In [1]:
```
!pip install keras-tuner
```

```
Requirement already satisfied: keras-tuner in c:\users\divya\anaconda3\l
ib\site-packages (1.4.7)
Requirement already satisfied: keras in c:\users\divya\anaconda3\lib\sit
e-packages (from keras-tuner) (3.0.5)
Requirement already satisfied: packaging in c:\users\divya\anaconda3\lib
\site-packages (from keras-tuner) (23.1)
Requirement already satisfied: requests in c:\users\divya\anaconda3\lib
\site-packages (from keras-tuner) (2.31.0)
Requirement already satisfied: kt-legacy in c:\users\divya\anaconda3\lib
\site-packages (from keras-tuner) (1.0.5)
Requirement already satisfied: absl-py in c:\users\divya\anaconda3\lib\s
ite-packages (from keras->keras-tuner) (2.1.0)
Requirement already satisfied: numpy in c:\users\divya\anaconda3\lib\sit
e-packages (from keras->keras-tuner) (1.26.4)
Requirement already satisfied: rich in c:\users\divya\anaconda3\lib\site
-packages (from keras->keras-tuner) (13.7.1)
Requirement already satisfied: namex in c:\users\divya\anaconda3\lib\sit
e-packages (from keras->keras-tuner) (0.0.7)
Requirement already satisfied: h5py in c:\users\divya\anaconda3\lib\site
-packages (from keras->keras-tuner) (3.10.0)
Requirement already satisfied: dm-tree in c:\users\divya\anaconda3\lib\s
ite-packages (from keras->keras-tuner) (0.1.8)
Requirement already satisfied: ml-dtypes in c:\users\divya\anaconda3\lib
\site-packages (from keras->keras-tuner) (0.3.2)
Requirement already satisfied: charset-normalizer<4,>=2 in c:\users\divy
a\anaconda3\lib\site-packages (from requests->keras-tuner) (2.0.4)
Requirement already satisfied: idna<4,>=2.5 in c:\users\divya\anaconda3
\lib\site-packages (from requests->keras-tuner) (3.4)
Requirement already satisfied: urllib3<3,>=1.21.1 in c:\users\divya\anac
onda3\lib\site-packages (from requests->keras-tuner) (2.0.7)
Requirement already satisfied: certifi>=2017.4.17 in c:\users\divya\anac
onda3\lib\site-packages (from requests->keras-tuner) (2024.2.2)
Requirement already satisfied: markdown-it-py>=2.2.0 in c:\users\divya\a
naconda3\lib\site-packages (from rich->keras->keras-tuner) (3.0.0)
Requirement already satisfied: pygments<3.0.0,>=2.13.0 in c:\users\divya
\anaconda3\lib\site-packages (from rich->keras->keras-tuner) (2.15.1)
Requirement already satisfied: mdurl~=0.1 in c:\users\divya\anaconda3\li
b\site-packages (from markdown-it-py>=2.2.0->rich->keras->keras-tuner)
(0.1.2)
```

In [2]:
```
from tensorflow import keras
from tensorflow.keras import layers
from kerastuner.tuners import RandomSearch
import keras_tuner
from kerastuner.tuners import RandomSearch
```

```
C:\Users\divya\AppData\Local\Temp\ipykernel_4808\1463078486.py:3: Deprec
ationWarning: `import kerastuner` is deprecated, please use `import kera
s_tuner`.
  from kerastuner.tuners import RandomSearch
```

In [3]:
```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import seaborn as sns
import yfinance as yf
```

In [4]:
```
# Download historical data
ticker_symbol = 'AAPL'
start_date = '2010-01-01'
end_date = '2024-04-01'
data = yf.download(ticker_symbol, start=start_date, end=end_date)
data
```

```
[*********************100%%**********************]  1 of 1 completed
```

Out[4]:

	Open	High	Low	Close	Adj Close	Volume
Date						
2010-01-04	7.622500	7.660714	7.585000	7.643214	6.470739	493729600
2010-01-05	7.664286	7.699643	7.616071	7.656429	6.481930	601904800
2010-01-06	7.656429	7.686786	7.526786	7.534643	6.378826	552160000
2010-01-07	7.562500	7.571429	7.466071	7.520714	6.367033	477131200
2010-01-08	7.510714	7.571429	7.466429	7.570714	6.409361	447610800
...	...	...	...	...	...	...
2024-03-22	171.759995	173.050003	170.059998	172.279999	172.279999	71106600
2024-03-25	170.570007	171.940002	169.449997	170.850006	170.850006	54288300
2024-03-26	170.000000	171.419998	169.580002	169.710007	169.710007	57388400
2024-03-27	170.410004	173.600006	170.110001	173.309998	173.309998	60273300
2024-03-28	171.750000	172.229996	170.509995	171.479996	171.479996	65672700

3583 rows × 6 columns

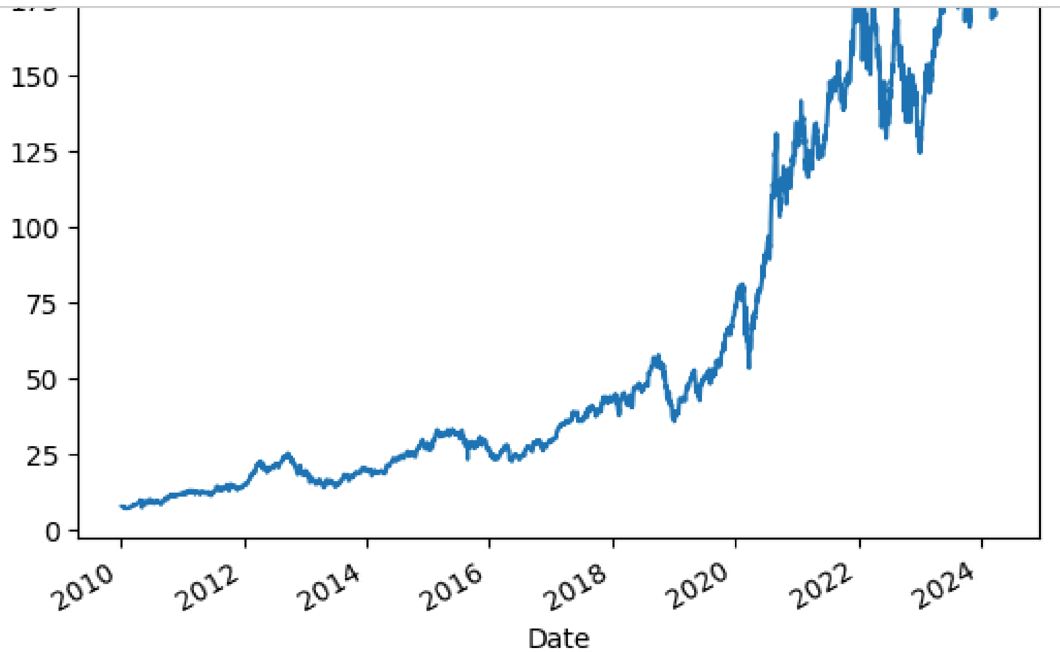
In [5]:
```
print(type(data))
```

```
<class 'pandas.core.frame.DataFrame'>
```

In [6]:
```
print(data.shape)
print(data.columns)
```

```
(3583, 6)
Index(['Open', 'High', 'Low', 'Close', 'Adj Close', 'Volume'], dtype='ob
ject')
```

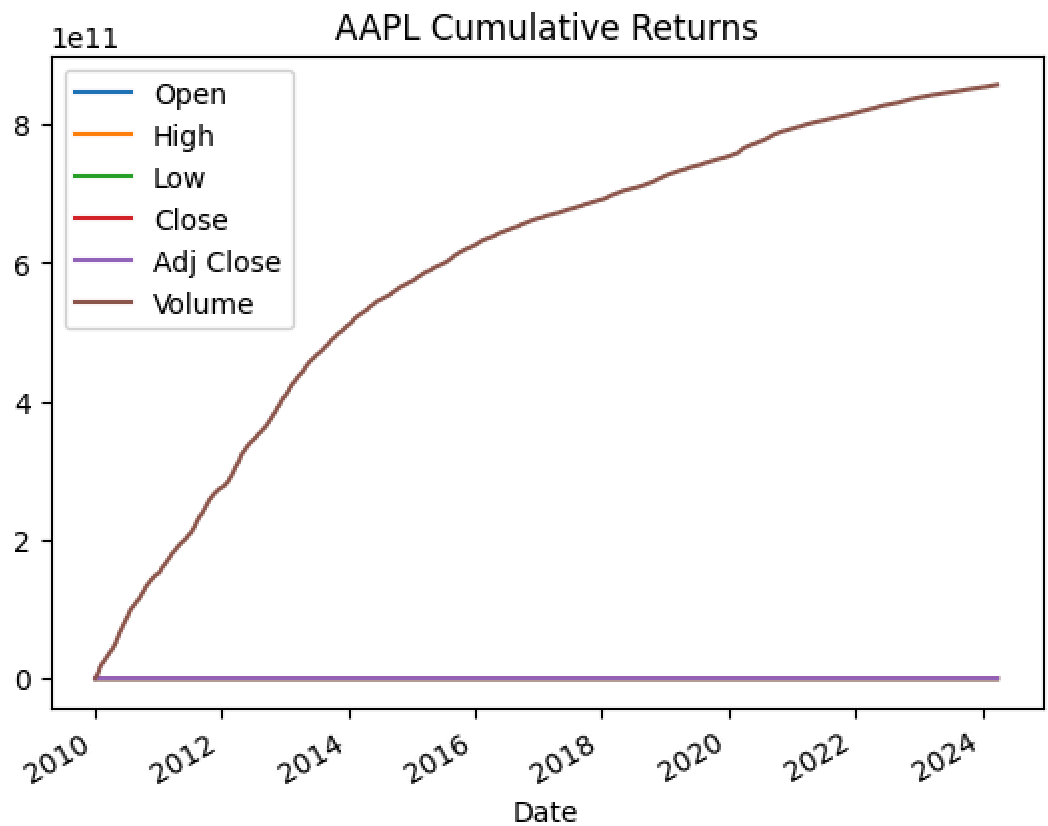
In [7]:
```
#plotting the every index plot with respect to date
for i in  data.columns:
  data[[i]].plot()
  plt.title("AAPL")
  plt.show()
```



In [8]:
```
# Comulative Return
plt.figure(figsize=(20,20))
data_1 = data.cumsum()
data_1.plot()
plt.title('AAPL Cumulative Returns')
```

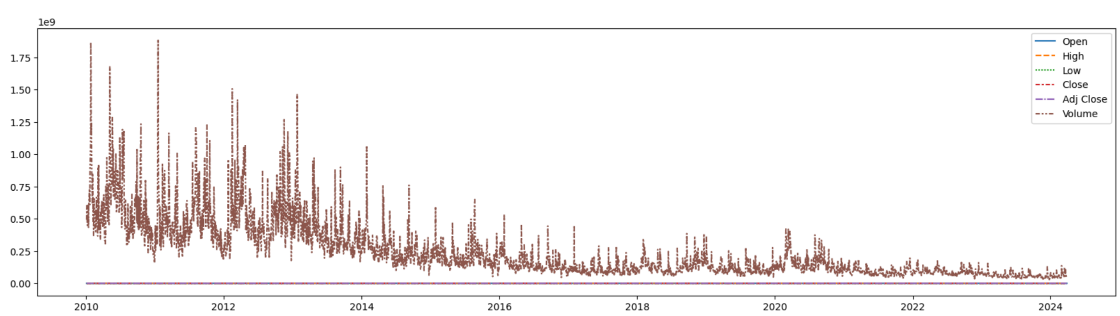
Out[8]:
```
Text(0.5, 1.0, 'AAPL Cumulative Returns')
```

```
<Figure size 2000x2000 with 0 Axes>
```



In [9]:
```
#ploting the line plot to see the trend in data set
plt.figure(figsize=(20,5))
sns.lineplot(data =data,)
```

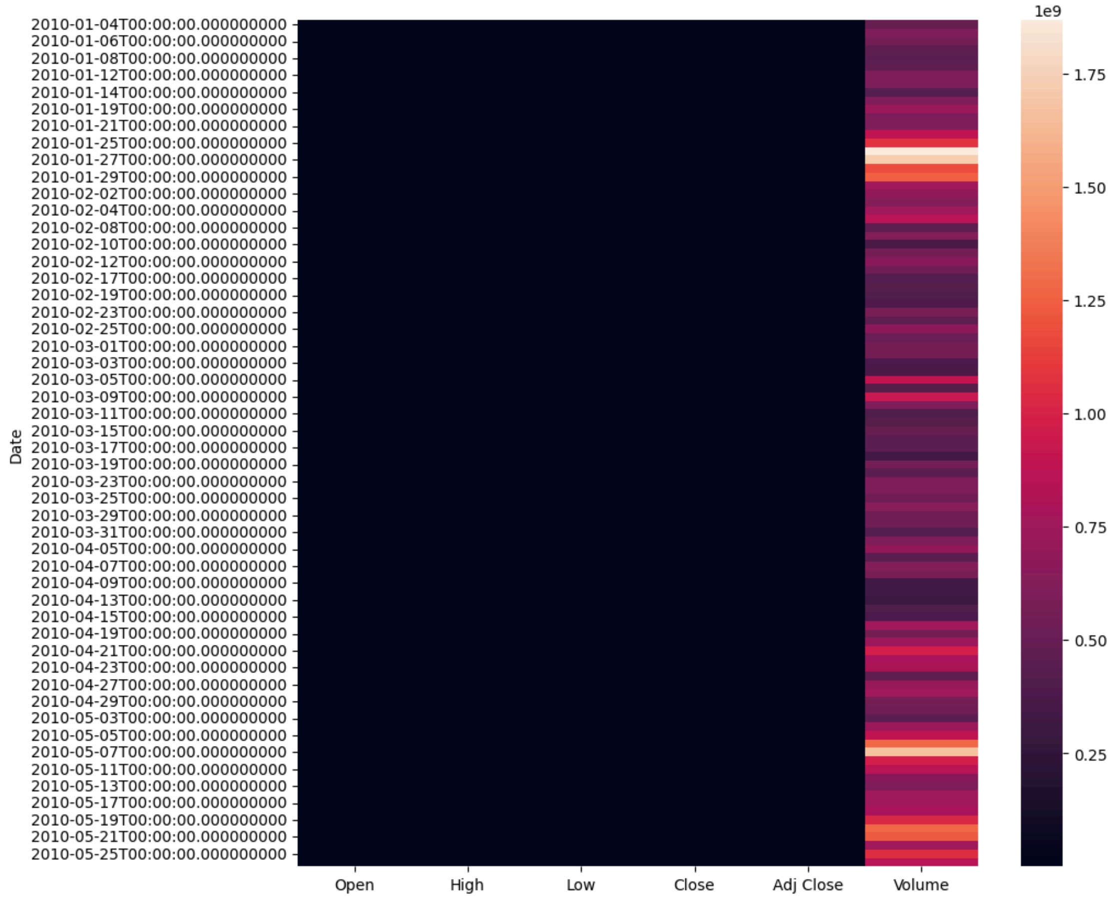
Out[9]:
```
<Axes: xlabel='Date'>
```



In [10]:

```
#drawing the heat map
plt.figure(figsize=(10,10))
sns.heatmap(data[:100],   robust=False,
                annot=None, fmt='.2g', annot_kws=None, linewidths=0, linec
                 square=False, xticklabels='auto', yticklabels='auto', mas
```

Out[10]: <Axes: ylabel='Date'>

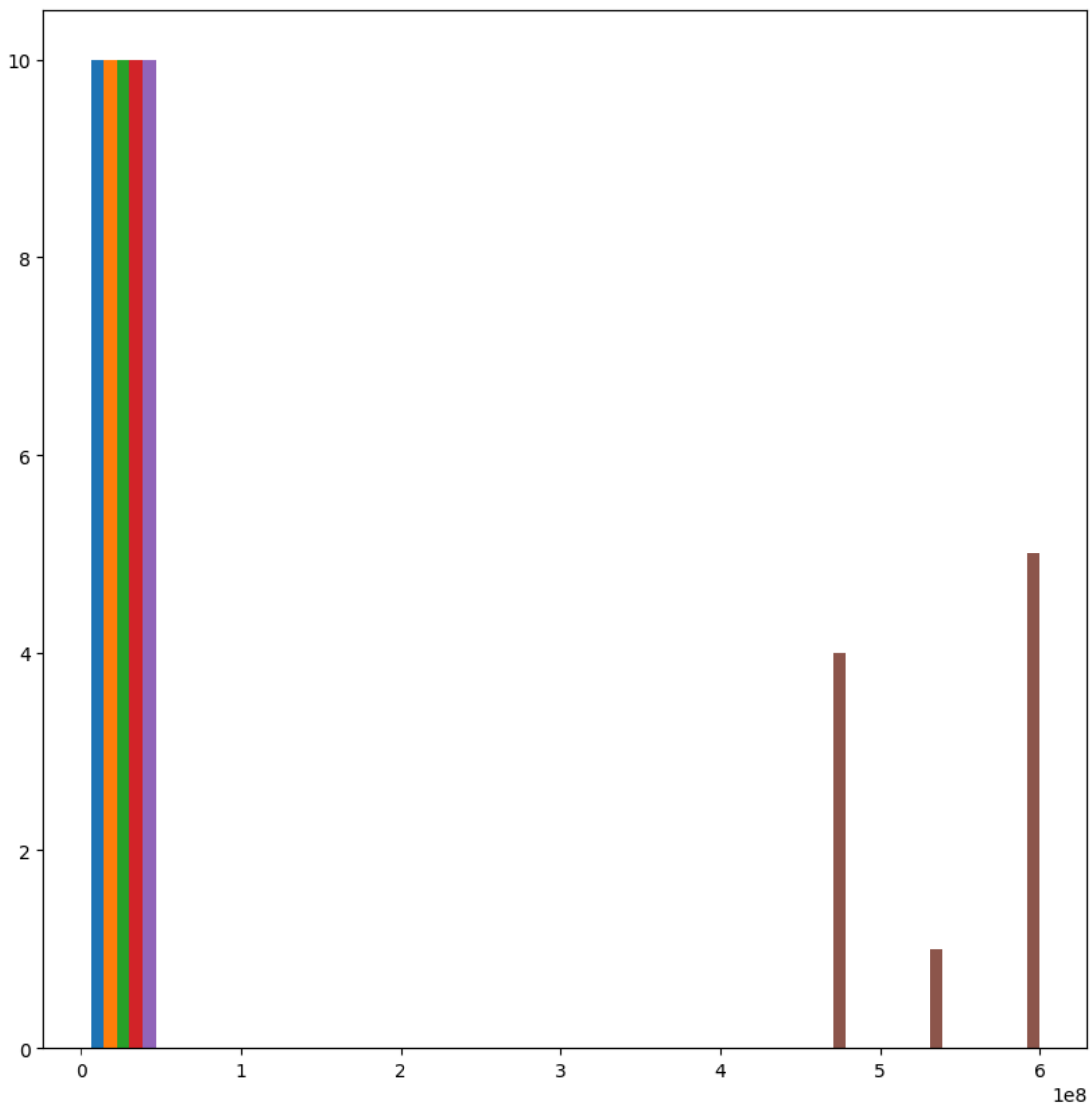


In [11]:

```python
# histogram
plt.figure(figsize=(10,10))
plt.hist(data[:10],
            bottom=None,
            histtype='bar',
            align='mid', orientation='vertical', rwidth=None)
```

Out[11]:

```
(array([[10.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
        [10.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
        [10.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
        [10.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
        [10.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  4.,  1.,  5.]]),
 array([6.22643757e+00, 6.05892056e+07, 1.21178405e+08, 1.81767604e+08,
        2.42356804e+08, 3.02946003e+08, 3.63535202e+08, 4.24124402e+08,
        4.84713601e+08, 5.45302801e+08, 6.05892000e+08]),
 <a list of 6 BarContainer objects>)
```



The model described in the code is a deep learning model for time series forecasting using Long Short-Term Memory (LSTM) networks. It consists of three LSTM layers with decreasing units, followed by a single dense layer for prediction. The model architecture is designed to capture temporal dependencies in the data over a sequence length of 100 time steps.

In [12]:
```
#LSTM is very sensitive neural network so we have to normalize the data se
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler(feature_range=(0,1))
data=scaler.fit_transform(data)
```

In [13]:
```
from sklearn.model_selection import train_test_split

train_size = int(len(data) * 0.8)
test_size = len(data) - train_size

# Split the data into training and testing sets
train, test = data[:train_size], data[train_size:]

# Print the shapes to verify the split
print("Training data shape:", train.shape)
print("Testing data shape:", test.shape)
```

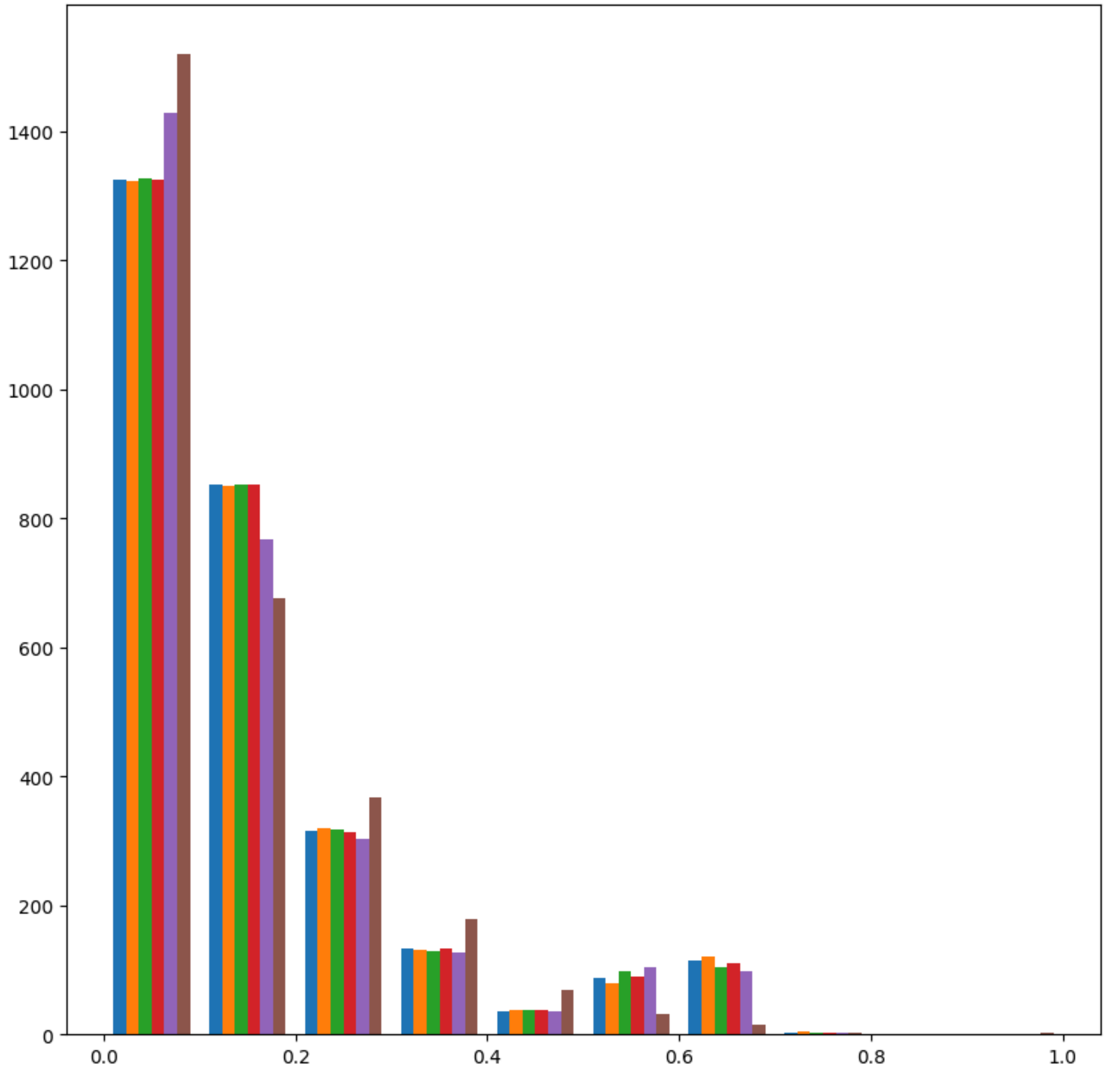
```
Training data shape: (2866, 6)
Testing data shape: (717, 6)
```

In [14]:

```
plt.figure(figsize=(10,10))
plt.hist(train,
            bottom=None,
            histtype='bar',
            align='mid', orientation='vertical', rwidth=None)
```

Out[14]:

```
(array([[1.325e+03, 8.530e+02, 3.160e+02, 1.330e+02, 3.500e+01, 8.700e+0
1,
         1.140e+02, 3.000e+00, 0.000e+00, 0.000e+00],
        [1.323e+03, 8.510e+02, 3.190e+02, 1.320e+02, 3.700e+01, 8.000e+0
1,
         1.200e+02, 4.000e+00, 0.000e+00, 0.000e+00],
        [1.327e+03, 8.520e+02, 3.170e+02, 1.290e+02, 3.800e+01, 9.700e+0
1,
         1.040e+02, 2.000e+00, 0.000e+00, 0.000e+00],
        [1.325e+03, 8.520e+02, 3.140e+02, 1.330e+02, 3.800e+01, 9.000e+0
1,
         1.110e+02, 3.000e+00, 0.000e+00, 0.000e+00],
        [1.428e+03, 7.680e+02, 3.030e+02, 1.270e+02, 3.600e+01, 1.040e+0
2,
         9.800e+01, 2.000e+00, 0.000e+00, 0.000e+00],
        [1.520e+03, 6.770e+02, 3.670e+02, 1.790e+02, 6.900e+01, 3.100e+0
1,
         1.600e+01, 3.000e+00, 1.000e+00, 3.000e+00]]),
 array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ]),
 <a list of 6 BarContainer objects>)
```


1400
1200
1000
800
600
400
200
0
0.0
0.2
0.4
0.6
0.8
1.0

In [15]:
```
plt.figure(figsize=(10,10))
plt.hist(test,
            bottom=None,
            histtype='bar',
            align='mid', orientation='vertical', rwidth=None)
```

Out[15]:
```
(array([[  0.,   0.,   0.,   0.,   0.,   0.,  84., 258., 230., 145.],
        [  0.,   0.,   0.,   0.,   0.,   0.,  75., 261., 237., 144.],
        [  0.,   0.,   0.,   0.,   0.,   0.,  97., 254., 233., 133.],
        [  0.,   0.,   0.,   0.,   0.,   0.,  82., 261., 228., 146.],
        [  0.,   0.,   0.,   0.,   0.,   0.,  94., 258., 227., 138.],
        [717.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
0.]]),
 array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ]),
 <a list of 6 BarContainer objects>)
```

In [16]:
```
import numpy as np
def create_dataset(dataset,time_stamp =1):
  X, Y = [], []
  for i in range(len(dataset)-time_stamp-1):
    a= dataset[i:(i+time_stamp),0]
    X.append(a)
    Y.append(data[i+time_stamp,0])
  return np.array(X),np.array(Y)
```

In [17]:
```
time_stamp=100
x_train, y_train=create_dataset(train,time_stamp)
x_test, y_test = create_dataset(test, time_stamp)
```

In [18]:
```
print(x_train.shape)
print(x_test.shape)
print(y_train.shape)
print(y_test.shape)
```

```
(2765, 100)
(616, 100)
(2765,)
(616,)
```

In [19]:
```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import LSTM
```

In [20]:
```
model=Sequential()
model.add(LSTM(100,return_sequences=True,input_shape=(100,1)))
model.add(LSTM(100,return_sequences=True))
model.add(LSTM(50,return_sequences=False))
model.add(Dense(1))
model.compile(loss='mean_squared_error',optimizer='adam')
```

```
C:\Users\divya\anaconda3\Lib\site-packages\keras\src\layers\rnn\rnn.py:2
05: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a
layer. When using Sequential models, prefer using an `Input(shape)` obje
ct as the first layer in the model instead.
  super().__init__(**kwargs)
```

In [21]:
```
model.summary()
```

Model: "sequential"

Layer (type)	Output Shape
lstm (LSTM)	(None, 100, 100)
lstm_1 (LSTM)	(None, 100, 100)
lstm_2 (LSTM)	(None, 50)
dense (Dense)	(None, 1)

Total params: 151,451 (591.61 KB)

Trainable params: 151,451 (591.61 KB)

Non-trainable params: 0 (0.00 B)

In [22]:
```
history=model.fit(
    x_train,y_train,
    validation_split=0.1,
    shuffle=False,
    epochs=50,batch_size=16,verbose=1)
```

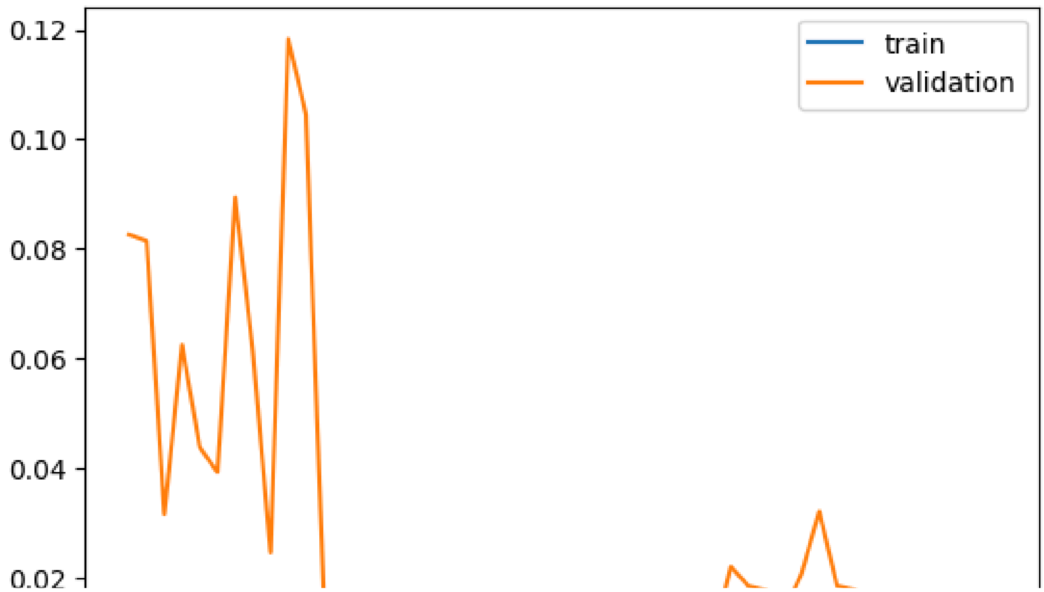
```
Epoch 1/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 21s 111ms/step - loss: 1.5010e-04 - val_los
s: 0.0826
Epoch 2/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 16s 100ms/step - loss: 0.0045 - val_loss:
0.0814
Epoch 3/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 17s 107ms/step - loss: 0.0072 - val_loss:
0.0315
Epoch 4/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 17s 106ms/step - loss: 0.0116 - val_loss:
0.0624
Epoch 5/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 17s 108ms/step - loss: 0.0136 - val_loss:
0.0437
Epoch 6/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 17s 109ms/step - loss: 0.0154 - val_loss:
0.0392
Epoch 7/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 16s 103ms/step - loss: 0.0156 - val_loss:
0.0893
Epoch 8/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 17s 106ms/step - loss: 0.0104 - val_loss:
0.0613
Epoch 9/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 17s 110ms/step - loss: 0.0129 - val_loss:
0.0245
Epoch 10/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 16s 101ms/step - loss: 0.0150 - val_loss:
0.1183
Epoch 11/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 18s 114ms/step - loss: 0.0129 - val_loss:
0.1045
Epoch 12/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 18s 113ms/step - loss: 0.0100 - val_loss:
0.0164
Epoch 13/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 16s 102ms/step - loss: 0.0071 - val_loss:
0.0073
Epoch 14/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 17s 111ms/step - loss: 0.0041 - val_loss:
5.2630e-04
Epoch 15/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 17s 112ms/step - loss: 0.0025 - val_loss:
0.0042
Epoch 16/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 16s 105ms/step - loss: 0.0012 - val_loss:
0.0041
Epoch 17/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 17s 108ms/step - loss: 8.3443e-04 - val_los
s: 0.0068
Epoch 18/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 18s 118ms/step - loss: 4.3544e-04 - val_los
s: 0.0070
Epoch 19/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 18s 115ms/step - loss: 2.1902e-04 - val_los
s: 0.0116
```

```
Epoch 20/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 17s 106ms/step - loss: 1.6886e-04 - val_los
s: 0.0124
Epoch 21/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 17s 109ms/step - loss: 1.7225e-04 - val_los
s: 0.0082
Epoch 22/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 16s 104ms/step - loss: 1.7936e-04 - val_los
s: 0.0071
Epoch 23/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 17s 111ms/step - loss: 1.8032e-04 - val_los
s: 0.0050
Epoch 24/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 17s 107ms/step - loss: 1.1137e-04 - val_los
s: 0.0063
Epoch 25/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 17s 109ms/step - loss: 1.0331e-04 - val_los
s: 0.0031
Epoch 26/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 18s 116ms/step - loss: 1.2443e-04 - val_los
s: 0.0038
Epoch 27/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 18s 113ms/step - loss: 1.1682e-04 - val_los
s: 0.0043
Epoch 28/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 17s 109ms/step - loss: 1.0123e-04 - val_los
s: 0.0046
Epoch 29/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 18s 115ms/step - loss: 8.2705e-05 - val_los
s: 0.0051
Epoch 30/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 16s 101ms/step - loss: 6.7399e-05 - val_los
s: 0.0061
Epoch 31/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 18s 112ms/step - loss: 5.8294e-05 - val_los
s: 0.0076
Epoch 32/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 18s 114ms/step - loss: 5.3483e-05 - val_los
s: 0.0097
Epoch 33/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 18s 113ms/step - loss: 5.0032e-05 - val_los
s: 0.0125
Epoch 34/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 17s 111ms/step - loss: 4.6678e-05 - val_los
s: 0.0088
Epoch 35/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 16s 105ms/step - loss: 7.7055e-05 - val_los
s: 0.0219
Epoch 36/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 18s 116ms/step - loss: 4.9409e-04 - val_los
s: 0.0184
Epoch 37/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 16s 105ms/step - loss: 2.9345e-04 - val_los
s: 0.0176
Epoch 38/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 17s 112ms/step - loss: 7.8401e-04 - val_los
s: 0.0137
```

```
Epoch 39/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 17s 110ms/step - loss: 0.0014 - val_loss:
0.0206
Epoch 40/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 17s 112ms/step - loss: 0.0013 - val_loss:
0.0320
Epoch 41/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 16s 100ms/step - loss: 9.9715e-04 - val_los
s: 0.0184
Epoch 42/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 18s 116ms/step - loss: 7.7219e-04 - val_los
s: 0.0177
Epoch 43/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 17s 111ms/step - loss: 2.9598e-04 - val_los
s: 0.0149
Epoch 44/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 17s 109ms/step - loss: 2.6773e-05 - val_los
s: 0.0137
Epoch 45/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 16s 101ms/step - loss: 8.1742e-06 - val_los
s: 0.0114
Epoch 46/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 17s 109ms/step - loss: 2.9948e-05 - val_los
s: 0.0102
Epoch 47/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 16s 105ms/step - loss: 3.6962e-05 - val_los
s: 0.0095
Epoch 48/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 17s 108ms/step - loss: 3.7652e-05 - val_los
s: 0.0090
Epoch 49/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 16s 104ms/step - loss: 3.8524e-05 - val_los
s: 0.0091
Epoch 50/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 16s 105ms/step - loss: 4.2038e-05 - val_los
s: 0.0096
```

In [23]:
```
plt.plot(history.history['loss'],label='train')
plt.plot(history.history['val_loss'],label='validation')
plt.legend()
```

Out[23]: <matplotlib.legend.Legend at 0x2e761f44090>



In [24]:
```
train_predict=model.predict(x_train)
test_predict=model.predict(x_test)
```

```
87/87 ━━━━━━━━━━━━━━━━━━━━ 5s 54ms/step
20/20 ━━━━━━━━━━━━━━━━━━━━ 1s 42ms/step
```

In [25]:
```
print(train_predict.shape)
print(test_predict.shape)
```

```
(2765, 1)
(616, 1)
```

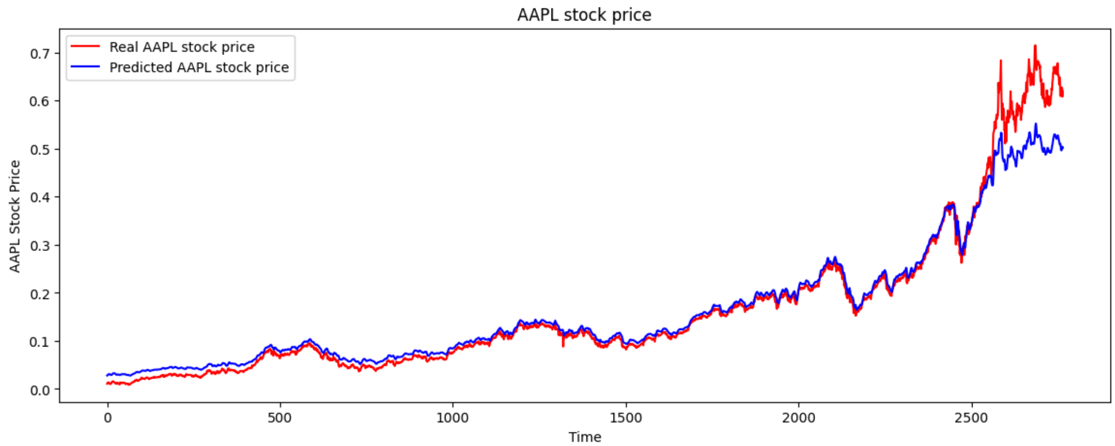
In [26]:
```
#calcualtion of RMSE
import math
from sklearn.metrics import mean_squared_error, precision_score,recall_sco
math.sqrt(mean_squared_error(y_train,train_predict))
from sklearn.metrics import confusion_matrix
x=confusion_matrix=(x_test, model.predict(x_test))
```

```
20/20 ━━━━━━━━━━━━━━━━━━━━ 1s 33ms/step
```

In [27]:
```
math.sqrt(mean_squared_error(y_test,test_predict))
```

Out[27]: 0.5419581749285586

In [28]:
```
# Visualising the results
plt.figure(figsize=(14,5))
plt.plot(y_train, color = 'red', label = 'Real AAPL stock price')
plt.plot(train_predict, color = 'blue', label = 'Predicted AAPL stock pric
plt.title('AAPL stock price')
plt.xlabel('Time')
plt.ylabel('AAPL Stock Price')
plt.legend()
plt.show()
```



#RNN

In [29]:
```
model=Sequential()
model.add(tf.keras.layers.GRU(100,return_sequences=True,input_shape=(100,1
model.add(tf.keras.layers.GRU(50,return_sequences=False))
model.add(Dense(1))
model.compile(loss='mean_squared_error',optimizer='adam')
```

```
C:\Users\divya\anaconda3\Lib\site-packages\keras\src\layers\rnn\rnn.py:2
05: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a
layer. When using Sequential models, prefer using an `Input(shape)` obje
ct as the first layer in the model instead.
  super().__init__(**kwargs)
```

In [30]:

```
history=model.fit(
    x_train,y_train,
    validation_split=0.1,
    shuffle=False,
    epochs=50,batch_size=16,verbose=1)
```

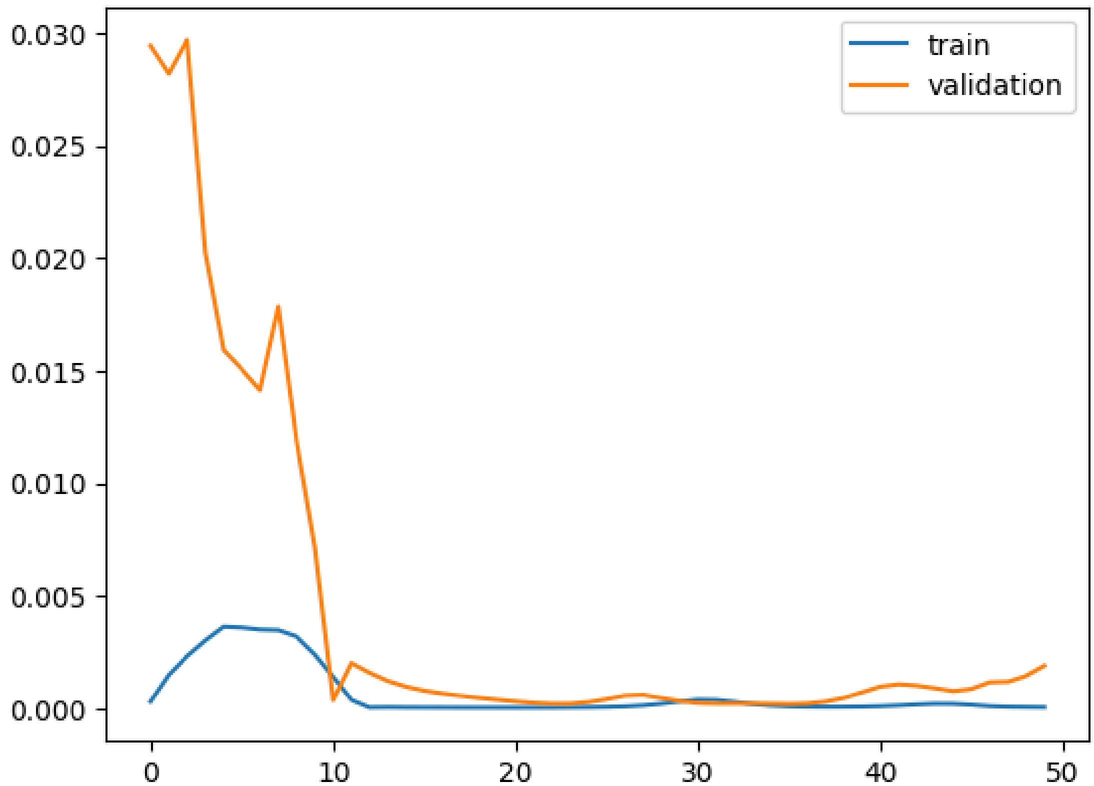
```
Epoch 1/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 13s 65ms/step - loss: 1.2052e-04 - val_los
s: 0.0294
Epoch 2/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 12s 78ms/step - loss: 0.0035 - val_loss: 0.
0282
Epoch 3/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 11s 72ms/step - loss: 0.0065 - val_loss: 0.
0297
Epoch 4/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 12s 78ms/step - loss: 0.0092 - val_loss: 0.
0202
Epoch 5/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 13s 81ms/step - loss: 0.0115 - val_loss: 0.
0159
Epoch 6/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 12s 80ms/step - loss: 0.0110 - val_loss: 0.
0151
Epoch 7/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 12s 79ms/step - loss: 0.0108 - val_loss: 0.
0141
Epoch 8/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 12s 77ms/step - loss: 0.0108 - val_loss: 0.
0179
Epoch 9/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 12s 79ms/step - loss: 0.0098 - val_loss: 0.
0119
Epoch 10/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 12s 78ms/step - loss: 0.0077 - val_loss: 0.
0072
Epoch 11/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 13s 81ms/step - loss: 0.0048 - val_loss: 3.
7478e-04
Epoch 12/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 12s 80ms/step - loss: 0.0013 - val_loss: 0.
0020
Epoch 13/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 12s 80ms/step - loss: 8.5979e-05 - val_los
s: 0.0016
Epoch 14/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 12s 79ms/step - loss: 1.2798e-04 - val_los
s: 0.0012
Epoch 15/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 12s 76ms/step - loss: 1.0682e-04 - val_los
s: 9.6994e-04
Epoch 16/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 13s 83ms/step - loss: 1.0608e-04 - val_los
s: 7.9246e-04
Epoch 17/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 12s 74ms/step - loss: 1.0199e-04 - val_los
s: 6.6353e-04
Epoch 18/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 11s 74ms/step - loss: 9.8231e-05 - val_los
s: 5.6226e-04
Epoch 19/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 13s 81ms/step - loss: 9.4905e-05 - val_los
s: 4.7427e-04
```

```
Epoch 20/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 12s 79ms/step - loss: 9.2605e-05 - val_los
s: 3.9199e-04
Epoch 21/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 12s 77ms/step - loss: 9.2075e-05 - val_los
s: 3.1511e-04
Epoch 22/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 12s 79ms/step - loss: 9.4170e-05 - val_los
s: 2.5082e-04
Epoch 23/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 13s 80ms/step - loss: 9.9876e-05 - val_los
s: 2.1259e-04
Epoch 24/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 13s 85ms/step - loss: 1.1051e-04 - val_los
s: 2.1733e-04
Epoch 25/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 11s 70ms/step - loss: 1.2831e-04 - val_los
s: 2.8199e-04
Epoch 26/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 12s 76ms/step - loss: 1.5783e-04 - val_los
s: 4.1467e-04
Epoch 27/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 13s 81ms/step - loss: 2.0914e-04 - val_los
s: 5.7579e-04
Epoch 28/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 12s 77ms/step - loss: 3.0451e-04 - val_los
s: 6.0889e-04
Epoch 29/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 12s 75ms/step - loss: 4.8595e-04 - val_los
s: 4.6219e-04
Epoch 30/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 12s 74ms/step - loss: 7.7935e-04 - val_los
s: 3.3353e-04
Epoch 31/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 14s 88ms/step - loss: 0.0010 - val_loss: 2.
4576e-04
Epoch 32/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 13s 86ms/step - loss: 0.0011 - val_loss: 2.
2189e-04
Epoch 33/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 14s 89ms/step - loss: 8.3118e-04 - val_los
s: 2.2319e-04
Epoch 34/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 12s 75ms/step - loss: 5.5710e-04 - val_los
s: 2.2473e-04
Epoch 35/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 13s 85ms/step - loss: 3.6222e-04 - val_los
s: 2.1410e-04
Epoch 36/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 13s 86ms/step - loss: 2.5089e-04 - val_los
s: 2.0274e-04
Epoch 37/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 13s 82ms/step - loss: 1.9528e-04 - val_los
s: 2.2303e-04
Epoch 38/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 13s 80ms/step - loss: 1.7302e-04 - val_los
s: 3.0511e-04
```

```
Epoch 39/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 13s 82ms/step - loss: 1.7180e-04 - val_los
s: 4.7098e-04
Epoch 40/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 13s 85ms/step - loss: 1.8846e-04 - val_los
s: 7.1638e-04
Epoch 41/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 13s 80ms/step - loss: 2.2678e-04 - val_los
s: 9.7116e-04
Epoch 42/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 13s 86ms/step - loss: 2.9595e-04 - val_los
s: 0.0011
Epoch 43/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 13s 84ms/step - loss: 4.0186e-04 - val_los
s: 0.0010
Epoch 44/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 13s 83ms/step - loss: 5.1264e-04 - val_los
s: 9.0182e-04
Epoch 45/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 14s 90ms/step - loss: 5.4439e-04 - val_los
s: 7.7222e-04
Epoch 46/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 13s 86ms/step - loss: 4.5451e-04 - val_los
s: 8.6871e-04
Epoch 47/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 12s 79ms/step - loss: 2.8124e-04 - val_los
s: 0.0012
Epoch 48/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 14s 90ms/step - loss: 1.7082e-04 - val_los
s: 0.0012
Epoch 49/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 13s 83ms/step - loss: 1.4786e-04 - val_los
s: 0.0014
Epoch 50/50
156/156 ━━━━━━━━━━━━━━━━━━━━ 14s 90ms/step - loss: 1.2229e-04 - val_los
s: 0.0019
```

In [31]:
```
plt.plot(history.history['loss'],label='train')
plt.plot(history.history['val_loss'],label='validation')
plt.legend()
```

Out[31]:
```
<matplotlib.legend.Legend at 0x2e772562990>
```



In [32]:
```
train_predict.shape
```

Out[32]:
```
(2765, 1)
```

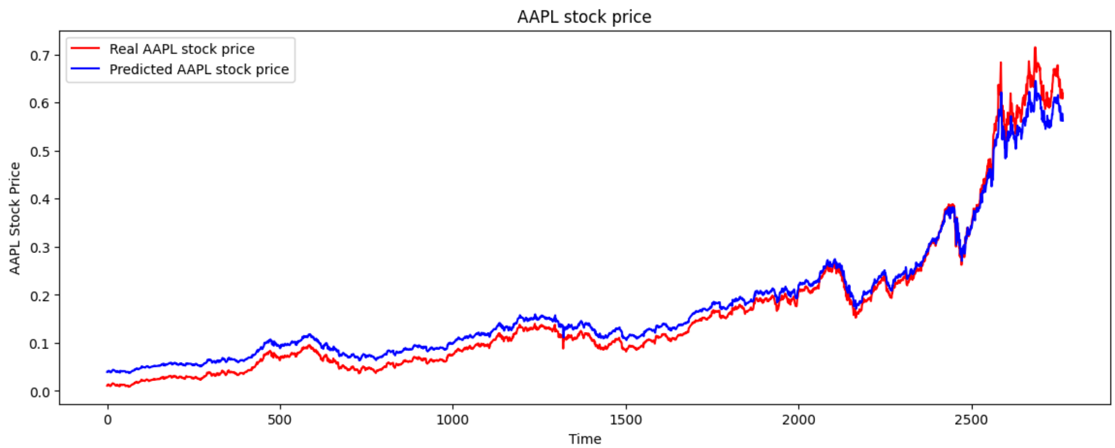
In [33]:
```
train_predict=model.predict(x_train)
test_predict=model.predict(x_test)
```

```
87/87 ━━━━━━━━━━━━━━━━━━━━ 4s 37ms/step
20/20 ━━━━━━━━━━━━━━━━━━━━ 1s 37ms/step
```

In [34]:
```
math.sqrt(mean_squared_error(y_train,train_predict))
```

Out[34]:
```
0.024831703098660302
```

In [35]:
```
# Visualising the results
plt.figure(figsize=(14,5))
plt.plot(y_train, color = 'red', label = 'Real AAPL stock price')
plt.plot(train_predict, color = 'blue', label = 'Predicted AAPL stock pric
plt.title('AAPL stock price')
plt.xlabel('Time')
plt.ylabel('AAPL Stock Price')
plt.legend()
plt.show()
```



In [36]:
```
#LSTM+RNN
```

The model employed in the code is a combination of LSTM and GRU layers, which are recurrent neural network (RNN) variants known for their ability to capture sequential patterns in data. Dropout layers are included to prevent overfitting by randomly dropping a proportion of connections during training. The model is trained using the Adam optimizer with a learning rate of 0.001 and mean squared error (MSE) loss function over 20 epochs with a batch size of 16.

In [37]:
```
model=Sequential()
model.add(LSTM(100,return_sequences=True,input_shape=(100,1)))
model.add(layer = tf.keras.layers.Dropout(.2,))
model.add(tf.keras.layers.GRU(50,return_sequences=False))
model.add(tf.keras.layers.Dropout(.2,))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer = tf.keras.optimizers.A
```

```
C:\Users\divya\anaconda3\Lib\site-packages\keras\src\layers\rnn\rnn.py:2
05: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a
layer. When using Sequential models, prefer using an `Input(shape)` obje
ct as the first layer in the model instead.
  super().__init__(**kwargs)
```

In [38]:

```
history=model.fit(
    x_train,y_train,
    validation_split=0.1,
    shuffle=False,
    epochs=20,batch_size=16,verbose=1,)
```

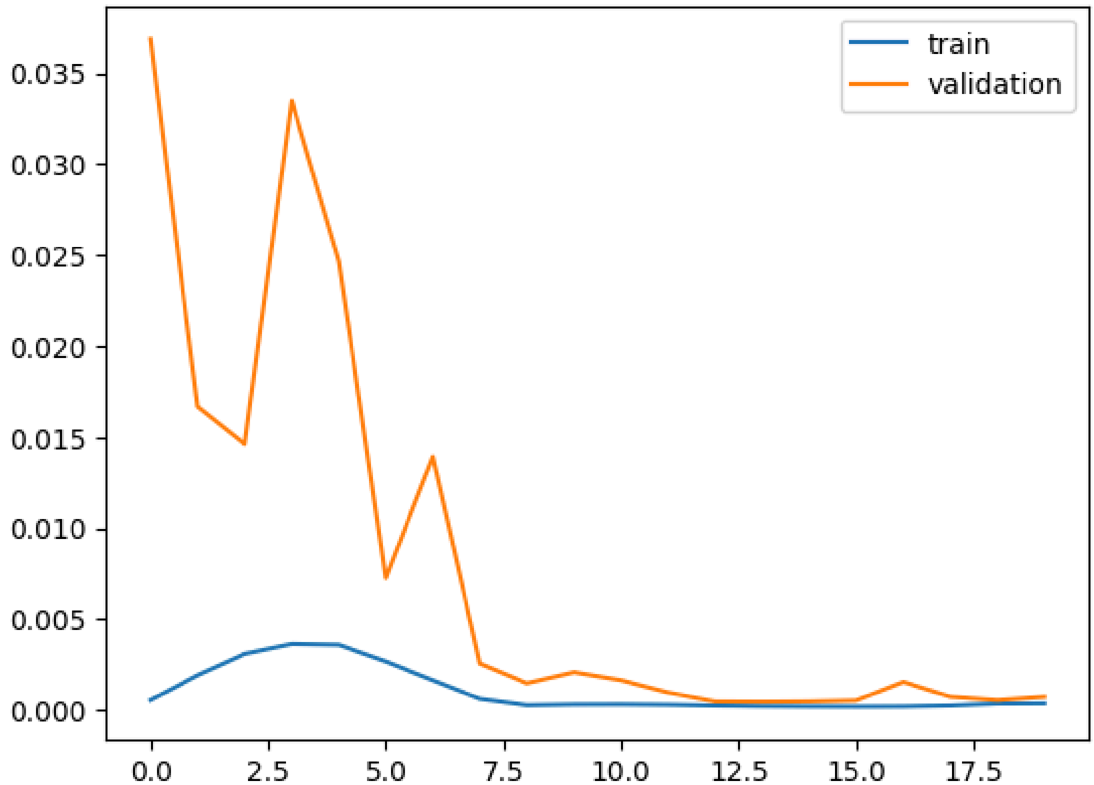
```
Epoch 1/20
156/156 ━━━━━━━━━━━━━━━━━━━━ 18s 92ms/step - loss: 1.7579e-04 - val_los
s: 0.0369
Epoch 2/20
156/156 ━━━━━━━━━━━━━━━━━━━━ 13s 84ms/step - loss: 0.0046 - val_loss: 0.
0167
Epoch 3/20
156/156 ━━━━━━━━━━━━━━━━━━━━ 15s 94ms/step - loss: 0.0087 - val_loss: 0.
0146
Epoch 4/20
156/156 ━━━━━━━━━━━━━━━━━━━━ 15s 97ms/step - loss: 0.0102 - val_loss: 0.
0335
Epoch 5/20
156/156 ━━━━━━━━━━━━━━━━━━━━ 15s 98ms/step - loss: 0.0106 - val_loss: 0.
0246
Epoch 6/20
156/156 ━━━━━━━━━━━━━━━━━━━━ 15s 98ms/step - loss: 0.0079 - val_loss: 0.
0073
Epoch 7/20
156/156 ━━━━━━━━━━━━━━━━━━━━ 15s 99ms/step - loss: 0.0050 - val_loss: 0.
0139
Epoch 8/20
156/156 ━━━━━━━━━━━━━━━━━━━━ 16s 104ms/step - loss: 0.0019 - val_loss:
0.0025
Epoch 9/20
156/156 ━━━━━━━━━━━━━━━━━━━━ 17s 106ms/step - loss: 4.5211e-04 - val_los
s: 0.0015
Epoch 10/20
156/156 ━━━━━━━━━━━━━━━━━━━━ 16s 104ms/step - loss: 5.0641e-04 - val_los
s: 0.0021
Epoch 11/20
156/156 ━━━━━━━━━━━━━━━━━━━━ 16s 104ms/step - loss: 5.2672e-04 - val_los
s: 0.0016
Epoch 12/20
156/156 ━━━━━━━━━━━━━━━━━━━━ 17s 110ms/step - loss: 6.3742e-04 - val_los
s: 9.5499e-04
Epoch 13/20
156/156 ━━━━━━━━━━━━━━━━━━━━ 17s 106ms/step - loss: 4.7004e-04 - val_los
s: 4.5583e-04
Epoch 14/20
156/156 ━━━━━━━━━━━━━━━━━━━━ 16s 104ms/step - loss: 3.7396e-04 - val_los
s: 4.4355e-04
Epoch 15/20
156/156 ━━━━━━━━━━━━━━━━━━━━ 16s 104ms/step - loss: 3.3420e-04 - val_los
s: 4.6182e-04
Epoch 16/20
156/156 ━━━━━━━━━━━━━━━━━━━━ 16s 102ms/step - loss: 3.2627e-04 - val_los
s: 5.1613e-04
Epoch 17/20
156/156 ━━━━━━━━━━━━━━━━━━━━ 16s 102ms/step - loss: 3.1760e-04 - val_los
s: 0.0015
Epoch 18/20
156/156 ━━━━━━━━━━━━━━━━━━━━ 16s 105ms/step - loss: 3.4798e-04 - val_los
s: 7.2076e-04
Epoch 19/20
156/156 ━━━━━━━━━━━━━━━━━━━━ 16s 103ms/step - loss: 7.2704e-04 - val_los
s: 5.3350e-04
```

```
Epoch 20/20
156/156 ━━━━━━━━━━━━━━━━━━━━ 16s 104ms/step - loss: 7.5948e-04 - val_los
s: 7.1973e-04
```

In [39]:

```
plt.plot(history.history['loss'],label='train')
plt.plot(history.history['val_loss'],label='validation')
plt.legend()
```

Out[39]: <matplotlib.legend.Legend at 0x2e75f9a9b90>



In [40]:

```
print(train_predict.shape)
print(test_predict.shape)
```

```
(2765, 1)
(616, 1)
```

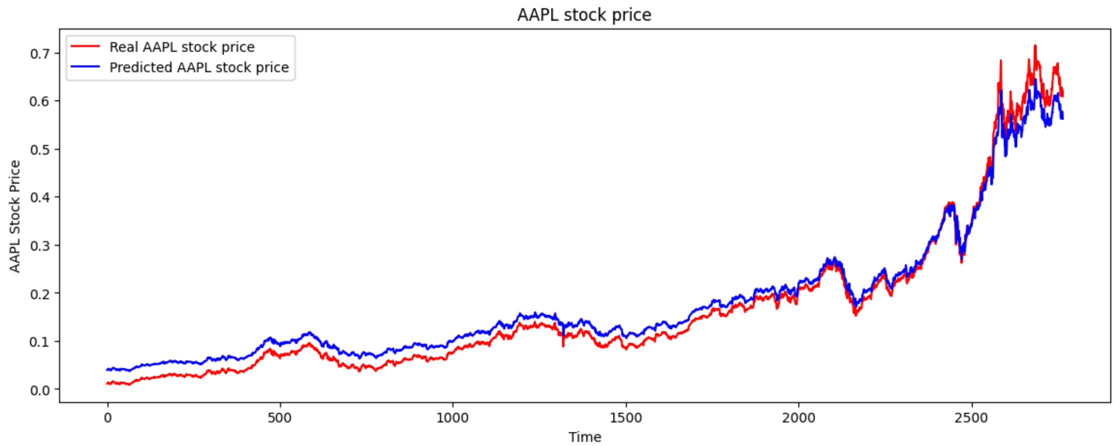
In [41]:

```
math.sqrt(mean_
0.024831703098660302squared_error(y_train,train_predict))
```

Out[41]: 0.024831703098660302

In [42]:

```
# Visualising the results
plt.figure(figsize=(14,5))
plt.plot(y_train, color = 'red', label = 'Real AAPL stock price')
plt.plot(train_predict, color = 'blue', label = 'Predicted AAPL stock pric
plt.title('AAPL stock price')
plt.xlabel('Time')
plt.ylabel('AAPL Stock Price')
plt.legend()
plt.show()
```



In [43]:

```
#SVM
```

This code segment downloads historical stock data using the Yahoo Finance API for the ticker symbol 'AAPL' within the specified date range. It then preprocesses the data by splitting it into features (X) and the target variable (y). After splitting the data into training and testing sets, it scales the features using StandardScaler. Finally, it trains a Support Vector Regression (SVR) model with a linear kernel on the scaled training data.

In [44]:

```
import yfinance as yf
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVR
from sklearn.metrics import mean_squared_error

ticker_symbol = 'AAPL'
start_date = '2010-01-01'
end_date = '2024-04-19'
data = yf.download(ticker_symbol, start=start_date, end=end_date)
# Convert data to DataFrame if it's not already
if not isinstance(data, pd.DataFrame):
    data = pd.DataFrame(data)

# Preprocess the data
X = data[['Open', 'High', 'Low', 'Close', 'Volume']]  # Features
y = data['Close']  # Target variable

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, r

# Scale the features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Train the SVM model
svm_model = SVR(kernel='linear')  # You can choose the kernel type (linear
svm_model.fit(X_train_scaled, y_train)
```

```
[*********************100%%**********************]  1 of 1 completed
```

Out[44]:

```
SVR(kernel='linear')
```

In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

In [45]:

```
# Make predictions
y_pred = svm_model.predict(X_test_scaled)

mse = mean_squared_error(y_test, y_pred)
print("Mean Squared Error:", mse)
```

```
Mean Squared Error: 0.11928499684428194
```

In [46]:
```
#LSTM+SVM
```

This code segment preprocesses the data by scaling it using MinMaxScaler, then splits it into sequences for input into the LSTM model. After splitting the data into training and testing sets, it trains an LSTM model to predict the next value in the sequence. The LSTM-generated features are then used to train an SVR model. Finally, both models are combined for predictions, and the RMSE is calculated for both the training and testing sets.

In [47]:

```
from sklearn.svm import SVR
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import MinMaxScaler
from keras.models import Sequential
from keras.layers import LSTM, Dense
import numpy as np



# Preprocess the data
scaler = MinMaxScaler(feature_range=(0, 1))
scaled_data = scaler.fit_transform(data['Close'].values.reshape(-1,1))

# Split the data into sequences
def create_sequences(data, seq_length):
    X, y = [], []
    for i in range(len(data)-seq_length):
        X.append(data[i:i+seq_length])
        y.append(data[i+seq_length])
    return np.array(X), np.array(y)

sequence_length = 10
X, y = create_sequences(scaled_data, sequence_length)

# Split the data into training and testing sets
train_size = int(len(X) * 0.80)
test_size = len(X) - train_size
X_train, X_test = X[0:train_size], X[train_size:len(X)]
y_train, y_test = y[0:train_size], y[train_size:len(y)]

# Train LSTM model
model = Sequential()
model.add(LSTM(50, activation='relu', input_shape=(sequence_length, 1)))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
model.fit(X_train, y_train, epochs=100, batch_size=32, verbose=1)

# Generate features using LSTM model
train_features = model.predict(X_train)
test_features = model.predict(X_test)

# Train SVM model
svm_model = SVR(kernel='rbf')
svm_model.fit(train_features, y_train)

# Combine models for prediction
def combined_predict(X):
    features = model.predict(X)
    return svm_model.predict(features)

# Predictions
train_predictions = combined_predict(X_train)
test_predictions = combined_predict(X_test)

# Evaluate the model
train_rmse = np.sqrt(mean_squared_error(y_train, train_predictions))
test_rmse = np.sqrt(mean_squared_error(y_test, test_predictions))
```

```
print("Train RMSE:", train_rmse)
print("Test RMSE:", test_rmse)
```

```
Epoch 1/100

C:\Users\divya\anaconda3\Lib\site-packages\keras\src\layers\rnn\rnn.p
y:205: UserWarning: Do not pass an `input_shape`/`input_dim` argument
to a layer. When using Sequential models, prefer using an `Input(shap
e)` object as the first layer in the model instead.
  super().__init__(**kwargs)

90/90 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - loss: 0.0197
Epoch 2/100
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 8.2440e-05
Epoch 3/100
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 7.4068e-05
Epoch 4/100
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 7.2254e-05
Epoch 5/100
90/90 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - loss: 5.7637e-05
Epoch 6/100
90/90 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - loss: 7.0178e-05
Epoch 7/100
```

In [48]:

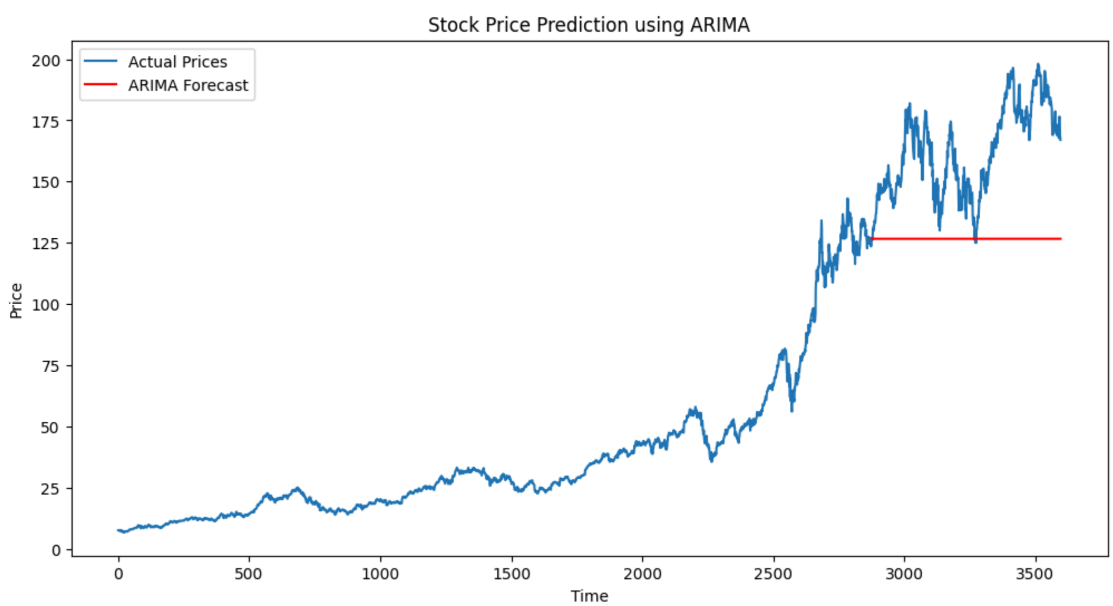
```
#ARIMA
```

This code segment checks if the provided data is either a pandas DataFrame or a NumPy array. If it's a DataFrame, it extracts the 'Close' column as a pandas Series. If it's a NumPy array, it assumes the first column contains the data. Then, it splits the data into training and testing sets, fits an ARIMA model to the training data, forecasts future values, calculates the RMSE between the forecasted values and the testing set. Finally plots the actual prices along with the ARIMA forecast.

In [49]:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from statsmodels.tsa.arima.model import ARIMA
from sklearn.metrics import mean_squared_error




if isinstance(data, pd.DataFrame):
    # Check if the 'Close' column exists
    if 'Close' in data.columns:
        # Convert the 'Close' column to a pandas Series
        series = data['Close']
    else:
        print("No 'Close' column found in the data.")
elif isinstance(data, np.ndarray):

    series = data[:, 0]
else:
    print("Unsupported data format. Please ensure 'data' is a pandas DataF

if 'series' in locals():
    # Split the data into training and testing sets
    train_size = int(len(series) * 0.80)
    train_data, test_data = series[0:train_size], series[train_size:]

    # Fit ARIMA model
    p, d, q = 5, 1, 0
    model = ARIMA(train_data, order=(p, d, q))
    fitted_model = model.fit()

    # Forecast
    forecast = fitted_model.forecast(steps=len(test_data))

    # Calculate RMSE
    mse = mean_squared_error(test_data, forecast)
    rmse = np.sqrt(mse)
    print("Root Mean Squared Error (RMSE):", rmse)

    # Plotting
    plt.figure(figsize=(12, 6))
    plt.plot(range(len(series)), series, label='Actual Prices')
    plt.plot(range(train_size, len(series)), forecast, color='red', label=
    plt.title('Stock Price Prediction using ARIMA')
    plt.xlabel('Time')
    plt.ylabel('Price')
    plt.legend()
    plt.show()
```

```
C:\Users\divya\anaconda3\Lib\site-packages\statsmodels\tsa\base\tsa_mode
l.py:473: ValueWarning: A date index has been provided, but it has no as
sociated frequency information and so will be ignored when e.g. forecast
ing.
  self._init_dates(dates, freq)
C:\Users\divya\anaconda3\Lib\site-packages\statsmodels\tsa\base\tsa_mode
l.py:473: ValueWarning: A date index has been provided, but it has no as
sociated frequency information and so will be ignored when e.g. forecast
ing.
  self._init_dates(dates, freq)
C:\Users\divya\anaconda3\Lib\site-packages\statsmodels\tsa\base\tsa_mode
l.py:473: ValueWarning: A date index has been provided, but it has no as
sociated frequency information and so will be ignored when e.g. forecast
ing.
  self._init_dates(dates, freq)
C:\Users\divya\anaconda3\Lib\site-packages\statsmodels\tsa\base\tsa_mode
l.py:836: ValueWarning: No supported index is available. Prediction resu
lts will be given with an integer index beginning at `start`.
  return get_prediction_index(
C:\Users\divya\anaconda3\Lib\site-packages\statsmodels\tsa\base\tsa_mode
l.py:836: FutureWarning: No supported index is available. In the next ve
rsion, calling this method in a model without a supported index will res
ult in an exception.
  return get_prediction_index(

Root Mean Squared Error (RMSE): 40.15777961679475
```



In [50]:
```
#LSTM, ARIMA(not of much use)
```

This code downloads historical stock data for the ticker symbol 'AAPL' from Yahoo Finance and performs time series analysis using both ARIMA and LSTM models for stock price prediction. The ARIMA model is trained on the closing prices, while the LSTM model is trained on normalized closing prices. After training, predictions are made using both models, and the results are evaluated using Root Mean Squared Error (RMSE). Finally, the actual prices, LSTM predictions, and ARIMA forecasts are plotted for comparison.

In [51]:
```
import pandas as pd
import numpy as np
from statsmodels.tsa.arima.model import ARIMA
from sklearn.preprocessing import MinMaxScaler
from keras.models import Sequential
from keras.layers import LSTM, Dense
import matplotlib.pyplot as plt
import yfinance as yf

# Download historical data
ticker_symbol = 'AAPL'
start_date = '2010-01-01'
end_date = '2024-04-01'
data = yf.download(ticker_symbol, start=start_date, end=end_date)

# Ensure that the data is in the expected format (pandas DataFrame)
if isinstance(data, pd.DataFrame):
    # Extract the 'Close' prices
    close_prices = data['Close']

    # Train ARIMA model
    model_arima = ARIMA(close_prices, order=(5, 1, 0))
    fitted_arima = model_arima.fit()

    # Generate ARIMA forecasts
    arima_forecast = fitted_arima.forecast(steps=len(close_prices))

    # Normalize data for LSTM
    scaler = MinMaxScaler(feature_range=(0, 1))
    scaled_data = scaler.fit_transform(np.array(close_prices).reshape(-1,

    # Prepare data for LSTM
    def create_dataset(data, time_steps=1):
        X, y = [], []
        for i in range(len(data) - time_steps):
            X.append(data[i:(i + time_steps), 0])
            y.append(data[i + time_steps, 0])
        return np.array(X), np.array(y)

    # Set time steps for LSTM
    time_steps = 100  # number of previous time steps to use as input feat

    # Split data into train and test sets
    X, y = create_dataset(scaled_data, time_steps)
    train_size = int(len(X) * 0.80)
    X_train, X_test = X[:train_size], X[train_size:]
    y_train, y_test = y[:train_size], y[train_size:]

    # Reshape input to be [samples, time steps, features]
    X_train = np.reshape(X_train, (X_train.shape[0], X_train.shape[1], 1))
    X_test = np.reshape(X_test, (X_test.shape[0], X_test.shape[1], 1))

    # Train LSTM model
    model_lstm = Sequential([
        LSTM(100, return_sequences=True, input_shape=(X_train.shape[1], 1)
        LSTM(100, return_sequences=True),
        LSTM(50),
```

```
        Dense(1)
    ])
    model_lstm.compile(optimizer='adam', loss='mean_squared_error')
    model_lstm.fit(X_train, y_train, epochs=50, batch_size=64, verbose=1)

    # Make predictions using LSTM
    lstm_predictions = model_lstm.predict(X_test)

    # Inverse transform predictions to original scale
    lstm_predictions = scaler.inverse_transform(lstm_predictions)

    # Calculate RMSE for LSTM
    rmse_lstm = np.sqrt(np.mean(np.square(y_test - lstm_predictions)))
    print("RMSE of LSTM model:", rmse_lstm)

    # Calculate RMSE for ARIMA
    rmse_arima = np.sqrt(np.mean(np.square(arima_forecast[:len(y_test)] -
    print("RMSE of ARIMA model:", rmse_arima)

    # Plotting
    plt.figure(figsize=(12, 6))
    plt.plot(y_test, label='Actual Prices')
    plt.plot(lstm_predictions, color='red', label='LSTM Forecast')
    plt.plot(arima_forecast[:len(y_test)], color='green', label='ARIMA For
    plt.title('Stock Price Prediction using ARIMA and LSTM')
    plt.xlabel('Time')
    plt.ylabel('Price')
    plt.legend()
    plt.show()
else:
    print("Unsupported data format. Please ensure 'data' is a pandas DataF
```

```
[*********************100%%**********************]  1 of 1 completed
C:\Users\divya\anaconda3\Lib\site-packages\statsmodels\tsa\base\tsa_m
odel.py:473: ValueWarning: A date index has been provided, but it has
no associated frequency information and so will be ignored when e.g.
forecasting.
  self._init_dates(dates, freq)
C:\Users\divya\anaconda3\Lib\site-packages\statsmodels\tsa\base\tsa_m
odel.py:473: ValueWarning: A date index has been provided, but it has
no associated frequency information and so will be ignored when e.g.
forecasting.
  self._init_dates(dates, freq)
C:\Users\divya\anaconda3\Lib\site-packages\statsmodels\tsa\base\tsa_m
odel.py:473: ValueWarning: A date index has been provided, but it has
no associated frequency information and so will be ignored when e.g.
forecasting.
  self._init_dates(dates, freq)

Epoch 1/50

C:\Users\divya\anaconda3\Lib\site-packages\statsmodels\tsa\base\tsa_m
```

In [54]:
```
#cnn
```

This code snippet downloads historical stock data for the ticker symbol 'AAPL' from Yahoo Finance, preprocesses the data using MinMaxScaler, and splits it into training and testing sets. The data is reshaped to fit the input requirements of a Convolutional Neural Network (CNN). A CNN model is then constructed with one convolutional layer, one max pooling layer, one flattening layer, and two dense layers. The model is compiled using the Adam optimizer and mean squared error loss function. It is then trained on the training data and evaluated on the test data. Predictions are made using the trained model, and the results are transformed back to the original scale using the inverse scaler. Finally, the Root Mean Squared Error (RMSE) is calculated to assess the model's performance.

In [55]:

```
import numpy as np
import pandas as pd
import yfinance as yf
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv1D, MaxPooling1D, Flatten, Dense
from sklearn.model_selection import train_test_split

# Download historical data
ticker_symbol = 'AAPL'
start_date = '2010-01-01'
end_date = '2024-04-01'
data = yf.download(ticker_symbol, start=start_date, end=end_date)

# Preprocess the data
def preprocess_data(data, window_size):
    scaler = MinMaxScaler(feature_range=(0, 1))
    scaled_data = scaler.fit_transform(data.values.reshape(-1, 1))
    X, y = [], []
    for i in range(len(scaled_data) - window_size):
        X.append(scaled_data[i:i + window_size, 0])
        y.append(scaled_data[i + window_size, 0])
    return np.array(X), np.array(y), scaler

window_size = 10
X, y, scaler = preprocess_data(data['Close'], window_size)

# Split the data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, r

# Reshape data for CNN
X_train = np.reshape(X_train, (X_train.shape[0], X_train.shape[1], 1))
X_test = np.reshape(X_test, (X_test.shape[0], X_test.shape[1], 1))

# Build the CNN model
model = Sequential()
model.add(Conv1D(filters=64, kernel_size=3, activation='relu', input_shape
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(50, activation='relu'))
model.add(Dense(1))

# Compile the model
model.compile(optimizer='adam', loss='mean_squared_error')

# Train the model
model.fit(X_train, y_train, epochs=50, batch_size=32, validation_data=(X_t

# Evaluate the model
loss = model.evaluate(X_test, y_test)
print("Test Loss:", loss)

# Predictions
predictions = model.predict(X_test)
predictions = scaler.inverse_transform(predictions)

# Actual vs Predicted
```

```
actual_prices = scaler.inverse_transform(y_test.reshape(-1, 1))
comparison = pd.DataFrame({'Actual': actual_prices.flatten(), 'Predicted':
print(comparison.head())
```

```
[*********************100%%**********************]  1 of 1 completed

Epoch 1/50


C:\Users\divya\anaconda3\Lib\site-packages\keras\src\layers\convoluti
onal\base_conv.py:99: UserWarning: Do not pass an `input_shape`/`inpu
t_dim` argument to a layer. When using Sequential models, prefer usin
g an `Input(shape)` object as the first layer in the model instead.
  super().__init__(

90/90 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - loss: 0.0332 - val_loss: 1.9
893e-04
Epoch 2/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 2.1641e-04 - val_loss:
2.7323e-04
Epoch 3/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 2.1391e-04 - val_loss:
2.2588e-04
Epoch 4/50
```

In [56]:

```
from sklearn.metrics import mean_squared_error

# Calculate RMSE
rmse = np.sqrt(mean_squared_error(actual_prices, predictions))
print("Root Mean Squared Error (RMSE):", rmse)
```

```
Root Mean Squared Error (RMSE): 2.2680914691978646
```

In [57]:

```
#cnn+lstm
```

This code snippet constructs a combined model architecture using both CNN (Convolutional Neural Network) and LSTM (Long Short-Term Memory) branches for time series forecasting. It defines separate input layers for each branch, applies convolutional and pooling layers for the CNN branch, and employs an LSTM layer for the LSTM branch. The outputs from both branches are concatenated and passed through dense layers to produce the final output. The model is then compiled and trained using training data, and evaluated on the test data to assess its performance using the mean squared error loss metric. Finally, predictions are made using the trained model, and the Root Mean Squared Error (RMSE) is calculated to quantify the forecasting accuracy.

In [58]:

```
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Conv1D, MaxPooling1D, LSTM, Der

# Define input shape
input_shape = (window_size, 1)

# CNN branch
cnn_input = Input(shape=input_shape)
cnn_layer = Conv1D(filters=64, kernel_size=3, activation='relu')(cnn_input
cnn_layer = MaxPooling1D(pool_size=2)(cnn_layer)
cnn_layer = Flatten()(cnn_layer)

# LSTM branch
lstm_input = Input(shape=input_shape)
lstm_layer = LSTM(50)(lstm_input)

# Concatenate CNN and LSTM branches
combined_layer = concatenate([cnn_layer, lstm_layer])

# Dense layers for combined branches
dense_layer = Dense(50, activation='relu')(combined_layer)
output_layer = Dense(1)(dense_layer)

# Combine both branches into a single model
combined_model = Model(inputs=[cnn_input, lstm_input], outputs=output_laye

# Compile the model
combined_model.compile(optimizer='adam', loss='mean_squared_error')

# Train the model
combined_model.fit([X_train, X_train], y_train, epochs=50, batch_size=32,

# Evaluate the model
loss = combined_model.evaluate([X_test, X_test], y_test)
print("Test Loss:", loss)

# Predictions
predictions = combined_model.predict([X_test, X_test])
predictions = scaler.inverse_transform(predictions)

# Calculate RMSE
rmse = np.sqrt(mean_squared_error(actual_prices, predictions))
print("Root Mean Squared Error (RMSE):", rmse)
```

```
Epoch 1/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 3s 9ms/step - loss: 0.0271 - val_loss: 2.4996
e-04
Epoch 2/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - loss: 2.0953e-04 - val_loss: 1.
7839e-04
Epoch 3/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 1s 7ms/step - loss: 2.2408e-04 - val_loss: 1.
6483e-04
Epoch 4/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 1s 9ms/step - loss: 1.7730e-04 - val_loss: 1.
7708e-04
Epoch 5/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 1s 8ms/step - loss: 2.0130e-04 - val_loss: 2.
0094e-04
Epoch 6/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 1s 7ms/step - loss: 1.7820e-04 - val_loss: 1.
3848e-04
Epoch 7/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - loss: 1.8706e-04 - val_loss: 1.
3848e-04
Epoch 8/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 1s 8ms/step - loss: 1.6453e-04 - val_loss: 1.
4373e-04
Epoch 9/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 1s 8ms/step - loss: 1.5086e-04 - val_loss: 1.
2545e-04
Epoch 10/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 1s 9ms/step - loss: 1.5548e-04 - val_loss: 1.
1341e-04
Epoch 11/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 1s 8ms/step - loss: 1.5432e-04 - val_loss: 1.
1666e-04
Epoch 12/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 1s 9ms/step - loss: 1.3583e-04 - val_loss: 1.
2126e-04
Epoch 13/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 1s 9ms/step - loss: 1.6081e-04 - val_loss: 1.
2300e-04
Epoch 14/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 1s 9ms/step - loss: 1.2445e-04 - val_loss: 1.
7218e-04
Epoch 15/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 1s 8ms/step - loss: 1.3173e-04 - val_loss: 2.
2693e-04
Epoch 16/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 1s 8ms/step - loss: 1.4616e-04 - val_loss: 1.
0998e-04
Epoch 17/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 1s 8ms/step - loss: 1.3955e-04 - val_loss: 2.
5244e-04
Epoch 18/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 1s 12ms/step - loss: 1.4458e-04 - val_loss:
9.8257e-05
Epoch 19/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 1s 9ms/step - loss: 1.1118e-04 - val_loss: 1.
3631e-04
```

```
Epoch 20/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 1s 9ms/step - loss: 1.6390e-04 - val_loss: 8.
2441e-05
Epoch 21/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - loss: 1.4232e-04 - val_loss:
9.2471e-05
Epoch 22/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 1s 8ms/step - loss: 1.1851e-04 - val_loss: 1.
5963e-04
Epoch 23/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 1s 7ms/step - loss: 1.1985e-04 - val_loss: 8.
2392e-05
Epoch 24/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 1s 8ms/step - loss: 1.0593e-04 - val_loss: 9.
1592e-05
Epoch 25/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 1s 8ms/step - loss: 1.3489e-04 - val_loss: 7.
4668e-05
Epoch 26/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 1s 7ms/step - loss: 1.0382e-04 - val_loss: 8.
9525e-05
Epoch 27/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 1s 8ms/step - loss: 1.2216e-04 - val_loss: 9.
2759e-05
Epoch 28/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - loss: 1.4030e-04 - val_loss:
7.4240e-05
Epoch 29/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 1s 9ms/step - loss: 1.1324e-04 - val_loss: 7.
3594e-05
Epoch 30/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 1s 8ms/step - loss: 1.1127e-04 - val_loss: 8.
8349e-05
Epoch 31/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 1s 9ms/step - loss: 9.1024e-05 - val_loss: 7.
3095e-05
Epoch 32/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - loss: 1.0183e-04 - val_loss:
7.6499e-05
Epoch 33/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 1s 9ms/step - loss: 9.9483e-05 - val_loss: 9.
3123e-05
Epoch 34/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 1s 8ms/step - loss: 1.2567e-04 - val_loss: 1.
1814e-04
Epoch 35/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 1s 8ms/step - loss: 8.9137e-05 - val_loss: 1.
5354e-04
Epoch 36/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - loss: 9.0661e-05 - val_loss:
9.1751e-05
Epoch 37/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - loss: 9.1891e-05 - val_loss:
7.6833e-05
Epoch 38/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 1s 9ms/step - loss: 9.1852e-05 - val_loss: 1.
5820e-04
```

```
Epoch 39/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 1s 8ms/step - loss: 1.0186e-04 - val_loss: 7.
8376e-05
Epoch 40/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 1s 9ms/step - loss: 1.5027e-04 - val_loss: 1.
3256e-04
Epoch 41/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 1s 9ms/step - loss: 1.2047e-04 - val_loss: 7.
1190e-05
Epoch 42/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 1s 9ms/step - loss: 9.4177e-05 - val_loss: 7.
8739e-05
Epoch 43/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 1s 9ms/step - loss: 8.7169e-05 - val_loss: 8.
7745e-05
Epoch 44/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 1s 9ms/step - loss: 8.6192e-05 - val_loss: 3.
0137e-04
Epoch 45/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 1s 9ms/step - loss: 1.4717e-04 - val_loss: 2.
1956e-04
Epoch 46/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - loss: 1.0352e-04 - val_loss:
9.1857e-05
Epoch 47/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 1s 8ms/step - loss: 1.0392e-04 - val_loss: 1.
3989e-04
Epoch 48/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 1s 9ms/step - loss: 1.1497e-04 - val_loss: 8.
4045e-05
Epoch 49/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 1s 8ms/step - loss: 1.1049e-04 - val_loss: 9.
4334e-05
Epoch 50/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 1s 8ms/step - loss: 1.0194e-04 - val_loss: 7.
6028e-05
23/23 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 7.3358e-05
Test Loss: 7.602754340041429e-05
23/23 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step
Root Mean Squared Error (RMSE): 1.672053432000006
```

In [5]:

```
#ARIMA+CNN
```

Combines ARIMA and CNN (Convolutional Neural Network) models for time series forecasting. It begins by training an ARIMA model on the closing prices of a dataset and then preprocesses the ARIMA residuals into windows suitable for CNN input. The CNN model architecture consists of a convolutional layer, max pooling layer, flattening layer, and two dense layers. After compiling and training the CNN model, predictions are made using both ARIMA and CNN models, which are then averaged to generate the final forecast. Finally, the Root Mean Squared Error (RMSE) is calculated to evaluate the performance of the combined approach.

In [60]:

```
from statsmodels.tsa.arima.model import ARIMA
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv1D, MaxPooling1D, Flatten, Dense
import numpy as np

# Train ARIMA model
arima_model = ARIMA(data['Close'], order=(5,1,0))
arima_result = arima_model.fit()

# Extract ARIMA residuals
arima_residuals = arima_result.resid

# Preprocess ARIMA residuals for CNN input
def preprocess_residuals(residuals, window_size):
    X = []
    for i in range(len(residuals) - window_size):
        X.append(residuals[i:i + window_size])
    return np.array(X)

window_size = 10
X_arima = preprocess_residuals(arima_residuals, window_size)

#  X_train, X_test, y_train, y_test

# Define CNN model
cnn_model = Sequential()
cnn_model.add(Conv1D(filters=64, kernel_size=3, activation='relu', input_s
cnn_model.add(MaxPooling1D(pool_size=2))
cnn_model.add(Flatten())
cnn_model.add(Dense(50, activation='relu'))
cnn_model.add(Dense(1))

# Compile CNN model
cnn_model.compile(optimizer='adam', loss='mean_squared_error')

# Train CNN model
cnn_model.fit(X_train, y_train, epochs=50, batch_size=32, validation_data=

# Make predictions using ARIMA residuals and CNN model
arima_forecast = arima_result.forecast(steps=len(X_test))
arima_predictions = arima_forecast  # Just the forecast without indexing
cnn_predictions = cnn_model.predict(X_test)

# Combine predictions using averaging
final_predictions = (arima_predictions + cnn_predictions.flatten()) / 2

# Calculate RMSE
rmse = np.sqrt(np.mean((y_test - final_predictions) ** 2))
print("Root Mean Squared Error (RMSE):", rmse)
```

```
C:\Users\divya\anaconda3\Lib\site-packages\statsmodels\tsa\base\tsa_m
odel.py:473: ValueWarning: A date index has been provided, but it has
no associated frequency information and so will be ignored when e.g.
forecasting.
  self._init_dates(dates, freq)
C:\Users\divya\anaconda3\Lib\site-packages\statsmodels\tsa\base\tsa_m
odel.py:473: ValueWarning: A date index has been provided, but it has
no associated frequency information and so will be ignored when e.g.
forecasting.
  self._init_dates(dates, freq)
C:\Users\divya\anaconda3\Lib\site-packages\statsmodels\tsa\base\tsa_m
odel.py:473: ValueWarning: A date index has been provided, but it has
no associated frequency information and so will be ignored when e.g.
forecasting.
  self._init_dates(dates, freq)

Epoch 1/50

C:\Users\divya\anaconda3\Lib\site-packages\keras\src\layers\convoluti
onal\base_conv.py:99: UserWarning: Do not pass an `input_shape`/`inpu
```

In [61]:

```
#ARIMA+SVM
```

Code combines two predictive models, ARIMA and SVM (Support Vector Machine), for time series forecasting. First trains an ARIMA model on the closing prices of a dataset, extracts the residuals, and preprocesses them into windows. Then, trains an SVM model on these preprocessed residuals. Finally, makes predictions using both models and averages the predictions to generate the final forecast, evaluating the performance with Root Mean Squared Error (RMSE).

In [62]:

```
from statsmodels.tsa.arima.model import ARIMA
from sklearn.svm import SVR
import numpy as np

# Train ARIMA model
arima_model = ARIMA(data['Close'], order=(5, 1, 0))  # Example ARIMA order
arima_result = arima_model.fit()

# Extract ARIMA residuals
arima_residuals = arima_result.resid

# Preprocess ARIMA residuals for SVM input
def preprocess_residuals(residuals, window_size):
    X = []
    for i in range(len(residuals) - window_size):
        X.append(residuals[i:i + window_size])
    return np.array(X)

window_size = 10
X_arima = preprocess_residuals(arima_residuals, window_size)

# Split the data into train and test sets
split_index = int(len(X_arima) * 0.8)
X_train, X_test = X_arima[:split_index], X_arima[split_index:]
y_train, y_test = data['Close'].values[window_size:split_index+window_size

# Train SVM model
svm_model = SVR(kernel='linear')
svm_model.fit(X_train, y_train)

# Make predictions using ARIMA residuals and SVM model
arima_predictions = arima_result.forecast(steps=len(X_test))  # Only forec
arima_predictions = arima_predictions[-len(X_test):]
svm_predictions = svm_model.predict(X_test)

# Combine predictions using averaging
final_predictions = (arima_predictions + svm_predictions) / 2
print(final_predictions)

# Calculate RMSE
rmse = np.sqrt(np.mean((y_test - final_predictions) ** 2))
print("Root Mean Squared Error (RMSE):", rmse)
```

```
C:\Users\divya\anaconda3\Lib\site-packages\statsmodels\tsa\base\tsa_mode
l.py:473: ValueWarning: A date index has been provided, but it has no as
sociated frequency information and so will be ignored when e.g. forecast
ing.
  self._init_dates(dates, freq)
C:\Users\divya\anaconda3\Lib\site-packages\statsmodels\tsa\base\tsa_mode
l.py:473: ValueWarning: A date index has been provided, but it has no as
sociated frequency information and so will be ignored when e.g. forecast
ing.
  self._init_dates(dates, freq)
C:\Users\divya\anaconda3\Lib\site-packages\statsmodels\tsa\base\tsa_mode
l.py:473: ValueWarning: A date index has been provided, but it has no as
sociated frequency information and so will be ignored when e.g. forecast
ing.
  self._init_dates(dates, freq)

3583    101.492857
3584    105.132245
3585    100.145048
3586     95.799290
3587     95.882811
           ...
4293    104.764777
4294    103.420353
4295     97.654873
4296     92.349498
4297    100.111830
Name: predicted_mean, Length: 715, dtype: float64
Root Mean Squared Error (RMSE): 65.3163697419207

C:\Users\divya\anaconda3\Lib\site-packages\statsmodels\tsa\base\tsa_mode
l.py:836: ValueWarning: No supported index is available. Prediction resu
lts will be given with an integer index beginning at `start`.
  return get_prediction_index(
C:\Users\divya\anaconda3\Lib\site-packages\statsmodels\tsa\base\tsa_mode
l.py:836: FutureWarning: No supported index is available. In the next ve
rsion, calling this method in a model without a supported index will res
ult in an exception.
  return get_prediction_index(
```

In [ ]: