

Dataset2-Dummy_Linear_output_30

October 7, 2021

1 Dataset 2 : Dummy Linear Data

1.1 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 0 ($\beta \sim N(0, \sigma)$) 2. std : $\sigma = 1, 0.1, 0.01$ (standard deviation) 3. prior: 0 (Correct) or 1 (Misspecified)

```
[1]: n_samples = 100

#Discriminator Parameters
hidden_nodes = 25

#ABC Generator Parameters
meanVal = 1
std = 1
prior = 0
```

```
[2]: # Parameters
sample_size = 1000000
std = 1
mean = 1
prior = 1
```

1.2 Import Libraries

```
[3]: import train_test
import ABC_train_test
import linearDataset
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 statistics import mean
from sklearn.metrics import mean_squared_error, mean_absolute_error
from torch import nn
import numpy as np
import warnings
warnings.filterwarnings('ignore')

```

1.3 Dataset

Generate the linear dataset

$y = m * x + c + e$ where $m = 1$, $c = 0.5$ and $e \sim N(0, 1)$

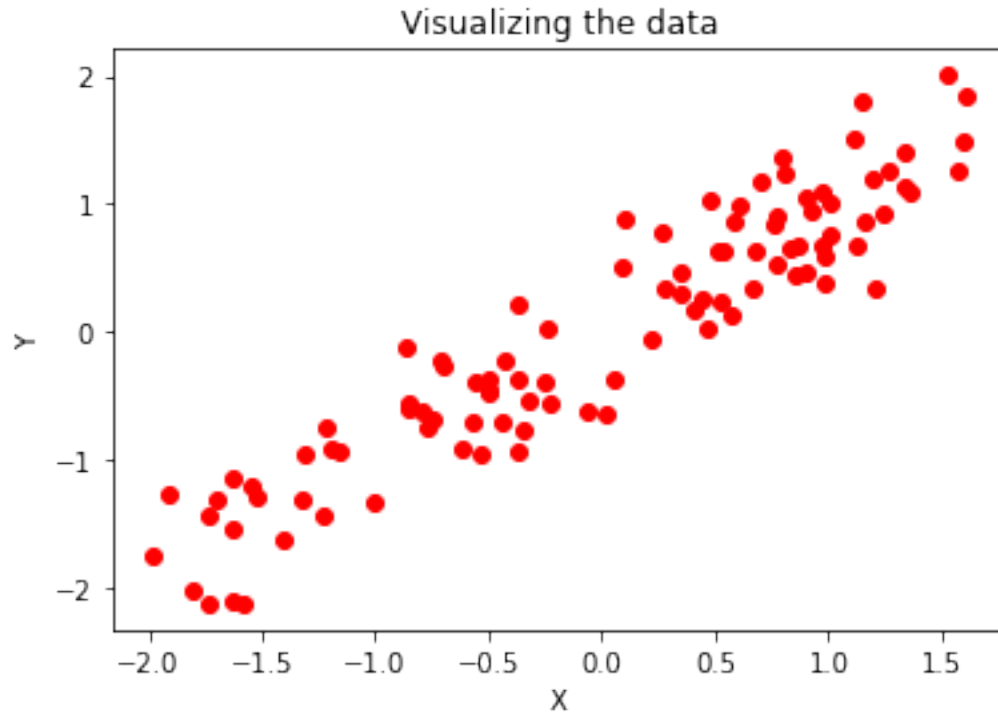
$x \sim 10 * U(-0.5, 0.5)$

```

[4]: X, Y = linearDataset.linear_data(n_samples)
     n_features = 1

```

	X	Y
0	-4.306785	-4.761985
1	-0.503024	-1.050802
2	-3.797499	-2.216484
3	2.025389	1.425009
4	-1.659397	-0.946918



1.4 Stats Model

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

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

No handles with labels found to put in legend.

OLS Regression Results

```
=====
=====
Dep. Variable:                Y    R-squared (uncentered):
0.876
Model:                        OLS    Adj. R-squared (uncentered):
0.875
Method:                        Least Squares    F-statistic:
702.5
Date:                          Thu, 07 Oct 2021    Prob (F-statistic):
9.35e-47
Time:                          16:33:28    Log-Likelihood:
-37.324
No. Observations:              100    AIC:
76.65
Df Residuals:                  99    BIC:
79.25
```

```

Df Model:                                1
Covariance Type:                        nonrobust
=====
              coef      std err          t      P>|t|      [0.025      0.975]
-----
x1              0.9362      0.035     26.505      0.000      0.866      1.006
=====
Omnibus:                3.229   Durbin-Watson:                2.218
Prob(Omnibus):           0.199   Jarque-Bera (JB):           1.972
Skew:                    0.082   Prob(JB):                  0.373
Kurtosis:                2.332   Cond. No.                  1.00
=====

```

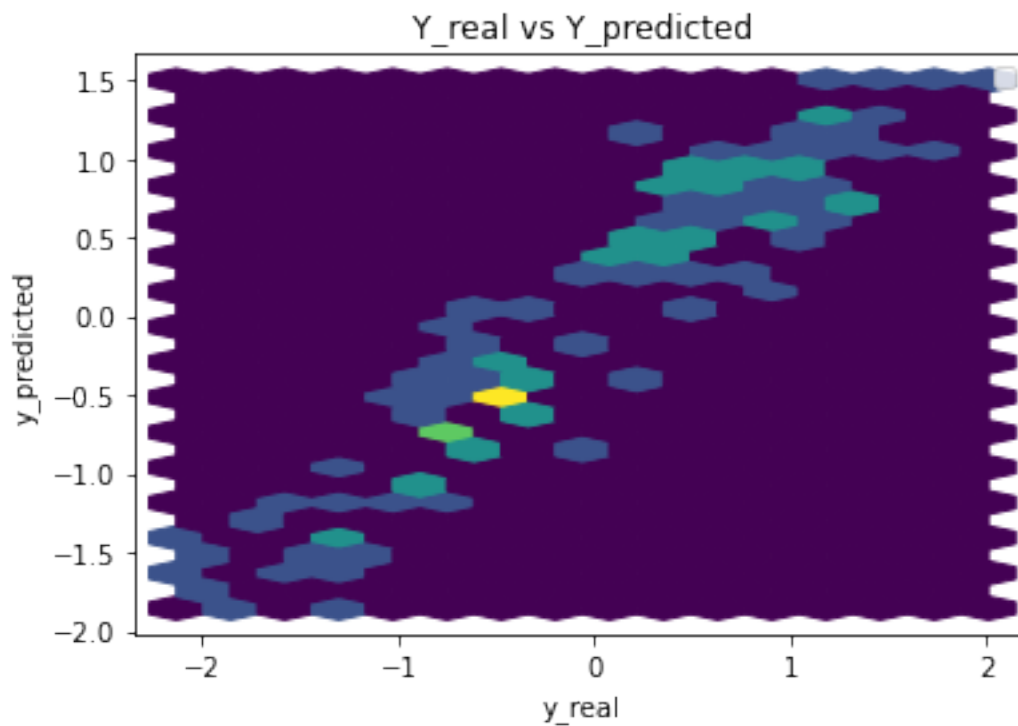
Notes:

[1] R^2 is computed without centering (uncentered) since the model does not contain a constant.

[2] Standard Errors assume that the covariance matrix of the errors is correctly specified.

Parameters: x1 0.936209

dtype: float64



Performance Metrics

Mean Squared Error: 0.12351344645629382

Mean Absolute Error: 0.29219087908507424

Manhattan distance: 29.219087908507436
Euclidean distance: 3.5144479859046687

1.5 Generator and Discriminator Networks

Generator Model

A simple generator consisting of 2 input nodes and an output node

```
[6]: 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
```

Discriminator Model

Discriminator Model consisting of 2 input nodes,1 hidden layer and one output node.The input to the discriminator will be (x, y_{real}) and (x, y_{pred})

```
[7]: 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 Generators

1. *Correctly Specified Prior* : The 1st ABC Generator is defined as $Y = m * X + c + e$ where $m \sim N(\mu, \sigma)$, $c = 0.5$ and $e \sim N(0,1)$
2. *Misspecified Prior* : The 2nd ABC Generator is defined as $Y = 1 + m * X + c + e$ where $m \sim N(\mu, \sigma)$, $c = 0.5$ and $e \sim N(0,1)$

Here μ and σ are parameters and can take the values $\mu = 0,1$ and $\sigma = 1,0.1,0.01$

```
[8]: def ABC_Generator_Correct(X,mu,sigma,batch_size,device):
      m = np.random.normal(mu,sigma)
      c = 0.5
      X = X.numpy().reshape(1,batch_size)[0]
      Y = m*X + c + np.random.normal(0,1,size = batch_size)
```

```

X = torch.from_numpy(X).reshape(batch_size,1)
Y = torch.from_numpy(Y).reshape(batch_size,1)
gen_input = torch.cat((X,Y),dim = 1).to(device)
return gen_input

```

```

[9]: def ABC_Generator_Misspecified(X,mu,sigma,batch_size,device):
    m = np.random.normal(mu,sigma)
    c = 0.5
    X = X.numpy().reshape(1,batch_size)[0]
    Y = 1 + m*X + c + np.random.normal(0,1,size = batch_size)
    X = torch.from_numpy(X).reshape(batch_size,1)
    Y = torch.from_numpy(Y).reshape(batch_size,1)
    gen_input = torch.cat((X,Y),dim = 1).to(device)
    return gen_input

```

1.6 GAN Model

```

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

```

```

[11]: generator = Generator(n_features+1)
discriminator = Discriminator(n_features+1,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))

```

```

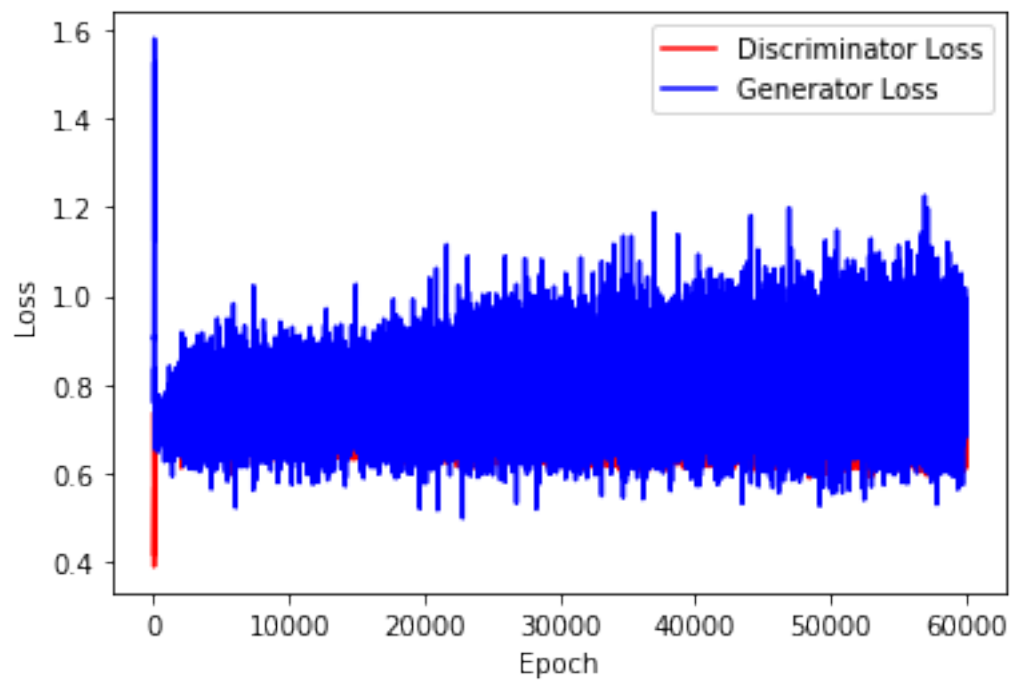
[12]: n_epochs = 30000
batch_size = n_samples//2

```

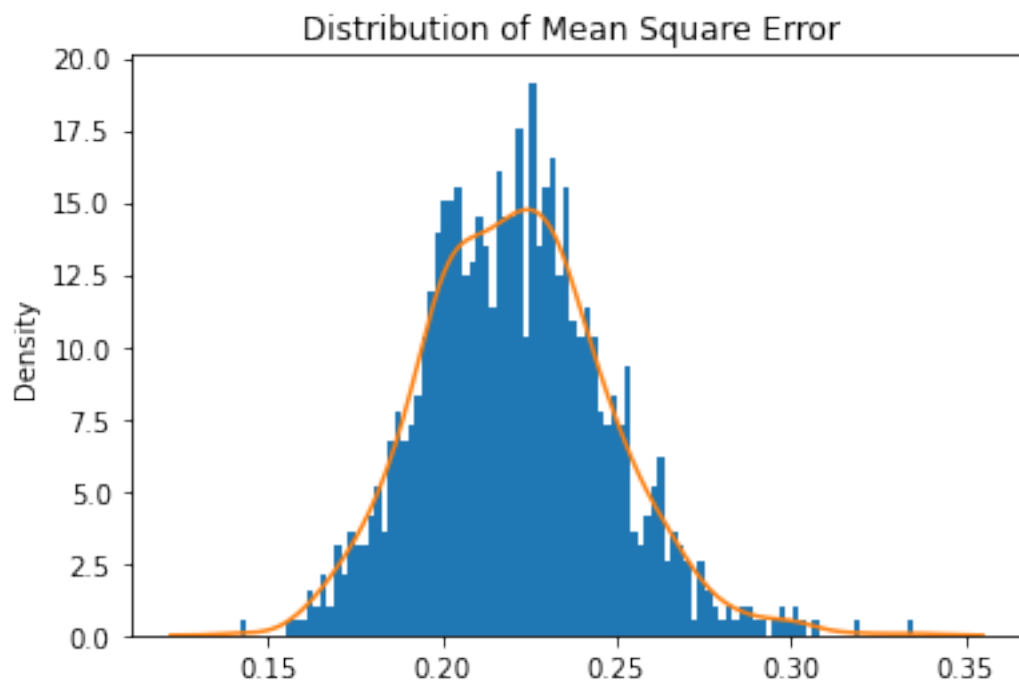
```

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

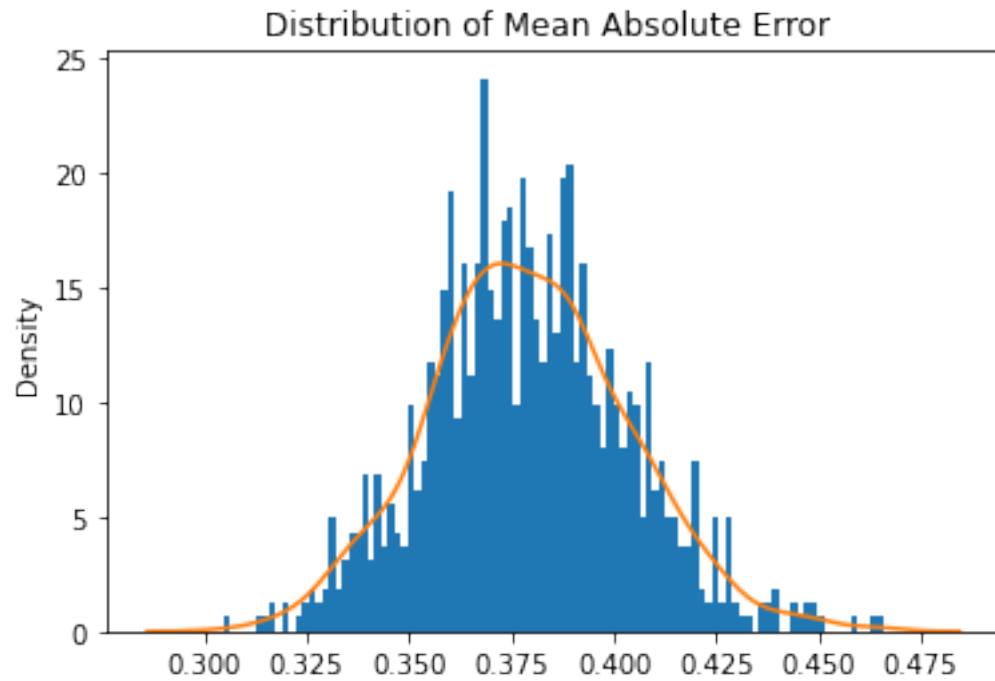
```



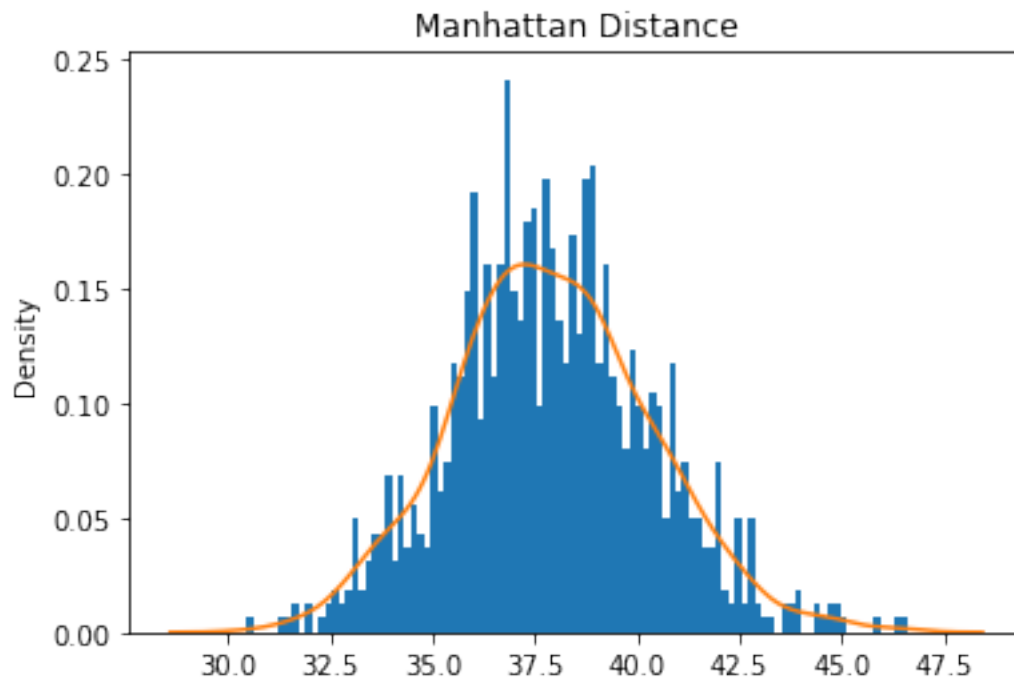
```
[14]: train_test.test_generator(generator,real_dataset,device)
```



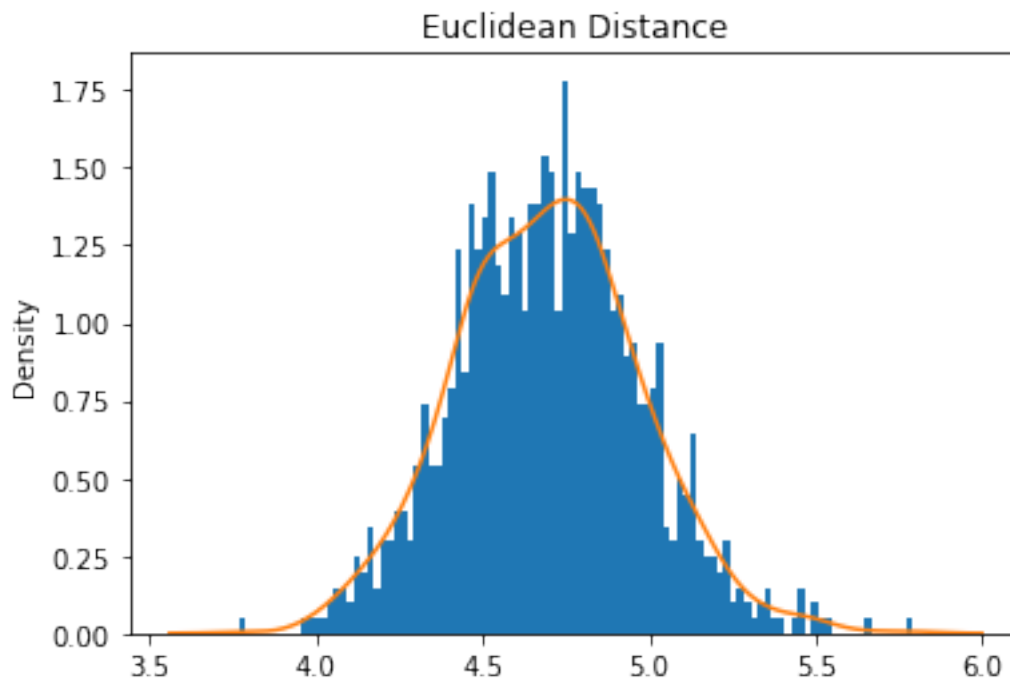
Mean Square Error: 0.2211768723040339



Mean Absolute Error: 0.37908661111619324



Mean Manhattan Distance: 37.908661111619324



Mean Euclidean Distance: 37.908661111619324

1.7 ABC - GAN Model

```
[15]: gen = Generator(n_features+1)
      disc = Discriminator(n_features+1,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 = n_samples//2
```

```
[17]: def training_GAN(disc, gen,disc_opt,gen_opt,dataset, batch_size,
      ↪n_epochs,criterion,coeff,mean,std,prior,device):
      discriminatorLoss = []
      generatorLoss = []
      train_loader = DataLoader(dataset, batch_size=batch_size, shuffle=True)

      for epoch in range(n_epochs):
```

```

for x_batch,y_batch in train_loader:
    y_shape = list(y_batch.size())
    curr_batch_size = y_shape[0]
    y_batch = torch.reshape(y_batch,(curr_batch_size,1))

    #Create the labels
    real_labels = torch.ones(curr_batch_size,1).to(device)
    fake_labels = torch.zeros(curr_batch_size,1).to(device)

    #-----
    #Update the discriminator
    #-----
    disc_opt.zero_grad()

    #Get discriminator loss for real data
    inputs_real = torch.cat((x_batch,y_batch),dim=1).to(device)
    disc_real_pred = disc(inputs_real)
    disc_real_loss = criterion(disc_real_pred,real_labels)

    #Get discriminator loss for fake data
    gen_input = ☐
    ↪ABC_Generator_Misspecified(x_batch,mean,std,curr_batch_size,device)
    if(prior == 0):
        gen_input = ☐
    ↪ABC_Generator_Correct(x_batch,mean,std,curr_batch_size,device)
    generated_y = gen(gen_input.float())
    x_batch = x_batch.to(device)
    inputs_fake = torch.cat((x_batch,generated_y),dim=1).to(device)
    x_batch = x_batch.detach().cpu()
    disc_fake_pred = disc(inputs_fake)
    disc_fake_loss = criterion(disc_fake_pred,fake_labels)

    #Get the discriminator loss
    disc_loss = (disc_fake_loss + disc_real_loss) / 2
    discriminatorLoss.append(disc_loss.item())

    # Update gradients
    disc_loss.backward(retain_graph=True)
    # Update optimizer
    disc_opt.step()

    #-----
    #Update the Generator
    #-----
    gen_opt.zero_grad()

    #Generate input to generator using ABC pre-generator

```

```

        gen_input = □
↪ABC_Generator_Misspecified(x_batch,mean,std,curr_batch_size,device)
        if(prior == 0):
            gen_input = □
↪ABC_Generator_Correct(x_batch,mean,std,curr_batch_size,device)
            generated_y = gen(gen_input.float())
            x_batch = x_batch.to(device)
            inputs_fake = torch.cat((x_batch,generated_y),dim=1).to(device)
            x_batch = x_batch.detach().cpu()
            disc_fake_pred = disc(inputs_fake)

            gen_loss = criterion(disc_fake_pred,real_labels)
            generatorLoss.append(gen_loss.item())

            #Update gradients
            gen_loss.backward()
            #Update optimizer
            gen_opt.step()

            #Plotting the Discriminator and Generator Loss
            plt.plot(discriminatorLoss,color = "red",label="Discriminator Loss")
            plt.plot(generatorLoss,color="blue",label = "Generator Loss")
            plt.xlabel("Epoch")
            plt.ylabel("Loss")
            plt.legend()
            plt.show()

```

```

[18]: def test_generator(gen,dataset,coeff,w,std,prior,device):
        test_loader = DataLoader(dataset, batch_size=len(dataset), shuffle=False)
        mse=[]
        mae=[]
        distp1 = []
        distp2 = []
        for epoch in range(1000):
            for x_batch, y_batch in test_loader:
                gen_input = □
↪ABC_Generator_Misspecified(x_batch,w,std,len(dataset),device)
                if(prior == 0):
                    gen_input = □
↪ABC_Generator_Correct(x_batch,w,std,len(dataset),device)
                    generated_y = gen(gen_input.float())
                    generated_y = generated_y.cpu().detach()
                    generated_data = torch.reshape(generated_y,(-1,))
                    gen_data = generated_data.numpy().reshape(1,len(dataset)).tolist()
                    real_data = y_batch.numpy().reshape(1,len(dataset)).tolist()
                    #Plot the data
                    if(epoch%200==0):

```

```

        gen_data1 = generated_data.numpy().tolist()
        real_data1 = y_batch.numpy().tolist()
        plt.hexbin(real_data1,gen_data1,gridsize=(15,15))
        plt.xlabel("Y")
        plt.ylabel("Y_Pred")
        plt.show()

    meanSquaredError = mean_squared_error(real_data,gen_data)
    meanAbsoluteError = mean_absolute_error(real_data, gen_data)
    mse.append(meanSquaredError)
    mae.append(meanAbsoluteError)
    dist1 = ABC_train_test.minkowski_distance(np.array(real_data)[0],np.
→array(gen_data)[0], 1)
    dist2 = ABC_train_test.minkowski_distance(np.array(real_data)[0],np.
→array(gen_data)[0], 2)
    distp1.append(dist1)
    distp2.append(dist2)

#Distribution of Metrics
#Mean Squared Error
n,x,_=plt.hist(mse,bins=100,density=True)
plt.title("Distribution of Mean Square Error ")
sns.distplot(mse,hist=False)
plt.show()
print("Mean Square Error:",mean(mse))

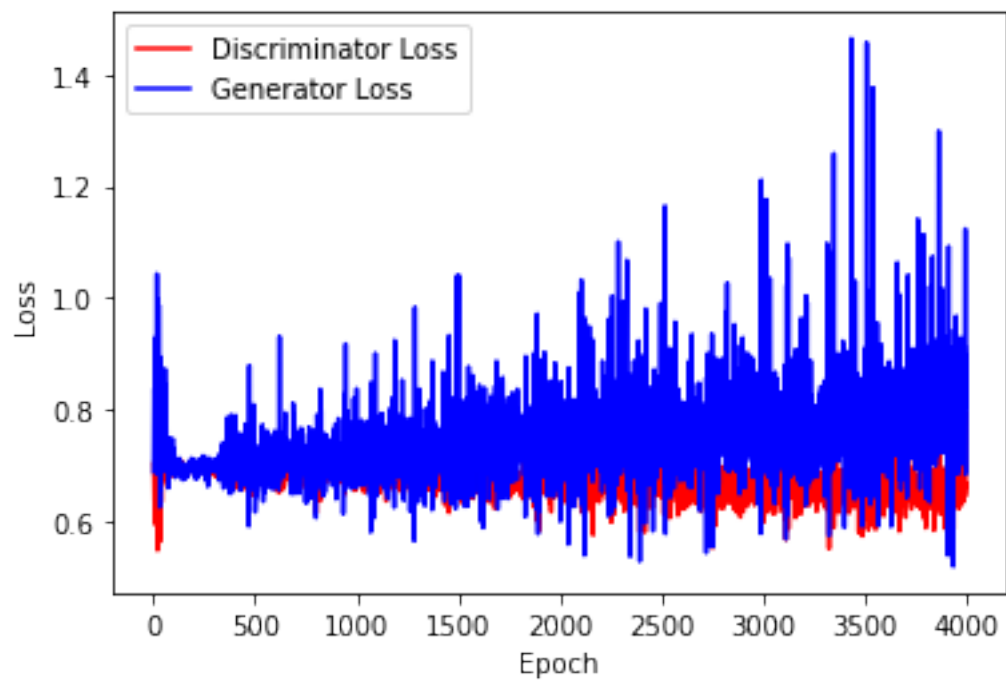
#Mean Absolute Error
n,x,_=plt.hist(mae,bins=100,density=True)
plt.title("Distribution of Mean Absolute Error ")
sns.distplot(mae,hist=False)
plt.show()
print("Mean Absolute Error:",mean(mae))

#Minkowski Distance 1st Order
n,x,_=plt.hist(distp1,bins=100,density=True)
plt.title("Manhattan Distance")
sns.distplot(distp1,hist=False)
print("Mean Manhattan Distance:",mean(distp1))
plt.show()

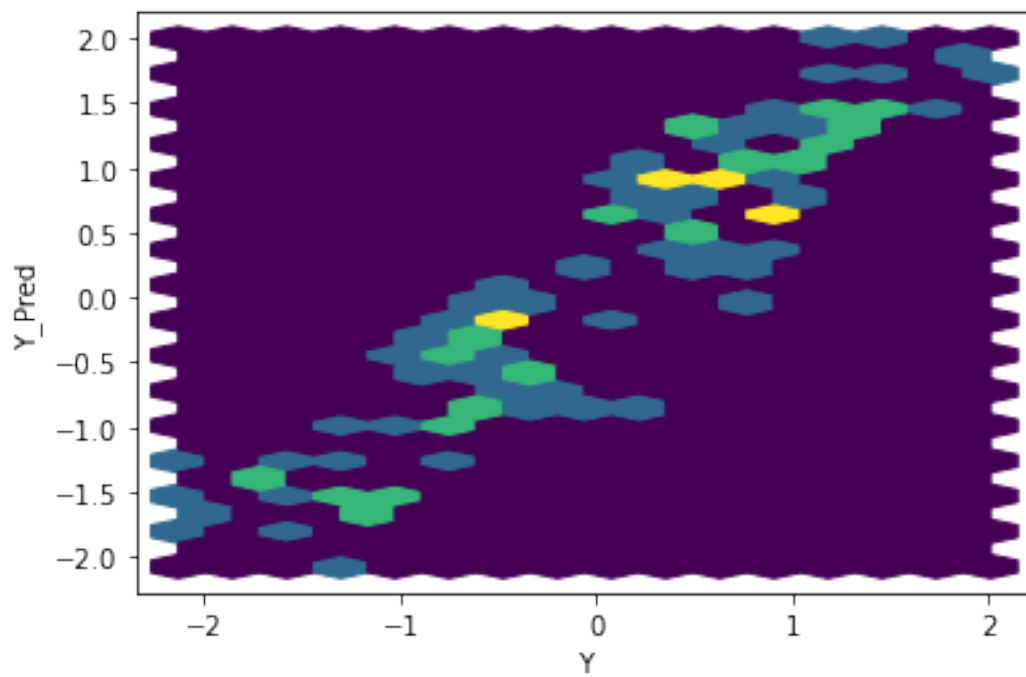
#Minkowski Distance 2nd Order
n,x,_=plt.hist(distp2,bins=100,density=True)
plt.title("Euclidean Distance")
sns.distplot(distp2,hist=False)
print("Mean Euclidean Distance:",mean(distp2))
plt.show()

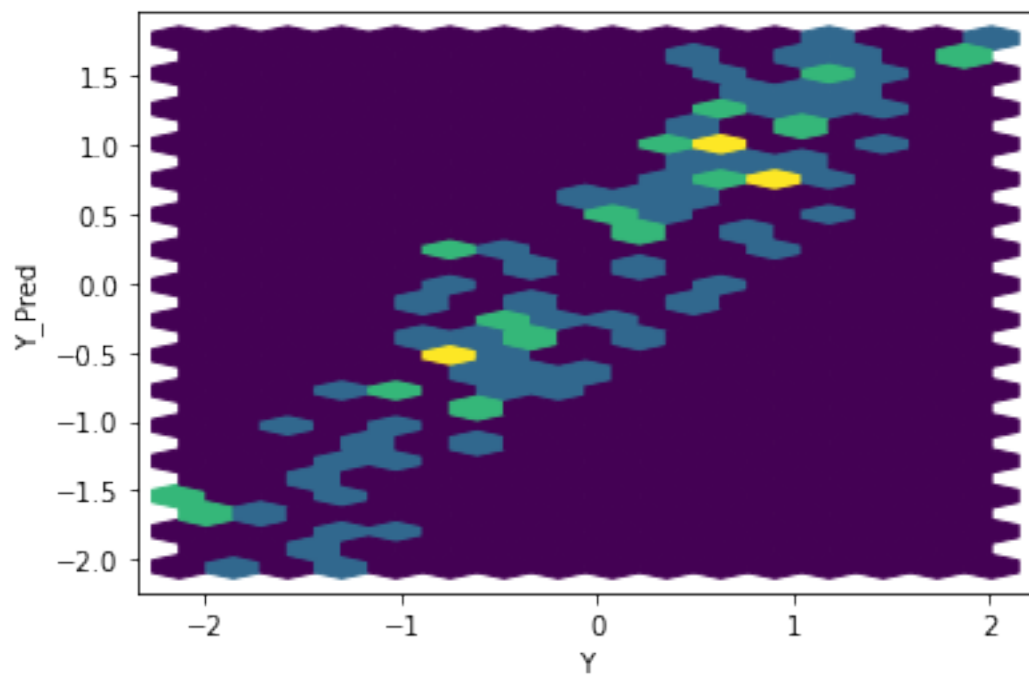
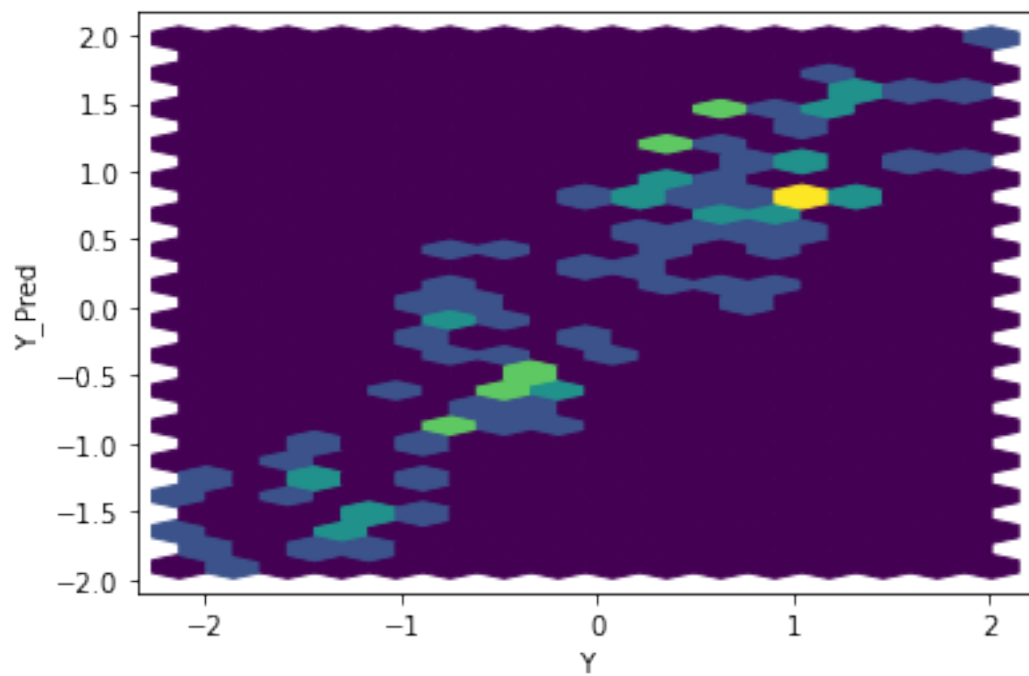
```

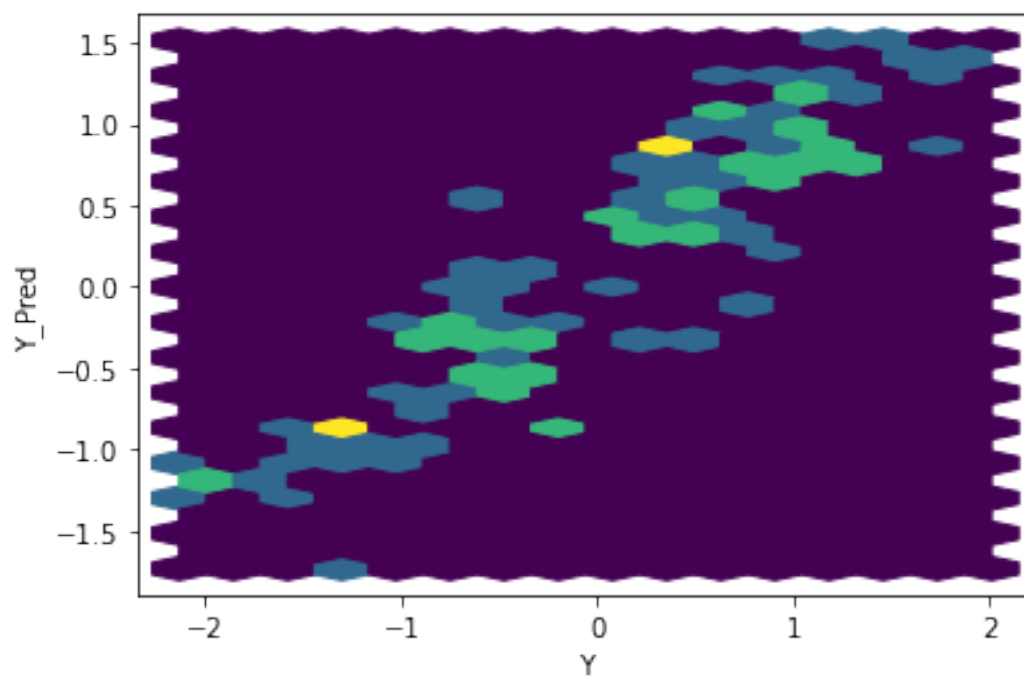
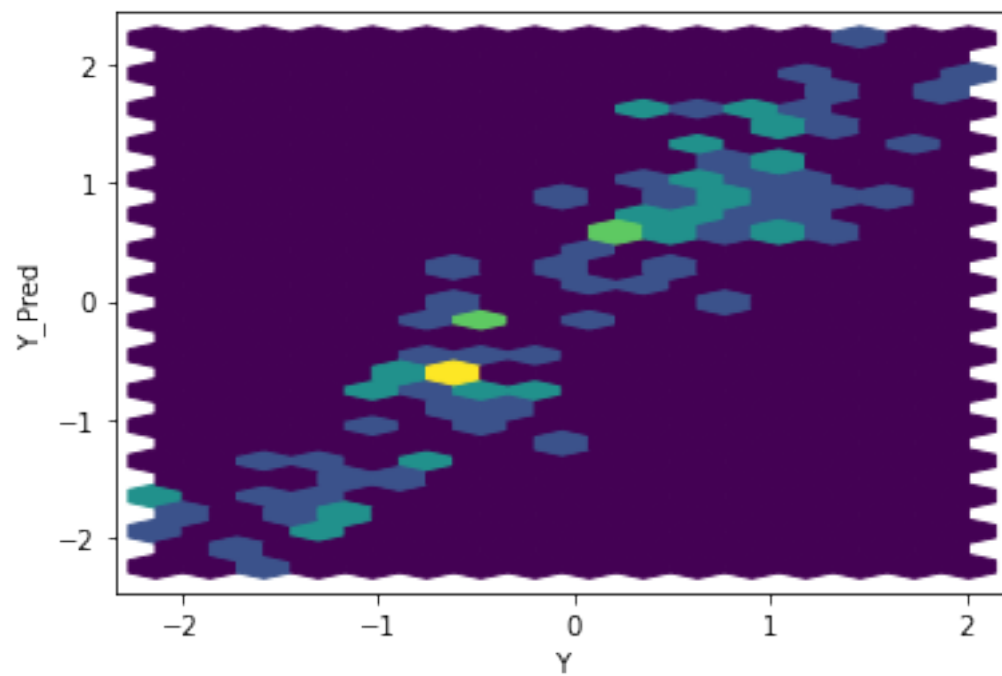
[19]: training_GAN(disc,gen,disc_opt,gen_opt,real_dataset,batch_size,n_epoch_abc,criterion,coeff,mea

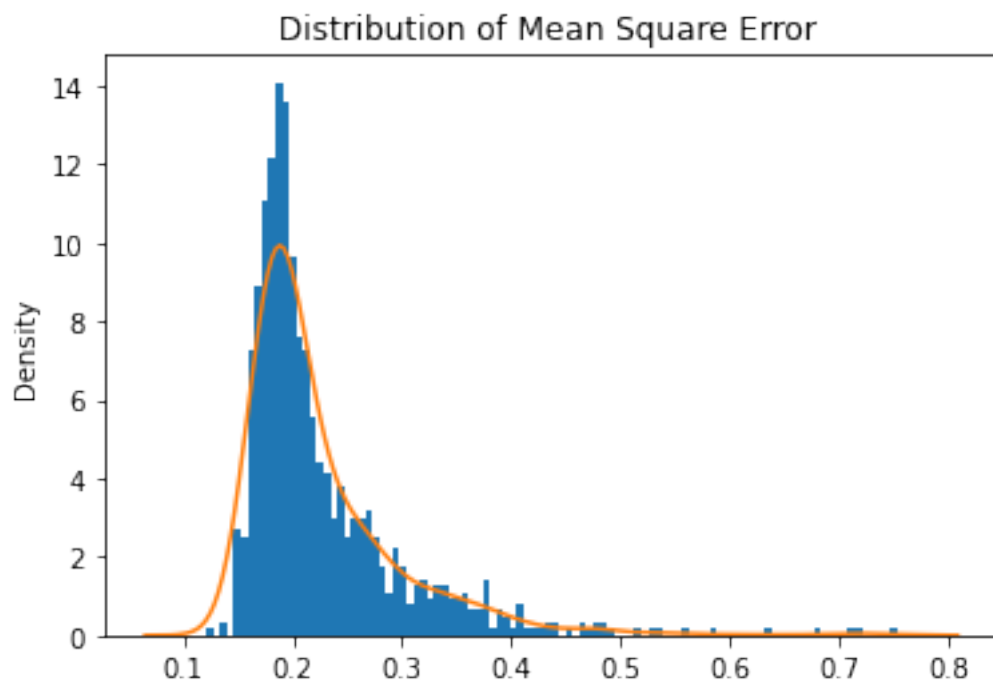


```
[20]: test_generator(gen,real_dataset,coeff,meanVal,std,prior,device)
```

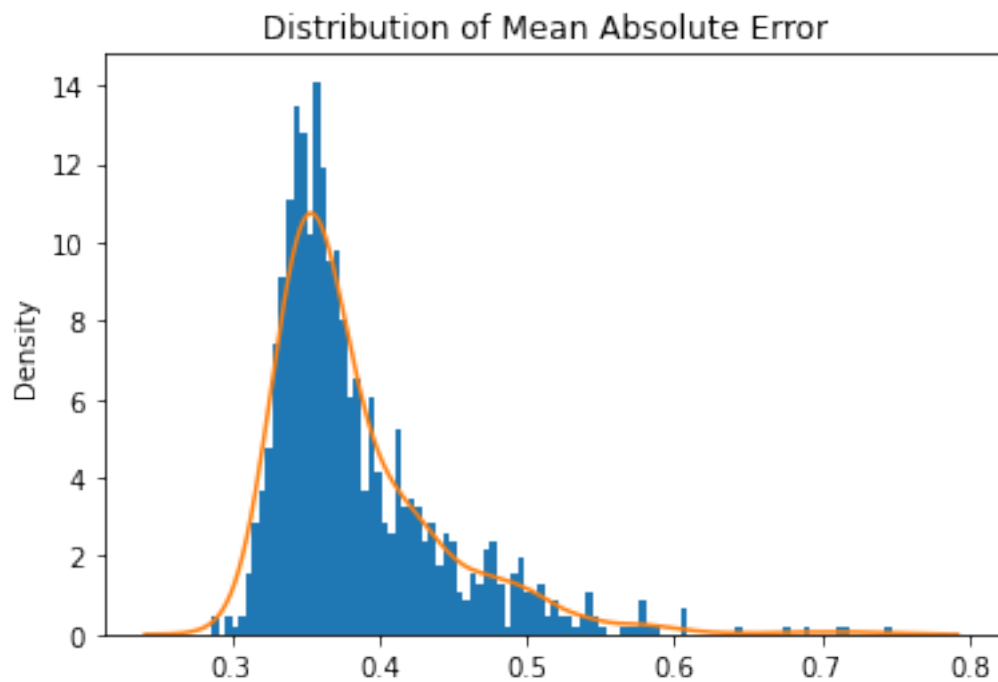




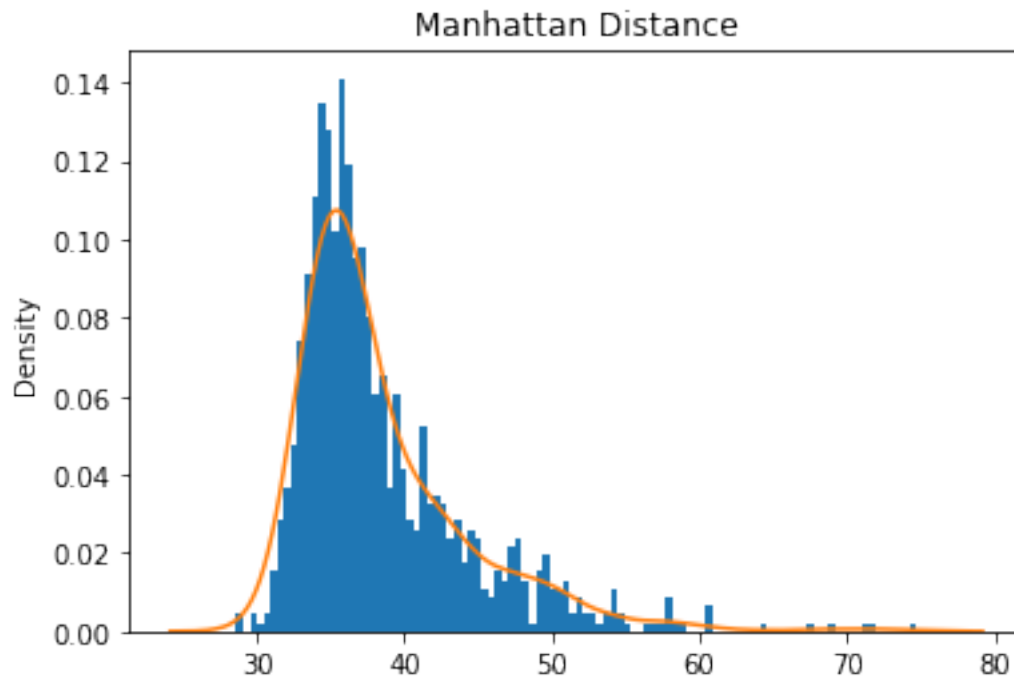




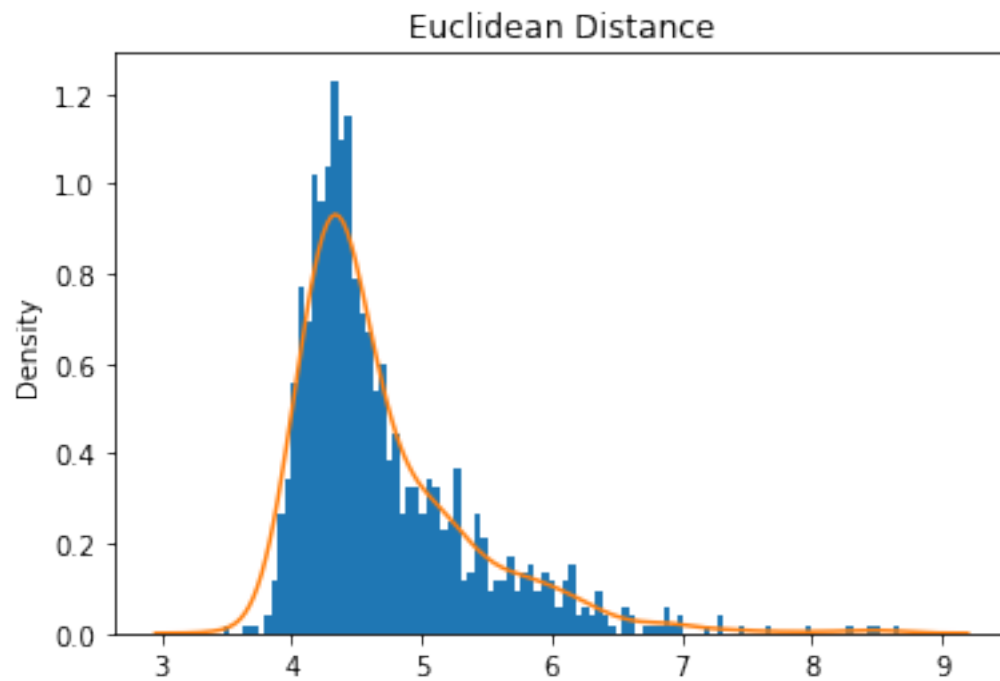
Mean Square Error: 0.22644308223418386



Mean Absolute Error: 0.38506293124847113
Mean Manhattan Distance: 38.50629312484711



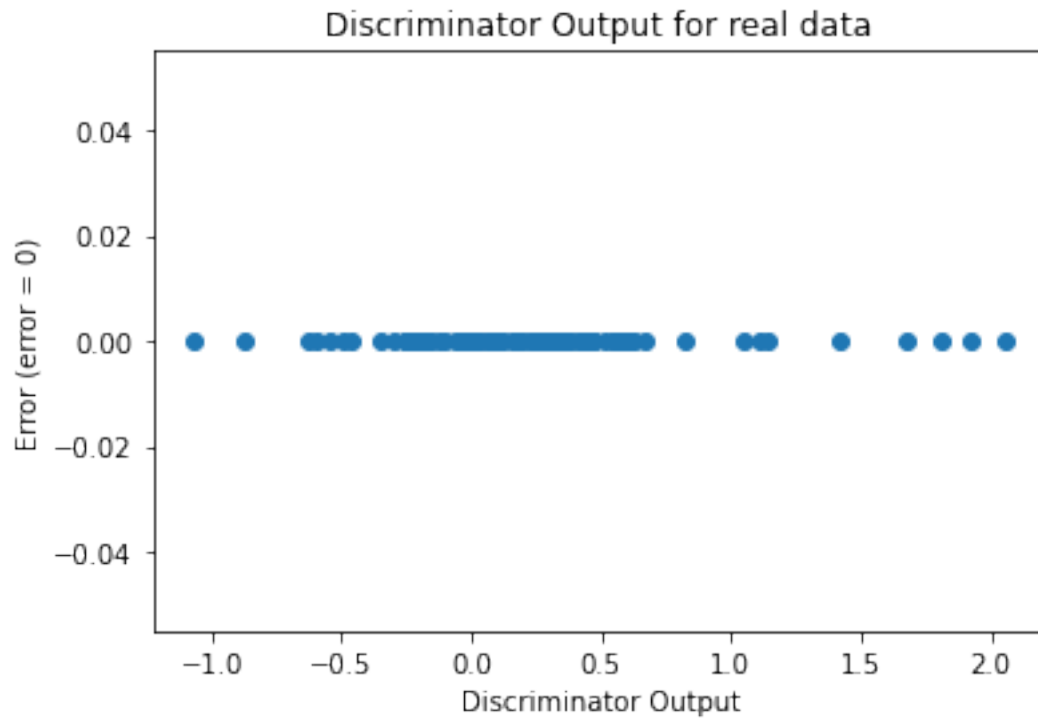
Mean Euclidean Distance: 4.708029028951709

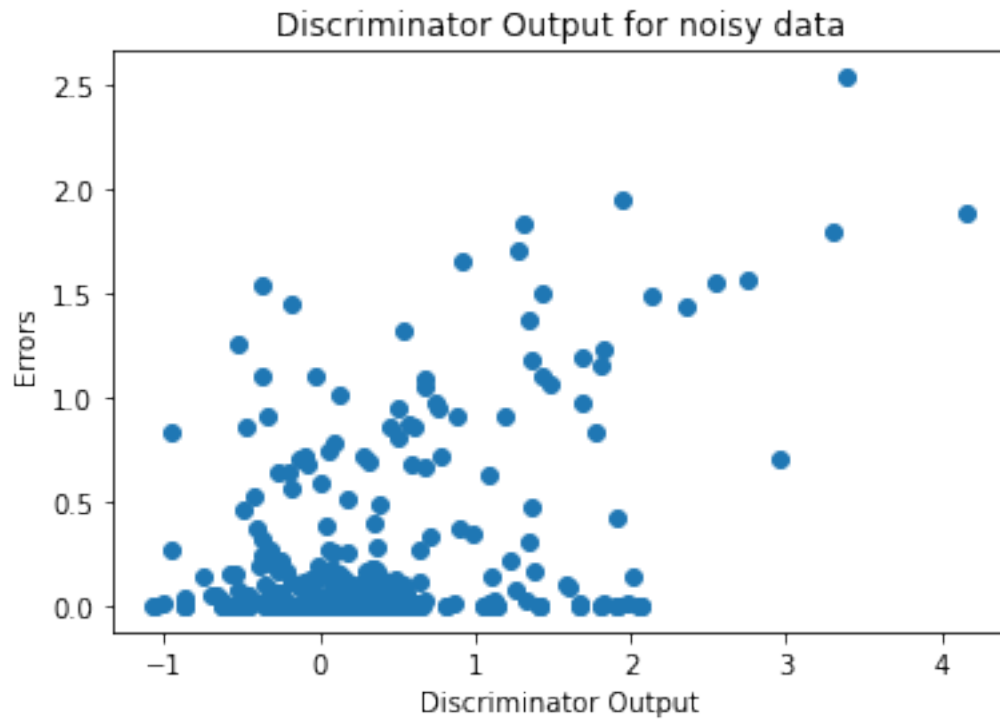


1.7.1 Sanity Check

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

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





1.7.2 Visualization of Trained GAN Generator

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

```
output.weight Parameter containing:  
tensor([[ 1.2192, -0.2080]], requires_grad=True)  
output.bias Parameter containing:  
tensor([0.4082], requires_grad=True)
```