Time Series Analysis

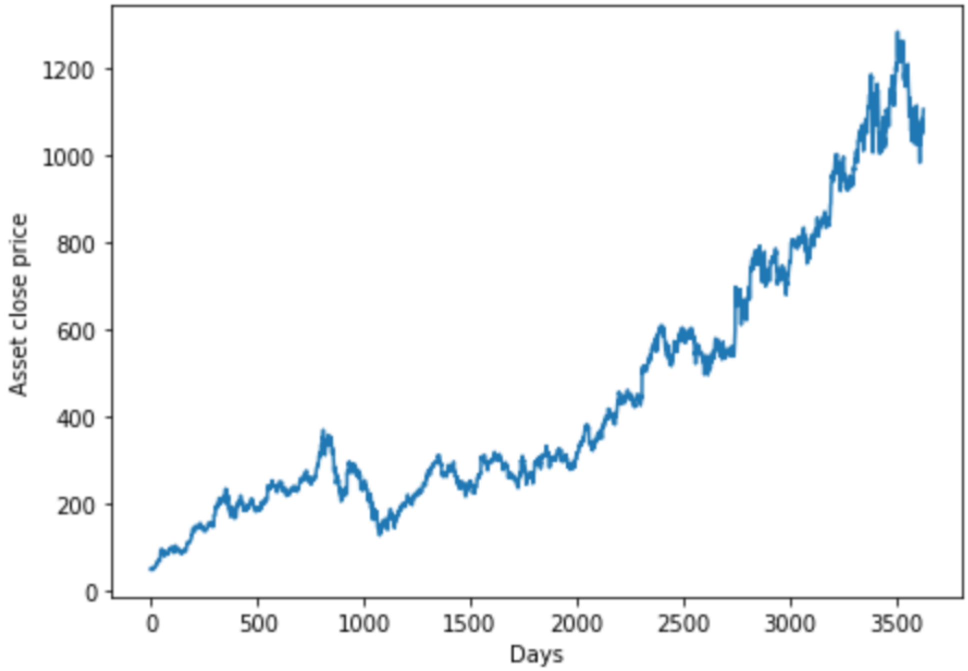
Fernando Montes

A time series is a series of data points indexed in time order. Modeling of a time series is usually done in order to extract meaningful statistics and/or to predict future values based on previously observed values. In doing so, it is assumed that what happened in the past is a good starting point for predicting what will happen in the future. Time series analysis can be useful, for example, in finance to see how a given asset, security, or economic variable changes over time. For this project a polynomial regression, an AutoRegressive Integrated Moving Average (ARIMA), a Recurrent Neural Network (RNN), and a Long Short-Term Network (LSTM) model were developed with the purpose of predicting the asset price N days ahead using asset prices M days in the past on a rolling basis.

The models were developed and optimized using data from price data 2004-08-19 to 2019-01-18. The data consisted of price of an asset at the start of the day, highest and lowest price reached that day, and closing daily price. The analysis used the closing daily price only.

```
In [10]: # Reading file and inspecting it
         seriesFull = pd.read_csv('../data/raw/example.csv', header=0)
         print(seriesFull.head())
         series = seriesFull[['Index','Close']]
         plt.plot(series['Close'])
         plt.show()
```

```
        Index       Open       High        Low      Close
0  2004-08-19  50.050049  52.082081  48.028027  50.220219
1  2004-08-20  50.555557  54.594593  50.300301  54.209209
2  2004-08-23  55.430431  56.796795  54.579578  54.754753
3  2004-08-24  55.675674  55.855854  51.836838  52.487488
4  2004-08-25  52.532532  54.054054  51.991993  53.053055
```



The metric used to quantify the accuracy of the predictions was the root mean square error (RMSE). All model parameters were optimized on a rolling basis using M days in the past to predict N days ahead. As such all the models predicted a rolling estimate of the future price over the whole time range excluding the first M days. Since for the ARIMA, polynomial and SVM models, the model is optimized without "seeing" the future (price asset N days ahead), their RMSE is calculated over the whole time range and it represents the out-of-bag error of the model. For the RNN and LSTM models, the whole time range is randomly separated into train (70%) and test (30%) data. For those models the RMSE is calculated on the test data only.

Comment about the code: The algorithms used in our analysis employ either native Scikit-Learn base estimator classes or have been written based on them. Two of the algorithms employed (RNN and LSTM) used the TensorFlow framework, and therefore two new classes, RNNClassifier and LSTMClassifier (Scikit-Learn compatible) were written. The strategies discussed here were coded using unique classes that rely heavily on parent inheritance.

```
In [8]: steps_ahead = 30 #N
        fit_range = 50   #M
```

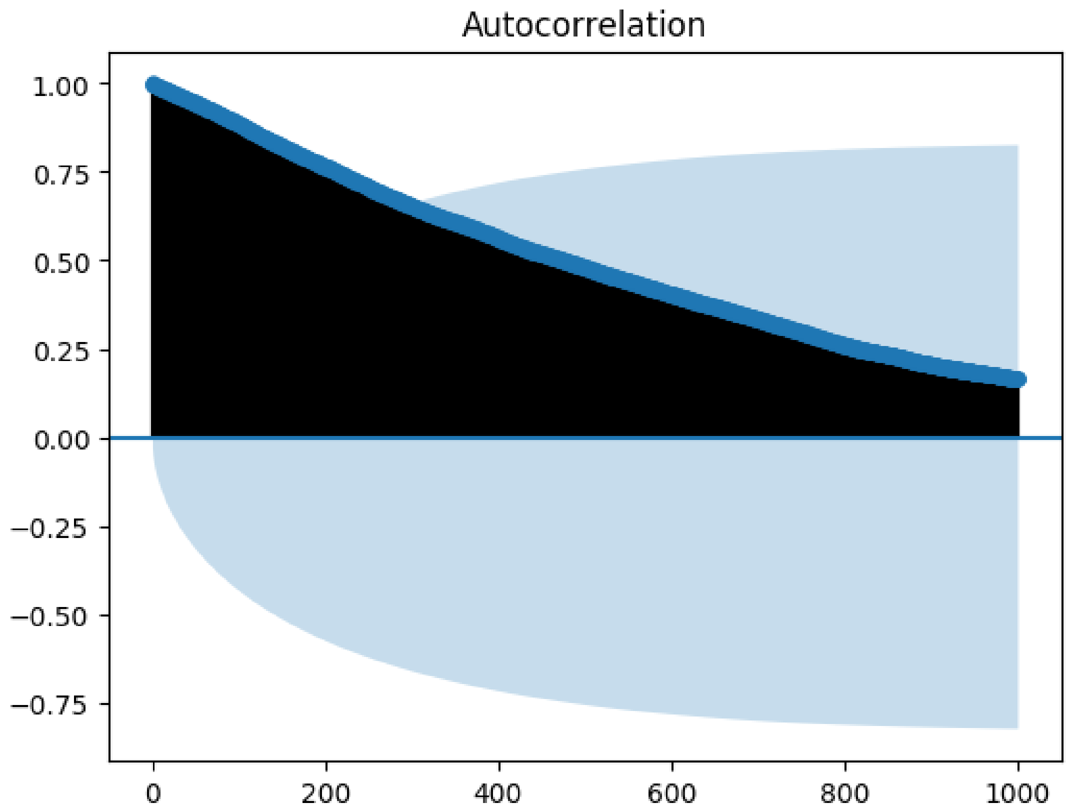
1 AutoRegressive Integrated Moving Average (ARIMA)

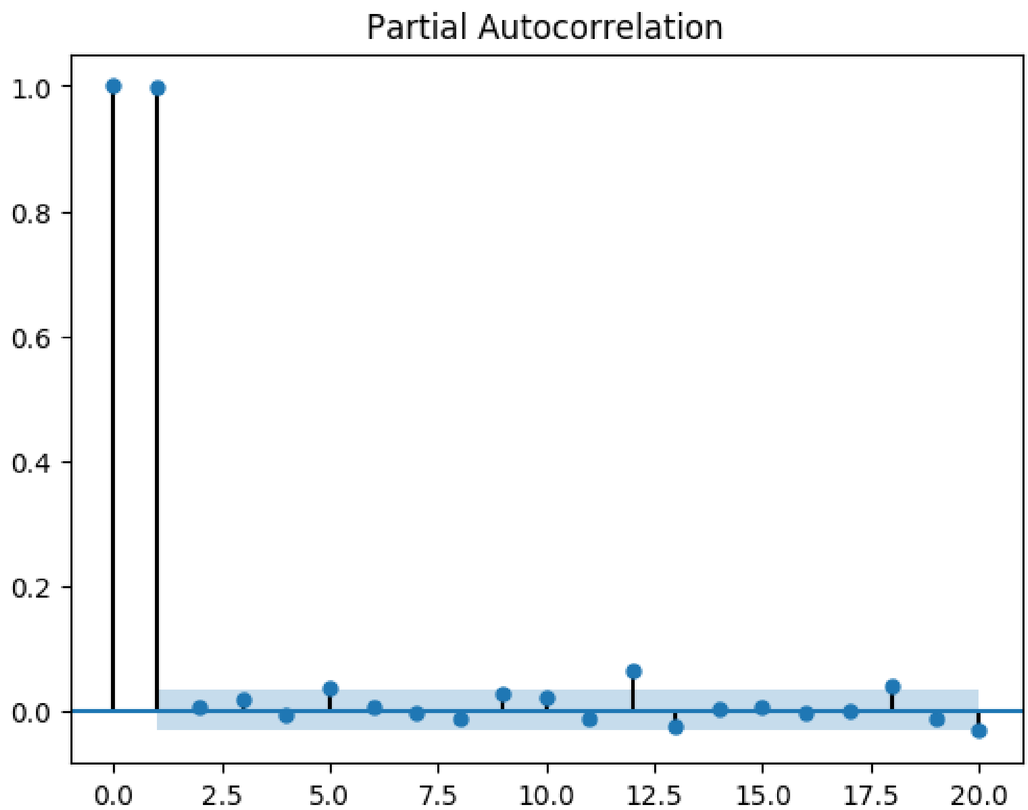
The **ARIMA** model has hyper-parameters (**p,d,q**) that represent different corrections or features of the model:

- Autogression: Relationship between an observation and a number of lag observations (**p**).
- Integrated: To make the time series stationary and remove trend and seasonal structures, the model subtracts an observation from an observation at a previous time step. The number of times that the raw observations are differenced (subtracted) is **d**.
- Moving average: The model uses the dependency between an observation and a residual error with a moving average (of length **q**)

It is possible to obtain insight into the autoregression parameter **p** by plotting the Autocorrelation (ACF) and partial (pACF) autocorrelation functions:

```
In [8]: # Finding the Autocorrelation (ACF) and partial (pACF) Autocorrelation functions
        tsaplots.plot_acf(series['Close'], lags = 1000)
        tsaplots.plot_pacf(series['Close'], lags = 20)
        plt.show()
```





The graphs above suggest that there is autoregression up to very large lags but mostly it can be explained by **p=1**. The optimal ARIMA parameters (**p,d,q**) can be found by iterating over the different combinations:

```
In [17]: p = d = q = range(0, 3)
         parameters = list(itertools.product(p, d, q))
         model_selection = ['ARIMA', parameters]
         bestParam, bestRMSE, gridResults = gridSearch(series, model_selection, steps_ahead, fit_range, fastR
```

```
Method: ARIMA - Param: (0, 0, 0) - RMSE: 50.3918
Method: ARIMA - Param: (0, 0, 1) - RMSE: 50.2967
Method: ARIMA - Param: (0, 1, 0) - RMSE: 53.2371
Method: ARIMA - Param: (0, 1, 1) - RMSE: 53.3756
Method: ARIMA - Param: (0, 1, 2) - RMSE: 53.7979
Method: ARIMA - Param: (0, 2, 0) - RMSE: 334.8272
Method: ARIMA - Param: (0, 2, 1) - RMSE: 139.2466
Method: ARIMA - Param: (1, 0, 0) - RMSE: 44.6161
Method: ARIMA - Param: (1, 1, 0) - RMSE: 54.5148
Method: ARIMA - Param: (1, 2, 0) - RMSE: 268.3600
Method: ARIMA - Param: (2, 1, 0) - RMSE: 53.8237
Best parameters for method ARIMA
Param: (1, 0, 0) - RMSE: 44.6161
```

The lowest RMSE (44.62) is obtained for ARIMA(**1,0,0**). It should be noted that ARIMA(0,0,0), consisting of a constant and white noise, has an RMSE of 50.39 which is not significantly worse than the optimized model.

2 Polynomial with ridge regularization

This model fits the asset prices M days in the past with a polynomial of order 3. A prediction N days ahead uses the fitted polynomial. The regression cost function uses a quadratic penalty (ridge) with weight α. This coefficient is a hyper-parameter of the model that was optimized:

```
In [16]: parameters = range(580, 620, 4)
         model_selection = ['poly', parameters]
         bestParam, bestRMSE, gridResults = gridSearch(series, model_selection, steps_ahead, fit_range, fastR
```

```
Method: poly - Param: 580 - RMSE: 43.5808
Method: poly - Param: 584 - RMSE: 43.5778
Method: poly - Param: 588 - RMSE: 43.5754
Method: poly - Param: 592 - RMSE: 43.5735
Method: poly - Param: 596 - RMSE: 43.5720
Method: poly - Param: 600 - RMSE: 43.5709
Method: poly - Param: 604 - RMSE: 43.5703
Method: poly - Param: 608 - RMSE: 43.5701
Method: poly - Param: 612 - RMSE: 43.5704
Method: poly - Param: 616 - RMSE: 43.5710
Best parameters for method poly
Param: 608 - RMSE: 43.5701
```

Even though the RMSE (43.57) is better than the ARIMA model, it is not a significant improvement. The lowest RMSE corresponds to $\alpha = 608$.

3 Support Vector Machine (SVM) regression

The SVM model used employed a polynomial of order 3 and as with the polynomial model, fitted the asset prices M days in the past. A prediction N days ahead used the fitted SVM model. The model has hyper-parameters $\mathbf{C}$ and ϵ representing the strength of the regularization and width of the margin, respectively. The optimization of the hyper-parameters is done by the following commands:

```
In [14]: parameters = [[10**np.random.uniform(0,2.5), np.random.uniform(0.01,0.1)] for i in range(0,100)]
         model_selection = ['SVM', parameters]
         bestParam, bestRMSE, gridResults = gridSearch(series, model_selection, steps_ahead, fit_range,
                                                       fastRMSE = True, numInstances=200)
         grid = pd.DataFrame(gridResults.Param.values.tolist(), columns=['C','epsilon'])
         grid['RMSE'] = gridResults['RMSE']
```

```
Method: SVM - Param: [1.440560504635875, 0.06650627453623208] - RMSE: 498.7121
Method: SVM - Param: [3.7891617208520905, 0.08967236597383675] - RMSE: 406.4787
Method: SVM - Param: [5.13719408098369, 0.020676711593975973] - RMSE: 341.0009
Method: SVM - Param: [253.3555614939664, 0.06156086252046523] - RMSE: 205.7539
Method: SVM - Param: [65.31810170418956, 0.09350412721023897] - RMSE: 155.1689
Method: SVM - Param: [49.11738512465951, 0.06511952774299071] - RMSE: 164.7664
Method: SVM - Param: [31.400209705373214, 0.07109275197796715] - RMSE: 154.8365
```

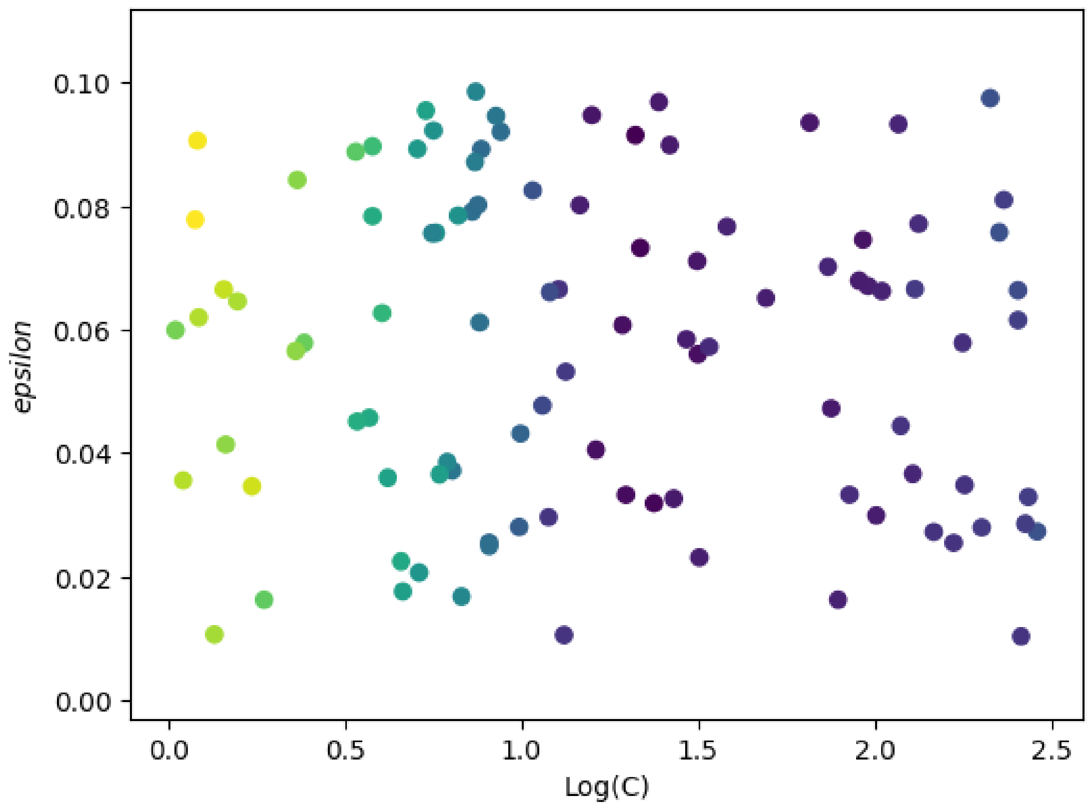
```
Method: SVM - Param: [23.692833770321467, 0.031896392115284805] - RMSE: 135.3943
Method: SVM - Param: [92.38949706316791, 0.07455168370009214] - RMSE: 150.1374
Method: SVM - Param: [3.7911416095663797, 0.07836212927670592] - RMSE: 377.9335
Method: SVM - Param: [1.2241768130023776, 0.06197801248392666] - RMSE: 488.7911
Method: SVM - Param: [5.367269566770042, 0.09546915952170885] - RMSE: 363.0664
Method: SVM - Param: [132.722601796458, 0.07714517968213323] - RMSE: 182.0614
Method: SVM - Param: [7.698690322498619, 0.08923748397497866] - RMSE: 269.8163
Method: SVM - Param: [1.4579994763700326, 0.041406045814192174] - RMSE: 466.4005
Method: SVM - Param: [4.038241209879603, 0.06268587355346504] - RMSE: 394.8992
Method: SVM - Param: [1.7292706584741064, 0.03464498625477038] - RMSE: 506.0298
Method: SVM - Param: [75.09960855627159, 0.04724842531462188] - RMSE: 159.6150
Method: SVM - Param: [6.767903090352112, 0.016800043800910346] - RMSE: 317.4601
Method: SVM - Param: [95.54724270041372, 0.0670569319871673] - RMSE: 163.6994
Method: SVM - Param: [104.48685326075915, 0.06622772753965016] - RMSE: 168.9789
Method: SVM - Param: [8.103983159583796, 0.025523148997074786] - RMSE: 266.9965
Method: SVM - Param: [7.387317937313566, 0.0871404202050945] - RMSE: 300.1637
Method: SVM - Param: [5.082577786738362, 0.089263100920925] - RMSE: 348.1087
Method: SVM - Param: [13.344347745036071, 0.05319228878453452] - RMSE: 197.6512
Method: SVM - Param: [6.375685256465348, 0.037187990154151945] - RMSE: 288.9277
Method: SVM - Param: [14.614214651500044, 0.08014499390568278] - RMSE: 163.8385
Method: SVM - Param: [287.0148431249829, 0.027324638944881012] - RMSE: 232.8770
Method: SVM - Param: [166.74770859758095, 0.025487930691606084] - RMSE: 189.1154
Method: SVM - Param: [2.3237550958551036, 0.084227046738877] - RMSE: 462.8802
Method: SVM - Param: [100.69407356451516, 0.02993507558403973] - RMSE: 161.2580
Method: SVM - Param: [270.94974948983025, 0.032918055012274676] - RMSE: 203.7914
Method: SVM - Param: [3.4011357661665182, 0.08879829809883184] - RMSE: 431.8483
Method: SVM - Param: [90.05132542494462, 0.06792943896696103] - RMSE: 161.6371
Method: SVM - Param: [19.742049637902316, 0.03326188540291181] - RMSE: 142.1230
Method: SVM - Param: [5.63738972956413, 0.09223012923881065] - RMSE: 333.7075
Method: SVM - Param: [31.544931549108924, 0.05603387171251998] - RMSE: 145.4496
Method: SVM - Param: [265.79151984530705, 0.028591470048510835] - RMSE: 203.1279
Method: SVM - Param: [29.272271190718637, 0.05844207261788697] - RMSE: 162.8640
Method: SVM - Param: [7.25632661015589, 0.07913741579844723] - RMSE: 274.6062
Method: SVM - Param: [118.0971133303757, 0.04441905106252618] - RMSE: 190.5709
Method: SVM - Param: [6.165574210007886, 0.038557623447957184] - RMSE: 320.6925
Method: SVM - Param: [1.8717010128775144, 0.016279600346457916] - RMSE: 436.8763
Method: SVM - Param: [258.58012144610916, 0.010353531044158232] - RMSE: 190.9843
Method: SVM - Param: [24.4677571195751, 0.09685863718013477] - RMSE: 160.2265
Method: SVM - Param: [5.867448342275172, 0.03657165765537754] - RMSE: 356.1027
Method: SVM - Param: [127.77463290757683, 0.036649589377813] - RMSE: 173.4681
Method: SVM - Param: [3.7120741441610523, 0.04573884750145543] - RMSE: 373.7717
Method: SVM - Param: [1.5746832951446257, 0.06458562929566392] - RMSE: 484.1671
Method: SVM - Param: [12.7490073287852, 0.06653289136020273] - RMSE: 190.0246
Method: SVM - Param: [5.737441600367675, 0.0756707175941256] - RMSE: 336.2927
Method: SVM - Param: [7.429012793029822, 0.0985405410035309] - RMSE: 314.5383
Method: SVM - Param: [13.196950863551795, 0.010573512449683454] - RMSE: 190.8353
Method: SVM - Param: [84.57171508042126, 0.03328397253944042] - RMSE: 181.0696
```

```
Method: SVM - Param: [4.620074477005239, 0.017617821052066277] - RMSE: 357.2563
Method: SVM - Param: [19.30860893900931, 0.06074356710650062] - RMSE: 147.0681
Method: SVM - Param: [211.6130648991773, 0.09747112060319887] - RMSE: 230.6867
Method: SVM - Param: [12.02876940086676, 0.06606576916002327] - RMSE: 224.2495
Method: SVM - Param: [129.5312233620595, 0.0665617887309087] - RMSE: 200.6294
Method: SVM - Param: [146.35306323454432, 0.027253412298480266] - RMSE: 190.1188
Method: SVM - Param: [2.431870500157728, 0.05786384107210471] - RMSE: 442.0046
Method: SVM - Param: [253.30467224329638, 0.06635768412184838] - RMSE: 223.2638
Method: SVM - Param: [1.2137857608567848, 0.09059980745109569] - RMSE: 529.0475
Method: SVM - Param: [16.236770296617756, 0.040552403294440986] - RMSE: 146.5961
Method: SVM - Param: [8.742049593229751, 0.09203235718517418] - RMSE: 273.3203
Method: SVM - Param: [8.469045646542268, 0.09460184373341014] - RMSE: 288.8295
Method: SVM - Param: [231.13121039967203, 0.08100316014809467] - RMSE: 204.4288
Method: SVM - Param: [1.0518966784035153, 0.059934895571975184] - RMSE: 450.6666
Method: SVM - Param: [73.55216849463388, 0.07017100671923775] - RMSE: 173.8353
Method: SVM - Param: [6.617522627481445, 0.07846985445256373] - RMSE: 337.1949
Method: SVM - Param: [5.593555684792854, 0.07561708363130884] - RMSE: 309.8378
Method: SVM - Param: [8.108132347852683, 0.02497130552979833] - RMSE: 276.5952
Method: SVM - Param: [4.5657111003050606, 0.02249211186177369] - RMSE: 373.6290
Method: SVM - Param: [1.3540605266527985, 0.010664082088107846] - RMSE: 480.3612
Method: SVM - Param: [9.938600811946412, 0.043176943270080814] - RMSE: 259.8311
Method: SVM - Param: [3.4331427176064158, 0.04513698725127714] - RMSE: 380.5125
Method: SVM - Param: [9.848236598217818, 0.028067355022279965] - RMSE: 249.0468
Method: SVM - Param: [7.538004768218792, 0.08019112232118908] - RMSE: 272.0432
Method: SVM - Param: [10.75549486390813, 0.08254387903301973] - RMSE: 234.1242
Method: SVM - Param: [38.08497854684913, 0.0766919994893021] - RMSE: 168.7422
Method: SVM - Param: [11.928744280247637, 0.029645590051260606] - RMSE: 193.3156
Method: SVM - Param: [176.7331197202595, 0.057870770561052] - RMSE: 186.9159
Method: SVM - Param: [179.07066714323616, 0.03484417709435937] - RMSE: 190.2520
Method: SVM - Param: [116.34662444480378, 0.09327238696090764] - RMSE: 172.0121
Method: SVM - Param: [200.05147096387742, 0.02798865447771695] - RMSE: 200.9005
Method: SVM - Param: [1.105417018681633, 0.03559500615309184] - RMSE: 489.3979
Method: SVM - Param: [15.805506472569196, 0.09472121202954999] - RMSE: 153.3738
Method: SVM - Param: [2.299927633764372, 0.056541594108725224] - RMSE: 466.7014
Method: SVM - Param: [20.998963987067544, 0.09148223927687062] - RMSE: 130.4959
Method: SVM - Param: [26.25612136249971, 0.08986131000921496] - RMSE: 159.2569
Method: SVM - Param: [31.837854872582994, 0.02310755369880901] - RMSE: 164.1384
Method: SVM - Param: [224.3653940152704, 0.07575107184890939] - RMSE: 230.7405
Method: SVM - Param: [7.6224783376264345, 0.06116423802853453] - RMSE: 279.8653
Method: SVM - Param: [1.1957698242046422, 0.07780879212838311] - RMSE: 534.8967
Method: SVM - Param: [78.53352795763877, 0.01628812435439324] - RMSE: 167.2795
Method: SVM - Param: [21.665924192694835, 0.07325091172502383] - RMSE: 135.0139
Method: SVM - Param: [26.948859189468436, 0.03264641807837956] - RMSE: 158.8602
Method: SVM - Param: [33.96017261950198, 0.05722787112358006] - RMSE: 178.0848
Method: SVM - Param: [4.186978108858816, 0.03604261978416694] - RMSE: 364.0887
Method: SVM - Param: [11.460730762818073, 0.04773619286612645] - RMSE: 220.4383
Best parameters for method SVM
```

```
Param: [20.998963987067544, 0.09148223927687062] - RMSE: 130.4959
```

The RMSE is high for all hyper-parameter combinations used. Furthermore, there is not a big dependence on **C** and ϵ:

```
In [15]: plt.scatter(x = np.log10(grid['C']), y = grid['epsilon'], c= grid['RMSE'])
         plt.xlabel('Log(C)')
         plt.ylabel('$epsilon$')
         plt.show()
```



Since the RMSE is significantly larger than for the other optimized models, this model was abandoned.

4 Recurrent Neural Network (RNN)

A RNN model was developed using a single layer due to the risk of overfitting. The hyper-parameters of the model, learning rate, sequence length (M number of days), number of neurons and activation function, were optimized using sklearn RandomizedSearchCV. Dropout was initially used as a regularizer but the predictions were significantly worse and it was not used in the optimized model.

```
In [344]: from sklearn.model_selection import RandomizedSearchCV

          def leaky_relu(alpha=0.01):
              def parametrized_leaky_relu(z, name=None):
                  return tf.maximum(alpha * z, z, name=name)
```

```python
            return parametrized_leaky_relu

        param_distribs = {
            "n_neurons": [10, 50, 100, 150],
            "fit_range": [50, 100, 200],
            "learning_rate": [0.001, 0.01],
            "activation": [tf.nn.relu, tf.nn.elu, leaky_relu(alpha=0.1)]
        }

        rnd_search = RandomizedSearchCV(RNNRegression(), param_distribs, n_iter=72,
                                        random_state=42, verbose=2, cv=3, n_jobs=-1, iid=False)
        rnd_search.fit(X=series, max_iterations=1500, keep_prob=1)
```

```
Fitting 3 folds for each of 72 candidates, totalling 216 fits


[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  25 tasks      | elapsed: 74.6min
[Parallel(n_jobs=-1)]: Done 146 tasks      | elapsed: 637.8min
[Parallel(n_jobs=-1)]: Done 216 out of 216 | elapsed: 1150.1min finished

Iteration 0 - model RMSE:628.085 - best RMSE:628.085
Iteration 20 - model RMSE:136.054 - best RMSE:136.054
Iteration 40 - model RMSE:88.510 - best RMSE:88.510
Iteration 60 - model RMSE:73.304 - best RMSE:73.304
Iteration 80 - model RMSE:68.908 - best RMSE:68.908
Iteration 100 - model RMSE:65.507 - best RMSE:65.507
Iteration 120 - model RMSE:62.478 - best RMSE:62.478
Iteration 140 - model RMSE:59.815 - best RMSE:59.815
Iteration 160 - model RMSE:57.537 - best RMSE:57.537
Iteration 180 - model RMSE:55.590 - best RMSE:55.590
Iteration 200 - model RMSE:53.988 - best RMSE:53.988
Iteration 220 - model RMSE:52.601 - best RMSE:52.601
Iteration 240 - model RMSE:51.321 - best RMSE:51.321
Iteration 260 - model RMSE:50.135 - best RMSE:50.135
Iteration 280 - model RMSE:49.039 - best RMSE:49.039
Iteration 300 - model RMSE:48.029 - best RMSE:48.029
Iteration 320 - model RMSE:47.104 - best RMSE:47.104
Iteration 340 - model RMSE:46.260 - best RMSE:46.260
Iteration 360 - model RMSE:45.493 - best RMSE:45.493
Iteration 380 - model RMSE:44.799 - best RMSE:44.799
Iteration 400 - model RMSE:44.175 - best RMSE:44.175
Iteration 420 - model RMSE:43.617 - best RMSE:43.617
Iteration 440 - model RMSE:43.119 - best RMSE:43.119
Iteration 460 - model RMSE:41.831 - best RMSE:41.831
Iteration 480 - model RMSE:41.357 - best RMSE:41.357
Iteration 500 - model RMSE:41.035 - best RMSE:41.035
Iteration 520 - model RMSE:40.763 - best RMSE:40.763
```

```
Iteration 540 - model RMSE:40.530 - best RMSE:40.530
Iteration 560 - model RMSE:40.333 - best RMSE:40.333
Iteration 580 - model RMSE:40.166 - best RMSE:40.166
Iteration 600 - model RMSE:40.027 - best RMSE:40.027
Iteration 620 - model RMSE:39.912 - best RMSE:39.912
Iteration 640 - model RMSE:39.816 - best RMSE:39.816
Iteration 660 - model RMSE:39.739 - best RMSE:39.739
Iteration 680 - model RMSE:39.676 - best RMSE:39.676
Iteration 700 - model RMSE:39.625 - best RMSE:39.625
Iteration 720 - model RMSE:39.584 - best RMSE:39.584
Iteration 740 - model RMSE:39.553 - best RMSE:39.553
Iteration 760 - model RMSE:39.528 - best RMSE:39.528
Iteration 780 - model RMSE:39.508 - best RMSE:39.508
Iteration 800 - model RMSE:39.493 - best RMSE:39.493
Iteration 820 - model RMSE:39.481 - best RMSE:39.481
Iteration 840 - model RMSE:39.471 - best RMSE:39.471
Iteration 860 - model RMSE:39.464 - best RMSE:39.464
Iteration 880 - model RMSE:39.459 - best RMSE:39.459
Iteration 900 - model RMSE:39.455 - best RMSE:39.455
Iteration 920 - model RMSE:39.451 - best RMSE:39.451
Iteration 940 - model RMSE:39.448 - best RMSE:39.448
Iteration 960 - model RMSE:39.446 - best RMSE:39.446
Iteration 980 - model RMSE:39.444 - best RMSE:39.444
Iteration 1000 - model RMSE:39.443 - best RMSE:39.443
Iteration 1020 - model RMSE:39.441 - best RMSE:39.441
Iteration 1040 - model RMSE:39.440 - best RMSE:39.440
Iteration 1060 - model RMSE:39.439 - best RMSE:39.439
Iteration 1080 - model RMSE:39.437 - best RMSE:39.437
Iteration 1100 - model RMSE:39.436 - best RMSE:39.436
Iteration 1120 - model RMSE:39.435 - best RMSE:39.435
Iteration 1140 - model RMSE:39.433 - best RMSE:39.433
Iteration 1160 - model RMSE:39.432 - best RMSE:39.432
Iteration 1180 - model RMSE:39.431 - best RMSE:39.431
Iteration 1200 - model RMSE:39.430 - best RMSE:39.430
Iteration 1220 - model RMSE:39.428 - best RMSE:39.428
Iteration 1240 - model RMSE:39.427 - best RMSE:39.427
Iteration 1260 - model RMSE:39.425 - best RMSE:39.425
Iteration 1280 - model RMSE:39.424 - best RMSE:39.424
Iteration 1300 - model RMSE:39.422 - best RMSE:39.422
Iteration 1320 - model RMSE:39.421 - best RMSE:39.421
Iteration 1340 - model RMSE:39.420 - best RMSE:39.420
Iteration 1360 - model RMSE:39.418 - best RMSE:39.418
Iteration 1380 - model RMSE:39.417 - best RMSE:39.417
Iteration 1400 - model RMSE:39.415 - best RMSE:39.415
Iteration 1420 - model RMSE:39.414 - best RMSE:39.414
Iteration 1440 - model RMSE:39.412 - best RMSE:39.412
Iteration 1460 - model RMSE:39.411 - best RMSE:39.411
```

```
Iteration 1480 - model RMSE:39.409 - best RMSE:39.409
```

```
Out[344]: RandomizedSearchCV(cv=3, error_score='raise-deprecating',
                    estimator=RNNRegression(activation=<function elu at 0x1303a5e18>, fit_range=50,
                  learning_rate=0.001, n_neurons=120,
                  optimizer_class=<class 'tensorflow.python.training.adam.AdamOptimizer'>,
                  steps_ahead=30),
                    fit_params=None, iid=False, n_iter=72, n_jobs=-1,
                    param_distributions={'n_neurons': [10, 50, 100, 150], 'fit_range': [50, 100, 200]
                    pre_dispatch='2*n_jobs', random_state=42, refit=True,
                    return_train_score='warn', scoring=None, verbose=2)
```

```
In [351]: rnd_search.best_estimator_.save("../models/RNNmodel-best")
          rnd_search.best_params_
```

```
Out[351]: {'n_neurons': 10,
           'learning_rate': 0.01,
           'fit_range': 50,
           'activation': <function tensorflow.python.ops.gen_nn_ops.relu(features, name=None)>}
```

5 Long Short-Term Memory Network (LSTM)

A LSTM model was developed using a single layer due to the risk of overfitting. The hyper-parameters of the model, learning rate, sequence length (M number of days), number of neurons and activation function, were optimized using scikit-learn RandomizedSearchCV method:

```
In [433]: from sklearn.model_selection import RandomizedSearchCV
          param_distribs = {
              "n_neurons": [10, 50, 120],
          #    "fit_range": [50, 100, 200],
              "learning_rate": [0.001, 0.01, 0.05]
          }

          rnd_search = RandomizedSearchCV(LSTMRegression(), param_distribs, n_iter=12,
                                          random_state=42, verbose=2, cv=3, n_jobs=-1, iid=False,
                                          return_train_score=True)
          rnd_search.fit(X=series, max_iterations=2500)
```

```
Fitting 3 folds for each of 9 candidates, totalling 27 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  27 out of  27 | elapsed: 181.7min finished

Iteration 0 - model RMSE:528.040 - best RMSE:528.040
Iteration 20 - model RMSE:526.442 - best RMSE:526.442
Iteration 40 - model RMSE:523.061 - best RMSE:523.061
```

```
Iteration 60 - model RMSE:518.868 - best RMSE:518.868
Iteration 80 - model RMSE:513.478 - best RMSE:513.478
Iteration 100 - model RMSE:509.775 - best RMSE:509.775
Iteration 120 - model RMSE:506.931 - best RMSE:506.931
Iteration 140 - model RMSE:504.446 - best RMSE:504.446
Iteration 160 - model RMSE:501.872 - best RMSE:501.872
Iteration 180 - model RMSE:499.267 - best RMSE:499.267
Iteration 200 - model RMSE:496.902 - best RMSE:496.902
Iteration 220 - model RMSE:494.557 - best RMSE:494.557
Iteration 240 - model RMSE:492.403 - best RMSE:492.403
Iteration 260 - model RMSE:490.200 - best RMSE:490.200
Iteration 280 - model RMSE:487.977 - best RMSE:487.977
Iteration 300 - model RMSE:485.892 - best RMSE:485.892
Iteration 320 - model RMSE:483.855 - best RMSE:483.855
Iteration 340 - model RMSE:481.846 - best RMSE:481.846
Iteration 360 - model RMSE:479.739 - best RMSE:479.739
Iteration 380 - model RMSE:477.743 - best RMSE:477.743
Iteration 400 - model RMSE:475.782 - best RMSE:475.782
Iteration 420 - model RMSE:473.835 - best RMSE:473.835
Iteration 440 - model RMSE:471.817 - best RMSE:471.817
Iteration 460 - model RMSE:469.875 - best RMSE:469.875
Iteration 480 - model RMSE:467.963 - best RMSE:467.963
Iteration 500 - model RMSE:466.075 - best RMSE:466.075
Iteration 520 - model RMSE:464.209 - best RMSE:464.209
Iteration 540 - model RMSE:462.361 - best RMSE:462.361
Iteration 560 - model RMSE:460.531 - best RMSE:460.531
Iteration 580 - model RMSE:458.717 - best RMSE:458.717
Iteration 600 - model RMSE:456.919 - best RMSE:456.919
Iteration 620 - model RMSE:455.135 - best RMSE:455.135
Iteration 640 - model RMSE:453.366 - best RMSE:453.366
Iteration 660 - model RMSE:451.611 - best RMSE:451.611
Iteration 680 - model RMSE:449.868 - best RMSE:449.868
Iteration 700 - model RMSE:448.138 - best RMSE:448.138
Iteration 720 - model RMSE:446.422 - best RMSE:446.422
Iteration 740 - model RMSE:444.717 - best RMSE:444.717
Iteration 760 - model RMSE:443.025 - best RMSE:443.025
Iteration 780 - model RMSE:441.344 - best RMSE:441.344
Iteration 800 - model RMSE:439.675 - best RMSE:439.675
Iteration 820 - model RMSE:438.017 - best RMSE:438.017
Iteration 840 - model RMSE:436.369 - best RMSE:436.369
Iteration 860 - model RMSE:434.731 - best RMSE:434.731
Iteration 880 - model RMSE:433.094 - best RMSE:433.094
Iteration 900 - model RMSE:431.469 - best RMSE:431.469
Iteration 920 - model RMSE:429.853 - best RMSE:429.853
Iteration 940 - model RMSE:428.250 - best RMSE:428.250
Iteration 960 - model RMSE:426.587 - best RMSE:426.587
Iteration 980 - model RMSE:424.932 - best RMSE:424.932
```

```
Iteration 1000 - model RMSE:423.302 - best RMSE:423.302
Iteration 1020 - model RMSE:421.692 - best RMSE:421.692
Iteration 1040 - model RMSE:420.099 - best RMSE:420.099
Iteration 1060 - model RMSE:418.519 - best RMSE:418.519
Iteration 1080 - model RMSE:416.953 - best RMSE:416.953
Iteration 1100 - model RMSE:415.400 - best RMSE:415.400
Iteration 1120 - model RMSE:413.858 - best RMSE:413.858
Iteration 1140 - model RMSE:412.327 - best RMSE:412.327
Iteration 1160 - model RMSE:410.806 - best RMSE:410.806
Iteration 1180 - model RMSE:409.297 - best RMSE:409.297
Iteration 1200 - model RMSE:407.797 - best RMSE:407.797
Iteration 1220 - model RMSE:406.308 - best RMSE:406.308
Iteration 1240 - model RMSE:404.829 - best RMSE:404.829
Iteration 1260 - model RMSE:403.358 - best RMSE:403.358
Iteration 1280 - model RMSE:401.898 - best RMSE:401.898
Iteration 1300 - model RMSE:400.447 - best RMSE:400.447
Iteration 1320 - model RMSE:399.005 - best RMSE:399.005
Iteration 1340 - model RMSE:397.572 - best RMSE:397.572
Iteration 1360 - model RMSE:396.148 - best RMSE:396.148
Iteration 1380 - model RMSE:394.733 - best RMSE:394.733
Iteration 1400 - model RMSE:393.327 - best RMSE:393.327
Iteration 1420 - model RMSE:391.930 - best RMSE:391.930
Iteration 1440 - model RMSE:390.544 - best RMSE:390.544
Iteration 1460 - model RMSE:389.164 - best RMSE:389.164
Iteration 1480 - model RMSE:387.792 - best RMSE:387.792
Iteration 1500 - model RMSE:386.428 - best RMSE:386.428
Iteration 1520 - model RMSE:385.073 - best RMSE:385.073
Iteration 1540 - model RMSE:383.725 - best RMSE:383.725
Iteration 1560 - model RMSE:382.387 - best RMSE:382.387
Iteration 1580 - model RMSE:381.057 - best RMSE:381.057
Iteration 1600 - model RMSE:379.733 - best RMSE:379.733
Iteration 1620 - model RMSE:378.418 - best RMSE:378.418
Iteration 1640 - model RMSE:377.108 - best RMSE:377.108
Iteration 1660 - model RMSE:375.810 - best RMSE:375.810
Iteration 1680 - model RMSE:374.514 - best RMSE:374.514
Iteration 1700 - model RMSE:373.227 - best RMSE:373.227
Iteration 1720 - model RMSE:371.947 - best RMSE:371.947
Iteration 1740 - model RMSE:370.680 - best RMSE:370.680
Iteration 1760 - model RMSE:369.411 - best RMSE:369.411
Iteration 1780 - model RMSE:368.150 - best RMSE:368.150
Iteration 1800 - model RMSE:366.903 - best RMSE:366.903
Iteration 1820 - model RMSE:365.653 - best RMSE:365.653
Iteration 1840 - model RMSE:364.414 - best RMSE:364.414
Iteration 1860 - model RMSE:363.182 - best RMSE:363.182
Iteration 1880 - model RMSE:361.968 - best RMSE:361.968
Iteration 1900 - model RMSE:360.741 - best RMSE:360.741
Iteration 1920 - model RMSE:359.526 - best RMSE:359.526
```

```
Iteration 1940 - model RMSE:358.315 - best RMSE:358.315
Iteration 1960 - model RMSE:357.115 - best RMSE:357.115
Iteration 1980 - model RMSE:355.926 - best RMSE:355.926
Iteration 2000 - model RMSE:354.738 - best RMSE:354.738
Iteration 2020 - model RMSE:353.553 - best RMSE:353.553
Iteration 2040 - model RMSE:352.377 - best RMSE:352.377
Iteration 2060 - model RMSE:351.203 - best RMSE:351.203
Iteration 2080 - model RMSE:350.048 - best RMSE:350.048
Iteration 2100 - model RMSE:348.883 - best RMSE:348.883
Iteration 2120 - model RMSE:347.730 - best RMSE:347.730
Iteration 2140 - model RMSE:346.584 - best RMSE:346.584
Iteration 2160 - model RMSE:345.437 - best RMSE:345.437
Iteration 2180 - model RMSE:344.304 - best RMSE:344.304
Iteration 2200 - model RMSE:343.163 - best RMSE:343.163
Iteration 2220 - model RMSE:342.037 - best RMSE:342.037
Iteration 2240 - model RMSE:340.914 - best RMSE:340.914
Iteration 2260 - model RMSE:339.800 - best RMSE:339.800
Iteration 2280 - model RMSE:338.683 - best RMSE:338.683
Iteration 2300 - model RMSE:337.571 - best RMSE:337.571
Iteration 2320 - model RMSE:336.472 - best RMSE:336.472
Iteration 2340 - model RMSE:335.370 - best RMSE:335.370
Iteration 2360 - model RMSE:334.285 - best RMSE:334.285
Iteration 2380 - model RMSE:333.185 - best RMSE:333.185
Iteration 2400 - model RMSE:332.098 - best RMSE:332.098
Iteration 2420 - model RMSE:331.024 - best RMSE:331.024
Iteration 2440 - model RMSE:329.930 - best RMSE:329.930
Iteration 2460 - model RMSE:328.850 - best RMSE:328.850
Iteration 2480 - model RMSE:327.809 - best RMSE:327.809
```

```
Out[433]: RandomizedSearchCV(cv=3, error_score='raise-deprecating',
                    estimator=LSTMRegression(fit_range=50, learning_rate=0.05, n_layers=1, n_neurons=
                  optimizer_class=<class 'tensorflow.python.training.adam.AdamOptimizer'>,
                  steps_ahead=30),
                    fit_params=None, iid=False, n_iter=12, n_jobs=-1,
                    param_distributions={'n_neurons': [10, 50, 120], 'learning_rate': [0.001, 0.01, 0
                    pre_dispatch='2*n_jobs', random_state=42, refit=True,
                    return_train_score='warn', scoring=None, verbose=2)
```

```
In [435]: rnd_search.cv_results_
```

```
Out[435]: {'mean_fit_time': array([ 504.14227907, 2736.57610424, 8424.39033262,  573.68816225,
                  2736.66937399, 7359.32454856,  629.21337549, 1680.82053836,
                  1897.33773494]),
           'std_fit_time': array([  1.00649638,  18.79009586,  10.9799733 ,   4.06966724,
                   42.96065895, 361.03666289,  33.90861493, 818.292878  ,
                  496.48952444]),
```

```
'mean_score_time': array([0.05381227, 0.07851076, 0.21243707, 0.02736831, 0.07229892,
       0.11959084, 0.05781452, 0.07188225, 0.18093975]),
'std_score_time': array([0.02981761, 0.01295513, 0.01627917, 0.00426053, 0.01436979,
       0.03431427, 0.02514526, 0.00982027, 0.08776949]),
'param_n_neurons': masked_array(data=[10, 50, 120, 10, 50, 120, 10, 50, 120],
             mask=[False, False, False, False, False, False, False, False,
                   False],
       fill_value='?',
            dtype=object),
'param_learning_rate': masked_array(data=[0.001, 0.001, 0.001, 0.01, 0.01, 0.01, 0.05, 0.0
                   0.05],
             mask=[False, False, False, False, False, False, False, False,
                   False],
       fill_value='?',
            dtype=object),
'params': [{'n_neurons': 10, 'learning_rate': 0.001},
 {'n_neurons': 50, 'learning_rate': 0.001},
 {'n_neurons': 120, 'learning_rate': 0.001},
 {'n_neurons': 10, 'learning_rate': 0.01},
 {'n_neurons': 50, 'learning_rate': 0.01},
 {'n_neurons': 120, 'learning_rate': 0.01},
 {'n_neurons': 10, 'learning_rate': 0.05},
 {'n_neurons': 50, 'learning_rate': 0.05},
 {'n_neurons': 120, 'learning_rate': 0.05}],
'split0_test_score': array([-202.97885132, -152.41622925,  -60.68752289,  -73.60482788,
       -282.89974976, -331.95944214, -186.01705933, -375.95498657,
       -375.7159729 ]),
'split1_test_score': array([-316.00466919, -211.96604919,  -86.98173523, -137.07229614,
        -37.6615181 ,  -57.5722084 , -165.86688232, -172.25794983,
       -203.9004364 ]),
'split2_test_score': array([-805.67321777, -702.15789795, -595.62860107, -643.64715576,
       -458.70626831, -381.3182373 , -569.3180542 , -568.75469971,
       -568.57629395]),
'mean_test_score': array([-441.55224609, -355.51339213, -247.76595306, -284.77475993,
       -259.75584539, -256.94996262, -307.06733195, -372.32254537,
       -382.73090108]),
'std_test_score': array([261.57442585, 246.31734994, 246.21015794, 255.08048044,
       172.66808362, 142.41415969, 185.62163739, 161.88949758,
       148.96090579]),
'rank_test_score': array([9, 6, 1, 4, 3, 2, 5, 7, 8], dtype=int32),
'split0_train_score': array([-643.52545166, -596.10131836, -483.09573364, -513.1852417 ,
       -284.38259888, -251.41241455, -356.6807251 , -293.58343506,
```

```
        -293.74215698]),
 'split1_train_score': array([-609.90649414, -523.77056885, -411.38232422, -462.7489624 ,
        -197.16665649, -101.04750061, -351.43505859, -317.26263428,
        -348.27575684]),
 'split2_train_score': array([-267.85256958, -172.74285889,  -94.00344086, -125.76622772,
         -44.8777504 ,  -30.70858955, -100.07324219, -100.13977814,
        -100.16195679]),
 'mean_train_score': array([-507.09483846, -430.87158203, -329.49383291, -367.23347727,
        -175.47566859, -127.7228349 , -269.39634196, -236.99528249,
        -247.3932902 ]),
 'std_train_score': array([169.72567276, 184.89774222, 169.07098476, 171.98019076,
         98.97311867,  92.05514538, 119.74866274,  97.25309909,
        106.46212952])}
```

Parameter combinations n_neurons, learning rate = [[10, 0.01],[120, 0.01]] result in the lowest RMSEs **but** there is a lot of variance in the cross-validation test results. The best RMSE result (37.66) was found for [120, 0.01]. The variance in the results is due to the slow convergence to the optimal solution (vanishing gradients) and to the fact that the number of iterations during the model optimization was not large enough. In total it takes about 10000 iteration to find optimal results.

6 Summary

After all models have been optimized they can be compared:

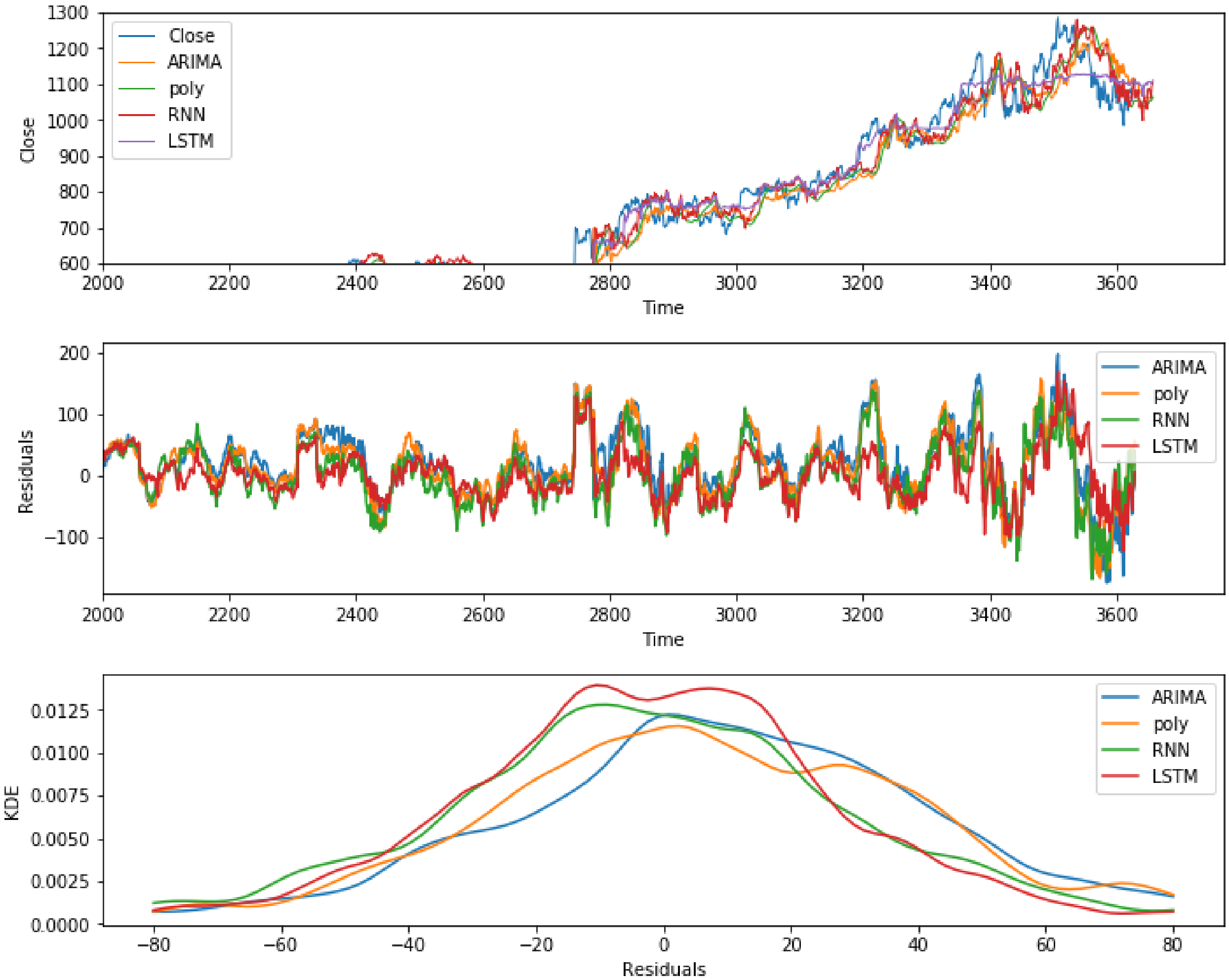
```
In [12]: np.warnings.filterwarnings("ignore") # specify to ignore warning messages
         sigma =[]

         model_sel = ['ARIMA', (1,0,0)]
         error, results = rollingEstimate(
             series, model_sel, steps_ahead = steps_ahead, fit_range = fit_range, verbose = False, fastRMSE =
         print('RMSE of {0} rolling estimate is {1:.2f}'.format(model_sel[0], error))
         resultsTotal = results.copy()
         resultsTotal.columns = ['Index', 'Close', 'ARIMA']
         sigma.append(error)

         model_sel = ['poly', [608]]
         error, results = rollingEstimate(
             series, model_sel, steps_ahead = steps_ahead, fit_range = fit_range, verbose = False, fastRMSE =
         print('RMSE of {0} rolling estimate is {1:.2f}'.format(model_sel[0], error))
         resultsTotal['poly'] = results['Pred']
         sigma.append(error)

         # Create RNN estimator
         rnn = RNNRegression()
         rnn.restore_model("../models/RNNmodel-best")
```

```
        print('RMSE of RNN rolling estimate is {0:.2f}'.format(-rnn.score(X=series)))
        resultsTotal['RNN'] = rnn.rolling_estimate(X=series)
        sigma.append(-rnn.score(X=series))

        # Create RNN estimator
        lstm = LSTMRegression()
        lstm.restore_model("../models/LSTMmodel-best")
        print('RMSE of LSTM rolling estimate is {0:.2f}'.format(-lstm.score(X=series)))
        resultsTotal['LSTM'] = lstm.rolling_estimate(X=series)
        sigma.append(-lstm.score(X=series))

        np.warnings.resetwarnings()
```

```
RMSE of ARIMA rolling estimate is 44.62
RMSE of poly rolling estimate is 43.57
INFO:tensorflow:Restoring parameters from ../models/RNNmodel-best
RMSE of RNN rolling estimate is 38.25
INFO:tensorflow:Restoring parameters from ../models/LSTMmodel-best
RMSE of LSTM rolling estimate is 36.90
```

The RNN and LSTM models show a smaller out-of-bag error, 38.25 and 36.90, respectively, than the ARIMA and polynomial estimates, 44.62 and 43.57, respectively. The predictions are shown in the following figure:

```
In [13]: plot_results(results=resultsTotal, steps_ahead=steps_ahead, fit_range=fit_range,
                      xlim_lo=2000, xlim_hi=3770, ylim_lo=600, ylim_hi=1300, xres_lo=-80, xres_hi=80,
                      lenSeries=len(series))
```



An average of the optimized predictions is also a prediction:

```
In [15]: sigma2Total = 1/(1/sigma[0]**2 + 1/sigma[1]**2 + 1/sigma[2]**2 + 1/sigma[3]**2)
         resultsTotal['average'] = (resultsTotal['ARIMA']/sigma[0]**2 + resultsTotal['poly']/sigma[1]**2 +
                                    resultsTotal['RNN']/sigma[2]**2    + resultsTotal['LSTM']/sigma[3]**2)*sig
         sigma.append( np.sqrt(mean_squared_error(
                 resultsTotal.loc[(fit_range+steps_ahead-1):(len(series)-1),'Close'],
                 resultsTotal.loc[(fit_range+steps_ahead-1):(len(series)-1),'average'])) )
         print(sigma[-1])
```

```
37.05654744616955
```

The average prediction is not lower than the RMSE from the LSTM model which is currently the best model. This indicates (and it can also be inferred from the residuals plot above) that the predictions of the models are not statistically independent. If they were independent, the expected error would be:

```
In [16]: print(np.sqrt(sigma2Total))
```

```
20.215214505196737
```

The predictions of all models over the whole time range are:

In [10]: resultsTotal

Out[10]:

	Index	Close	ARIMA	poly	RNN	LSTM	\
79	2004-12-10	85.910912	91.6124	82.791	104.558868	92.862923	
80	2004-12-13	85.310310	90.0358	84.7562	104.022583	92.654099	
81	2004-12-14	89.434433	98.1131	86.8874	107.306732	93.003403	
82	2004-12-15	89.979980	97.5751	88.8106	107.847649	92.913948	
83	2004-12-16	88.323326	90.8135	90.3968	106.028473	92.720551	
84	2004-12-17	90.130127	86.8142	91.518	103.751785	92.194839	
85	2004-12-20	92.602600	78.3221	91.7986	97.712532	90.266945	
86	2004-12-21	91.966965	80.0495	92.1601	97.105003	91.209511	
87	2004-12-22	93.243240	78.2377	92.2383	95.191109	90.399681	
88	2004-12-23	94.044044	77.9814	92.2249	93.519608	90.458672	
89	2004-12-27	96.051048	84.5837	92.8841	98.288261	92.323685	
90	2004-12-28	96.476479	84.2224	93.4157	97.891975	91.983597	
91	2004-12-29	96.546547	85.6342	93.9963	98.125481	92.298126	
92	2004-12-30	98.898895	79.3663	93.9064	94.625084	91.037933	
93	2004-12-31	96.491493	79.418	93.7746	94.746101	91.818558	
94	2005-01-03	101.456459	77.4077	93.3605	93.815857	91.463692	
95	2005-01-04	97.347351	78.1856	93.0328	94.176620	91.999474	
96	2005-01-05	96.851852	76.7729	92.4841	92.140343	91.330574	
97	2005-01-06	94.369370	77.6762	92.0677	92.073898	92.241959	
98	2005-01-07	97.022018	80.2269	91.9882	94.591454	93.141136	
99	2005-01-10	97.627625	81.9754	92.1112	96.480812	93.272194	
100	2005-01-11	96.866867	82.6738	92.2935	96.685104	93.468552	
101	2005-01-12	97.787788	83.1652	92.4864	97.692215	93.791626	
102	2005-01-13	97.762764	82.4911	92.5233	97.572166	93.439171	
103	2005-01-14	100.085083	82.3844	92.4976	98.090935	93.270500	
104	2005-01-18	102.052055	82.8275	92.5026	99.172737	94.177956	
105	2005-01-19	98.748749	81.5445	92.2659	97.627754	93.910172	
106	2005-01-20	97.057060	80.6695	91.7604	95.432320	93.113548	
107	2005-01-21	94.234238	80.7682	91.2387	94.398529	93.210701	
108	2005-01-24	90.450447	81.5638	90.9135	95.355446	93.845772	
...	...	...	...	...	...	...	
3629	2019-01-18	1107.300049	1107.7	1052.17	1067.974365	1102.104614	
3630	NaN	NaN	1108.77	1053.85	1079.568848	1105.669312	
3631	NaN	NaN	1102.98	1052.54	1060.887451	1100.605347	
3632	NaN	NaN	1102.01	1052.29	1065.748291	1101.349121	
3633	NaN	NaN	1101.09	1053.03	1078.847168	1102.691162	
3634	NaN	NaN	1099.61	1055.08	1084.283691	1104.772339	
3635	NaN	NaN	1096.86	1057.15	1081.482422	1104.754517	
3636	NaN	NaN	1087.37	1057.2	1060.380615	1101.366943	

3637	NaN	NaN	1082.19	1054.4	1034.465698	1096.937500
3638	NaN	NaN	1079.79	1053.47	1050.064453	1098.840698
3639	NaN	NaN	1076.57	1051.78	1048.050781	1098.120972
3640	NaN	NaN	1071.85	1048.99	1038.717896	1095.627808
3641	NaN	NaN	1068.17	1042.58	1009.707703	1084.182007
3642	NaN	NaN	1066.8	1035.92	998.195679	1066.202515
3643	NaN	NaN	1068.06	1036.25	1051.626953	1080.127930
3644	NaN	NaN	1068.8	1036.88	1062.812500	1096.354858
3645	NaN	NaN	1066.48	1037.38	1054.011963	1099.718994
3646	NaN	NaN	1063.64	1037.72	1048.674805	1099.878662
3647	NaN	NaN	1063.39	1038.66	1050.437866	1101.411987
3648	NaN	NaN	1061.1	1036.88	1030.057495	1096.894775
3649	NaN	NaN	1061.84	1040.47	1079.124756	1104.482056
3650	NaN	NaN	1059.53	1043.77	1084.414307	1105.142700
3651	NaN	NaN	1061.92	1047.07	1087.556641	1106.916138
3652	NaN	NaN	1060.67	1050.49	1085.480835	1106.051025
3653	NaN	NaN	1059	1053.23	1079.255127	1105.613770
3654	NaN	NaN	1059.57	1053.76	1068.843750	1103.257935
3655	NaN	NaN	1060.61	1053.19	1061.437134	1101.322388
3656	NaN	NaN	1061.51	1056.63	1092.361572	1107.036255
3657	NaN	NaN	1061.2	1060.14	1099.166504	1107.627075
3658	NaN	NaN	1061.59	1064.2	1104.355469	1109.635132

	average
79	93.7052
80	93.5921
81	96.7313
82	97.1591
83	95.5463
84	94.1728
85	90.2243
86	90.7699
87	89.6372
88	89.1324
89	92.5215
90	92.349
91	92.9235
92	90.2613
93	90.5117
94	89.6435
95	89.9942
96	88.8165

97	89.1673
98	90.6469
99	91.5994
100	91.898
101	92.4187
102	92.149
103	92.2158
104	92.8824
105	92.0561
106	90.9153
107	90.5636
108	91.1149
...	...
3629	1082.97
3630	1087.86
3631	1079.65
3632	1080.98
3633	1085.01
3634	1087.29
3635	1086.39
3636	1077.54
3637	1067.3
3638	1071.54
3639	1069.74
3640	1064.81
3641	1051.14
3642	1040.81
3643	1060.24
3644	1068.53
3645	1066.71
3646	1064.75
3647	1065.86
3648	1057.96
3649	1074.86
3650	1076.78
3651	1079.39
3652	1079.03
3653	1077.4
3654	1074.02
3655	1071.46
3656	1082.74
3657	1085.51

```
3658  1088.52

[3580 rows x 7 columns]
```

7 Future work

- Develop predictions as a function of N days ahead.
- Quantify uncertainties in model predictions.
- Extend input data to include correlated assets and asset data other than price.