7.185  61.1  4.9671  2.0  242.0
3  0.03237   0.0   2.18   0.0  0.458  6.998  45.8  6.0622  3.0  222.0
4  0.06905   0.0   2.18   0.0  0.458  7.147  54.2  6.0622  3.0  222.0

   PTRATIO       B  LSTAT  TARGET
0     15.3  396.90   4.98    24.0
1     17.8  396.90   9.14    21.6
2     17.8  392.83   4.03    34.7
3     18.7  394.63   2.94    33.4
4     18.7  396.90   5.33    36.2
```

1.3 Stats Model

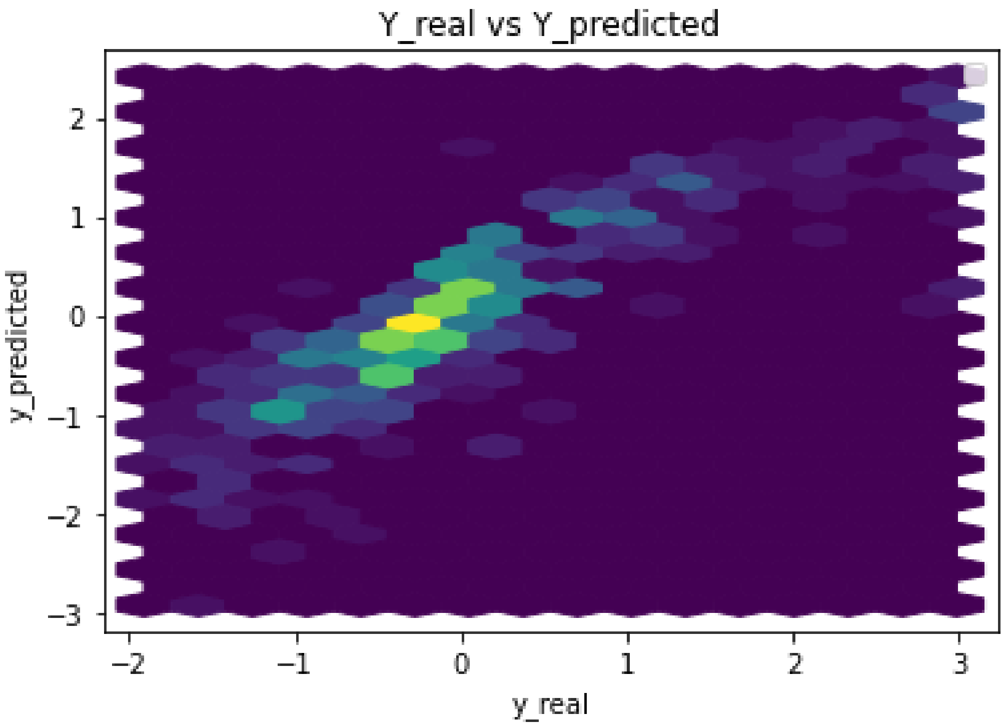
The statistical model is assumed to be $Y = \beta X + \mu$ where $\mu \sim N(0,1)$

To analyze the performance of the statistical model, we plot a graph of y_{real} vs y_{pred} and calculate performance metrics like mean squared error, mean absolute error, manhattan distance and euclidean distance between y_{real} and y_{pred}

[6]:
```
[coeff,y_pred] = statsModel.statsModel(X,Y)
```

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

                            OLS Regression Results
==============================================================================
Dep. Variable:                 TARGET   R-squared:                       0.741
Model:                            OLS   Adj. R-squared:                  0.734
Method:                 Least Squares   F-statistic:                     108.1
Date:                Thu, 07 Oct 2021   Prob (F-statistic):          6.72e-135
Time:                        14:29:22   Log-Likelihood:                -376.55
No. Observations:                 506   AIC:                             781.1
Df Residuals:                     492   BIC:                             840.3
Df Model:                          13
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const      -5.235e-16      0.023  -2.28e-14      1.000      -0.045       0.045
x1            -0.1010      0.031     -3.287      0.001      -0.161      -0.041
x2             0.1177      0.035      3.382      0.001       0.049       0.186
x3             0.0153      0.046      0.334      0.738      -0.075       0.105
x4             0.0742      0.024      3.118      0.002       0.027       0.121
```

```
x5            -0.2238      0.048     -4.651      0.000      -0.318      -0.129
x6             0.2911      0.032      9.116      0.000       0.228       0.354
x7             0.0021      0.040      0.052      0.958      -0.077       0.082
x8            -0.3378      0.046     -7.398      0.000      -0.428      -0.248
x9             0.2897      0.063      4.613      0.000       0.166       0.413
x10           -0.2260      0.069     -3.280      0.001      -0.361      -0.091
x11           -0.2243      0.031     -7.283      0.000      -0.285      -0.164
x12            0.0924      0.027      3.467      0.001       0.040       0.145
x13           -0.4074      0.039    -10.347      0.000      -0.485      -0.330
==============================================================================
Omnibus:                      178.041   Durbin-Watson:                   1.078
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              783.126
Skew:                           1.521   Prob(JB):                    8.84e-171
Kurtosis:                       8.281   Cond. No.                         9.82
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
Parameters:  const   -5.234528e-16
x1      -1.010171e-01
x2       1.177152e-01
x3       1.533520e-02
x4       7.419883e-02
x5      -2.238480e-01
x6       2.910565e-01
x7       2.118638e-03
x8      -3.378363e-01
x9       2.897491e-01
x10     -2.260317e-01
x11     -2.242712e-01
x12      9.243223e-02
x13     -4.074469e-01
dtype: float64
```




```
Performance Metrics
Mean Squared Error: 0.2593573358905906
Mean Absolute Error: 0.35599245764784004
Manhattan distance: 180.13218356980693
Euclidean distance: 11.455776357830965
```

1.4 Generator and Discriminator Networks

1.4.1 Discriminator

[7]:
```python
class Discriminator(nn.Module):
  def __init__(self,n_input):
    super().__init__()
    self.hidden = nn.Linear(n_input,10)
    self.output = nn.Linear(10,1)
    self.sigmoid = nn.Sigmoid()
    self.leakyRelu = nn.LeakyReLU()

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

1.4.2 Generator

[8]:
```
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
```

1.4.3 ABC Pre 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

[9]:
```
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
```

1.5 GAN Model

We are using a Conditional GAN network as a baseline. The input to the GAN generator is (X,z) where X are the features of the dataset and z is gaussian noise

[10]:
```
real_dataset = dataset.CustomDataset(X,Y)
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

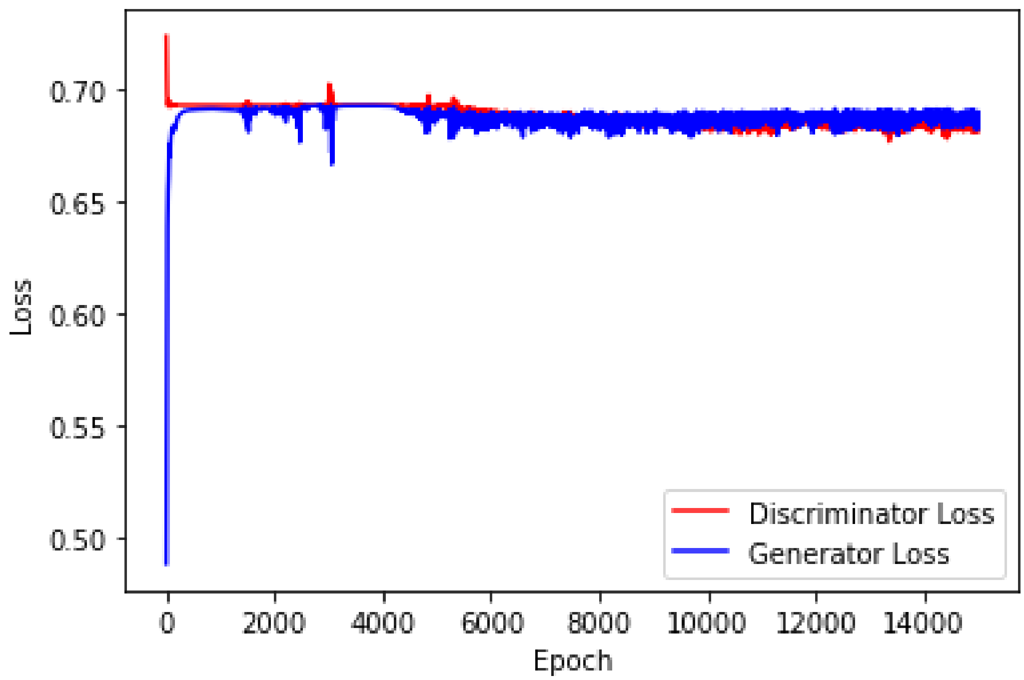
[11]:
```
generator = Generator(n_features+2)
discriminator = Discriminator(n_features+2)

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

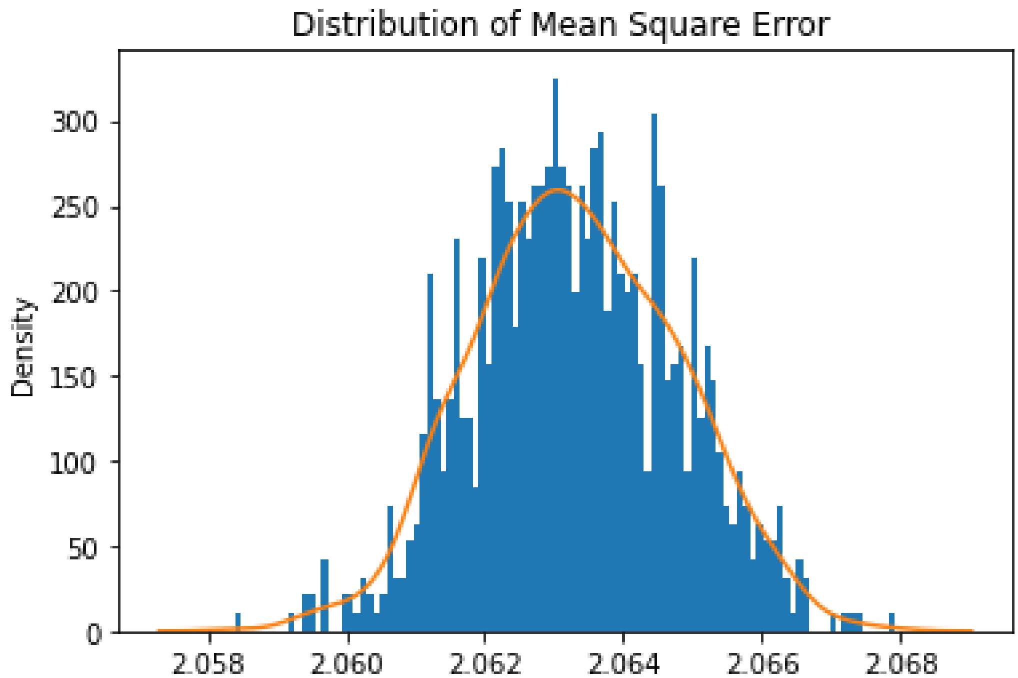
```
Discriminator(
  (hidden): Linear(in_features=15, out_features=10, bias=True)
  (output): Linear(in_features=10, out_features=1, bias=True)
  (sigmoid): Sigmoid()
  (leakyRelu): LeakyReLU(negative_slope=0.01)
)
Generator(
  (output): Linear(in_features=15, out_features=1, bias=True)
)
```

[12]:
```
sample_size = len(real_dataset)
n_epochs = 15000
batch_size = sample_size
```

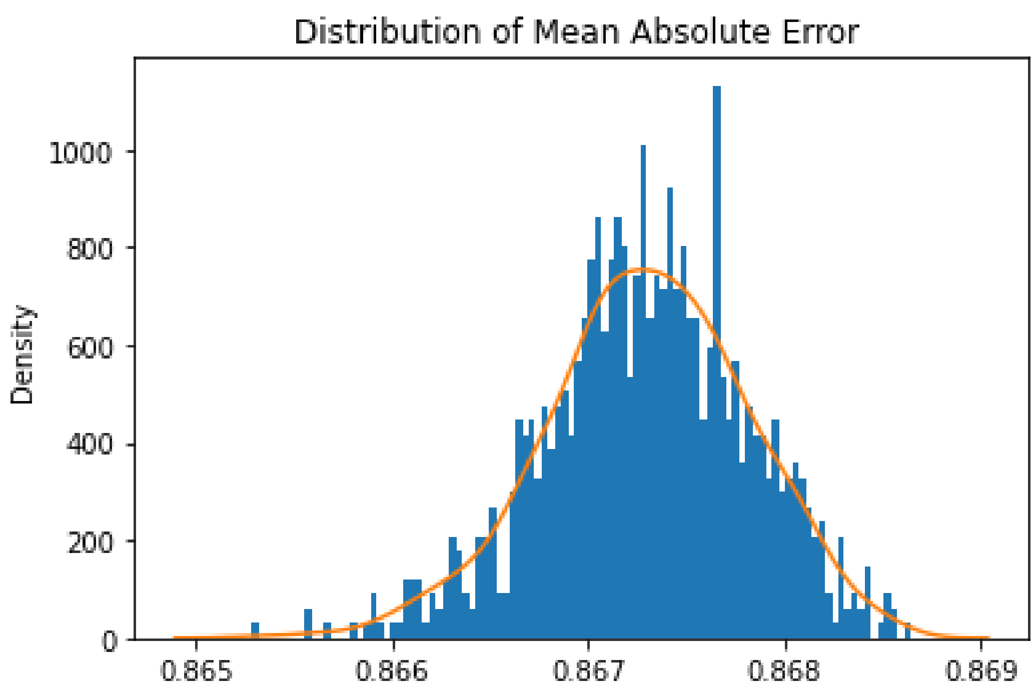
[13]:
```
train_test.
 ↪training_GAN(discriminator,generator,disc_opt,gen_opt,real_dataset,batch_size,␣
 ↪n_epochs,criterion,device)
```



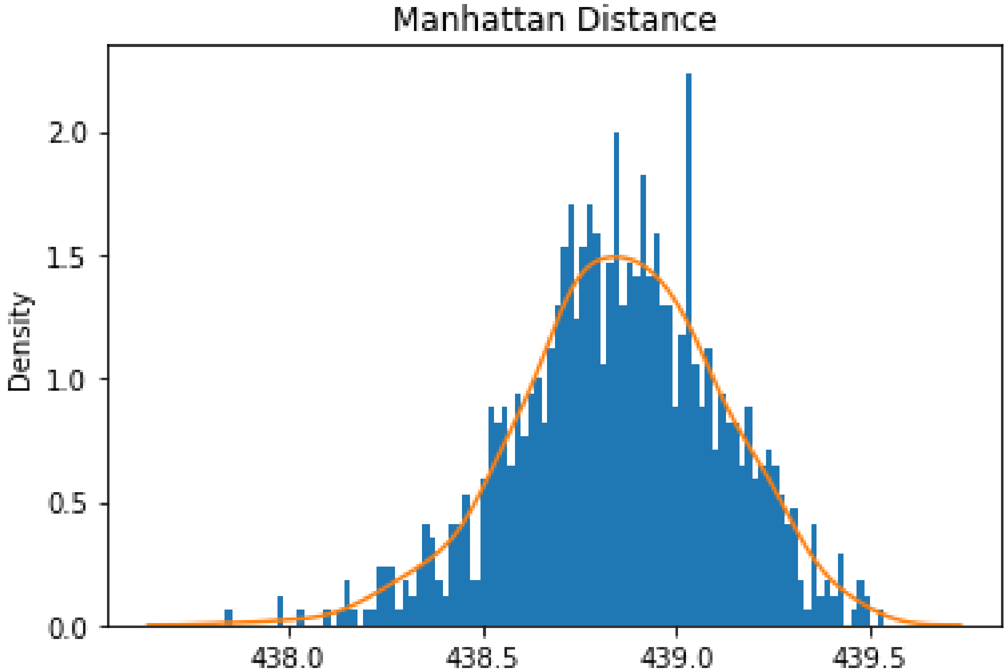
[14]:
```
train_test.test_generator(generator,real_dataset,device)
```



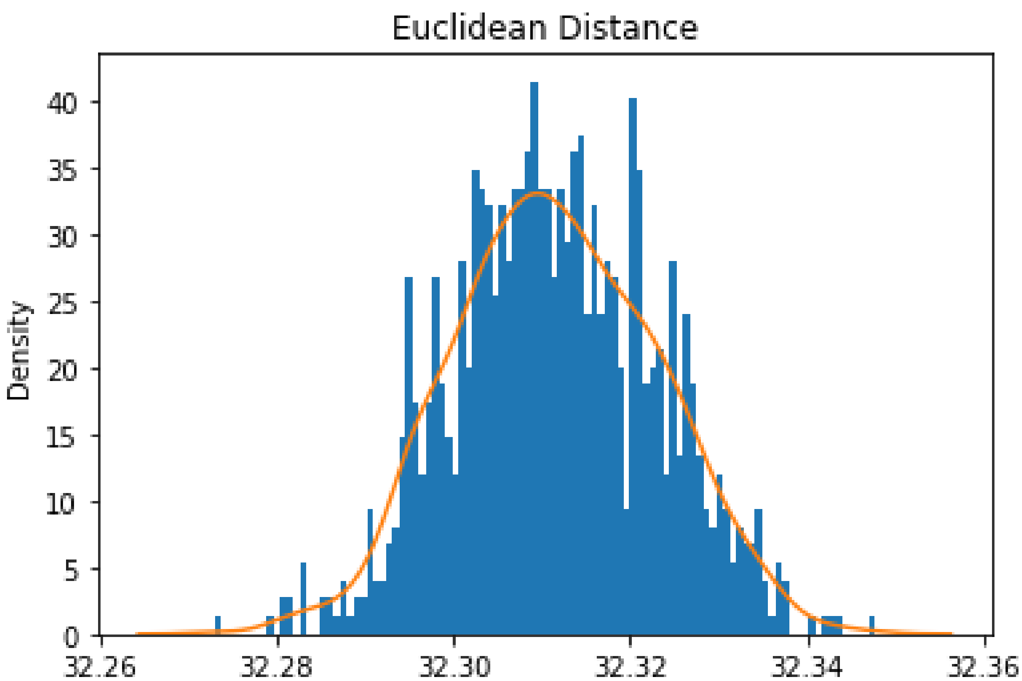

```
Mean Square Error: 2.063314873633221
```



```
Mean Absolute Error: 0.867299038175015
```




```
Mean Manhattan Distance: 438.85331331655755
```



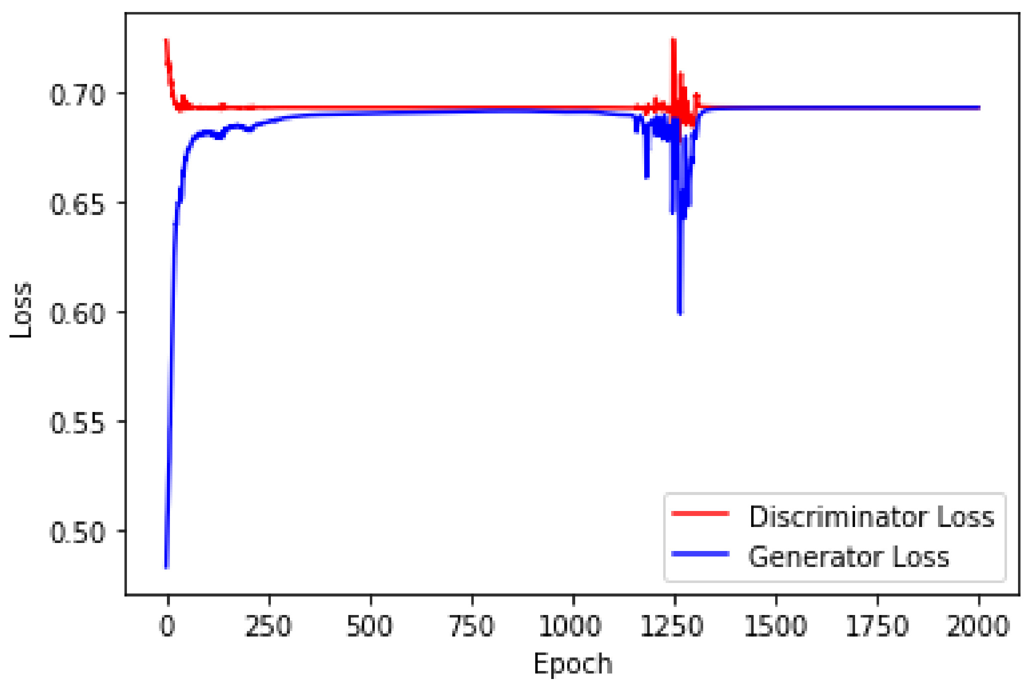
```
Mean Euclidean Distance: 438.85331331655755
```

2 ABC GAN Model

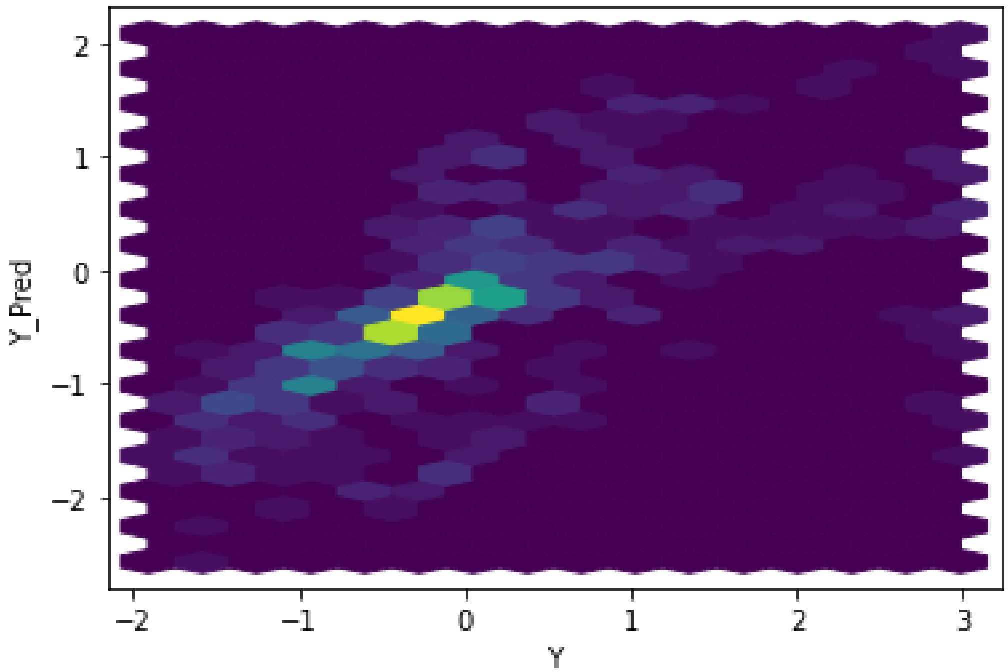
[15]:
```
gen = Generator(n_features+2)
disc = Discriminator(n_features+2)

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))
```

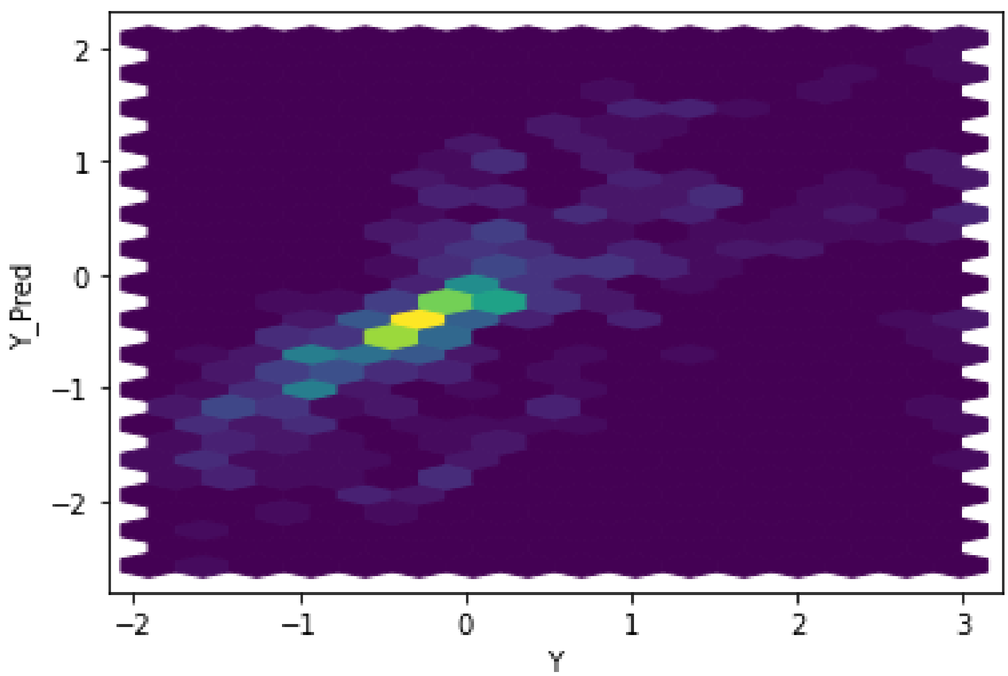
[16]:
```
n_epoch_abc = 2000
batch_size = sample_size
```

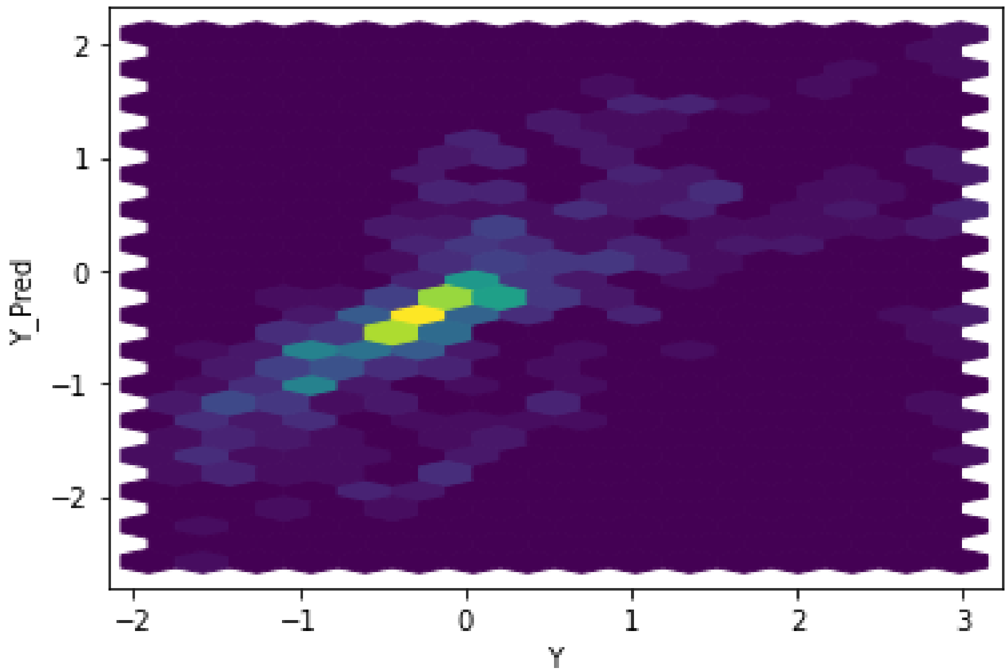
[17]:
```
ABC_train_test.training_GAN(disc, gen,disc_opt,gen_opt,real_dataset,␣
↪batch_size, n_epoch_abc,criterion,coeff,mean,std,device)
```



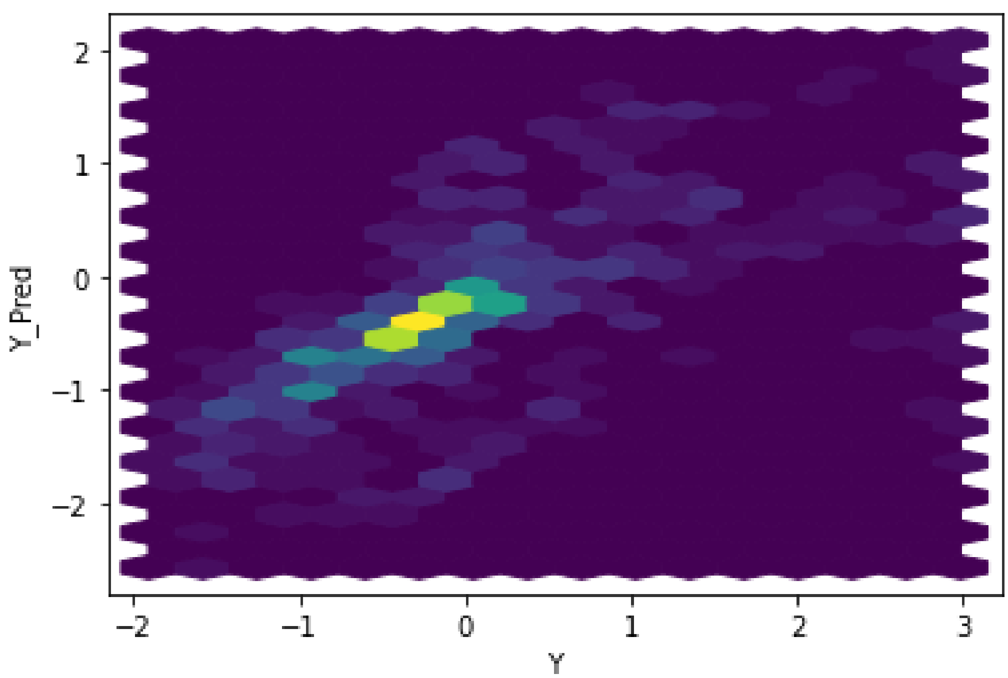
[18]:
```
ABC_train_test.test_generator(gen,real_dataset,coeff,mean,std,device)
```


Y_Pred
Y
2
1
0
−1
−2
−2
−1
0
1
2
3

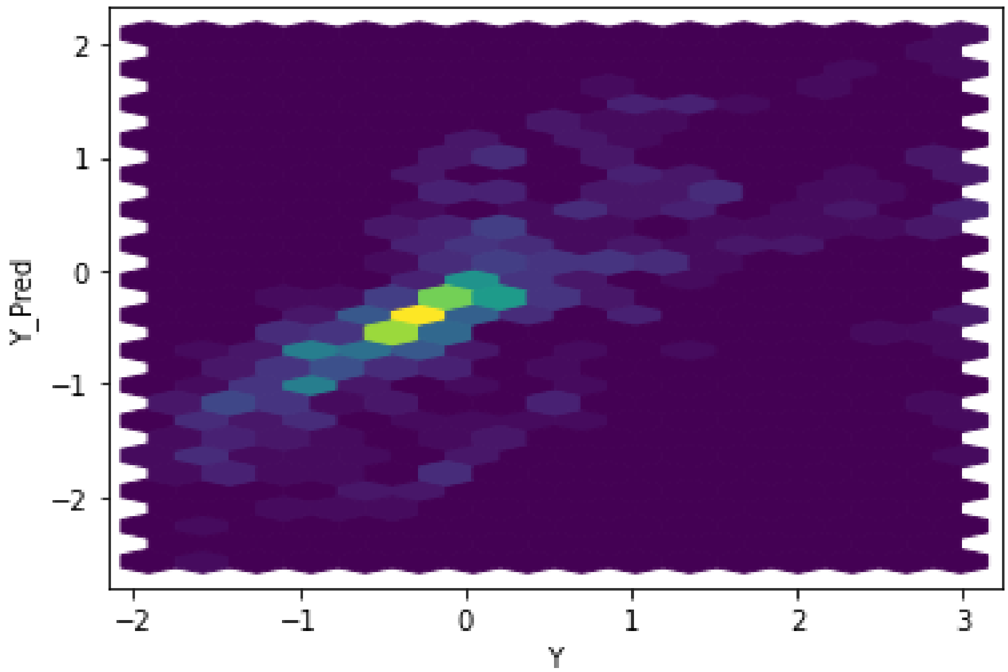

Y_Pred
Y
2
1
0
−1
−2
−2
−1
0
1
2
3

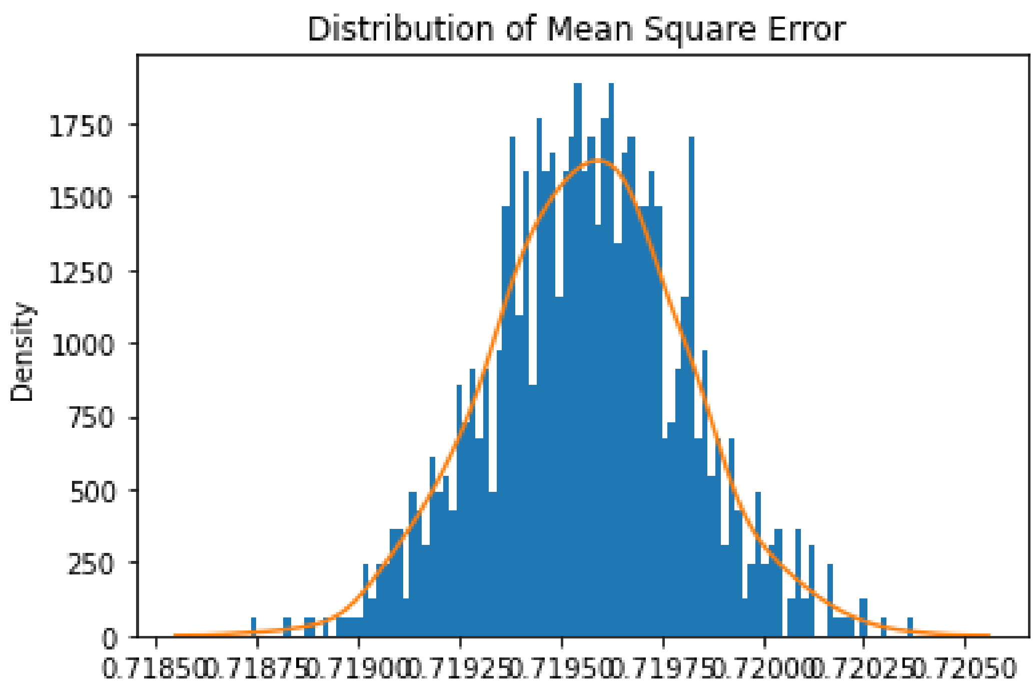


2
1
0
−1
−2
Y_Pred
−2
−1
0
1
2
3
Y

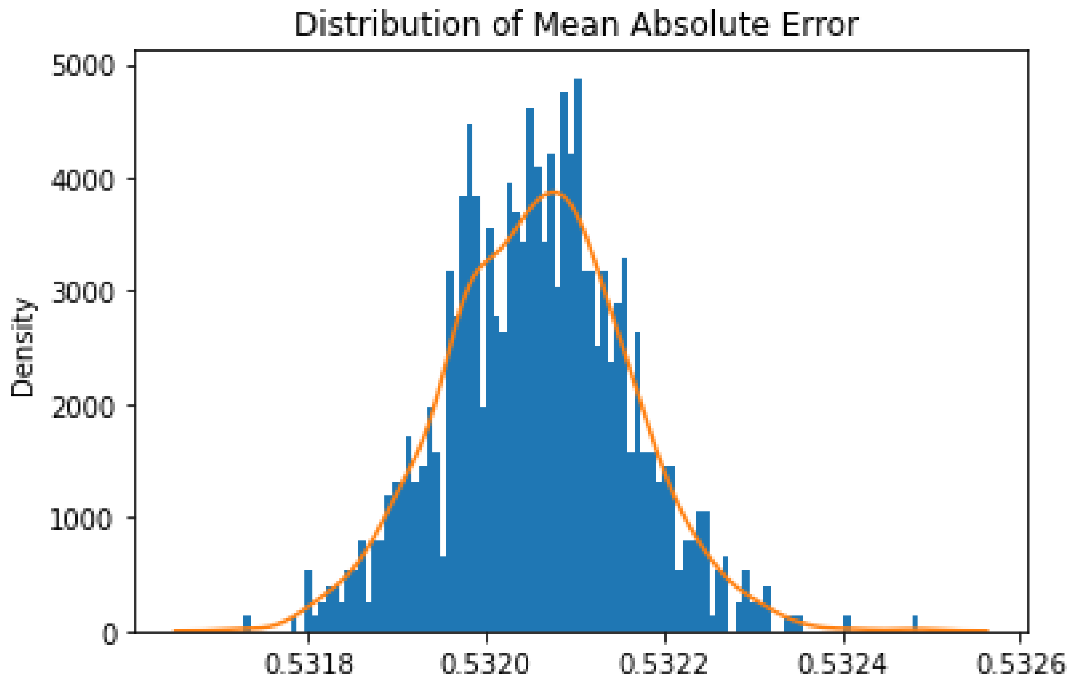


2
1
0
−1
−2
Y_Pred
−2
−1
0
1
2
3
Y

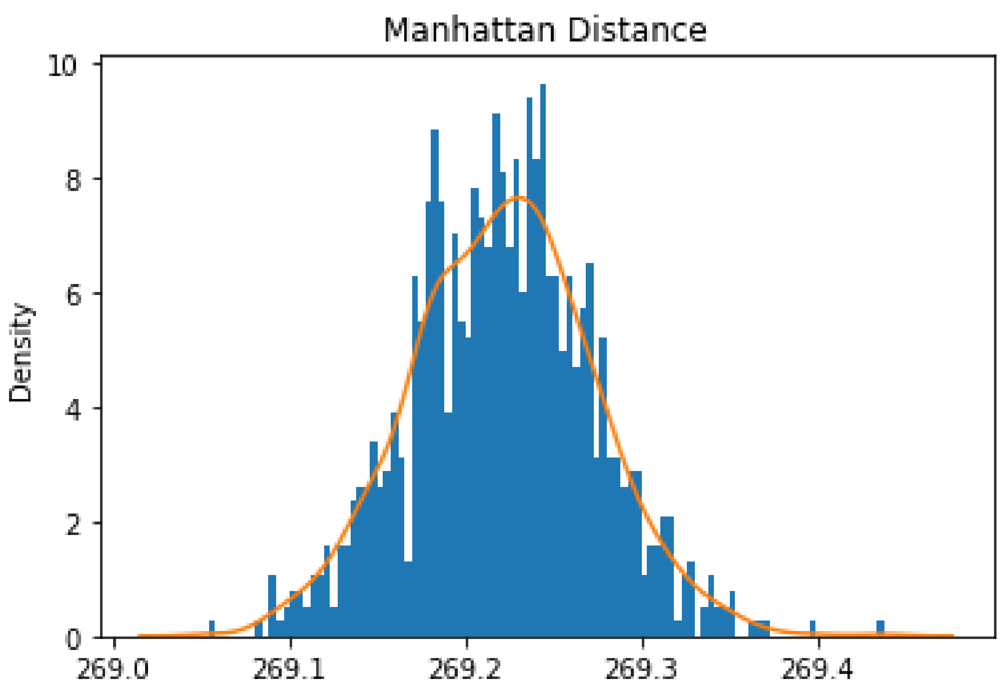




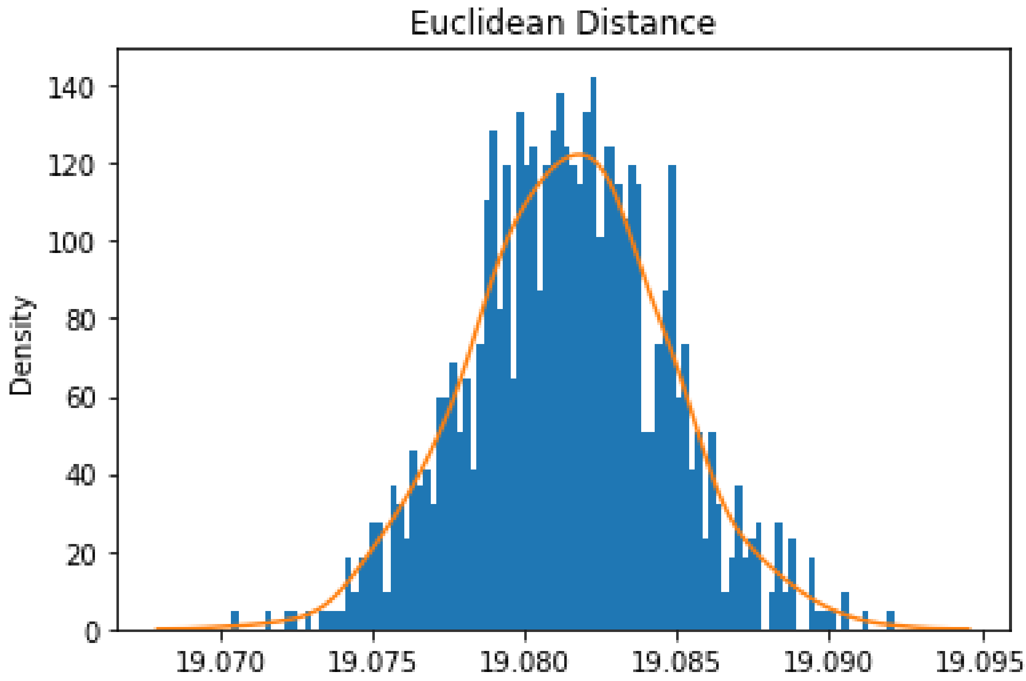

```
Mean Square Error: 0.7195666207058606
```




```
Mean Absolute Error: 0.5320579777624727
Mean Manhattan Distance: 269.22133674781116
```



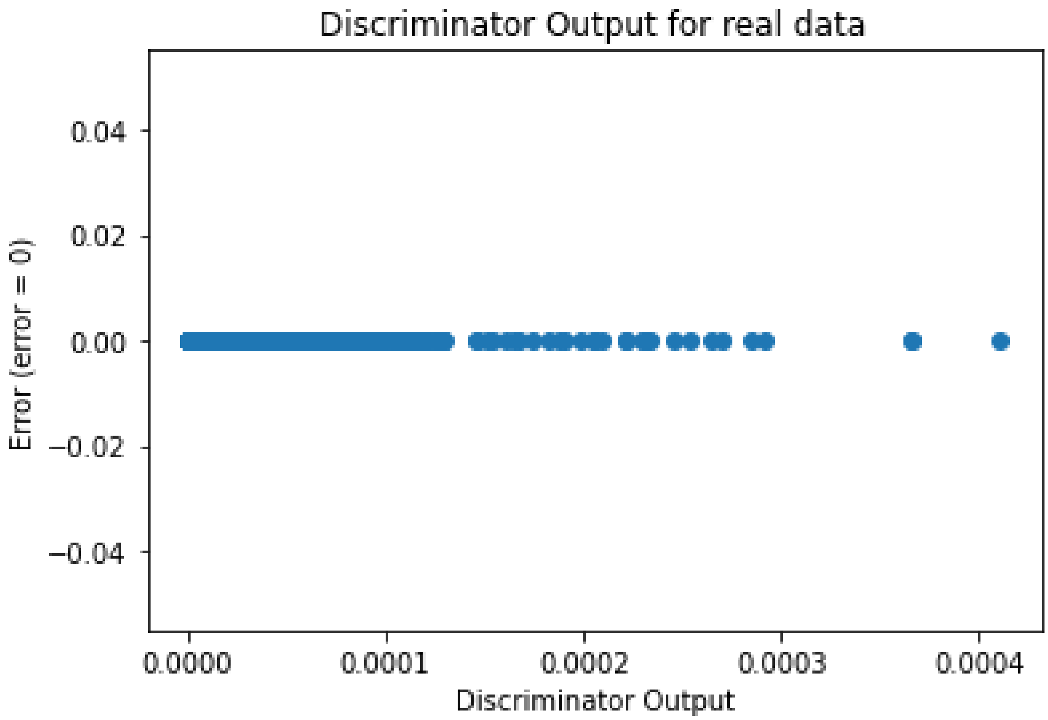
```
Mean Euclidean Distance: 19.08142289330327
```

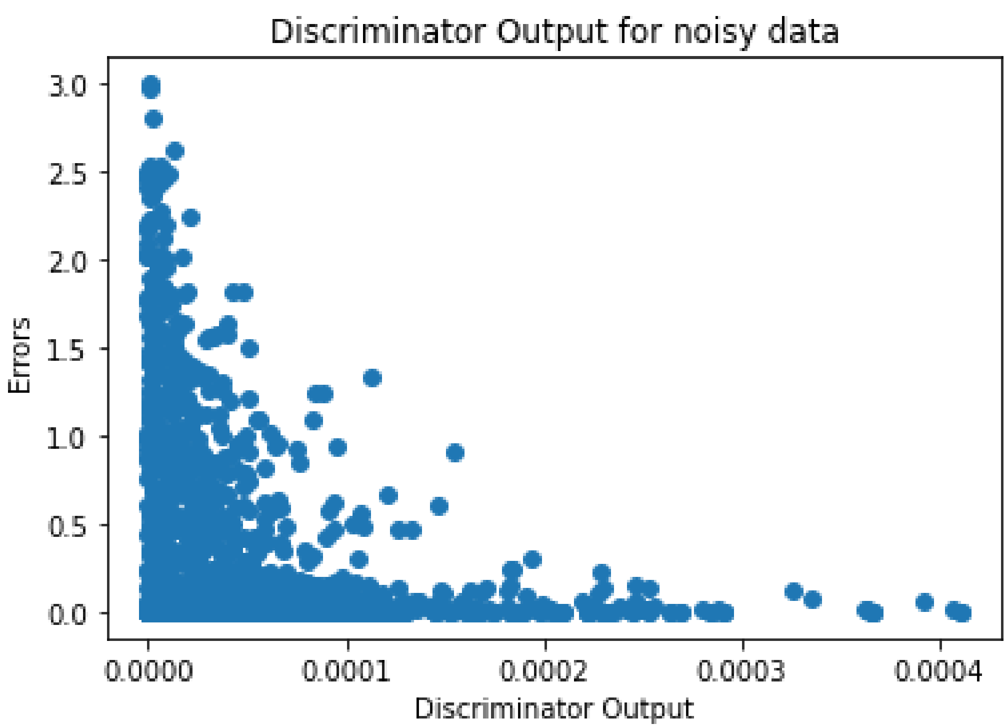


2.0.1 Sanity Check

We plot the discriminator output vs the noise in the input to verify that the discriminator functions correctly. We expect that discriminator output and noise are inversely proportional

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


Discriminator Output for real data
Error (error = 0)
0.04
0.02
0.00
−0.02
−0.04
0.0000
0.0001
0.0002
0.0003
0.0004
Discriminator Output


Discriminator Output for noisy data
Errors
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0.0000
0.0001
0.0002
0.0003
0.0004
Discriminator Output

2.0.2 Visualization of Trained GAN Generator

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

```
output.weight Parameter containing:
tensor([[-3.3236e-01, -1.2182e-01,  3.7743e-01,  5.5747e-02, -3.0517e-01,
         -6.3161e-02,  1.6531e-01, -2.1374e-02, -1.1724e-01,  8.7821e-02,
         -1.1384e-01, -1.0055e-01,  1.3363e-01, -2.0199e-01,  1.7138e-04]],
       requires_grad=True)
output.bias Parameter containing:
tensor([-0.0092], requires_grad=True)
```

[ ]:
```
```