In [1]:

```
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
from statsmodels.tsa.arima_model import ARIMA
import pmdarima as pm
import datetime
from pmdarima.arima import ADFTest
```

Usage of Time Series model: ARIMA to predict stock prices, Usage of Auto ARIMA: It is known to be more accurate and powerful compared to the manual ARIMA, as it will select more appropriate parameters. More things to explore: Selecting the correct time frame, how to improve the accuracy of ARIMA

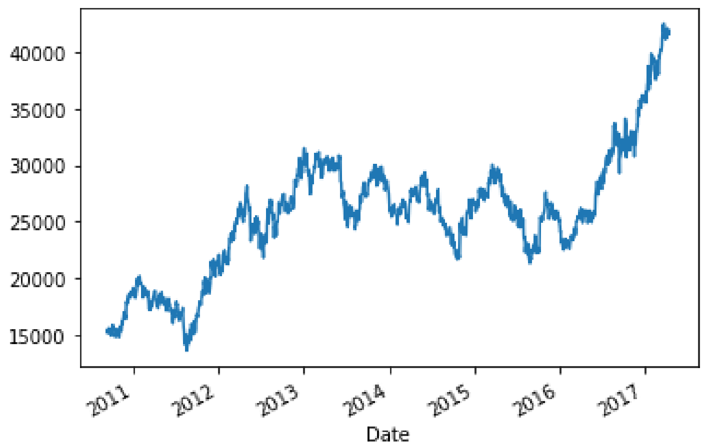
In [145]:

```
def write_ans_toCSV(filename,answers):
    submissions = pd.read_csv("sample_submission.csv")
    print(submissions)
    submissions['Predicted']= answers
    submissions.to_csv(filename,index=False)
```

In [140]:

```
df = pd.read_csv('data.csv')
ans = pd.read_csv('sample_submission.csv')
ans['Date']=pd.to_datetime(ans['Date'])
df['Date']=pd.to_datetime(df["Date"])
df.set_index("Date",inplace=True)
stocks=df["Close"]


print(stocks)
print(stocks.plot())
```

```
Date
2010-09-13    15400
2010-09-14    15200
2010-09-15    15140
2010-09-16    15140
2010-09-17    15460
              ...
2017-04-06    41840
2017-04-07    41600
2017-04-10    41940
2017-04-11    41600
2017-04-12    41900
Name: Close, Length: 1628, dtype: int64
AxesSubplot(0.125,0.2;0.775x0.68)
```



We will now test is the data is stationary using the ADF Test. This will indicate whether we have to use

differencing to make the data stationary (although this will already be automatically computed by the Auto ARIMA model).

In [3]:

```
adf_test = ADFTest(alpha=0.05)
adf_test.should_diff(stocks)
```

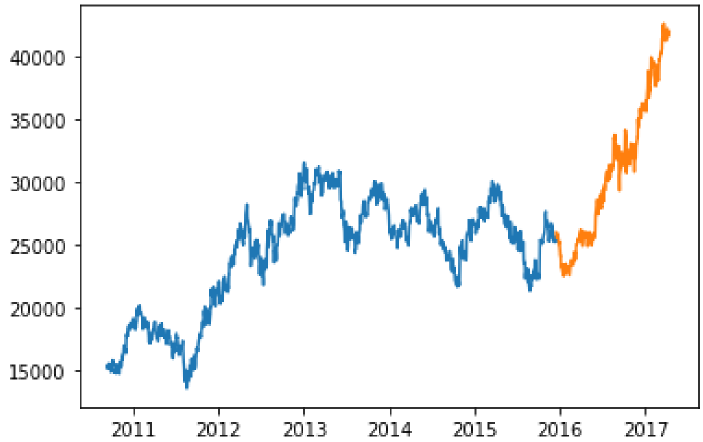
Out[3]:

```
(0.8958134521967053, True)
```

ADF Test shows that the data is already stationary, hence integration/ differencing is not necessary. Now it is time to split the data into a training and test set. Split according to the 80-20 split rule (this follows from the Pareto principle, where an 80/20 statistical split is a commonly occuring ratio). This 80-20 ratio ensures that there is sufficient training and testing data to minimise the variance of the trained model and the performance statistic respectively.

In [4]:

```
length = 1628
upper_limit=int(0.8*1628)
ARIMA_train_data = stocks[:upper_limit]
ARIMA_test_data=stocks[upper_limit:]
#print(len(ARIMA_train_data))
#print(len(ARIMA_test_data))
plt.plot(ARIMA_train_data)
plt.plot(ARIMA_test_data)
plt.show()
```



Parameters for auto_arima. Set seasonal to False because stock prices are not known to be seasonal, and test each parameter(p and q) from 0 to 3. d=None essentially means that we do not have to use any differentials/ integration to make the data stationary. Auto_ARIMA will then use AIC (a means of measuring the error), to figure out the best parameters to use for the stock prediction model.

In [5]:

```
model = pm.auto_arima(ARIMA_train_data,start_p=0,start_q=0,test='adf', d=None,seasonal=F
alse,
                      suppress_warnings=True, error_action="ignore", max_p=3,max_q=3,
                      trace=True,m=1,D=0,start_P=0,stepwise=True)
print(model.summary())
```

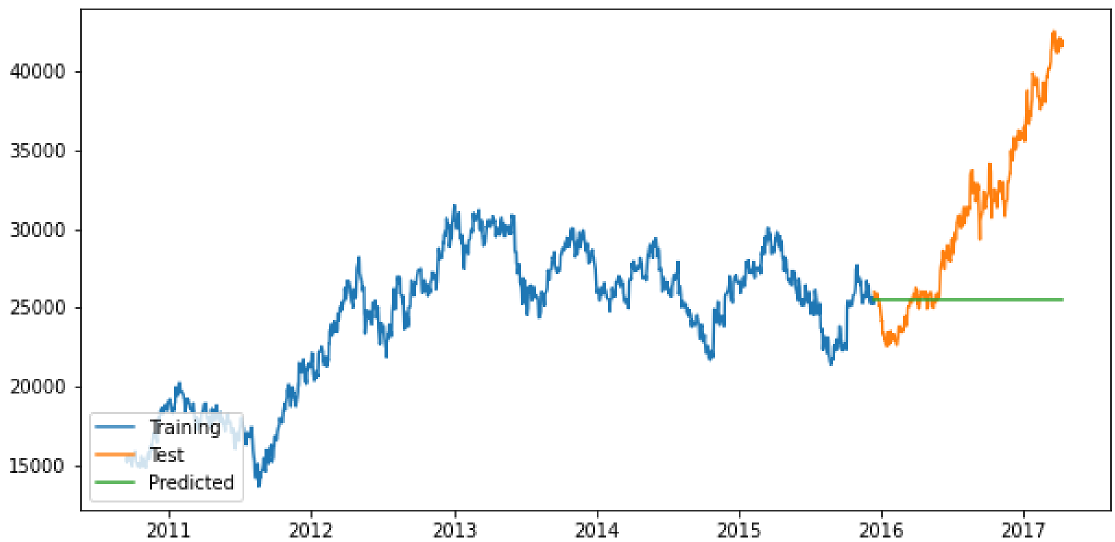
```
Performing stepwise search to minimize aic
 ARIMA(0,1,0)(0,0,0)[0] intercept   : AIC=19394.563, Time=0.18 sec
 ARIMA(1,1,0)(0,0,0)[0] intercept   : AIC=19395.227, Time=0.28 sec
 ARIMA(0,1,1)(0,0,0)[0] intercept   : AIC=19394.992, Time=0.29 sec
 ARIMA(0,1,0)(0,0,0)[0]             : AIC=19393.017, Time=0.07 sec
 ARIMA(1,1,1)(0,0,0)[0] intercept   : AIC=19394.104, Time=1.10 sec

Best model:  ARIMA(0,1,0)(0,0,0)[0]
Total fit time: 1.956 seconds
                               SARIMAX Results
==============================================================================
```

```
Dep. Variable:                      y   No. Observations:                 1302
Model:               SARIMAX(0, 1, 0)   Log Likelihood               -9695.509
Date:                Sat, 02 Oct 2021   AIC                          19393.017
Time:                        10:38:59   BIC                          19398.188
Sample:                             0   HQIC                         19394.957
                               - 1302
Covariance Type:                  opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
sigma2      1.739e+05   5200.201     33.435      0.000    1.64e+05    1.84e+05
===================================================================================
Ljung-Box (L1) (Q):                   1.55   Jarque-Bera (JB):               112.21
Prob(Q):                              0.21   Prob(JB):                         0.00
Heteroskedasticity (H):               1.19   Skew:                             0.10
Prob(H) (two-sided):                  0.07   Kurtosis:                         4.43
===================================================================================

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).
```

Utilising the model to predict data, and thereafter comparing the predicted data with the test data

In [6]:

```
ARIMA_predicted_stock_prices=pd.DataFrame(model.predict(n_periods=326),index=ARIMA_test_d
ata.index)
ARIMA_predicted_stock_prices.columns=['predicted_price']
plt.figure(figsize=(10,5))
plt.plot(ARIMA_train_data,label="Training")
plt.plot(ARIMA_test_data,label="Test")
plt.plot(ARIMA_predicted_stock_prices,label="Predicted")
plt.legend(loc='lower left')
plt.show()
```



Validating the data through RMSE. We will use RMSE to compare how accurate the models are.

In [7]:

```
from sklearn.metrics import mean_squared_error

#print(ARIMA_test_data)
#print(ARIMA_predicted_stock_prices)

ARIMA_test_data_lst = ARIMA_test_data.tolist()
ARIMA_predicted_stock_prices_lst=ARIMA_predicted_stock_prices["predicted_price"].tolist()

#print(ARIMA_test_data_lst)
#print(ARIMA_predicted_stock_prices_lst)

print("The mean squared error is", mean_squared_error(ARIMA_test_data_lst,ARIMA_predicted
```

```
_stock_prices_lst,squared=False))
```

```
The mean squared error is 7892.814857854459
```

Using the ARIMA model to make predictions, but using the entire dataset. In my opinion, this would be more accurate than using just the training dataset. This is because, the entire dataset also contains more recent values.

In [127]:

```
model = pm.auto_arima(stocks,start_p=0,start_q=0,test='adf', d=None,seasonal=False,
                      suppress_warnings=True, error_action="ignore", max_p=3,max_q=3,
                      trace=True,m=1,D=0,start_P=0,stepwise=True)
print(model.summary())
```

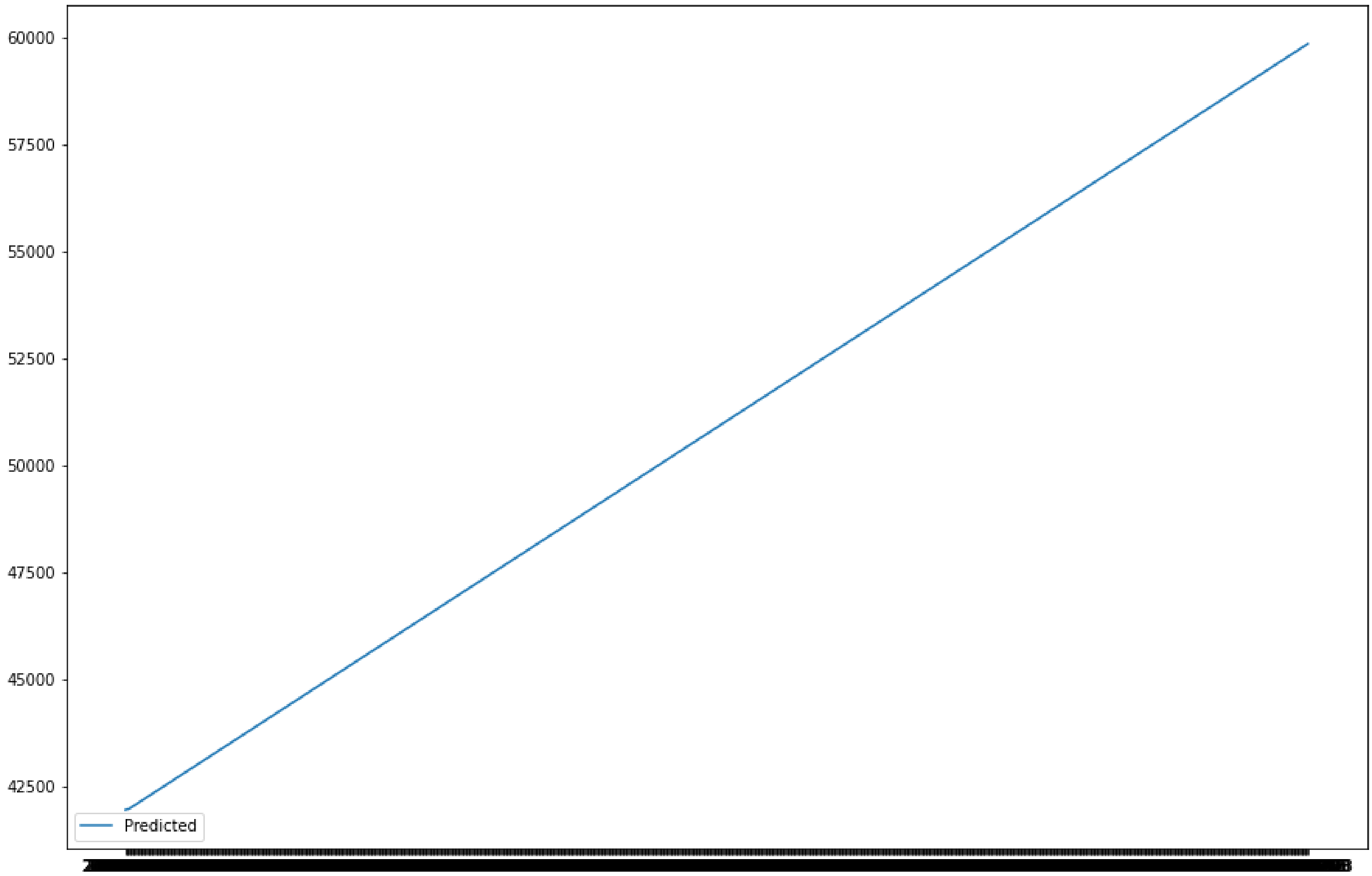
```
Performing stepwise search to minimize aic
 ARIMA(0,1,0)(0,0,0)[0] intercept   : AIC=24390.269, Time=0.05 sec
 ARIMA(1,1,0)(0,0,0)[0] intercept   : AIC=24390.072, Time=0.10 sec
 ARIMA(0,1,1)(0,0,0)[0] intercept   : AIC=24389.640, Time=0.11 sec
 ARIMA(0,1,0)(0,0,0)[0]             : AIC=24390.549, Time=0.02 sec
 ARIMA(1,1,1)(0,0,0)[0] intercept   : AIC=24386.516, Time=0.79 sec
 ARIMA(2,1,1)(0,0,0)[0] intercept   : AIC=24368.388, Time=0.90 sec
 ARIMA(2,1,0)(0,0,0)[0] intercept   : AIC=24379.311, Time=0.13 sec
 ARIMA(3,1,1)(0,0,0)[0] intercept   : AIC=24370.041, Time=0.57 sec
 ARIMA(2,1,2)(0,0,0)[0] intercept   : AIC=24368.911, Time=1.13 sec
 ARIMA(1,1,2)(0,0,0)[0] intercept   : AIC=24370.108, Time=0.75 sec
 ARIMA(3,1,0)(0,0,0)[0] intercept   : AIC=24369.957, Time=0.21 sec
 ARIMA(3,1,2)(0,0,0)[0] intercept   : AIC=24371.688, Time=1.00 sec
 ARIMA(2,1,1)(0,0,0)[0]             : AIC=24369.958, Time=0.31 sec

Best model:  ARIMA(2,1,1)(0,0,0)[0] intercept
Total fit time: 6.099 seconds
                               SARIMAX Results
==============================================================================
Dep. Variable:                      y   No. Observations:                 1628
Model:               SARIMAX(2, 1, 1)   Log Likelihood              -12179.194
Date:                Sat, 02 Oct 2021   AIC                          24368.388
Time:                        15:35:28   BIC                          24395.360
Sample:                             0   HQIC                         24378.395
                               - 1628
Covariance Type:                  opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
intercept      7.9227      4.807      1.648      0.099      -1.499      17.344
ar.L1          0.6413      0.127      5.049      0.000       0.392       0.890
ar.L2         -0.1204      0.024     -5.045      0.000      -0.167      -0.074
ma.L1         -0.6107      0.126     -4.849      0.000      -0.858      -0.364
sigma2      1.864e+05   4716.203     39.513      0.000    1.77e+05    1.96e+05
===================================================================================
Ljung-Box (L1) (Q):                   0.00   Jarque-Bera (JB):               300.20
Prob(Q):                              1.00   Prob(JB):                         0.00
Heteroskedasticity (H):               1.33   Skew:                            -0.12
Prob(H) (two-sided):                  0.00   Kurtosis:                         5.09
===================================================================================

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).
```

In [147]:

```
ans = pd.read_csv('sample_submission.csv')
ans.set_index("Date",inplace=True)
new_ARIMA_predicted_stock_prices=pd.DataFrame(model.predict(n_periods=1086),index=ans.ind
ex)
new_ARIMA_predicted_stock_prices.columns=['predicted_price']
#print(ARIMA_predicted_stock_prices)
print(stocks)
print(ARIMA_predicted_stock_prices)
plt.figure(figsize=(15,10))
```

```
#plt.plot(stocks,label="Historical")
plt.plot(ARIMA_predicted_stock_prices,label="Predicted")
plt.legend(loc='lower left')
plt.show()
#We can see that there is a 'linear' trend when we plot the graph

submit_values=new_ARIMA_predicted_stock_prices['predicted_price'].tolist()
print(submit_values)
write_ans_toCSV("Keane Ong_PredictedValues2.csv",submit_values)
```

```
Date
2010-09-13    15400
2010-09-14    15200
2010-09-15    15140
2010-09-16    15140
2010-09-17    15460
              ...
2017-04-06    41840
2017-04-07    41600
2017-04-10    41940
2017-04-11    41600
2017-04-12    41900
Name: Close, Length: 1628, dtype: int64
            predicted_price
Date
2017-04-13     41953.247619
2017-04-14     41959.193845
2017-04-17     41964.518057
2017-04-18     41975.139191
2017-04-19     41989.232195
...                     ...
2021-09-07     59797.763531
2021-09-08     59814.299929
2021-09-09     59830.836328
2021-09-10     59847.372726
2021-09-13     59863.909124

[1086 rows x 1 columns]
```




```
[41953.24761932145, 41959.19384482033, 41964.51805675454, 41975.13919130015, 41989.23219480339, 42004.91391002688, 42021.19641456814, 42037.67290554967, 42054.201457541894, 42070.740037779775, 42087.27878027248, 42103.816419263516, 42120.35333103002, 42136.8899092990
```

6, 42153.42636126207, 42169.962772382074, 42186.49917251827, 42203.03557052858, 42219.571
96849815, 42236.1083666976, 42252.644765049365, 42269.18116347114, 42285.71756191947, 423
02.253960376394, 42318.790358835635, 42335.32675729533, 42351.86315575503, 42368.39955421
469, 42384.935952674314, 42401.472351133925, 42418.00874959353, 42434.545148053134, 42451
.08154651274, 42467.61794497234, 42484.15434343195, 42500.69074189155, 42517.227140351155
, 42533.76353881076, 42550.29993727036, 42566.83633572997, 42583.37273418957, 42599.90913
2649176, 42616.44553110878, 42632.981929568385, 42649.51832802799, 42666.05472648759, 426
82.5911249472, 42699.1275234068, 42715.663921866406, 42732.20032032601, 42748.73671878561
4, 42765.27311724522, 42781.80951570482, 42798.34591416443, 42814.88231262403, 42831.4187
11083636, 42847.95510954324, 42864.491508002844, 42881.02790646245, 42897.56430492205, 42
914.10070338166, 42930.63710184126, 42947.173500300865, 42963.70989876047, 42980.24629722
0074, 42996.78269567968, 43013.31909413928, 43029.85549259889, 43046.39189105849, 43062.9
28289518095, 43079.4646879777, 43096.0010864373, 43112.53748489691, 43129.07388335651, 43
145.610281816116, 43162.14668027572, 43178.683078735325, 43195.21947719493, 43211.7558756
5453, 43228.29227411414, 43244.82867257374, 43261.365071033346, 43277.90146949295, 43294.
437867952554, 43310.97426641216, 43327.51066487176, 43344.04706333137, 43360.58346179097,
43377.119860250576, 43393.65625871018, 43410.192657169784, 43426.72905562939, 43443.26545
408899, 43459.8018525486, 43476.3382510082, 43492.874649467805, 43509.41104792741, 43525.
947446387014, 43542.48384484662, 43559.02024330622, 43575.55664176583, 43592.09304022543,
43608.629438685035, 43625.16583714464, 43641.70223560424, 43658.23863406385, 43674.775032
52345, 43691.311430983056, 43707.84782944266, 43724.384227902265, 43740.92062636187, 4375
7.45702482147, 43773.99342328108, 43790.52982174068, 43807.066220200286, 43823.6026186598
9, 43840.139017119494, 43856.6754155791, 43873.2118140387, 43889.74821249831, 43906.28461
095791, 43922.821009417516, 43939.35740787712, 43955.893806336724, 43972.43020479633, 439
88.96660325593, 44005.50300171554, 44022.03940017514, 44038.575798634745, 44055.112197094
35, 44071.648595553954, 44088.18499401356, 44104.72139247316, 44121.25779093277, 44137.79
418939237, 44154.330587851975, 44170.86698631158, 44187.40338477118, 44203.93978323079, 4
4220.47618169039, 44237.012580149996, 44253.5489786096, 44270.085377069205, 44286.6217755
2881, 44303.15817398841, 44319.69457244802, 44336.23097090762, 44352.767369367226, 44369.
30376782683, 44385.840166286434, 44402.37656474604, 44418.91296320564, 44435.44936166525,
44451.98576012485, 44468.522158584456, 44485.05855704406, 44501.594955503664, 44518.13135
396327, 44534.66775242287, 44551.20415088248, 44567.74054934208, 44584.276947801685, 4460
0.81334626129, 44617.349744720894, 44633.8861431805, 44650.4225416401, 44666.95894009971,
44683.49533855931, 44700.031737018915, 44716.56813547852, 44733.10453393812, 44749.640932
39773, 44766.17733085733, 44782.713729316936, 44799.25012777654, 44815.786526236145, 4483
2.32292469575, 44848.85932315535, 44865.39572161496, 44881.93212007456, 44898.46851853416
6, 44915.00491699377, 44931.541315453374, 44948.07771391298, 44964.61411237258, 44981.150
51083219, 44997.68690929179, 45014.223307751396, 45030.759706211, 45047.296104670604, 450
63.83250313021, 45080.36890158981, 45096.90530004942, 45113.44169850902, 45129.9780969686
25, 45146.51449542823, 45163.050893887834, 45179.58729234744, 45196.12369080704, 45212.66
008926665, 45229.19648772625, 45245.732886185855, 45262.26928464546, 45278.80568310506, 4
5295.34208156467, 45311.87848002427, 45328.414878483876, 45344.95127694348, 45361.4876754
03085, 45378.02407386269, 45394.56047232229, 45411.0968707819, 45427.6332692415, 45444.16
9667701106, 45460.70606616071, 45477.242464620314, 45493.77886307992, 45510.31526153952,
45526.85165999913, 45543.38805845873, 45559.924456918336, 45576.46085537794, 45592.997253
837544, 45609.53365229715, 45626.07005075675, 45642.60644921636, 45659.14284767596, 45675
.679246135565, 45692.21564459517, 45708.752043054774, 45725.28844151438, 45741.8248399739
8, 45758.36123843359, 45774.89763689319, 45791.434035352795, 45807.9704338124, 45824.5068
32272, 45841.04323073161, 45857.57962919121, 45874.116027650816, 45890.65242611042, 45907
.188824570025, 45923.72522302963, 45940.26162148923, 45956.79801994884, 45973.33441840844
, 45989.870816868046, 46006.40721532765, 46022.943613787254, 46039.48001224686, 46056.016
41070646, 46072.55280916607, 46089.08920762567, 46105.625606085276, 46122.16200454488, 46
138.698403004484, 46155.23480146409, 46171.77119992369, 46188.3075983833, 46204.843996842
9, 46221.380395302505, 46237.91679376211, 46254.453192221714, 46270.98959068132, 46287.52
598914092, 46304.06238760053, 46320.59878606013, 46337.135184519735, 46353.67158297934, 4
6370.20798143894, 46386.74437989855, 46403.28077835815, 46419.817176817756, 46436.3535752
7736, 46452.889973736965, 46469.42637219657, 46485.96277065617, 46502.49916911578, 46519.
03556757538, 46535.571966034986, 46552.10836449459, 46568.644762954194, 46585.1811614138,
46601.7175598734, 46618.25395833301, 46634.79035679261, 46651.326755252216, 46667.8631537
1182, 46684.399552171424, 46700.93595063103, 46717.47234909063, 46734.00874755024, 46750.
54514600984, 46767.081544469445, 46783.61794292905, 46800.154341388654, 46816.69073984826
, 46833.22713830786, 46849.76353676747, 46866.29993522707, 46882.836333686675, 46899.3727
3214628, 46915.90913060588, 46932.44552906549, 46948.98192752509, 46965.518325984696, 469
82.0547244443, 46998.591122903905, 47015.12752136351, 47031.66391982311, 47048.2003182827
2, 47064.73671674232, 47081.273115201926, 47097.80951366153, 47114.345912121134, 47130.88
231058074, 47147.41870904034, 47163.95510749995, 47180.49150595955, 47197.027904419156, 4
7213.56430287876, 47230.100701338364, 47246.63709979797, 47263.17349825757, 47279.7098967
1718, 47296.24629517678, 47312.782693636385, 47329.31909209599, 47345.855490555594, 47362
.3918890152, 47378.9282874748, 47395.46468593441, 47412.00108439401, 47428.537482853615,
47445.07388131322, 47461.61027977282, 47478.14667823243, 47494.68307669203, 47511.2194751
51636, 47527.75587361124, 47544.292272070845, 47560.82867053045, 47577.36506899005, 47593
.90146744966, 47610.43786590926, 47626.974264368866, 47643.51066282847, 47660.04706128807

4, 47676.58345974768, 47693.11985820728, 47709.65625666689, 47726.19265512649, 47742.7290
53586096, 47759.2654520457, 47775.801850505304, 47792.33824896491, 47808.87464742451, 478
25.41104588412, 47841.94744434372, 47858.483842803325, 47875.02024126293, 47891.556639722
534, 47908.09303818214, 47924.62943664174, 47941.16583510135, 47957.70223356095, 47974.23
8632020555, 47990.77503048016, 48007.31142893976, 48023.84782739937, 48040.38422585897, 4
8056.920624318576, 48073.45702277818, 48089.993421237785, 48106.52981969739, 48123.066218
15699, 48139.6026166166, 48156.1390150762, 48172.675413535806, 48189.21181199541, 48205.7
48210455014, 48222.28460891462, 48238.82100737422, 48255.35740583383, 48271.89380429343,
48288.430202753036, 48304.96660121264, 48321.502999672244, 48338.03939813185, 48354.57579
659145, 48371.11219505106, 48387.64859351066, 48404.184991970265, 48420.72139042987, 4843
7.257788889474, 48453.79418734908, 48470.33058580868, 48486.86698426829, 48503.4033827278
9, 48519.939781187495, 48536.4761796471, 48553.0125781067, 48569.54897656631, 48586.08537
502591, 48602.621773485516, 48619.15817194512, 48635.694570404725, 48652.23096886433, 486
68.76736732393, 48685.30376578354, 48701.84016424314, 48718.376562702746, 48734.912961162
35, 48751.449359621954, 48767.98575808156, 48784.52215654116, 48801.05855500077, 48817.59
495346037, 48834.131351919976, 48850.66775037958, 48867.204148839184, 48883.74054729879,
48900.27694575839, 48916.813344218, 48933.3497426776, 48949.886141137205, 48966.422539596
81, 48982.958938056414, 48999.49533651602, 49016.03173497562, 49032.56813343523, 49049.10
453189483, 49065.640930354435, 49082.17732881404, 49098.71372727364, 49115.25012573325, 4
9131.78652419285, 49148.322922652456, 49164.85932111206, 49181.395719571665, 49197.932118
03127, 49214.46851649087, 49231.00491495048, 49247.54131341008, 49264.077711869686, 49280
.61411032929, 49297.150508788895, 49313.6869072485, 49330.2233057081, 49346.75970416771,
49363.29610262731, 49379.832501086916, 49396.36889954652, 49412.905298006124, 49429.44169
646573, 49445.97809492533, 49462.51449338494, 49479.05089184454, 49495.587290304145, 4951
2.12368876375, 49528.660087223354, 49545.19648568296, 49561.73288414256, 49578.2692826021
7, 49594.80568106177, 49611.342079521375, 49627.87847798098, 49644.41487644058, 49660.951
27490019, 49677.48767335979, 49694.024071819396, 49710.560470279, 49727.096868738605, 497
43.63326719821, 49760.16966565781, 49776.70606411742, 49793.24246257702, 49809.7788610366
26, 49826.31525949623, 49842.851657955835, 49859.38805641544, 49875.92445487504, 49892.46
085333465, 49908.99725179425, 49925.533650253856, 49942.07004871346, 49958.606447173064,
49975.14284563267, 49991.67924409227, 50008.21564255188, 50024.75204101148, 50041.2884394
71085, 50057.82483793069, 50074.361236390294, 50090.8976348499, 50107.4340333095, 50123.9
7043176911, 50140.50683022871, 50157.043228688315, 50173.57962714792, 50190.11602560752,
50206.65242406713, 50223.18882252673, 50239.725220986336, 50256.26161944594, 50272.798017
905545, 50289.33441636515, 50305.87081482475, 50322.40721328436, 50338.94361174396, 50355
.480010203566, 50372.01640866317, 50388.552807122775, 50405.08920558238, 50421.6256040419
8, 50438.16200250159, 50454.69840096119, 50471.234799420796, 50487.7711978804, 50504.3075
96340004, 50520.84399479961, 50537.38039325921, 50553.91679171882, 50570.45319017842, 505
86.989588638025, 50603.52598709763, 50620.062385557234, 50636.59878401684, 50653.13518247
644, 50669.67158093605, 50686.20797939565, 50702.744377855255, 50719.28077631486, 50735.8
1717477446, 50752.35357323407, 50768.88997169367, 50785.42637015328, 50801.96276861288, 5
0818.499167072485, 50835.03556553209, 50851.57196399169, 50868.1083624513, 50884.64476091
09, 50901.181159370506, 50917.71755783011, 50934.253956289715, 50950.79035474932, 50967.3
2675320892, 50983.86315166853, 51000.39955012813, 51016.935948587736, 51033.47234704734,
51050.008745506944, 51066.54514396655, 51083.08154242615, 51099.61794088576, 51116.154339
34536, 51132.690737804965, 51149.22713626457, 51165.763534724174, 51182.29993318378, 5119
8.83633164338, 51215.37273010299, 51231.90912856259, 51248.445527022195, 51264.9819254818
, 51281.518323941404, 51298.05472240101, 51314.59112086061, 51331.12751932022, 51347.6639
1777982, 51364.200316239425, 51380.73671469903, 51397.27311315863, 51413.80951161824, 514
30.34591007784, 51446.882308537446, 51463.41870699705, 51479.955105456655, 51496.49150391
626, 51513.02790237586, 51529.56430083547, 51546.10069929507, 51562.637097754676, 51579.1
7349621428, 51595.709894673884, 51612.24629313349, 51628.78269159309, 51645.3190900527, 5
1661.8554885123, 51678.391886971905, 51694.92828543151, 51711.464683891114, 51728.0010823
5072, 51744.53748081032, 51761.07387926993, 51777.61027772953, 51794.14667619135, 51810.
68307464874, 51827.219473108344, 51843.75587156795, 51860.29227002755, 51876.82866848716,
51893.36506694676, 51909.901465406365, 51926.43786386597, 51942.97426232557, 51959.510660
78518, 51976.04705924478, 51992.583457704386, 52009.11985616399, 52025.656254623595, 5204
2.1926530832, 52058.7290515428, 52075.26545000241, 52091.80184846201, 52108.338246921616,
52124.87464538122, 52141.411043840824, 52157.94744230043, 52174.48384076003, 52191.020239
21964, 52207.55663767924, 52224.093036138845, 52240.62943459845, 52257.165833058054, 5227
3.70223151766, 52290.23862997726, 52306.77502843687, 52323.31142689647, 52339.84782535607
5, 52356.38422381568, 52372.920622275284, 52389.45702073489, 52405.99341919449, 52422.529
8176541, 52439.0662161137, 52455.602614573305, 52472.13901303291, 52488.67541149251, 5250
5.21180995212, 52521.74820841172, 52538.284606871326, 52554.82100533093, 52571.3574037905
35, 52587.89380225014, 52604.43020070974, 52620.96659916935, 52637.50299762895, 52654.039
396088556, 52670.57579454816, 52687.112193007764, 52703.64859146737, 52720.18498992697, 5
2736.72138838658, 52753.25778684618, 52769.794185305786, 52786.33058376539, 52802.8669822
24994, 52819.4033806846, 52835.9397791442, 52852.47617760381, 52869.01257606341, 52885.54
8974523015, 52902.08537298262, 52918.621771442224, 52935.15816990183, 52951.69456836143,
52968.23096682104, 52984.76736528064, 53001.303763740245, 53017.84016219985, 53034.376560
65945, 53050.91295911906, 53067.44935757866, 53083.985756038266, 53100.52215449787, 53117
.058552957475, 53133.59495141708, 53150.13134987668, 53166.66774833629, 53183.20414679589

, 53199.740545255496, 53216.2769437151, 53232.813342174704, 53249.34974063431, 53265.8861
3909391, 53282.42253755352, 53298.95893601312, 53315.495334472726, 53332.03173293233, 533
48.568131391934, 53365.10452985154, 53381.64092831114, 53398.17732677075, 53414.713725230
35, 53431.250123689955, 53447.78652214956, 53464.322920609164, 53480.85931906877, 53497.3
9571752837, 53513.93211598798, 53530.46851444758, 53547.004912907185, 53563.54131136679,
53580.07770982639, 53596.614108286, 53613.1505067456, 53629.686905205206, 53646.223303664
81, 53662.759702124415, 53679.29610058402, 53695.83249904362, 53712.36889750323, 53728.90
529596283, 53745.441694422436, 53761.97809288204, 53778.514491341644, 53795.05088980125,
53811.58728826085, 53828.12368672046, 53844.66008518006, 53861.196483639666, 53877.732882
09927, 53894.269280558874, 53910.80567901848, 53927.34207747808, 53943.87847593769, 53960
.41487439729, 53976.951272856895, 53993.4876713165, 54010.024069776104, 54026.56046823571
, 54043.09686669531, 54059.63326515492, 54076.16966361452, 54092.706062074125, 54109.2424
6053373, 54125.77885899333, 54142.31525745294, 54158.85165591254, 54175.388054372146, 541
91.92445283175, 54208.460851291355, 54224.99724975096, 54241.53364821056, 54258.070046670
17, 54274.60644512977, 54291.142843589376, 54307.67924204898, 54324.215640508584, 54340.7
5203896819, 54357.28843742779, 54373.8248358874, 54390.361234347, 54406.897632806606, 544
23.43403126621, 54439.970429725814, 54456.50682818542, 54473.04322664502, 54489.579625104
63, 54506.11602356423, 54522.652422023835, 54539.18882048344, 54555.725218943044, 54572.2
6161740265, 54588.79801586225, 54605.33441432186, 54621.87081278146, 54638.407211241065,
54654.94360970067, 54671.48000816027, 54688.01640661988, 54704.55280507948, 54721.0892035
39086, 54737.62560199869, 54754.162000458295, 54770.6983989179, 54787.2347973775, 54803.7
7119583711, 54820.30759429671, 54836.843992756316, 54853.38039121592, 54869.916789675524,
54886.45318813513, 54902.98958659473, 54919.52598505434, 54936.06238351394, 54952.5987819
73546, 54969.13518043315, 54985.671578892754, 55002.20797735236, 55018.74437581196, 55035
.28077427157, 55051.81717273117, 55068.353571190775, 55084.88996965038, 55101.42636810998
4, 55117.96276656959, 55134.49916502919, 55151.0355634888, 55167.5719619484, 55184.108360
408005, 55200.64475886761, 55217.18115732721, 55233.71755578682, 55250.25395424642, 55266
.790352706026, 55283.32675116563, 55299.863149625235, 55316.39954808484, 55332.9359465444
4, 55349.47234500405, 55366.00874346365, 55382.545141923256, 55399.08154038286, 55415.617
938842464, 55432.15433730207, 55448.69073576167, 55465.22713422128, 55481.76353268088, 55
498.299931140486, 55514.83632960009, 55531.372728059694, 55547.9091265193, 55564.44552497
89, 55580.98192343851, 55597.51832189811, 55614.054720357715, 55630.59111881732, 55647.12
7517276924, 55663.66391573653, 55680.20031419613, 55696.73671265574, 55713.27311111534, 5
5729.809509574945, 55746.34590803455, 55762.88230649415, 55779.41870495376, 55795.9551034
1336, 55812.491501872966, 55829.02790033257, 55845.564298792175, 55862.10069725178, 55878
.63709571138, 55895.17349417099, 55911.70989263059, 55928.246291090196, 55944.7826895498,
55961.319088009404, 55977.85548646901, 55994.39188492861, 56010.92828338822, 56027.464681
84782, 56044.001080307426, 56060.53747876703, 56077.073877226634, 56093.61027568624, 5611
0.14667414584, 56126.68307260545, 56143.21947106505, 56159.755869524655, 56176.2922679842
6, 56192.828666443864, 56209.36506490347, 56225.90146336307, 56242.43786182268, 56258.974
26028228, 56275.510658741885, 56292.04705720149, 56308.58345566109, 56325.1198541207, 563
41.6562525803, 56358.192651039906, 56374.72904949951, 56391.265447959115, 56407.801846418
72, 56424.33824487832, 56440.87464333793, 56457.41104179753, 56473.947440257136, 56490.48
383871674, 56507.020237176344, 56523.55663563595, 56540.09303409555, 56556.62943255516, 5
6573.16583101476, 56589.702229474366, 56606.23862793397, 56622.775026393574, 56639.311424
85318, 56655.84782331278, 56672.38422177239, 56688.92062023199, 56705.457018691595, 56721
.9934171512, 56738.529815610804, 56755.06621407041, 56771.60261253001, 56788.13901098962,
56804.67540944922, 56821.211807908825, 56837.74820636843, 56854.28460482803, 56870.821003
28764, 56887.35740174724, 56903.893800206846, 56920.43019866645, 56936.966597126055, 5695
3.50299558566, 56970.03939404526, 56986.57579250487, 57003.11219096447, 57019.64858942407
6, 57036.18498788368, 57052.721386343284, 57069.25778480289, 57085.79418326249, 57102.330
5817221, 57118.8669801817, 57135.403378641306, 57151.93977710091, 57168.476175560514, 571
85.01257402012, 57201.54897247972, 57218.08537093933, 57234.62176939893, 57251.1581678585
35, 57267.69456631814, 57284.230964777744, 57300.76736323735, 57317.30376169695, 57333.84
016015656, 57350.37655861616, 57366.912957075765, 57383.44935553537, 57399.98575399497, 5
7416.52215245458, 57433.05855091418, 57449.594949373786, 57466.13134783339, 57482.6677462
92995, 57499.2041447526, 57515.7405432122, 57532.27694167181, 57548.81334013141, 57565.34
9738591016, 57581.88613705062, 57598.422535510224, 57614.95893396983, 57631.49533242943,
57648.03173088904, 57664.56812934864, 57681.104527808246, 57697.64092626785, 57714.177324
727454, 57730.71372318706, 57747.25012164666, 57763.78652010627, 57780.32291856587, 57796
.859317025475, 57813.39571548508, 57829.932113944684, 57846.46851240429, 57863.0049108638
9, 57879.5413093235, 57896.0777077831, 57912.614106242705, 57929.15050470231, 57945.68690
316191, 57962.22330162152, 57978.75970008112, 57995.296098540726, 58011.83249700033, 5802
8.368895459935, 58044.90529391954, 58061.44169237914, 58077.97809083875, 58094.5144892983
5, 58111.050887757956, 58127.58728621756, 58144.123684677164, 58160.66008313677, 58177.19
648159637, 58193.73288005598, 58210.26927851558, 58226.805676975186, 58243.34207543479, 5
8259.878473894394, 58276.414872354, 58292.9512708136, 58309.48766927321, 58326.0240677328
1, 58342.560466192415, 58359.09686465202, 58375.633263111624, 58392.16966157123, 58408.70
606003083, 58425.24245849044, 58441.77885695004, 58458.315255409645, 58474.85165386925, 5
8491.38805232885, 58507.92445078846, 58524.46084924806, 58540.997247707666, 58557.5336461
6727, 58574.070044626875, 58590.60644308648, 58607.14284154608, 58623.67924000569, 58640.
21563846529, 58656.752036924896, 58673.2884353845, 58689.824833844104, 58706.36123230371,

```
58722.89763076331, 58739.43402922292, 58755.97042768252, 58772.506826142126, 58789.043224
60173, 58805.579623061334, 58822.11602152094, 58838.65241998054, 58855.18881844015, 58871
.72521689975, 58888.261615359355, 58904.79801381896, 58921.334412278564, 58937.8708107381
7, 58954.40720919777, 58970.94360765738, 58987.48000611698, 59004.016404576585, 59020.552
80303619, 59037.08920149579, 59053.6255999554, 59070.161998415, 59086.698396874606, 59103
.23479533421, 59119.771193793815, 59136.30759225342, 59152.84399071302, 59169.38038917263
, 59185.91678763223, 59202.453186091836, 59218.98958455144, 59235.525983011044, 59252.062
38147065, 59268.59877993025, 59285.13517838986, 59301.67157684946, 59318.207975309066, 59
334.74437376867, 59351.280772228274, 59367.81717068788, 59384.35356914748, 59400.88996760
709, 59417.42636606669, 59433.962764526295, 59450.4991629859, 59467.035561445504, 59483.5
7195990511, 59500.10835836471, 59516.64475682432, 59533.18115528392, 59549.717553743525,
59566.25395220313, 59582.79035066273, 59599.32674912234, 59615.86314758194, 59632.3995460
41546, 59648.93594450115, 59665.472342960755, 59682.00874142036, 59698.54513987996, 59715
.08153833957, 59731.61793679917, 59748.154335258776, 59764.69073371838, 59781.22713217798
4, 59797.76353063759, 59814.29992909719, 59830.8363275568, 59847.3727260164, 59863.909124
476006]
            Date  Predicted
0     2017-04-13          0
1     2017-04-14          0
2     2017-04-17          0
3     2017-04-18          0
4     2017-04-19          0
...          ...        ...
1081  2021-09-07          0
1082  2021-09-08          0
1083  2021-09-09          0
1084  2021-09-10          0
1085  2021-09-13          0

[1086 rows x 2 columns]
```

The ARIMA model may not be the best means of capturing unobservable data in the stock market - i.e. consumer sentiment etc., a better, more accurate means of forecasting may entail the use of Neural Networks.

USE of LSTM (Long Short Term Memory) Model

LSTM is able to more accurately weigh previous information for stock prices, it is able to classify, process and predict time series given lags of unknown duration.

We will start by manipulating the data to make it compatible with the Stacked LSTM Model. This involves reshaping the data such that the values are scaled between 0 and 1, and such that there is only 1 column in an array. We will split the training and test data according to the 80/20 split - 80% of the data will be training data, whereas 20% of the data will be the test data. The Stacked LSTM works by utilising the previous data to predict future data. Hence, we will arrange the data in accordance with this via the create_LSTMdataset function.

Note: For the min max scaler/ normalisation, we need to seperate the training and test sets before normalising, otherwise there will be leakage between the test & training sets.

In [67]:

```
import array
from tensorflow import keras
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
import tensorflow as tf
from tensorflow.keras import layers
from sklearn.preprocessing import MinMaxScaler

#obtaining the correct data from the dataset
LSTM_Data=(df.reset_index()['Close'])
print(len(LSTM_Data))

#performing the splitting of DataSets into train and test
LSTMtraining_len=int(len(LSTM_Data)*0.80)
LSTMtrain=np.array(LSTM_Data[:LSTMtraining_len])
LSTMtest=np.array(LSTM_Data[LSTMtraining_len:])
```

```
#scaling the data to the range between 0 and 1
scaler=MinMaxScaler(feature_range=(0,1))
LSTMtrain=scaler.fit_transform(LSTMtrain.reshape(-1,1))
LSTMtest=scaler.fit_transform(LSTMtest.reshape(-1,1))

print(LSTMtrain.shape)

#creating a function to split the data into x and y, where x input predicts y. setting th
e timestep as 200
def create_LSTMdataset(dataset,time_step):
    X=[]
    Y=[]
    for i in range(len(dataset)-time_step-1):
        #print(dataset[0:i,0])
        a=dataset[i:(i+time_step),0]
        X.append(a)
        Y.append(dataset[i+time_step,0])
    X=np.array(X)
    Y=np.array(Y)
    return X,Y

timesteps=200
print(LSTMtrain.shape)
LSTM_X_train, LSTM_Y_train = create_LSTMdataset(LSTMtrain,timesteps)
print(LSTM_X_train.shape)
LSTM_X_test, LSTM_Y_test= create_LSTMdataset(LSTMtest,timesteps)

# Conducting masking (not necessary)
# embedding = layers.Embedding(input_dim=5000, output_dim=16, mask_zero=True)
# LSTM_X_train_masked_output = embedding(LSTM_X_train)
#LSTM_X_test_masked_output = embedding(LSTM_X_test)

#print(LSTM_X_train_masked_output._keras_mask)
#print(LSTM_X_test_masked_output._keras_mask)

# def create_LSTMdataset2(dataset):
#     X=[]
#     Y=[]
#     for i in range(1, len(dataset)):
#         #print(dataset[0:i,0])
#         X.append(dataset[:i,0])
#         Y.append(dataset[i,0])
#     #X=tf.keras.preprocessing.sequence.pad_sequences(np.array(X),padding="post")
#     X=np.array(X)
#     Y=np.array(Y)
#     return X,Y
```

```
1628
(1302, 1)
(1302, 1)
(1101, 200)
```

There is a need to reshape the X data, such that it is in 3D. The LSTM only accepts 3D data.

In [9]:

```
# print(LSTM_X_train.shape)
# print(LSTM_X_test.shape)

#reshaping the 2D arrays into 3D arrays
LSTM_X_train=LSTM_X_train.reshape(LSTM_X_train.shape[0],LSTM_X_train.shape[1],1)
LSTM_X_test=LSTM_X_test.reshape(LSTM_X_test.shape[0],LSTM_X_test.shape[1],1)
LSTM_Y_train=LSTM_Y_train.reshape(LSTM_Y_train.shape[0],1)
LSTM_Y_test=LSTM_Y_test.reshape(LSTM_Y_test.shape[0],1)


# print(LSTM_X_train.shape)
# print(LSTM_X_test.shape)
# print(LSTM_Y_train.shape)
# print(LSTM_Y_test.shape)
```

Now, it is time to build,compile and train the LSTM model.

A few conceptual notes:

- **LSTM takes an input of fixed lengths**
- **X_train must be the shape of (batch_size, window_size or time_steps, number of features); batch size is the number of samples(or datapoints); the window_size/ time_steps is basically "How many features/ preceding dates we shall use to predict the next date"), the 1 at the end is the number of features per date/ iteration.**
- **y_train must be the shape of (batch_size, n) > batch size is the number of output units; n is the number of timesteps ahead that you are predicting the value, relative to the previous y value (not how many x values you are taking to predict the y value). If you are predicting the immediate next value, n=1.**
- **The arrays within each array need to be of regular size**
- **The input_shape of the LSTM is [batch, timesteps, number of features], but when we specify only 2 parameters the batchsize will be any size and the timesteps and features will be covered by the parameters, i.e input_shape=(2,10), this means batch=any size, timesteps=2,number of features=10**
- **Batch size is the number of samples to work through before updating the model's parameters internally. We choose a mini-batch gradient descent here, of roughly 32.**
- **Epoch is the hyperparameter that defines the number of times the algorithm will work through the entire training set. We iterate 15 times.**

In [10]:

```
model=Sequential()
model.add(LSTM(50,return_sequences=True,input_shape=(LSTM_X_train.shape[1],1))) #this en
sures that the input shape follows the correct 'format'
model.add(LSTM(50,return_sequences=True))
model.add(LSTM(50))
model.add(Dense(1))
model.compile(loss='mean_squared_error',optimizer='adam')

### Printing the inputs of the model to double check what inputs are required for the mod
el
# [print(i.shape, i.dtype) for i in model.inputs]
# [print(o.shape, o.dtype) for o in model.outputs]
# [print(l.name, l.input_shape, l.dtype) for l in model.layers]

print(LSTM_X_train.shape)
model.fit(LSTM_X_train,LSTM_Y_train,epochs=10,batch_size=32)
#try arrays of same sizes for the training input.
```

```
(1101, 200, 1)
Epoch 1/10
35/35 [==============================] - 15s 208ms/step - loss: 0.0526
Epoch 2/10
35/35 [==============================] - 7s 197ms/step - loss: 0.0056
Epoch 3/10
35/35 [==============================] - 7s 202ms/step - loss: 0.0041
Epoch 4/10
35/35 [==============================] - 6s 177ms/step - loss: 0.0040
Epoch 5/10
35/35 [==============================] - 6s 173ms/step - loss: 0.0037
Epoch 6/10
35/35 [==============================] - 7s 190ms/step - loss: 0.0036
Epoch 7/10
17/35 [=============>................] - ETA: 3s - loss: 0.0034

---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_26836/2498118478.py in <module>
     12 
     13 print(LSTM_X_train.shape)
---> 14 model.fit(LSTM_X_train,LSTM_Y_train,epochs=10,batch_size=32)
     15 #try arrays of same sizes for the training input.

~\AppData\Local\Programs\Python\Python39\lib\site-packages\keras\engine\training.py in fi
t(self, x, y, batch_size, epochs, verbose, callbacks, validation_split, validation_data,
shuffle, class_weight, sample_weight, initial_epoch, steps_per_epoch, validation_steps, v
alidation_batch_size, validation_freq, max_queue_size, workers, use_multiprocessing)
```

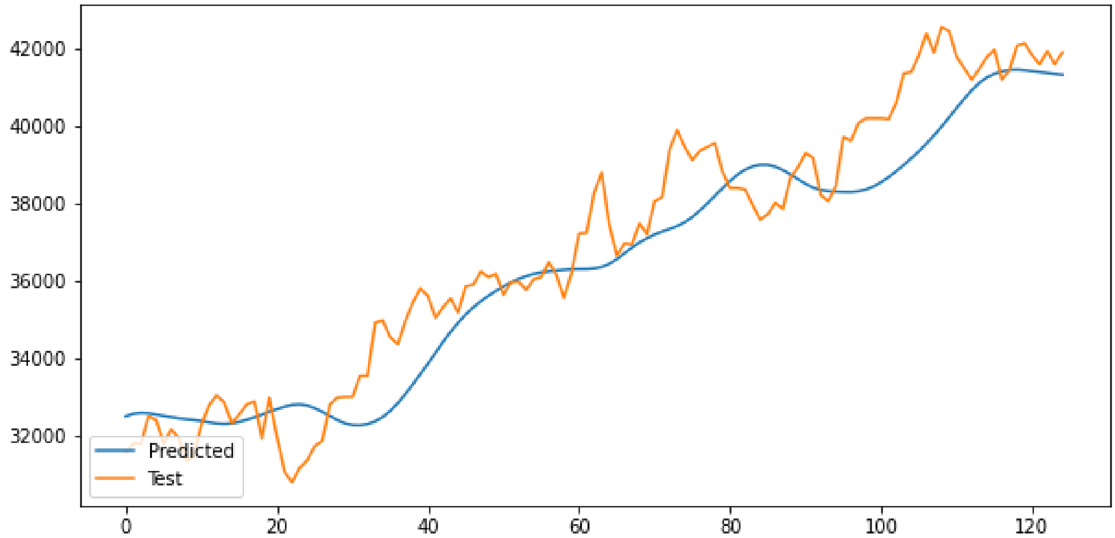
```
alidation_batch_size, validation_freq, max_queue_size, workers, use_multiprocessing,
   1182                 _r=1):
   1183               callbacks.on_train_batch_begin(step)
-> 1184               tmp_logs = self.train_function(iterator)
   1185               if data_handler.should_sync:
   1186                 context.async_wait()

~\AppData\Local\Programs\Python\Python39\lib\site-packages\tensorflow\python\eager\def_fu
nction.py in __call__(self, *args, **kwds)
    883 
    884       with OptionalXlaContext(self._jit_compile):
--> 885         result = self._call(*args, **kwds)
    886 
    887       new_tracing_count = self.experimental_get_tracing_count()

~\AppData\Local\Programs\Python\Python39\lib\site-packages\tensorflow\python\eager\def_fu
nction.py in _call(self, *args, **kwds)
    915       # In this case we have created variables on the first call, so we run the
    916       # defunned version which is guaranteed to never create variables.
--> 917       return self._stateless_fn(*args, **kwds)  # pylint: disable=not-callable
    918     elif self._stateful_fn is not None:
    919       # Release the lock early so that multiple threads can perform the call

~\AppData\Local\Programs\Python\Python39\lib\site-packages\tensorflow\python\eager\functi
on.py in __call__(self, *args, **kwargs)
   3037       (graph_function,
   3038        filtered_flat_args) = self._maybe_define_function(args, kwargs)
-> 3039     return graph_function._call_flat(
   3040         filtered_flat_args, captured_inputs=graph_function.captured_inputs)  # py
lint: disable=protected-access
   3041 

~\AppData\Local\Programs\Python\Python39\lib\site-packages\tensorflow\python\eager\functi
on.py in _call_flat(self, args, captured_inputs, cancellation_manager)
   1961         and executing_eagerly):
   1962       # No tape is watching; skip to running the function.
-> 1963       return self._build_call_outputs(self._inference_function.call(
   1964           ctx, args, cancellation_manager=cancellation_manager))
   1965     forward_backward = self._select_forward_and_backward_functions(

~\AppData\Local\Programs\Python\Python39\lib\site-packages\tensorflow\python\eager\functi
on.py in call(self, ctx, args, cancellation_manager)
    589       with _InterpolateFunctionError(self):
    590         if cancellation_manager is None:
--> 591           outputs = execute.execute(
    592               str(self.signature.name),
    593               num_outputs=self._num_outputs,

~\AppData\Local\Programs\Python\Python39\lib\site-packages\tensorflow\python\eager\execut
e.py in quick_execute(op_name, num_outputs, inputs, attrs, ctx, name)
     57   try:
     58     ctx.ensure_initialized()
---> 59     tensors = pywrap_tfe.TFE_Py_Execute(ctx._handle, device_name, op_name,
     60                                         inputs, attrs, num_outputs)
     61   except core._NotOkStatusException as e:

KeyboardInterrupt:
```

Making predictions using the stacked LSTM. Thereafter, we validate the model by producing graphs and calculating the RMSE.

In [ ]:

```
LSTM_test_predict=model.predict(LSTM_X_test)   #prediction will output an array
LSTM_test_predict=scaler.inverse_transform(LSTM_test_predict) #we will then convert the a
rray from scaled values to the original values

LSTM_testDatavalues=df.reset_index()['Close'][LSTMtraining_len+timesteps+1:] #converting
to series object in order to plot graph
LSTM_testDatavalues=LSTM_testDatavalues.reset_index(drop=True)
```

```
LSTM_test_predictvalues = pd.DataFrame(LSTM_test_predict).iloc[:,0] #converting to series object in order to plot graph
plt.figure(figsize=(10,5))
plt.plot(LSTM_test_predictvalues,label="Predicted")
plt.plot(LSTM_testDatavalues,label="Test")
plt.legend(loc='lower left')
plt.show()

from sklearn.metrics import mean_squared_error
print(type(LSTM_test_predictvalues))
print(type(LSTM_testDatavalues))
LSTM_test_predictvalues_lst = LSTM_test_predictvalues.tolist()
LSTM_testDatavalues_lst = LSTM_testDatavalues.tolist()
#print(LSTM_test_predictvalues_lst)
#print(LSTM_testDatavalues_lst)
print(mean_squared_error(LSTM_testDatavalues_lst,LSTM_test_predictvalues_lst,squared=False))
```




```
<class 'pandas.core.series.Series'>
<class 'pandas.core.series.Series'>
1195.478899256284
```

The above is an explanation of using an LSTM model to predict the stock prices. It used a timestep of about 200, essentially meaning that 200 previous dates were used to predict the next date. However, the optimal number of timesteps is debatable: Given that different investors use different time frames for prediction. To obtain the most accurate model, we will try different timesteps of about 50, 100, 150, 200. These timestep values are not randomly selected. Although its not entirely the same, this 'kind-of' draws parallels with how investors use information from the previous 50, 100, 200 days (through moving averages), to predict the stock trends of the next day.

Hence, we will now create a function that we will iterate with, to obtain the best model parameters.

In [ ]:

```
def optimal_LSTM_model(timesteps):
    #obtaining the correct data from the dataset
    LSTM_Data=(df.reset_index()['Close'])

    #performing the splitting of DataSets into train and test

    LSTMtraining_len=int(len(LSTM_Data)*0.80)
    LSTMtrain=np.array(LSTM_Data[:LSTMtraining_len])
    LSTMtest=np.array(LSTM_Data[LSTMtraining_len:])

    #scaling the data to the range between 0 and 1

    scaler=MinMaxScaler(feature_range=(0,1))
    LSTMtrain=scaler.fit_transform(LSTMtrain.reshape(-1,1))
    LSTMtest=scaler.fit_transform(LSTMtest.reshape(-1,1))
    LSTM_X_train, LSTM_Y_train = create_LSTMdataset(LSTMtrain,timesteps)
    LSTM_X_test, LSTM_Y_test= create_LSTMdataset(LSTMtest,timesteps)
```

```python
    LSTM_X_train=LSTM_X_train.reshape(LSTM_X_train.shape[0],LSTM_X_train.shape[1],1)
    LSTM_X_test=LSTM_X_test.reshape(LSTM_X_test.shape[0],LSTM_X_test.shape[1],1)
    LSTM_Y_train=LSTM_Y_train.reshape(LSTM_Y_train.shape[0],1)
    LSTM_Y_test=LSTM_Y_test.reshape(LSTM_Y_test.shape[0],1)

    model=Sequential()
    model.add(LSTM(50,return_sequences=True,input_shape=(LSTM_X_train.shape[1],1))) #this ensures that the input shape follows the correct 'format'
    model.add(LSTM(50,return_sequences=True))
    model.add(LSTM(50))
    model.add(Dense(1))
    model.compile(loss='mean_squared_error',optimizer='adam')

    model.fit(LSTM_X_train,LSTM_Y_train,epochs=10,batch_size=32)

    LSTM_test_predict=model.predict(LSTM_X_test)  #prediction will output an array
    LSTM_test_predict=scaler.inverse_transform(LSTM_test_predict) #we will then convert the array from scaled values to the original values

    LSTM_testDatavalues=df.reset_index()['Close'][LSTMtraining_len+timesteps+1:] #converting to series object in order to plot graph
    LSTM_testDatavalues=LSTM_testDatavalues.reset_index(drop=True)
    LSTM_test_predictvalues = pd.DataFrame(LSTM_test_predict).iloc[:,0] #converting to series object in order to plot graph
    # plt.figure(figsize=(10,5))
    # plt.plot(LSTM_test_predictvalues,label="Predicted")
    # plt.plot(LSTM_testDatavalues,label="Test")
    # plt.legend(loc='lower left')
    # plt.show()
    LSTM_test_predictvalues_lst = LSTM_test_predictvalues.tolist()
    LSTM_testDatavalues_lst = LSTM_testDatavalues.tolist()
    #print(LSTM_test_predictvalues_lst)
    #print(LSTM_testDatavalues_lst)
    return mean_squared_error(LSTM_testDatavalues_lst,LSTM_test_predictvalues_lst,squared=False)

lst_timesteps={50:[],100:[],200:[]}

def obtainRSME(lst_of_timesteps):
    for time in lst_of_timesteps:
        lst_of_timesteps[time].append(optimal_LSTM_model(time))
    return lst_of_timesteps

obtainRSME(lst_timesteps)

new_lst_timesteps={75:[],125:[],150:[]}
print(obtainRSME(new_lst_timesteps))
```

```
Epoch 1/10
40/40 [==============================] - 12s 54ms/step - loss: 0.0530
Epoch 2/10
40/40 [==============================] - 2s 52ms/step - loss: 0.0048
Epoch 3/10
40/40 [==============================] - 2s 51ms/step - loss: 0.0035
Epoch 4/10
40/40 [==============================] - 2s 52ms/step - loss: 0.0034
Epoch 5/10
40/40 [==============================] - 2s 51ms/step - loss: 0.0032
Epoch 6/10
40/40 [==============================] - 2s 51ms/step - loss: 0.0032
Epoch 7/10
40/40 [==============================] - 2s 53ms/step - loss: 0.0029
Epoch 8/10
40/40 [==============================] - 2s 53ms/step - loss: 0.0027
Epoch 9/10
40/40 [==============================] - 2s 51ms/step - loss: 0.0026
Epoch 10/10
40/40 [==============================] - 2s 53ms/step - loss: 0.0025
Epoch 1/10
38/38 [==============================] - 16s 112ms/step - loss: 0.0592
Epoch 2/10
```

```
38/38 [==============================] - 4s 115ms/step - loss: 0.0056
Epoch 3/10
38/38 [==============================] - 4s 116ms/step - loss: 0.0036
Epoch 4/10
38/38 [==============================] - 4s 115ms/step - loss: 0.0034
Epoch 5/10
38/38 [==============================] - 4s 118ms/step - loss: 0.0033
Epoch 6/10
38/38 [==============================] - 5s 133ms/step - loss: 0.0030
Epoch 7/10
38/38 [==============================] - 4s 116ms/step - loss: 0.0028
Epoch 8/10
38/38 [==============================] - 4s 109ms/step - loss: 0.0030
Epoch 9/10
38/38 [==============================] - 4s 110ms/step - loss: 0.0030
Epoch 10/10
38/38 [==============================] - 4s 113ms/step - loss: 0.0026
Epoch 1/10
35/35 [==============================] - 13s 192ms/step - loss: 0.0461
Epoch 2/10
35/35 [==============================] - 7s 188ms/step - loss: 0.0052
Epoch 3/10
35/35 [==============================] - 7s 192ms/step - loss: 0.0039
Epoch 4/10
35/35 [==============================] - 7s 189ms/step - loss: 0.0036
Epoch 5/10
35/35 [==============================] - 7s 191ms/step - loss: 0.0035
Epoch 6/10
35/35 [==============================] - 7s 193ms/step - loss: 0.0031
Epoch 7/10
35/35 [==============================] - 7s 189ms/step - loss: 0.0030
Epoch 8/10
35/35 [==============================] - 7s 193ms/step - loss: 0.0028
Epoch 9/10
35/35 [==============================] - 7s 188ms/step - loss: 0.0029
Epoch 10/10
35/35 [==============================] - 7s 189ms/step - loss: 0.0028
```

Out[ ]:

```
{50: [1140.2485123779595],
 100: [1061.1554992884144],
 200: [1258.0689310005735]}
```

NOTE **Initially I ran the LSTM with a lag of 1. This essentially means that a set number of all of the previous values would predict the next immediate value. However, I realise that this has to be adapted because the forecasting requires you to predict about 1087 values in advance. Thus, I have to modify the algorithm to include the lag. As it seems like the dataset (of about 1628 values) is not large enough to be split into 2 non-overlapping/ seperate portions, we need to split the dataset into the training set containing the first 1387 values, and the test set containing the final 1387 values of the dataset. This will require improvisation.**

I will try to implement this in a new algorithm below, but definitely, I would the validation with a pinch of salt (because of the overlap between the different datasets) - the validation may not be all that useful, but I guess it opens our minds to new ideas.

For more accuracy, the final model should be trained on all the data.

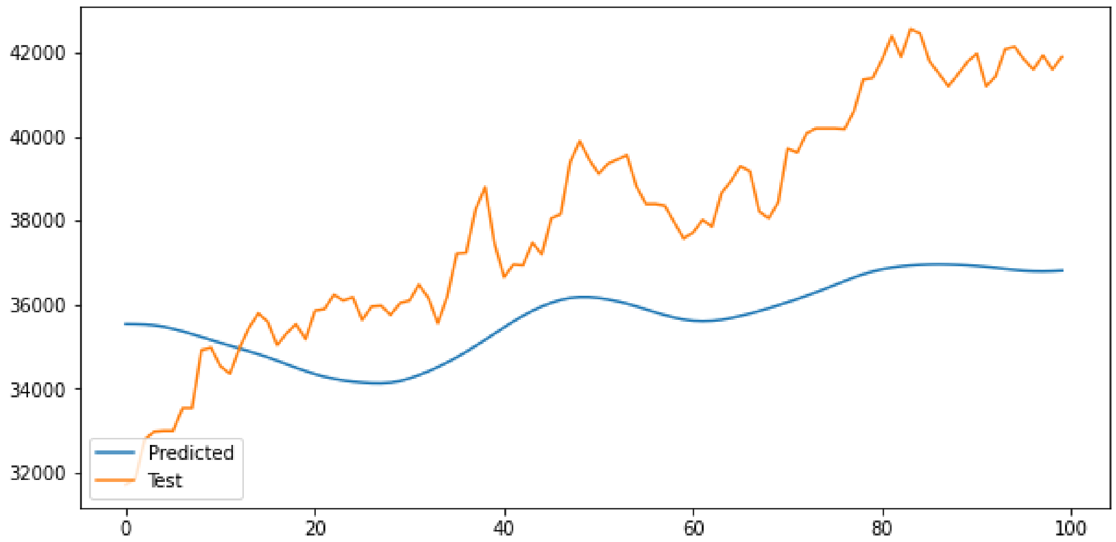
In [114]:

```
def create_LR_LSTMdataset(dataset,time_step):
    X=[]
    Y=[]
    lag=1087
    for i in range(len(dataset)-time_step-1-lag):
        #print(dataset[0:i,0])
        a=dataset[i:(i+time_step),0]
        X.append(a)
        Y.append(dataset[i+time_step+lag,0])
    X=np.array(X)
    Y=np.array(Y)
```

```
    return X,Y

def long_range_LSTM_model(timesteps):
    #obtaining the correct data from the dataset

    LSTM_Data=(df.reset_index()['Close'])

    #performing the splitting of DataSets into train and test

    #LSTMtraining_len=int(len(LSTM_Data)*0.80)
    LSTMtrain=np.array(LSTM_Data[:1388])
    LSTMtest=np.array(LSTM_Data[240:])

    #scaling the data to the range between 0 and 1

    scaler=MinMaxScaler(feature_range=(0,1))
    LSTMtrain=scaler.fit_transform(LSTMtrain.reshape(-1,1))
    LSTMtest=scaler.fit_transform(LSTMtest.reshape(-1,1))
    print(len(LSTMtrain))
    print(len(LSTMtest))
    LSTM_X_train, LSTM_Y_train = create_LR_LSTMdataset(LSTMtrain,timesteps)
    LSTM_X_test, LSTM_Y_test= create_LR_LSTMdataset(LSTMtest,timesteps)

    LSTM_X_train=LSTM_X_train.reshape(LSTM_X_train.shape[0],LSTM_X_train.shape[1],1)
    LSTM_X_test=LSTM_X_test.reshape(LSTM_X_test.shape[0],LSTM_X_test.shape[1],1)
    LSTM_Y_train=LSTM_Y_train.reshape(LSTM_Y_train.shape[0],1)
    LSTM_Y_test=LSTM_Y_test.reshape(LSTM_Y_test.shape[0],1)

    model=Sequential()
    model.add(LSTM(50,return_sequences=True,input_shape=(LSTM_X_train.shape[1],1))) #thi
s ensures that the input shape follows the correct 'format'
    model.add(LSTM(50,return_sequences=True))
    model.add(LSTM(50))
    model.add(Dense(1))
    model.compile(loss='mean_squared_error',optimizer='adam')

    model.fit(LSTM_X_train,LSTM_Y_train,epochs=10,batch_size=32)

    LSTM_test_predict=model.predict(LSTM_X_test)  #prediction will output an array
    LSTM_test_predict=scaler.inverse_transform(LSTM_test_predict) #we will then convert
the array from scaled values to the original values

    LSTM_testDatavalues=df.reset_index()['Close'][240+timesteps+1+1087:] #converting to
series object in order to plot graph
    LSTM_testDatavalues=LSTM_testDatavalues.reset_index(drop=True)
    LSTM_test_predictvalues = pd.DataFrame(LSTM_test_predict).iloc[:,0] #converting to se
ries object in order to plot graph

    plt.figure(figsize=(10,5))
    plt.plot(LSTM_test_predictvalues,label="Predicted")
    plt.plot(LSTM_testDatavalues,label="Test")
    plt.legend(loc='lower left')
    plt.show()

    LSTM_test_predictvalues_lst = LSTM_test_predictvalues.tolist()
    LSTM_testDatavalues_lst = LSTM_testDatavalues.tolist()
    #print(LSTM_test_predictvalues_lst)
    #print(LSTM_testDatavalues_lst)
    print(mean_squared_error(LSTM_testDatavalues_lst,LSTM_test_predictvalues_lst,squared=
False))

long_range_LSTM_model(200)
```

```
1388
1388
Epoch 1/10
4/4 [==============================] - 5s 173ms/step - loss: 0.3509
Epoch 2/10
4/4 [==============================] - 1s 181ms/step - loss: 0.1595
Epoch 3/10
4/4 [==============================] - 1s 228ms/step - loss: 0.0414
Epoch 4/10
```

```
4/4 [==============================] - 1s 209ms/step - loss: 0.0085
Epoch 5/10
4/4 [==============================] - 1s 208ms/step - loss: 0.0266
Epoch 6/10
4/4 [==============================] - 1s 173ms/step - loss: 0.0131
Epoch 7/10
4/4 [==============================] - 1s 170ms/step - loss: 0.0039
Epoch 8/10
4/4 [==============================] - 1s 175ms/step - loss: 0.0093
Epoch 9/10
4/4 [==============================] - 1s 212ms/step - loss: 0.0029
Epoch 10/10
4/4 [==============================] - 1s 215ms/step - loss: 0.0038
```




```
3173.232323277622
```

Due to limitations with the training set, we had to make do with a small timestep of only 200. Any timestep greater and we would have greater overlapping portions between the test and the training set. However, as we have to predict 1086 days into the future, I would train the model with the entirety of the existing dataset to maximise the accuracy in the CSV file submission, and with a very large timestep of about 1-2 years (500 days/ timesteps).

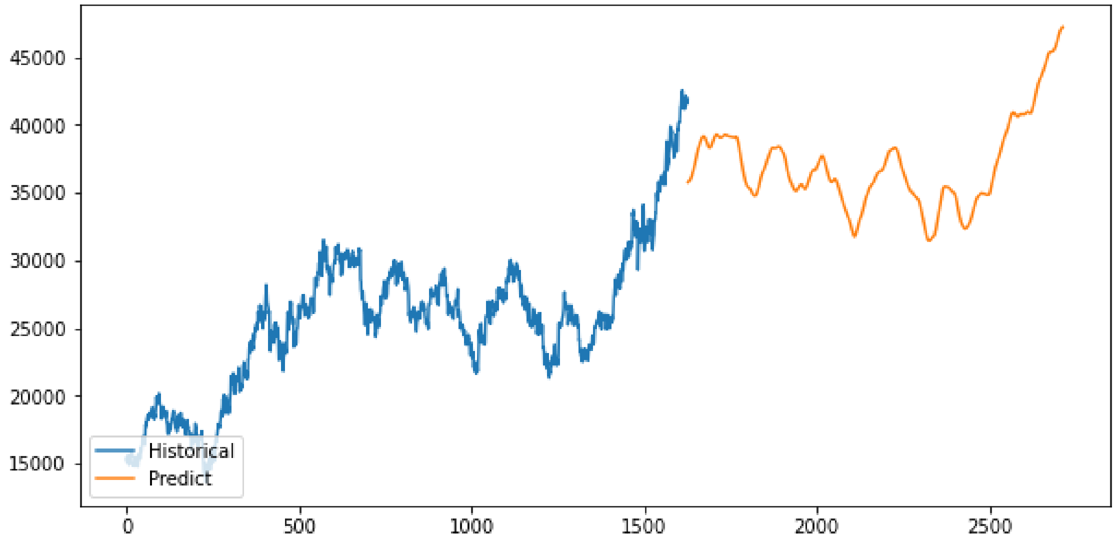
In [111]:

```
def create_LR_LSTMdataset(dataset,time_step):
    X=[]
    Y=[]
    lag=1086
    for i in range(len(dataset)-time_step-1-lag):
        #print(dataset[0:i,0])
        a=dataset[i:(i+time_step),0]
        X.append(a)
        Y.append(dataset[i+time_step+lag,0])
    X=np.array(X)
    Y=np.array(Y)
    return X,Y

def create_actualpredict_LSTMdataset(dataset,time_step):
    X=[]
    lag=1086
    for i in range((len(dataset)-time_step-lag),(len(dataset)-time_step)):
        a=dataset[i:(i+time_step),0]
        X.append(a)
    X=np.array(X)
    X.reshape(X.shape[0],time_step)
    return X

#LSTM_Data=(df.reset_index()['Close'])

#performing the splitting of DataSets into train and test
```

```
# #LSTMtraining_len=int(len(LSTM_Data)*0.80)
# LSTMtrain=np.array(LSTM_Data[:])
# LSTMtest=np.array(LSTM_Data[240:])

# #scaling the data to the range between 0 and 1

# scaler=MinMaxScaler(feature_range=(0,1))
# LSTMtrain=scaler.fit_transform(LSTMtrain.reshape(-1,1))
# LSTMtest=scaler.fit_transform(LSTMtest.reshape(-1,1))

# LSTM_X_train = create_actualpredict_LSTMdataset(LSTMtrain,500)
# # LSTM_X_test, LSTM_Y_test= create_LR_LSTMdataset(LSTMtest,100)
# print(LSTM_X_train.shape)
# print("This is the",LSTM_X_train.shape)

# LSTM_X_train=LSTM_X_train.reshape(LSTM_X_train.shape[0],LSTM_X_train.shape[1],1)
# print("This is the new",LSTM_X_train.shape)
# # LSTM_X_test=LSTM_X_test.reshape(LSTM_X_test.shape[0],LSTM_X_test.shape[1],1)
# # LSTM_Y_train=LSTM_Y_train.reshape(LSTM_Y_train.shape[0],1)
# # LSTM_Y_test=LSTM_Y_test.reshape(LSTM_Y_test.shape[0],1)

def actual_LSTM_model(timesteps):
    #obtaining the correct data from the dataset

    LSTM_DataFrame=df.reset_index()['Close']
    LSTM_Data=np.array(LSTM_DataFrame[:])

    #performing the splitting of DataSets into train and test

    #LSTMtraining_len=int(len(LSTM_Data)*0.80)

    #scaling the data to the range between 0 and 1

    scaler=MinMaxScaler(feature_range=(0,1))
    LSTMtrain=scaler.fit_transform(LSTM_Data.reshape(-1,1))
    #LSTMtest=scaler.fit_transform(LSTMtest.reshape(-1,1))
    LSTM_X_train, LSTM_Y_train = create_LR_LSTMdataset(LSTMtrain,timesteps)
    LSTM_X_actual = create_actualpredict_LSTMdataset(LSTMtrain,timesteps)

    LSTM_X_train=LSTM_X_train.reshape(LSTM_X_train.shape[0],LSTM_X_train.shape[1],1)
    LSTM_X_actual=LSTM_X_actual.reshape(LSTM_X_actual.shape[0],LSTM_X_actual.shape[1],1)
    LSTM_Y_train=LSTM_Y_train.reshape(LSTM_Y_train.shape[0],1)
    # LSTM_Y_test=LSTM_Y_test.reshape(LSTM_Y_test.shape[0],1)

    model=Sequential()
    model.add(LSTM(50,return_sequences=True,input_shape=(LSTM_X_train.shape[1],1))) #thi
s ensures that the input shape follows the correct 'format'
    model.add(LSTM(50,return_sequences=True))
    model.add(LSTM(50))
    model.add(Dense(1))
    model.compile(loss='mean_squared_error',optimizer='adam')

    model.fit(LSTM_X_train,LSTM_Y_train,epochs=10,batch_size=32)

    LSTM_test_predict=model.predict(LSTM_X_actual)  #prediction will output an array
    LSTM_test_predict=scaler.inverse_transform(LSTM_test_predict) #we will then convert
the array from scaled values to the original values

    # LSTM_testDatavalues=df.reset_index()['Close'][240+timesteps+1+1087:] #converting to
series object in order to plot graph
    # LSTM_testDatavalues=LSTM_testDatavalues.reset_index(drop=True)
    LSTM_test_predictvalues = pd.DataFrame(LSTM_test_predict).iloc[:,0] #converting to se
ries object in order to plot graph
    LSTM_test_predictvalues.index=pd.RangeIndex(1628,1628+1086)

    plt.figure(figsize=(10,5))
    plt.plot(LSTM_DataFrame,label="Historical")
    plt.plot(LSTM_test_predictvalues,label="Predict")
    plt.legend(loc='lower left')
    plt.show()

    LSTM_test_predictvalues_lst = LSTM_test_predictvalues.tolist()
```

```
    # LSTM_testDatavalues_lst = LSTM_testDatavalues.tolist()
    #print(LSTM_test_predictvalues_lst)
    #print(LSTM_testDatavalues_lst)
    # print(mean_squared_error(LSTM_testDatavalues_lst,LSTM_test_predictvalues_lst,square
d=False))
    return LSTM_test_predictvalues_lst
final_values = actual_LSTM_model(500)
```

```
Epoch 1/10
2/2 [==============================] - 5s 343ms/step - loss: 0.8636
Epoch 2/10
2/2 [==============================] - 1s 429ms/step - loss: 0.6386
Epoch 3/10
2/2 [==============================] - 1s 430ms/step - loss: 0.3877
Epoch 4/10
2/2 [==============================] - 1s 505ms/step - loss: 0.1069
Epoch 5/10
2/2 [==============================] - 1s 475ms/step - loss: 0.0469
Epoch 6/10
2/2 [==============================] - 1s 434ms/step - loss: 0.1201
Epoch 7/10
2/2 [==============================] - 1s 433ms/step - loss: 0.0178
Epoch 8/10
2/2 [==============================] - 1s 485ms/step - loss: 0.0087
Epoch 9/10
2/2 [==============================] - 1s 500ms/step - loss: 0.0356
Epoch 10/10
2/2 [==============================] - 1s 445ms/step - loss: 0.0441
```



Create a function to write a list of data (the predicted stock prices) into a CSV file.

In [ ]:

```
write_ans_toCSV(final_values)
```

Usage of SVM for stock prices. The SVM is a usually more robust version of linear regression, in the sense that it is less sensitive to outliers in the dataset. Applied in the context of the stock market, the algorithm produced will more closely match the general trend of the stock (and be less sensitive to market fluctuations). Thus, the algorithm will be more accurate for long-term investing (for which the day-to-day/ week-to-week fluctuations occur less often).

In [112]:

```
from sklearn.svm import SVR
from sklearn.svm import LinearSVR


history_close_prices = df['Close'].reset_index(drop=True)
critical_length = int(len(history_close_prices)*0.8)
train_history_close_prices = history_close_prices[:critical_length].tolist()
```

```
train_days = [[i] for i in range(1,critical_length+1)]
test_history_close_prices = history_close_prices[critical_length:].tolist()
new_day = critical_length + 1
end_day = new_day + len(test_history_close_prices)
test_days = [[i] for i in range(new_day,end_day)]
```

Using LinearSVR to predict stock prices

In [113]:

```
#print(train_days)
#print(train_history_close_prices)

lin_svr = LinearSVR()
lin_svr.fit(train_days, train_history_close_prices)

# poly_svr = SVR(kernel='poly',C=1000, degree=2)
# poly_svr.fit(train_days, train_history_close_prices)

# rbf_svr = SVR(kernel='rbf',C=1000, gamma=0.15)
# rbf_svr.fit(train_days, train_history_close_prices)
```

```
C:\Users\65932\AppData\Local\Programs\Python\Python39\lib\site-packages\sklearn\svm\_base
.py:1199: ConvergenceWarning: Liblinear failed to converge, increase the number of iterat
ions.
  warnings.warn(
```

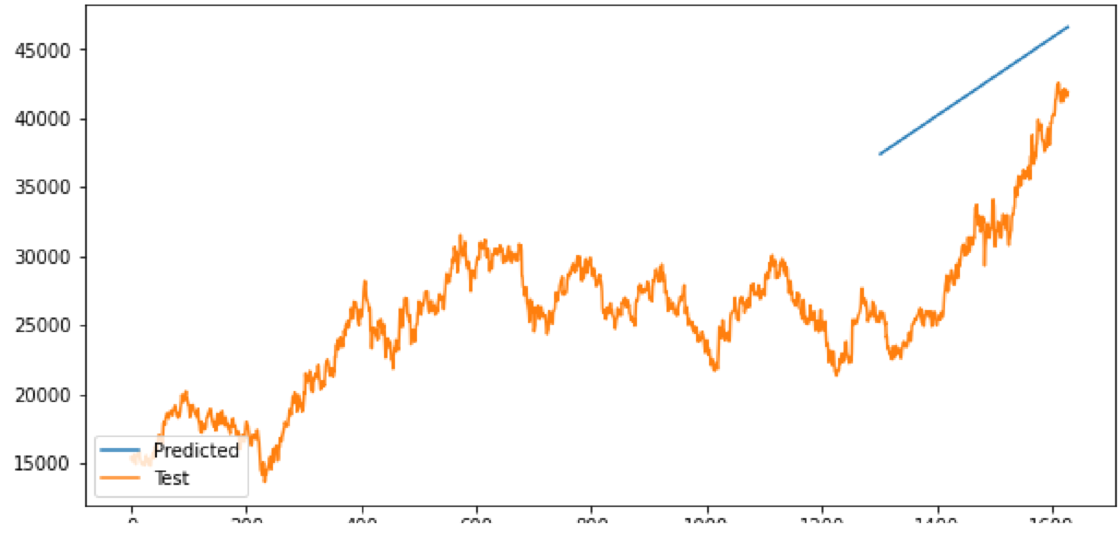
Out[113]:

```
LinearSVR()
```

Stock Price Predictions using the SVR model. Validating the model.

In [125]:

```
svr_test_predict=lin_svr.predict(test_days)
svr_predictdf = pd.DataFrame(svr_test_predict)
svr_testdf = pd.DataFrame(test_history_close_prices)
#LSTM_test_predictvalues = pd.DataFrame(LSTM_test_predict).iloc[:,0] #converting to serie
s object in order to plot graph
svr_predictdf.index=pd.RangeIndex(critical_length,1628)
plt.figure(figsize=(10,5))
plt.plot(svr_predictdf,label="Predicted")
plt.plot(history_close_prices,label="Test")
plt.legend(loc='lower left')
plt.show()

#obtaining the RMSE
print(type(svr_predictdf))
print(type(svr_testdf))
lst_svr_predictdf=svr_predictdf.iloc[:,0].tolist()
lst_svr_testdf=svr_testdf.iloc[:,0].tolist()
print(mean_squared_error(lst_svr_predictdf,lst_svr_testdf,squared=False))
```



```
<class 'pandas.core.frame.DataFrame'>
<class 'pandas.core.frame.DataFrame'>
11469.80289419135
```

SVR yields a result that is pretty far off > Seemingly even further off compared to our ARIMA model. As the LSTM model cannot be validated, we will use our ARIMA model as our main result.