```
!pip install pyramid
```

```
!pip install pyramid-arima
```

```
!pip install pmdarima
```

```
!pip install talib
```

```
!wget http://prdownloads.sourceforge.net/ta-lib/ta-lib-0.4.0-src.tar.gz
```

```
!tar -xzvf ta-lib-0.4.0-src.tar.gz
```

```
%cd ta-lib
```

```
!./configure --prefix=/usr
```

```
!make
```

```
!make install
```

```
!pip install Ta-Lib
```

```
import talib
```

```
import pandas as pd
import numpy as np
from scipy.stats import kurtosis
from pmdarima import auto_arima
import pmdarima as pm
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import MinMaxScaler
from keras.models import Sequential
from keras.layers import Dense, LSTM
from keras.callbacks import EarlyStopping
from talib import abstract
import json
import warnings
warnings.simplefilter("ignore")
```

```
def mean_absolute_percentage_error(actual, prediction):
    actual = pd.Series(actual)
    prediction = pd.Series(prediction)
    return 100 * np.mean(np.abs((actual - prediction))/actual)
```

```
def get_arima(data, train_len, test_len):
    # prepare train and test data
```

```
    data = data.tail(test_len + train_len).reset_index(drop=True)
    train = data.head(train_len).values.tolist()
    test = data.tail(test_len).values.tolist()

    # Initialize model
    model = auto_arima(train, max_p=3, max_q=3, seasonal=False, trace=True,
                       error_action='ignore', suppress_warnings=True)

    # Determine model parameters
    model.fit(train)
    order = model.get_params()['order']
    print('ARIMA order:', order, '\n')

    # Genereate predictions
    prediction = []
    for i in range(len(test)):
        model = pm.ARIMA(order=order)
        model.fit(train)
        print('working on', i+1, 'of', test_len, '-- ' + str(int(100 * (i + 1) / test_len)
        prediction.append(model.predict()[0])
        train.append(test[i])

    # Generate error data
    mse = mean_squared_error(test, prediction)
    rmse = mse ** 0.5
    mape = mean_absolute_percentage_error(pd.Series(test), pd.Series(prediction))
    return prediction, mse, rmse, mape



def get_lstm(data, train_len, test_len, lstm_len=4):
    # prepare train and test data
    data = data.tail(test_len + train_len).reset_index(drop=True)
    dataset = np.reshape(data.values, (len(data), 1))
    scaler = MinMaxScaler(feature_range=(0, 1))
    dataset_scaled = scaler.fit_transform(dataset)
    x_train = []
    y_train = []
    x_test = []

    for i in range(lstm_len, train_len):
        x_train.append(dataset_scaled[i - lstm_len:i, 0])
        y_train.append(dataset_scaled[i, 0])
    for i in range(train_len, len(dataset_scaled)):
        x_test.append(dataset_scaled[i - lstm_len:i, 0])

    x_train = np.array(x_train)
    y_train = np.array(y_train)
    x_train = np.reshape(x_train, (x_train.shape[0], x_train.shape[1], 1))
    x_test = np.array(x_test)
    x_test = np.reshape(x_test, (x_test.shape[0], x_test.shape[1], 1))

    # Set up & fit LSTM RNN
    model = Sequential()
    model.add(LSTM(units=lstm_len, return_sequences=True, input_shape=(x_train.shape[1], 1
```

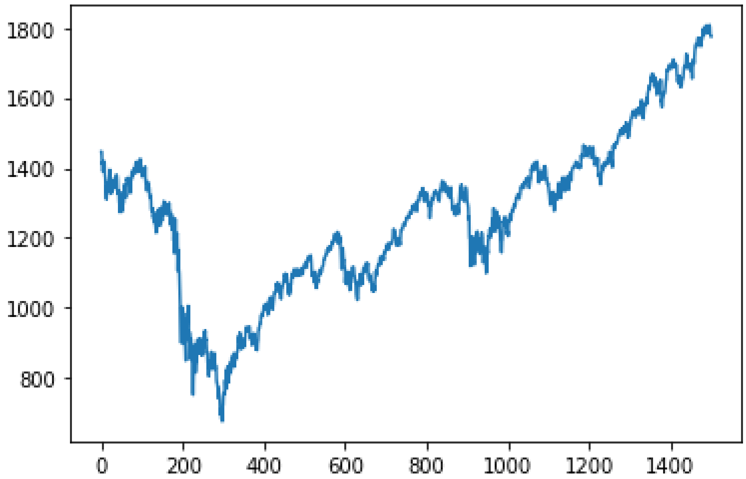
```
    model.add(LSTM(units=lstm_len, return_sequences=True, input_shape=(x_train.shape[1], 1
    model.add(LSTM(units=int(lstm_len/2)))
    model.add(Dense(1, activation='sigmoid'))

    model.compile(loss='mean_squared_error', optimizer='adam')
    early_stopping = EarlyStopping(monitor='loss', mode='min', verbose=1, patience=5)
    model.fit(x_train, y_train, epochs=500, batch_size=1, verbose=2, callbacks=[early_stop

    # Generate predictions
    prediction = model.predict(x_test)
    prediction = scaler.inverse_transform(prediction).tolist()

    output = []
    for i in range(len(prediction)):
        output.extend(prediction[i])
    prediction = output

    # Generate error data
    mse = mean_squared_error(data.tail(len(prediction)).values, prediction)
    rmse = mse ** 0.5
    mape = mean_absolute_percentage_error(data.tail(len(prediction)).reset_index(drop=True
    return prediction, mse, rmse, mape
```

```
%cd ..
!pwd
```

```
/content
/content
```

```
if __name__ == '__main__':
    # Load historical data
    # CSV should have columns: ['date', 'open', 'high', 'low', 'close', 'volume']
    data = pd.read_csv('data.csv', index_col=0, header=0, error_bad_lines=False).tail(1500
    print(data.shape)
    print(data.columns)
    data['close'].plot()
```

```
(1500, 5)
Index(['open', 'high', 'low', 'close', 'volume'], dtype='object')
```

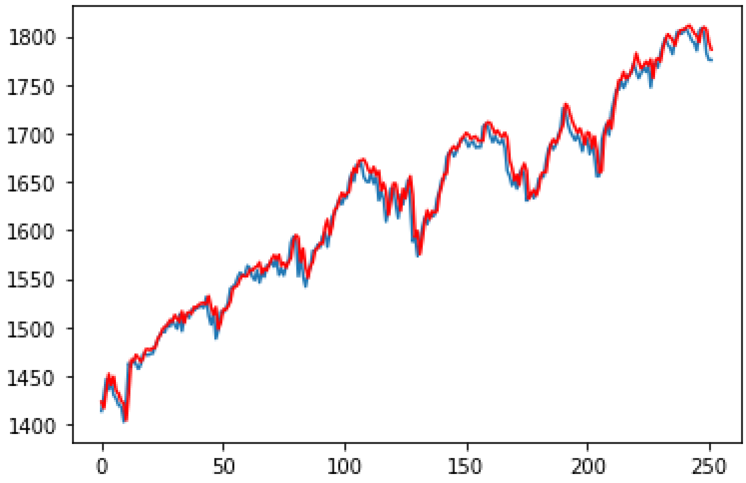



```
# Initialize moving averages from Ta-Lib, store functions in dictionary
  talib_moving_averages = ['SMA', 'EMA', 'WMA', 'DEMA', 'KAMA', 'MIDPOINT', 'MIDPRICE',
  functions = {}
  for ma in talib_moving_averages:
      functions[ma] = abstract.Function(ma)


# Determine kurtosis "K" values for MA period 4-99
  kurtosis_results = {'period': []}
  for i in range(4, 100):
      kurtosis_results['period'].append(i)
      for ma in talib_moving_averages:
          # Run moving average, remove last 252 days (used later for test data set), tri
          ma_output = functions[ma](data[:-252], i).tail(60)

          # Determine kurtosis "K" value
          k = kurtosis(ma_output, fisher=False)

          # add to dictionary
          if ma not in kurtosis_results.keys():
              kurtosis_results[ma] = []
          kurtosis_results[ma].append(k)

  kurtosis_results = pd.DataFrame(kurtosis_results)
  kurtosis_results.to_csv('kurtosis_results.csv')



# Determine period with K closest to 3 +/-5%
  optimized_period = {}
  for ma in talib_moving_averages:
      difference = np.abs(kurtosis_results[ma] - 3)
      df = pd.DataFrame({'difference': difference, 'period': kurtosis_results['period']}
      df = df.sort_values(by=['difference'], ascending=True).reset_index(drop=True)
      if df.at[0, 'difference'] < 3 * 0.05:
          optimized_period[ma] = df.at[0, 'period']
      else:
          print(ma + ' is not viable, best K greater or less than 3 +/-5%')
```

```
print('\nOptimized periods:', optimized_period)

simulation = {}
for ma in optimized_period:
    # Split data into low volatility and high volatility time series
    low_vol = functions[ma](data, optimized_period[ma])
    high_vol = data['close'] - low_vol

    # Generate ARIMA and LSTM predictions
    print('\nWorking on ' + ma + ' predictions')
    try:
        low_vol_prediction, low_vol_mse, low_vol_rmse, low_vol_mape = get_arima(low_vc
    except:
        print('ARIMA error, skipping to next MA type')
        continue

    high_vol_prediction, high_vol_mse, high_vol_rmse, high_vol_mape = get_lstm(high_vc

    final_prediction = pd.Series(low_vol_prediction) + pd.Series(high_vol_prediction)
    mse = mean_squared_error(final_prediction.values, data['close'].tail(252).values)
    rmse = mse ** 0.5
    mape = mean_absolute_percentage_error(data['close'].tail(252).reset_index(drop=Tru

    # Generate prediction accuracy
    actual = data['close'].tail(252).values
    result_1 = []
    result_2 = []
    for i in range(1, len(final_prediction)):
        # Compare prediction to previous close price
        if final_prediction[i] > actual[i-1] and actual[i] > actual[i-1]:
            result_1.append(1)
        elif final_prediction[i] < actual[i-1] and actual[i] < actual[i-1]:
            result_1.append(1)
        else:
            result_1.append(0)

        # Compare prediction to previous prediction
        if final_prediction[i] > final_prediction[i-1] and actual[i] > actual[i-1]:
            result_2.append(1)
        elif final_prediction[i] < final_prediction[i-1] and actual[i] < actual[i-1]:
            result_2.append(1)
        else:
            result_2.append(0)

    accuracy_1 = np.mean(result_1)
    accuracy_2 = np.mean(result_2)

    simulation[ma] = {'low_vol': {'prediction': low_vol_prediction, 'mse': low_vol_mse
                                  'rmse': low_vol_rmse, 'mape': low_vol_mape},
                      'high_vol': {'prediction': high_vol_prediction, 'mse': high_vol_
                                   'rmse': high_vol_rmse},
                      'final': {'prediction': final_prediction.values.tolist(), 'mse':
                                'rmse': rmse, 'mape': mape},
                      'accuracy': {'prediction vs close': accuracy_1, 'prediction vs p
```

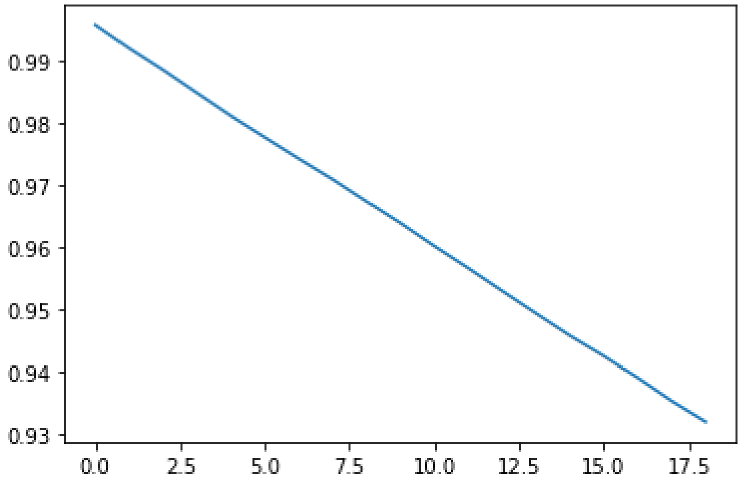
```
        # save simulation data here as checkpoint
        with open('simulation_data.json', 'w') as fp:
            json.dump(simulation, fp)

    for ma in simulation.keys():
        print('\n' + ma)
        print('Prediction vs Close:\t\t' + str(round(100*simulation[ma]['accuracy']['predi
              + '% Accuracy')
        print('Prediction vs Prediction:\t' + str(round(100*simulation[ma]['accuracy']['pr
              + '% Accuracy')
        print('MSE:\t', simulation[ma]['final']['mse'],
              '\nRMSE:\t', simulation[ma]['final']['rmse'],
              '\nMAPE:\t', simulation[ma]['final']['mape'])
```

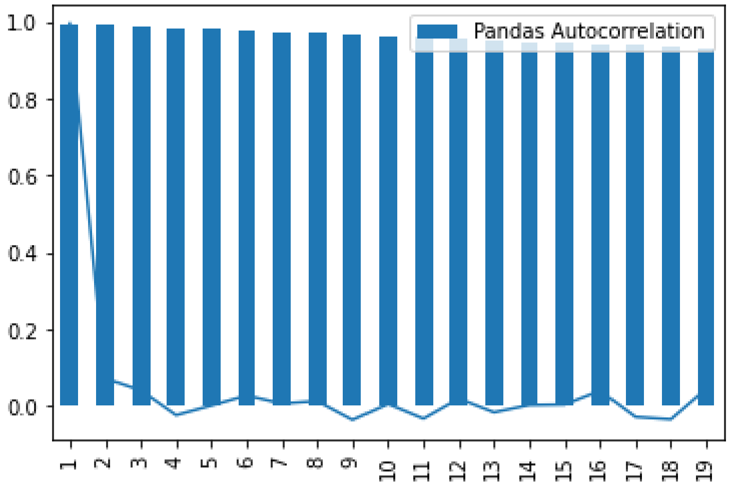
```
import matplotlib.pyplot as plt
Y = pd.DataFrame(actual)
pred = pd.DataFrame(final_prediction)
plt.plot(Y)
plt.plot(pred , color = 'r')
#p.plot()
plt.show()
```



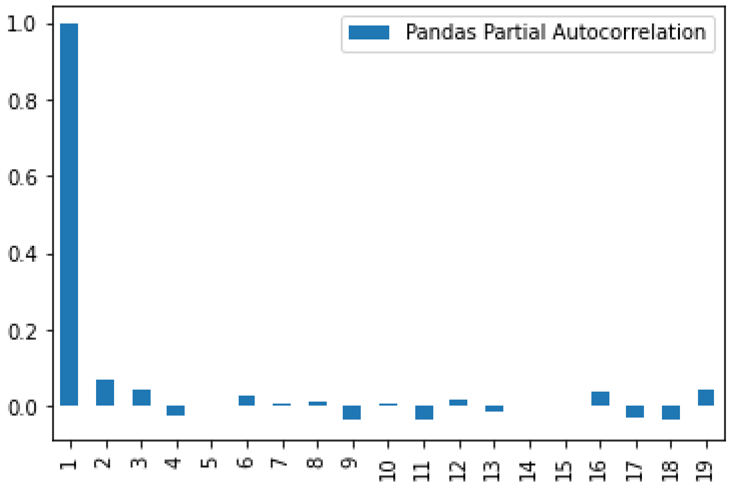

```
data = data['close'].tolist()
```

```
from statsmodels.tsa.stattools import acf, pacf
acf_1 = acf(data)[1:20]
plt.plot(acf_1)
test_df = pd.DataFrame([acf_1]).T
test_df.columns = ["Pandas Autocorrelation"]
test_df.index += 1
test_df.plot(kind='bar')
pacf_1 = pacf(data)[1:20]
plt.plot(pacf_1)
plt.show()
test_df = pd.DataFrame([pacf_1]).T
test_df.columns = ['Pandas Partial Autocorrelation']
test_df.index += 1
test_df.plot(kind='bar')
#from the figures we conclude that it is an AR process with a lag of 8-9
```






```
<matplotlib.axes._subplots.AxesSubplot at 0x7f0c4c05c0f0>
```




```
from sklearn.metrics import mean_absolute_error as mae
from sklearn.utils import check_array
#from sklearn.metrics import mean_absolute_percentage_error as mape
from sklearn.metrics import mean_squared_error
from math import sqrt
```

```
error = mean_squared_error(actual, final_prediction)
print('Test MSE: %.3f' % error)
```

```
Test MSE: 153.728
```

```
error = mae(actual,final_prediction)
print('Test MAE: %.3f' % error)
```

```
Test MAE: 9.287
```

```
def mean_absolute_percentage_error(y_true, y_pred):
    y_true, y_pred = np.array(y_true), np.array(y_pred)
    return np.mean(np.abs((y_true - y_pred) / y_true)) * 100


print('Test MAPE:%.3f'% mean_absolute_percentage_error(actual, final_prediction))
```

```
Test MAPE:0.574
```

```
rms = sqrt(mean_squared_error(actual, final_prediction))
print('Test RMS:%.3f'% rms)
```

```
Test RMS:12.399
```