Model to Predict ND2

In [2]:
```python
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
from sklearn.model_selection import train_test_split
from scipy.stats import norm
import liberator
```

In [95]:
```python
%time df = liberator.get_dataframe(liberator.query(symbols = ['AAPL'],
```

```
CPU times: total: 2.64 s
Wall time: 4.89 s
```

In [4]:
```python
df
```

Out[4]:

	_seq	muts	timestamp	symbol	okey_at	okey_ts	okey_tk	oke
0	1	1686805200000000	2023-06-15 01:00:00.000	AAPL	EQT	NMS	AAPL	2
1	2	1686805200000000	2023-06-15 01:00:00.000	AAPL	EQT	NMS	AAPL	2
2	3	1686805200000000	2023-06-15 01:00:00.000	AAPL	EQT	NMS	AAPL	2
3	4	1686805200000000	2023-06-15 01:00:00.000	AAPL	EQT	NMS	AAPL	2
4	5	1686805200000000	2023-06-15 01:00:00.000	AAPL	EQT	NMS	AAPL	2
...	...	...	...	...	...	...	...	
22905	22906	1688187600000000	2023-07-01 01:00:00.000	AAPL	EQT	NMS	AAPL	2
22906	22907	1688187600000000	2023-07-01 01:00:00.000	AAPL	EQT	NMS	AAPL	2
22907	22908	1688187600000000	2023-07-01 01:00:00.000	AAPL	EQT	NMS	AAPL	2
22908	22909	1688187600000000	2023-07-01 01:00:00.000	AAPL	EQT	NMS	AAPL	2
22909	22910	1688187600000000	2023-07-01 01:00:00.000	AAPL	EQT	NMS	AAPL	2

22910 rows × 69 columns

In [96]:
```
S = df['uClose'].values  # Underlying asset price
K = df['okey_xx'].values  # Strike price
r = df['rate'].values  # Risk-free interest rate
T = df['years'].values  # Time to expiration in years
sigma = df['srVol']  # Volatility
option_price = df['closePrc'].values  # Given put option price
nd2 = df['de'].values # Delta
```

In [97]:
```
kappa = 0.5 # reversion rate of volaitility i.e. how fast does the vola
theta = df['th'].values
rho = df['rh'].values
v0 = df['srVol'].iloc[0] # volatility at time zero
```

In [98]:
```
# Filter the data for put options only
df_put = df[df['okey_cp'] == 'Put']
```

In [99]:
```
df_put['in_the_money'] = (df_put['uClose'] < df_put['okey_xx']).astype(
df_put['in_the_money']
```

```
C:\Users\Kanika\AppData\Local\Temp\ipykernel_7232\2969976795.py:1: Se
ttingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/panda
s-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
(https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.htm
l#returning-a-view-versus-a-copy)
  df_put['in_the_money'] = (df_put['uClose'] < df_put['okey_xx']).ast
ype(int)
```

Out[99]:
```
1        0
3        0
5        0
7        0
9        0
        ..
22901    1
22903    1
22905    1
22907    1
22909    1
Name: in_the_money, Length: 11455, dtype: int32
```

In [100]:
```
# Separate the features (input parameters) and target variable (N(D2))
X = df_put[['uClose','okey_xx','rate','years','askIV','in_the_money']]
y = df_put['de']
```

In [84]:
```

```

In [134]:

```
# rolling window to split into training, testing

def rolling_window_split(X, y, window_size, test_size):
    """
    Perform a rolling window training-testing split on time series data

    Parameters:
        X (array-like): The input features (time series data).
        y (array-like): The target variable (labels or predictions).
        window_size (int): The size of the rolling window.
        test_size (float or int): The proportion or absolute number of

    Returns:
        X_train, X_test, y_train, y_test: The rolled training and testi
    """
    # Calculate the number of samples for the test set based on the prc

    if window_size == 1:
        # If the window size is 1, directly use the original X and y dc
        return train_test_split(X, y, test_size=test_size, random_state

    # Calculate the number of samples for the test set based on the prc
    if isinstance(test_size, float):
        test_size = int(len(X) * test_size)
    else:
        test_size = int(test_size)


    # Calculate the number of samples for the training set
    train_size = len(X) - test_size - window_size + 1

    # Initialize arrays to store the rolling windows
    X_train, X_test = [], []
    y_train, y_test = [], []

    # Create the rolling windows
    for i in range(train_size):
        X_train.append(X[i:i+window_size])
        y_train.append(y[i+window_size-1])

    for i in range(train_size, len(X) - window_size + 1):
        X_test.append(X[i:i+window_size])
        y_test.append(y[i+window_size-1])

    # Convert the lists to numpy arrays
    X_train = np.array(X_train)
    X_test = np.array(X_test)
    y_train = np.array(y_train)
    y_test = np.array(y_test)

    return X_train, X_test, y_train, y_test

# Convert data to PyTorch tensors
X_tensor = torch.tensor(X.values, dtype=torch.float32)
y_tensor = torch.tensor(y.values, dtype=torch.float32).view(-1, 1)

# Example usage:
# X_tensor and y_tensor are your time series data and target variable c
window_size = 1
test_size = 0.2  # or an absolute number of samples
```

```
# X_tensor = np.expand_dims(X_tensor, axis=2) # unsqueezing - adding ar


X_train_rolled, X_test_rolled, y_train_rolled, y_test_rolled = rolling_

# Convert the rolled data to PyTorch tensors
X_train_rolled = torch.tensor(X_train_rolled, dtype=torch.float32)
X_test_rolled = torch.tensor(X_test_rolled, dtype=torch.float32)
y_train_rolled = torch.tensor(y_train_rolled, dtype=torch.float32)
y_test_rolled = torch.tensor(y_test_rolled, dtype=torch.float32)
```

```
C:\Users\Kanika\AppData\Local\Temp\ipykernel_7232\3335210378.py:68: U
serWarning: To copy construct from a tensor, it is recommended to use
sourceTensor.clone().detach() or sourceTensor.clone().detach().requir
es_grad_(True), rather than torch.tensor(sourceTensor).
  X_train_rolled = torch.tensor(X_train_rolled, dtype=torch.float32)
C:\Users\Kanika\AppData\Local\Temp\ipykernel_7232\3335210378.py:69: U
serWarning: To copy construct from a tensor, it is recommended to use
sourceTensor.clone().detach() or sourceTensor.clone().detach().requir
es_grad_(True), rather than torch.tensor(sourceTensor).
  X_test_rolled = torch.tensor(X_test_rolled, dtype=torch.float32)
C:\Users\Kanika\AppData\Local\Temp\ipykernel_7232\3335210378.py:70: U
serWarning: To copy construct from a tensor, it is recommended to use
sourceTensor.clone().detach() or sourceTensor.clone().detach().requir
es_grad_(True), rather than torch.tensor(sourceTensor).
  y_train_rolled = torch.tensor(y_train_rolled, dtype=torch.float32)
C:\Users\Kanika\AppData\Local\Temp\ipykernel_7232\3335210378.py:71: U
serWarning: To copy construct from a tensor, it is recommended to use
sourceTensor.clone().detach() or sourceTensor.clone().detach().requir
es_grad_(True), rather than torch.tensor(sourceTensor).
  y_test_rolled = torch.tensor(y_test_rolled, dtype=torch.float32)
```

In [163]:

```
X_train_rolled = X_train_rolled.unsqueeze(1)
X_test_rolled = X_test_rolled.unsqueeze(1)
```

In [170]:

```
# original
class LSTMModel(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(LSTMModel, self).__init__()
        self.hidden_size = hidden_size
        self.lstm_cell = nn.LSTMCell(input_size, hidden_size)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        batch_size, seq_length, _ = x.size()
        h_t = torch.zeros(batch_size, self.hidden_size).to(x.device)
        c_t = torch.zeros(batch_size, self.hidden_size).to(x.device)

        for t in range(seq_length):
            h_t, c_t = self.lstm_cell(x[:, t, :], (h_t, c_t))

        out = self.fc(h_t)
        return out
```

In [182]:
```python
# applying sigmoid function so predicted values are bw 0-1 ----> led tc
class LSTMModel(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(LSTMModel, self).__init__()
        self.hidden_size = hidden_size
        self.lstm_cell = nn.LSTMCell(input_size, hidden_size)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        batch_size, seq_length, _ = x.size()
        h_t = torch.zeros(batch_size, self.hidden_size).to(x.device)
        c_t = torch.zeros(batch_size, self.hidden_size).to(x.device)

        for t in range(seq_length):
            h_t, c_t = self.lstm_cell(x[:, t, :], (h_t, c_t))
# Apply sigmoid activation to the output
        out = torch.sigmoid(self.fc(h_t))
        return out
```

In [183]:
```python
input_size = 6  # Size of input features
output_size = 1  # Size of output (target) variable
hidden_size = 64  # Number of LSTM units (hidden layer size)
learning_rate = 0.001
num_epochs = 100
batch_size = 16

# Create an instance of the LSTM model
model = LSTMModel(input_size, hidden_size, output_size)
```

In [184]:
```python
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
```

In [186]:
```python
train_dataset = torch.utils.data.TensorDataset(X_train_rolled, y_train_
test_dataset = torch.utils.data.TensorDataset(X_test_rolled, y_test_rol

train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=ba
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=batc
```

In [187]:
```python
X_train_rolled.shape
```

Out[187]:
```
torch.Size([9164, 1, 6])
```

In [188]:

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model.to(device)

for epoch in range(num_epochs):
    model.train()
    total_loss = 0

    for batch_x, batch_y in train_loader:
        batch_x, batch_y = batch_x.to(device), batch_y.to(device)

        # Initialize hidden state and cell state with zeros
        batch_size = batch_x.size(0)
        h0 = torch.zeros(1, batch_size, hidden_size).to(device)
        c0 = torch.zeros(1, batch_size, hidden_size).to(device)

        optimizer.zero_grad()
        outputs = model(batch_x)
        loss = criterion(outputs, batch_y)
        loss.backward()
        optimizer.step()

        total_loss += loss.item()

    print(f'Epoch {epoch + 1}/{num_epochs}, Loss: {total_loss / len(tra

print("Training finished.")
```

```
Epoch 1/100, Loss: 0.4055428061939867
Epoch 2/100, Loss: 0.36228827342885117
Epoch 3/100, Loss: 0.36169635919759774
Epoch 4/100, Loss: 0.36135482829694765
Epoch 5/100, Loss: 0.36137838122706345
Epoch 6/100, Loss: 0.3612306211826377
Epoch 7/100, Loss: 0.3613130573644896
Epoch 8/100, Loss: 0.36118996301974715
Epoch 9/100, Loss: 0.36128411136476574
Epoch 10/100, Loss: 0.3611968945692347
Epoch 11/100, Loss: 0.3612831754522174
Epoch 12/100, Loss: 0.3613288370635601
Epoch 13/100, Loss: 0.3612576723293796
Epoch 14/100, Loss: 0.3612772281140245
Epoch 15/100, Loss: 0.361140133263657
Epoch 16/100, Loss: 0.36133646282382037
Epoch 17/100, Loss: 0.36125360737452333
Epoch 18/100, Loss: 0.36122210193782994
Epoch 19/100, Loss: 0.36121766326419136
Epoch 20/100, Loss: 0.3612383550315783
Epoch 21/100, Loss: 0.3612295801492886
Epoch 22/100, Loss: 0.3612466533692719
Epoch 23/100, Loss: 0.36119571585780297
Epoch 24/100, Loss: 0.3612853663520039
Epoch 25/100, Loss: 0.36129102632273347
Epoch 26/100, Loss: 0.3611778490648428
Epoch 27/100, Loss: 0.361187185582999
Epoch 28/100, Loss: 0.361234654215722
Epoch 29/100, Loss: 0.3612405352723536
Epoch 30/100, Loss: 0.361292483970965
Epoch 31/100, Loss: 0.36116762474881414
Epoch 32/100, Loss: 0.36117716675484474
Epoch 33/100, Loss: 0.36123068992822166
Epoch 34/100, Loss: 0.3612274048788177
Epoch 35/100, Loss: 0.36134763350661514
Epoch 36/100, Loss: 0.36113581678994244
Epoch 37/100, Loss: 0.3611555074001898
Epoch 38/100, Loss: 0.36131357986733553
Epoch 39/100, Loss: 0.3611494643877002
Epoch 40/100, Loss: 0.3612027037596203
Epoch 41/100, Loss: 0.36128882750426705
Epoch 42/100, Loss: 0.3612353178765137
Epoch 43/100, Loss: 0.3612817321590312
Epoch 44/100, Loss: 0.36126940580406736
Epoch 45/100, Loss: 0.3612146032470267
Epoch 46/100, Loss: 0.36120327451895357
Epoch 47/100, Loss: 0.36124708538592176
Epoch 48/100, Loss: 0.36121636592696893
Epoch 49/100, Loss: 0.3611937385266988
Epoch 50/100, Loss: 0.36115869021540536
Epoch 51/100, Loss: 0.3611953931992279
Epoch 52/100, Loss: 0.3612693358784831
Epoch 53/100, Loss: 0.3611704619117016
Epoch 54/100, Loss: 0.36124157905578613
Epoch 55/100, Loss: 0.36120894707310264
Epoch 56/100, Loss: 0.3612172333929967
Epoch 57/100, Loss: 0.36131512840484864
Epoch 58/100, Loss: 0.3613123623085376
Epoch 59/100, Loss: 0.3612178709442495
Epoch 60/100, Loss: 0.3611601871709757
Epoch 61/100, Loss: 0.3612185873832378
```

```
Epoch 62/100, Loss: 0.3612036115341994
Epoch 63/100, Loss: 0.3612366166830479
Epoch 64/100, Loss: 0.361149908113334
Epoch 65/100, Loss: 0.3612137464237567
Epoch 66/100, Loss: 0.36123645298482027
Epoch 67/100, Loss: 0.3611600012055242
Epoch 68/100, Loss: 0.36128587585534205
Epoch 69/100, Loss: 0.3613189504486728
Epoch 70/100, Loss: 0.36128154910619764
Epoch 71/100, Loss: 0.3612771328103064
Epoch 72/100, Loss: 0.36122661684013996
Epoch 73/100, Loss: 0.36117341266668695
Epoch 74/100, Loss: 0.36116128176918827
Epoch 75/100, Loss: 0.3612770434553503
Epoch 76/100, Loss: 0.36116012775162026
Epoch 77/100, Loss: 0.36126519800740803
Epoch 78/100, Loss: 0.36127166646365305
Epoch 79/100, Loss: 0.3612959075712617
Epoch 80/100, Loss: 0.3612192595023239
Epoch 81/100, Loss: 0.36119182182908266
Epoch 82/100, Loss: 0.3612741826014369
Epoch 83/100, Loss: 0.3612471751699689
Epoch 84/100, Loss: 0.3611557433449941
Epoch 85/100, Loss: 0.36124750067452277
Epoch 86/100, Loss: 0.361163464854317
Epoch 87/100, Loss: 0.36121640840698077
Epoch 88/100, Loss: 0.3611945513486342
Epoch 89/100, Loss: 0.36113340385992854
Epoch 90/100, Loss: 0.361242101883701
Epoch 91/100, Loss: 0.3611939219858686
Epoch 92/100, Loss: 0.3612358482395271
Epoch 93/100, Loss: 0.36119242195839657
Epoch 94/100, Loss: 0.36121542014532687
Epoch 95/100, Loss: 0.3611795441954548
Epoch 96/100, Loss: 0.36113957635551225
Epoch 97/100, Loss: 0.36110157005969973
Epoch 98/100, Loss: 0.3612346141672259
Epoch 99/100, Loss: 0.36118528501733643
Epoch 100/100, Loss: 0.36124704654505685
Training finished.
```

In [189]:

```python
average_loss = total_loss / len(train_loader)
average_loss
```

Out[189]:

```
0.36124704654505685
```

In [190]:

```
# Evaluating the model on the test set
model.eval()  # Set the model to evaluation mode (important for dropout

y_test_pred = []  # List to store the predicted values

with torch.no_grad():  # No need to track gradients during evaluation
    for batch_x, _ in test_loader:
        batch_x = batch_x.to(device)

        # Initialize hidden state and cell state with zeros
        batch_size = batch_x.size(0)
        h0 = torch.zeros(1, batch_size, hidden_size).to(device)
        c0 = torch.zeros(1, batch_size, hidden_size).to(device)

        # Pass the input through the model
        outputs = model(batch_x)

        # Add the predictions to the y_test_pred list
        y_test_pred.append(outputs.cpu().numpy())

# Concatenate the predictions into a single numpy array
y_test_pred = np.concatenate(y_test_pred, axis=0)

# Convert the numpy array to a tensor
y_test_pred = torch.tensor(y_test_pred, dtype=torch.float32)
y_test_pred
```

Out[190]:

```
tensor([[1.8948e-11],
        [1.8934e-11],
        [1.9423e-11],
        ...,
        [1.8944e-11],
        [1.8416e-11],
        [1.8947e-11]])
```

In [ ]:

```
```

##

In [ ]:

```
# trying the model on a highly volatile stock - BANC - Bank of Califorr
# no rolling window, just RNN
```

In [57]:

```
%time df = liberator.get_dataframe(liberator.query(symbols = ['BANC'],
```

```
CPU times: total: 156 ms
Wall time: 1.01 s
```

In [58]:
```
S = df['uClose'].values  # Underlying asset price
K = df['okey_xx'].values  # Strike price
r = df['rate'].values  # Risk-free interest rate
T = df['years'].values  # Time to expiration in years
sigma = df['srVol']  # Volatility
option_price = df['closePrc'].values  # Given put option price
nd2 = df['de'].values # Delta
```

In [59]:
```
kappa = 0.5 # reversion rate of volaitility i.e. how fast does the vola
theta = df['th'].values
rho = df['rh'].values
v0 = df['srVol'].iloc[0] # volatility at time zero
```

In [60]:
```
# Filter the data for put options only
df_put = df[df['okey_cp'] == 'Put']
```

In [61]:
```
df_put['in_the_money'] = (df_put['uClose'] < df_put['okey_xx']).astype(
df_put['in_the_money']
```

```
C:\Users\Kanika\AppData\Local\Temp\ipykernel_7232\2969976795.py:1: Se
ttingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/panda
s-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
(https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.htm
l#returning-a-view-versus-a-copy)
  df_put['in_the_money'] = (df_put['uClose'] < df_put['okey_xx']).ast
ype(int)
```

Out[61]:
```
1       0
3       0
5       0
7       0
9       1
       ..
1009    0
1011    1
1013    1
1015    1
1017    1
Name: in_the_money, Length: 509, dtype: int32
```

In [62]:
```
# Separate the features (input parameters) and target variable (N(D2))
X = df_put[['uClose','okey_xx','rate','years','askIV','in_the_money']]
y = df_put['de']
```

In [63]:
```
# Convert data to PyTorch tensors
X_tensor = torch.tensor(X.values, dtype=torch.float32)
y_tensor = torch.tensor(y.values, dtype=torch.float32).view(-1, 1)
```

In [64]:
```
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_tensor, y_tensor,
```

In [65]:
```
# Reshape X_train and X_test to 3D tensors with sequence length = 1
X_train = X_train.unsqueeze(1)  # Shape: (9164, 1, 6)
X_test = X_test.unsqueeze(1)    # Shape: (2291, 1, 6)
```

In [66]:

```
# Define the Recurrent Neural Network (RNN) model using LSTM

class RNNModel(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(RNNModel, self).__init__()
        self.hidden_size = hidden_size
        self.lstm = nn.LSTM(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)
        self.sigmoid = nn.Sigmoid()  # Add sigmoid activation function

    def forward(self, x):
        h0 = torch.zeros(1, x.size(0), self.hidden_size).to(x.device)
        c0 = torch.zeros(1, x.size(0), self.hidden_size).to(x.device)

        out, _ = self.lstm(x, (h0, c0))
        out = self.fc(out[:, -1, :])
        out = self.sigmoid(out)  # Apply sigmoid activation function sc
        return out


# Hyperparameters
input_size = 6  # Number of input features (variables)
hidden_size = 16
output_size = 1  # Number of output units (1 for a single output variab
learning_rate = 0.001
num_epochs = 100

# Create the RNN model, loss function, and optimizer
model = RNNModel(input_size, hidden_size, output_size)
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

# Training loop
for epoch in range(num_epochs):
    # Forward pass
    outputs = model(X_train)
    loss = criterion(outputs, y_train)

    # Backward and optimize
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    # Print the progress
    if (epoch + 1) % 10 == 0:
        print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}

# After training, you can use the model to make predictions on the test
model.eval()
with torch.no_grad():
    y_pred = model(X_test)

# Assuming your problem is regression, you can evaluate the performance
mse = criterion(y_pred, y_test)
print(f"Test MSE: {mse.item():.4f}")
```

```
Epoch [10/100], Loss: 1.3818
Epoch [20/100], Loss: 1.3198
Epoch [30/100], Loss: 1.2491
Epoch [40/100], Loss: 1.1705
Epoch [50/100], Loss: 1.0874
Epoch [60/100], Loss: 1.0047
Epoch [70/100], Loss: 0.9275
Epoch [80/100], Loss: 0.8593
Epoch [90/100], Loss: 0.8033
Epoch [100/100], Loss: 0.7599
Test MSE: 0.6789
```

In [67]:

```
# Calculate the total loss function for the model
def calculate_total_loss(model, criterion, data, targets):
    model.eval()
    with torch.no_grad():
        # Forward pass
        outputs = model(data)
        # Calculate the loss
        loss = criterion(outputs, targets)
    return loss.item()

# Calculate the total loss on the test set (X_test, y_test)
total_test_loss = calculate_total_loss(model, criterion, X_test, y_test
print(f"Total Test Loss: {total_test_loss:.4f}")

# Calculate the total loss on the training set
total_train_loss = calculate_total_loss(model, criterion, X_train, y_tr
print(f"Total Training Loss: {total_train_loss:.4f}")
```

```
Total Test Loss: 0.6789
Total Training Loss: 0.7561
```

In [68]:

```
model.eval()
with torch.no_grad():
    y_pred = model(X_test)

# Convert the predicted values to a numpy array for further analysis if
y_pred_numpy = y_pred.numpy()

# Print the predicted values
print("Predicted Values:")
print(y_pred_numpy)
```

```
Predicted Values:
[[0.15978126]
 [0.189027  ]
 [0.16422713]
 [0.2651917 ]
 [0.16333908]
 [0.16427779]
 [0.17924026]
 [0.16948265]
 [0.18769814]
 [0.1663567 ]
 [0.17814718]
 [0.2609908 ]
 [0.17190062]
 [0.16054906]
 [0.26169193]
 [0.16977373]
 [0.1680419 ]
 [0.16524127]
 [0.2668142 ]
 [0.16441076]
 [0.16358934]
 [0.26123318]
 [0.16857757]
 [0.21086295]
 [0.17339294]
 [0.17512158]
 [0.16966414]
 [0.16760346]
 [0.26699466]
 [0.16457818]
 [0.1910118 ]
 [0.15779579]
 [0.19029386]
 [0.2213602 ]
 [0.1647706 ]
 [0.16074005]
 [0.16544819]
 [0.16396268]
 [0.19206822]
 [0.17933388]
 [0.22121829]
 [0.16256104]
 [0.19895662]
 [0.21095672]
 [0.16239724]
 [0.1632809 ]
 [0.26124865]
 [0.19095545]
 [0.16279769]
 [0.16198768]
 [0.16247652]
 [0.17972526]
 [0.16186269]
 [0.16418327]
 [0.21123402]
 [0.15948287]
 [0.25484806]
 [0.1985161 ]
 [0.16262972]
 [0.2689214 ]
```

```
 [0.16177619]
 [0.19162804]
 [0.16639541]
 [0.26154026]
 [0.19029985]
 [0.16788861]
 [0.16037475]
 [0.18650974]
 [0.18840307]
 [0.1772943 ]
 [0.2562323 ]
 [0.16232844]
 [0.19230506]
 [0.18017992]
 [0.16812058]
 [0.2695864 ]
 [0.15745087]
 [0.16497359]
 [0.16513693]
 [0.16469188]
 [0.19212921]
 [0.22167009]
 [0.1694005 ]
 [0.16564064]
 [0.26693678]
 [0.16450904]
 [0.17690483]
 [0.17645079]
 [0.1607324 ]
 [0.16172029]
 [0.2107264 ]
 [0.16820645]
 [0.19154856]
 [0.16452679]
 [0.19074231]
 [0.17700365]
 [0.16269419]
 [0.17640232]
 [0.26513946]
 [0.16351742]
 [0.22301409]
 [0.22153352]]
```

In [ ]: