Stock Market Prediction

Autoregressive Integrated Moving Average (ARIMA) is a statistical analysis model that uses time series data to either better understand the data set or to predict future trends. A statistical model is autoregressive if it predicts future values based on past values. In this project, we will use ARIMA to illustrate how to predict stock market price.

Data Loading

Basic packages Numpy, Pandas, Matplotlib and pmdarima are needed. Matplotlib is highly recommended in order to qualitatively validate the results - "A picture is worth a thousand words".

Relatively current version of the packages should work; however, pmdarima version is 2.0.3. In Jupyter Notebook, "%matplotlib inline" is needed to have plots show up in the cell.

In [ ]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline

import pmdarima as pm
print(f"Using pmdarima {pm.__version__}")
# Using pmdarima 2.0.3
```

In this exercise, I will use S&P500 historical prices:

In [56]:
```
df = pd.read_csv("sp500.csv",header=0,index_col='Date',parse_dates=['Date'])
df.head()
```

Out[56]:

Date	Open	High	Low	Close	Adj Close	Volu
2000-01-03	1469.250000	1478.000000	1438.359985	1455.219971	1455.219971	931800(
2000-01-04	1455.219971	1455.219971	1397.430054	1399.420044	1399.420044	1009000(
2000-01-05	1399.420044	1413.270020	1377.680054	1402.109985	1402.109985	1085500(
2000-01-06	1402.109985	1411.900024	1392.099976	1403.449951	1403.449951	1092300(
2000-01-07	1403.449951	1441.469971	1400.729980	1441.469971	1441.469971	1225200(

Data Engineering

As with all statistical and ML modeling, we separate the data into training and testing datasets, and sometimes cross-validation dataset, so we can evaluate model performance using desired metrics such as Mean Square Error (MSE) in regression problem and Accuracy in classification problem. However, time series models intrinsically introduce endogenous temporality, which means that the values at any given point y_t likely have some effect on some future value, y_{t+n}. Therefore, we have to split the data sequentially into two contiguous chunks.

In [ ]:
```
from pmdarima.model_selection import train_test_split

train_len = int(df.shape[0] * 0.8)
train_data, test_data = train_test_split(df, train_size=train_len)

y_train = train_data['Open'].values
y_test = test_data['Open'].values

print(f"{train_len} train samples")
print(f"{df.shape[0] - train_len} test samples")
# 4727 train samples
# 1182 test samples
```

Data Analysis

An ARIMA model has 3 core hyper-parameters, known as "order":

- p: The order of the auto-regressive (AR) model (i.e., the number of lag observations)
- d: The degree of differencing.

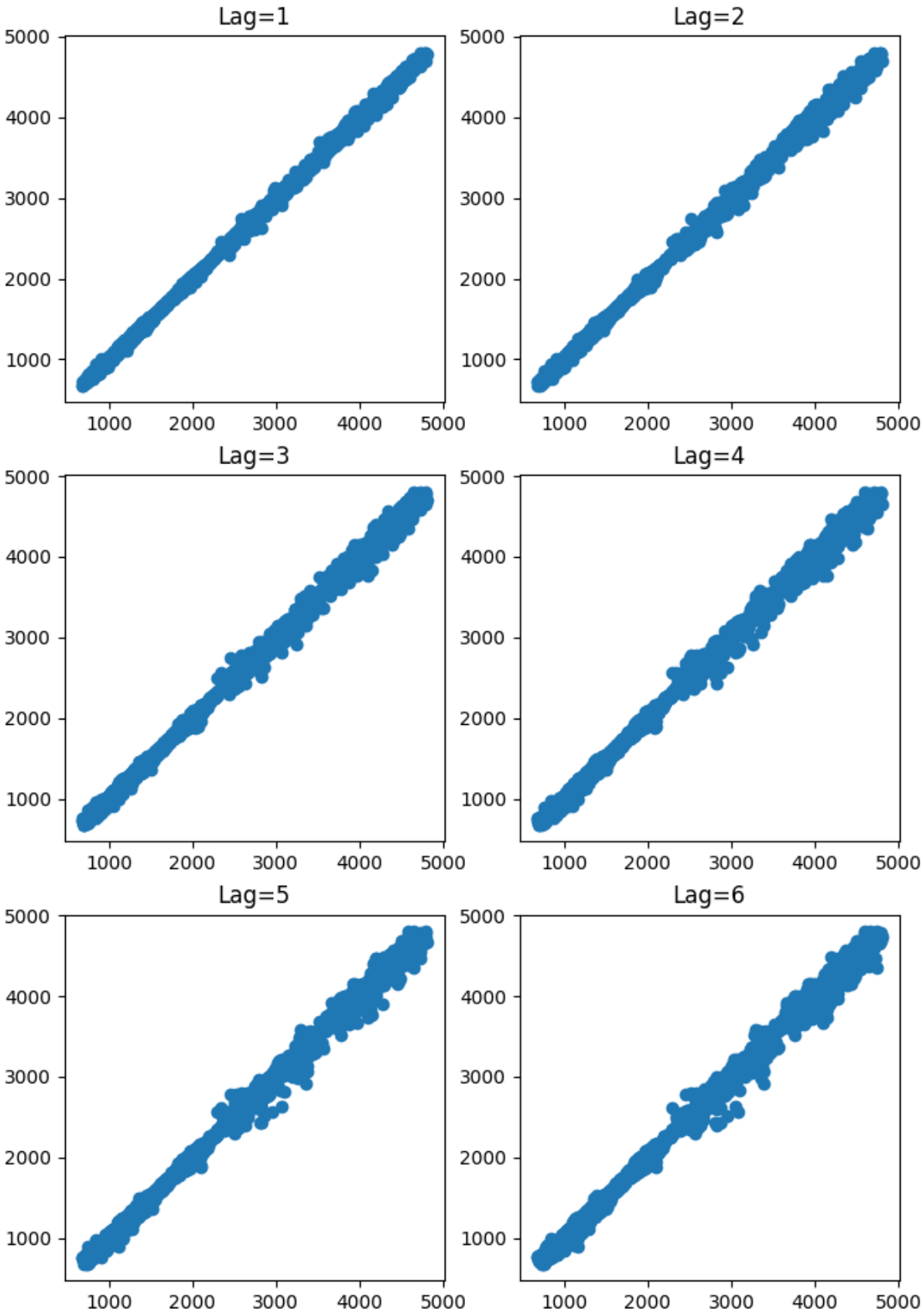
- q: The order of the moving average (MA) model. This is essentially the size of the "window" function over your time series data.

The main idea for the auto-arima approach is intelligently finding the proper combination of p, d, q such that you achieve the best fit.

A lag plot can provide clues about the underlying structure of your data:

- A linear shape to the plot suggests that an autoregressive model is probably a better choice.
- An elliptical plot suggests that the data comes from a single-cycle sinusoidal model.

In [59]:
```
from pandas.plotting import lag_plot

fig, axes = plt.subplots(3, 2, figsize=(8, 12))
plt.title('S&P500 Autocorrelation plot')

# The axis coordinates for the plots
ax_idcs = [
    (0, 0),
    (0, 1),
    (1, 0),
    (1, 1),
    (2, 0),
    (2, 1)
]

for lag, ax_coords in enumerate(ax_idcs, 1):
    ax_row, ax_col = ax_coords
    axis = axes[ax_row][ax_col]
    lag_plot(df['Open'], lag=lag, ax=axis)
    axis.set_title(f"Lag={lag}")
    axis.xaxis.label.set_visible(False)
    axis.yaxis.label.set_visible(False)

plt.show()
```



As above, all the lags look fairly linear, so it's a good indication that an auto-regressive model is a good choice. But since we don't want to allow simple visual bias to impact our decision here, we'll allow the auto_arima to select the proper lag term for us.

Estimate Differencing Parameter d

With pmdarima, we can run several differencing tests against the time series to select the best number of differences such that the time series will be stationary.

Here, we'll use the KPSS test and ADF test selecting the maximum value between the two to be conservative. Fortunately, in this case, both tests indicated that $d = 2$ was the best answer, but in the case where they disagreed, we could try both or allow auto_arima to auto-select the d term.

In [ ]:
```
from pmdarima.arima import ndiffs

kpss_diffs = ndiffs(y_train, alpha=0.05, test='kpss', max_d=6)
adf_diffs = ndiffs(y_train, alpha=0.05, test='adf', max_d=6)
n_diffs = max(adf_diffs, kpss_diffs)

print(f"Estimated differencing term: {n_diffs}")
# Estimated differencing term: 2
```

Therefore, d is set to 2.

Create Model

Let's plug into the model:

In [ ]:
```
auto = pm.auto_arima(y_train, d=n_diffs, seasonal=False, stepwise=True,
                     suppress_warnings=True, error_action="ignore", max_p=6,
                     max_order=None, trace=True)
```

Notice that we preset d=n_diffs, since we've already settled on a value for d. However, we're allowing our ARIMA models explore various values of p and q. After a few seconds, we arrive at the following solution:

In [ ]:
```
print(auto.order)
# (6, 2, 0)
```

Updating Model

Now that the heavy lifting of selecting model hyper-parameters has been performed, we can update our model by simulating days passing with our test set. For each new observation, we'll let our model progress for several more iterations, allowing MLE to update its discovered parameters and shifting the latest observed value. Then we can measure the error on the forecasts:

In [ ]:
```
from sklearn.metrics import mean_squared_error
from pmdarima.metrics import smape

model = auto

def forecast_one_step():
    fc, conf_int = model.predict(n_periods=1, return_conf_int=True)
    return (
        fc.tolist()[0],
        np.asarray(conf_int).tolist()[0])

forecasts = []
confidence_intervals = []

for new_ob in y_test:
    fc, conf = forecast_one_step()
    forecasts.append(fc)
    confidence_intervals.append(conf)

    # Updates the existing model with a small number of MLE steps
    model.update(new_ob)

print(f"Mean squared error: {mean_squared_error(y_test, forecasts)}")
print(f"symmetric Mean Absolute Percentage Error (sMAPE): {smape(y_test, for
# Mean squared error: 1917.3445329085707
# SMAPE: 0.8885413986916558
```

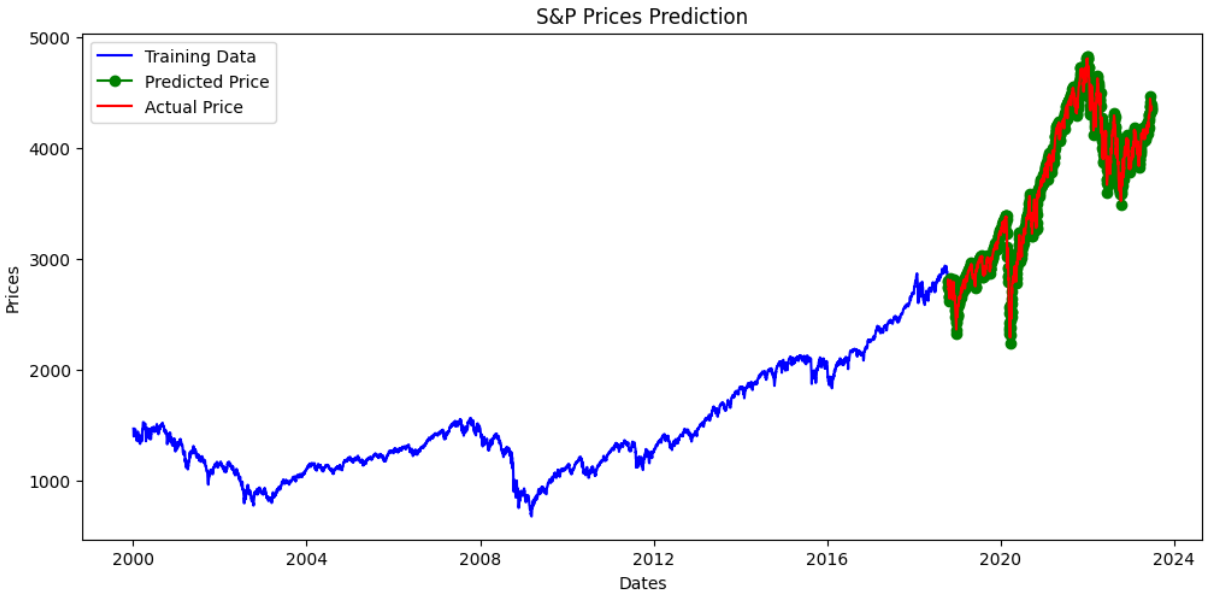
Symmetric Mean Absolute Percentage (SMAPE) at 88.9% not perfect but acceptable with the work so far.

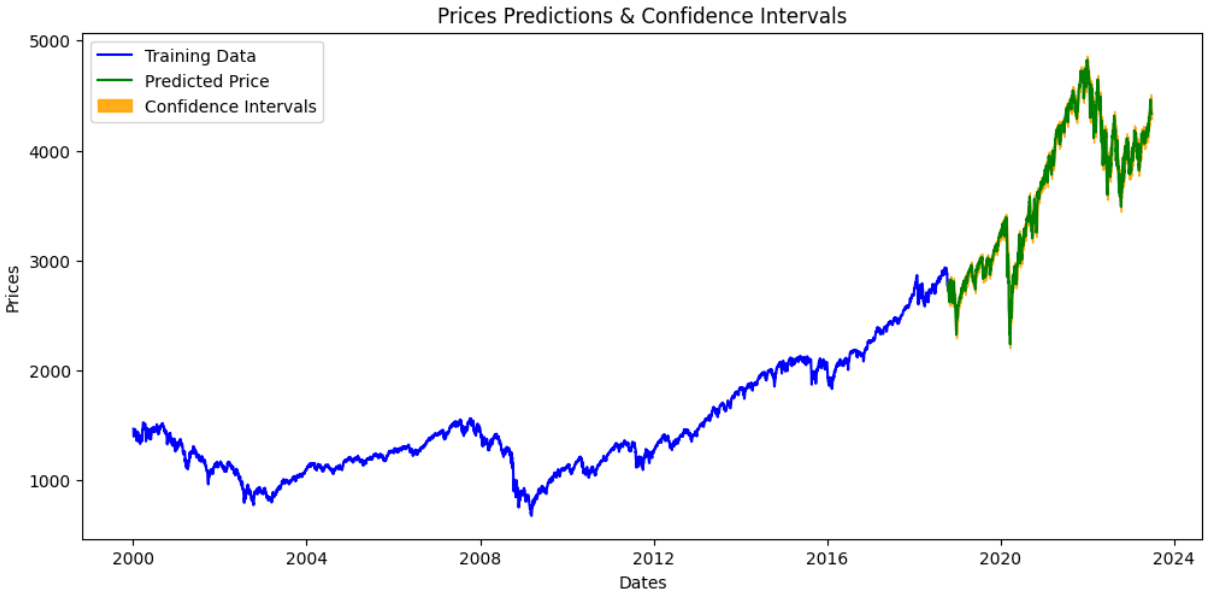
Plot Frecasts

We wil take a look at the forecasts our model produces overlaid on the actuals (in the first plot), and the confidence intervals of the forecasts (in the second plot).

In [73]:
```
fig, axes = plt.subplots(2, 1, figsize=(12, 12))

# -------------------- Actual vs. Predicted --------------------------
axes[0].plot(train_data.index, y_train, color='blue', label='Training Data')
axes[0].plot(test_data.index, forecasts, color='green', marker='o',label='Pr
axes[0].plot(test_data.index, y_test, color='red', label='Actual Price')
axes[0].set_title('S&P Prices Prediction')
axes[0].set_xlabel('Dates')
axes[0].set_ylabel('Prices')
axes[0].legend()

# ------------------ Predicted with confidence intervals ----------------
axes[1].plot(train_data.index, y_train, color='blue', label='Training Data')
axes[1].plot(test_data.index, forecasts, color='green',label='Predicted Pric

axes[1].set_title('Prices Predictions & Confidence Intervals')
axes[1].set_xlabel('Dates')
```

```
axes[1].set_ylabel('Prices')

conf_int = np.asarray(confidence_intervals)
axes[1].fill_between(test_data.index,
                     conf_int[:, 0], conf_int[:, 1],
                     alpha=0.9, color='orange',
                     label="Confidence Intervals")
axes[1].legend()
plt.show()
```





Summary

Stock price prediction is obviously not as simple as this notebook. The goal is to demonstrate some of basic mechanics running in Wall Street compute engines.

In [ ]: