ARIMA for Time Series Forecasting: A Complete Guide

Learn the key components of the ARIMA model, how to build and optimize it for accurate forecasts in Python, and explore its applications across industries.

Contents Updated Jan 7, 2025 · 12 min read

Zaina Saadeddin
ML Engineer. Lecturer, Data Mentor, EdTech Researcher. Passionate about AI.

TOPICS

Data Science

Python

Let's take a look at ARIMA, which is one of the most popular (if not the most popular) time series forecasting techniques. ARIMA is popular because it effectively models time series data by capturing both the autoregressive (AR) and moving average (MA) components, while also addressing non-stationarity through differencing (I). This combination makes ARIMA models especially flexible, which is why they are used across very different industries, like finance and weather prediction.

ARIMA models are highly technical, but I will break down the parts so you can develop a strong understanding. Before getting started, it's a good idea to familiarize yourself with some foundational tools. DataCamp offers a lot of good resources, such as our **ARIMA Models in Python** or **ARIMA Models in R** courses. You can choose either depending on the language you prefer.

Why Use ARIMA Forecasting?

Throughout finance, economics and environmental sciences etc., ARIMA has great interest because it can identify many complex patterns of our past observations with future needs which makes it a state-of-the-art technique. From **predicting the price of stocks**, forecasting weather patterns to getting an idea about consumer demand, ARIMA is a great way to make accurate and actionable predictive analyses.

By using ARIMA, we are able to both analyze and forecast time series data in a sophisticated manner that accounts for patterns, trends, and seasonality. This facilitates a 360-degree view of the underlying dynamics for making informed decisions.

Key Components of ARIMA Models

In order to really understand ARIMA, we need to deconstruct its building blocks. Once we have the components down, it will become easier to understand how this time series forecasting method works as a whole. Here, I'll give a detailed explanation of every component.

Autoregressive (AR) part

The Autoregressive (AR) component builds a trend from past values in the AR framework for predictive models. For clarification, the 'autoregression framework' works like a regression model where you use the lags of the time series' own past values as the regressors.

Integrated (I) part

The Integrated (I) part involves the differencing of the time series component keeping in mind that our time series should be stationary, which really means that the mean and variance should remain constant over a period of time. Basically, we subtract one observation from another so that trends and seasonality are eliminated. By performing differencing we get stationarity. This step is necessary because it helps the model fit the data and not the noise.

Moving average (MA) part

The moving average (MA) component focuses on the relationship between an observation and a residual error. Looking at how the present observation is related to those of the past errors, we can then infer some helpful information about any possible trend in our data.

We can consider the residuals among one of these errors, and the moving average model concept estimates or considers their impact on our latest observation. This is particularly useful for tracking and trapping short-term changes in the data or random shocks. In the (MA) part of a time series, we can gain valuable information about its behavior which in turn allows us to forecast and predict with greater accuracy.

How to Build an ARIMA Model in Python

To build an ARIMA model for forecasting, like gold prices, you can follow these steps. Let's break it down together.

Data collection

The first step is to tee up an appropriate dataset and prepare our environment.

Find a dataset

Collect or search for a dataset from data source platforms. You want one that has historical data over time. Here is a link to the Kaggle dataset related to gold future prices.

Install packages

We install the packages we need, including statsmodels and sklearn .

```python
import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.tsa.stattools import adfuller
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.arima.model import ARIMA
from sklearn.metrics import mean_squared_error
```

POWERED BY datalab

Load the data

We then read the data into our local environment. I'm taking the extra step to make sure the dates are recognized in order.

```python
# Define the path to your CSV file
# You may have to make this path more specific to the location of your file.
csv_path = "future-gc00-daily-prices.csv"

# Read the CSV, parse 'Date' column as datetime, and set it as the index
```

```
data = pd.read_csv(
    csv_path,
    parse_dates=["Date"],
    dayfirst=False,
    index_col="Date"
)

# Sort the DataFrame by the Date index in ascending order
data.sort_index(inplace=True)
```

POWERED BY datalab

Data preprocessing

Our dataset is pretty clean, but in other contexts, we would have to handle indexing issues, which is important in time series forecasting. For example, since we are forecasting the closing value of a stock on a particular exchange, we have to consider that the stock market is not open on weekends.

Before continuing, I'm also taking the quick step of removing commas from the `Close` variable, which is what I will be using.

```
# Clean the "Close" column
data["Close"] = data["Close"].replace(',', '', regex=True)
data["Close"] = pd.to_numeric(data["Close"], errors='coerce')
data["Close"].replace([np.inf, -np.inf], np.nan, inplace=True)
data.dropna(subset=["Close"], inplace=True)
```

POWERED BY datalab

Handle missing values

As part of data preprocessing, we also often have to consider **how to handle missing values** using an imputation method like forward filling or mean replacement. Do know that even one `NA` value, depending on the programming language and the library you are using, can prevent an ARIMA from running.

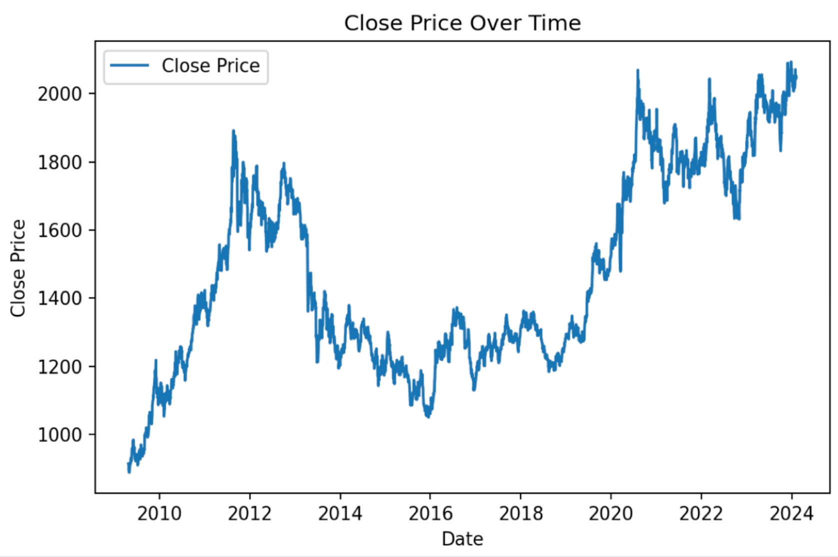
I had mentioned that weekend days might be recognized as `NA`. That's can be the case, and would require an extra step. Luckily, `statsmodels` treats the data as a sequential index without strictly enforcing regular time intervals, allowing our model to compile even weekend gaps.

Create a time plot

Now is a good time to create a time plot so we can see the series that we are working with:

```
# Plotting the original Close price
plt.figure(figsize=(14, 7))
plt.plot(data.index, data["Close"], label='Close Price')
plt.title('Close Price Over Time')
plt.xlabel('Date')
plt.ylabel('Close Price')
plt.legend()
plt.show()
```

POWERED BY datalab



Check for stationarity and perform differencing if necessary

While ARIMA models can deal with non-stationarity up to a point, they cannot effectively account for time-varying variance. In other words, for an ARIMA model to really work, the data has to be stationary.

Looking at the plot, above, we can see that the data is, in fact, not stationary because there is a clear trend. Also, it looks like there is non-constant variance at different time points. We can use the Augmented Dickey-Fuller test to test our intuition and see if our data has a constant mean and variance, and put numbers to it.

```
# Perform the Augmented Dickey-Fuller test on the original series
result_original = adfuller(data["Close"])

print(f"ADF Statistic (Original): {result_original[0]:.4f}")
print(f"p-value (Original): {result_original[1]:.4f}")

if result_original[1] < 0.05:
    print("Interpretation: The original series is Stationary.\n")
else:
    print("Interpretation: The original series is Non-Stationary.\n")

# Apply first-order differencing
data['Close_Diff'] = data['Close'].diff()

# Perform the Augmented Dickey-Fuller test on the differenced series
result_diff = adfuller(data["Close_Diff"].dropna())
print(f"ADF Statistic (Differenced): {result_diff[0]:.4f}")
print(f"p-value (Differenced): {result_diff[1]:.4f}")
if result_diff[1] < 0.05:
    print("Interpretation: The differenced series is Stationary.")
else:
    print("Interpretation: The differenced series is Non-Stationary.")
```

POWERED BY datalab

```
ADF Statistic (Original): -1.4367
p-value (Original): 0.5646
Interpretation: The original series is Non-Stationary.

ADF Statistic (Differenced): -19.1308
p-value (Differenced): 0.0000
Interpretation: The differenced series is Stationary.
```

POWERED BY datalab

The results of our Augmented Dickey-Fuller test indicate that our original series is, in fact, non-stationary, so using an ARIMA out-of-the-box on the original data would be a mistake.

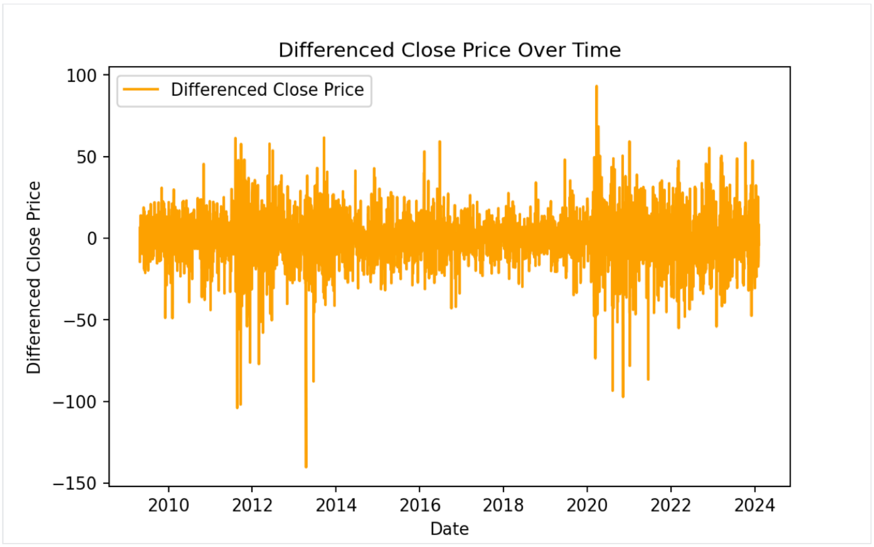
The ADF test on a differenced version of the data indicates that the differenced version is stationary, however.

A quick note on differencing. I want to take the time to explain this, since a lot is happening under the hood. To perform differencing, we subtract each observation from the previous one to give us a new time series of first differences. (The new time series is now one element shorter than the original.) If the differenced series is still not stationary, we can take a second difference by differencing the original series again, and we can continue differencing the series until it finally becomes stationary. The order of differencing required is the minimum number of differences needed to get a series with no autocorrelation.

I do want to say, finally - don't perform differencing more than you have to, otherwise you might create some false dynamics in your data. Also, if you were going to do seasonal ARIMA model, which we aren't doing in this tutorial, you would always consider a seasonal difference before a first difference.

```
# Plotting the differenced Close price
plt.figure(figsize=(14, 7))
plt.plot(data.index, data['Close_Diff'], label='Differenced Close Price', color='
plt.title('Differenced Close Price Over Time')
plt.xlabel('Date')
plt.ylabel('Differenced Close Price')
plt.legend()
plt.show()
```

POWERED BY datalab



Model identification

When we build an ARIMA model, we have to consider the *p*, *d*, and *q* terms that go into our ARIMA model.

- The first parameter, *p*, is the number of lagged observations. By considering *p*, we effectively determine how far back in time we go when trying to predict the current observation. We do this by looking at the autocorrelations of our time series, which are the correlations in our series at previous time lags.
- The second parameter, *d,* refers to the order of differencing, which we talked about. Again, differencing simply means finding the differences between consecutive timesteps. It is a way to make our data stationary, which means removing the trends and any changes in variance over time. *d* indicates differencing at which order you get a process stationary.

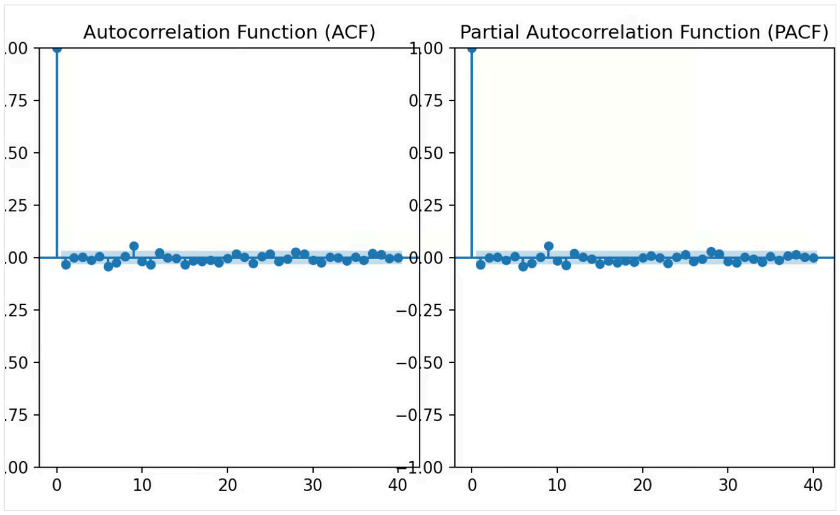
- The third parameter *q* refers to the order of the moving average (MA) part of the model. It represents the number of lagged forecast errors included in the model. Unlike a simple moving average, which smooths data, the moving average in ARIMA captures the relationship between an observation and the residual errors from a moving average model applied to lagged observations.

Finding the ARIMA terms

We use tools like ACF (Autocorrelation Function) and PACF (Partial Autocorrelation Function) to determine the values of *p*, *d*, and *q*. The number of lags where ACF cuts off is *q*, and where PACF cuts off is *p*. We also have to choose the appropriate value for *d* by creating a situation where, after differencing, the data resembles white noise. For our data, we choose 1 for both *p* and *q* because we see a significant spike in the first lag for each.

```
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
import matplotlib.pyplot as plt

# Plot ACF and PACF for the differenced series
fig, axes = plt.subplots(1, 2, figsize=(16, 4))

# ACF plot
plot_acf(data['Close_Diff'].dropna(), lags=40, ax=axes[0])
axes[0].set_title('Autocorrelation Function (ACF)')

# PACF plot
plot_pacf(data['Close_Diff'].dropna(), lags=40, ax=axes[1])
axes[1].set_title('Partial Autocorrelation Function (PACF)')

plt.tight_layout()
plt.show()
```

POWERED BY datalab



Parameter estimation

To be clear, the *p*, *d*, and *q* values in ARIMA represent the model's order (lags for autoregression, differencing, and moving average terms), but they are not the actual parameters being estimated. Once the *p*, *d*, and *q* values are chosen, the model estimates additional parameters, such as coefficients for the autoregressive and moving average terms, through maximum likelihood estimation (MLE).

Forecasting

To forecast using an ARIMA model, start by using the fitted model to predict future values based on the data. Once predictions are made, it's helpful to visualize them by plotting the predicted values alongside the actual values. This is accomplished because we use a train/test workflow, where the data is split into training and testing sets. Doing this lets us see how well the model performs on unseen data. If you are unclear on this, take our **Model Validation in Python** course, which is a great resource to learn the ins and outs of training and testing.

1. Use a train/test workflow

Our first step is to split the data into training and testing versions.

```
# Split data into train and test
train_size = int(len(data) * 0.8)
train, test = data.iloc[:train_size], data.iloc[train_size:]

# Fit ARIMA model
model = ARIMA(train["Close"], order=(1,1,1))
model_fit = model.fit()
```

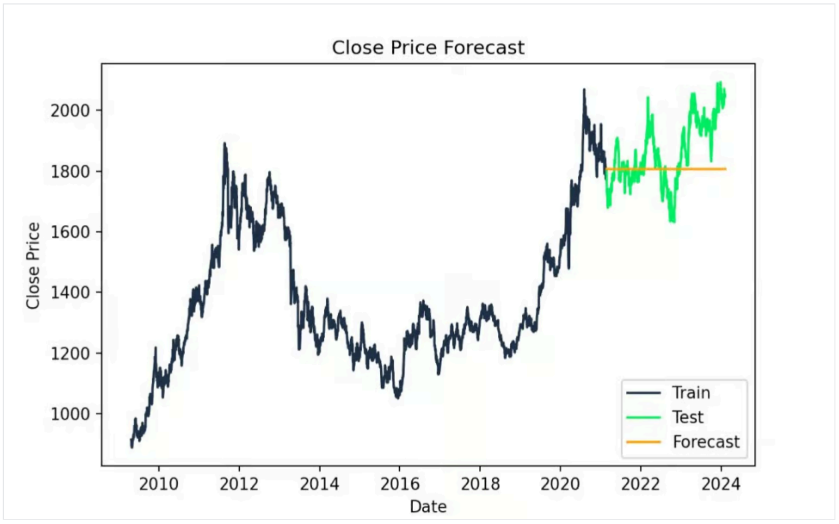
POWERED BY datalab

2. Visualize our time series

Our next step is to create our forecast and also to visually inspect it. We can see how our forecast performs against the testing version of our data.

```
# Forecast
forecast = model_fit.forecast(steps=len(test))

# Plot the results with specified colors
plt.figure(figsize=(14,7))
plt.plot(train.index, train["Close"], label='Train', color='#203147')
plt.plot(test.index, test["Close"], label='Test', color='#01ef63')
plt.plot(test.index, forecast, label='Forecast', color='orange')
plt.title('Close Price Forecast')
plt.xlabel('Date')
plt.ylabel('Close Price')
plt.legend()
plt.show()
```

POWERED BY datalab



ARIMA forecast actual vs. predicted values. Image by Author

3. Evaluate model statistics

We check out the AIC and BIC model statistics. Lower values mean the model fits better, but we might also compare the results with those from simpler models to avoid overfitting. I'm printing the numbers here but they make the most sense in the context of comparing to other ARIMA models on the same data, in order to find the ARIMA model that works best.

```
print(f"AIC: {model_fit.aic}")
print(f"BIC: {model_fit.bic}")
```

Explain code

POWERED BY datalab

```
AIC: 24602.97426066606
BIC: 24620.97128230579
```

POWERED BY datalab

We can also evaluate the mean squared error, to assess our model's fit. This is a practical metric. A lower RMSE indicates a better ARIMA model, reflecting smaller differences between actual and predicted values, and it's on the scale of the data. That is, the RMSE of 118.5339 signifies that, on average, our model's predictions deviate from the actual Close prices by about $118.53.

```
forecast = forecast[:len(test)]
test_close = test["Close"][:len(forecast)]

# Calculate RMSE
rmse = np.sqrt(mean_squared_error(test_close, forecast))
print(f"RMSE: {rmse:.4f}")
```

POWERED BY datalab

```
RMSE: 118.5339
```

POWERED BY datalab

Common Uses of ARIMA Forecasting

Now, let's discuss the applications of ARIMA in different industries. A variety of sectors - everything from economics and finance to weather forecasting and health - make use of ARIMA models to derive insights from data and in the quest for predictive accuracy as well. Some big ones are as follows:

Economics and finance

ARIMA's strength lies in its capacity to handle financial time series that often contain complex autocorrelations and interactions between multiple economic indicators. Its ability to model lag effects and incorporate differencing makes it ideal for forecasting volatile metrics like stock prices or exchange rates.

- **Financial Forecasting:** Through predicting stock prices, exchange rates and other financial instruments, ARIMA can be used to support investment strategies.
- **Economic Modeling:** ARIMA models help predict the future of a country or global economy, informing economic policy decisions.

- **Demand Planning:** ARIMA predicts the demand for consumer goods and services, helping to optimize production planning to control inventory.

Weather forecasting

ARIMA models leverage historical weather patterns to provide short- and long-term forecasts, so they are flexible enough to predict both typical and extreme weather conditions.

- **Predicting Temperature and Precipitation**: ARIMA models are used in short-term and long-term weather forecasts, incorporating air-sea interactions and many other factors.
- **Climate Change Modeling**: ARIMA models use historical weather data to better understand trends in the climate and predict what future climates will look like.

Supply chain management

ARIMA's ability to model lag effects helps supply chain managers manage inventory or anticipate disruptions based on historical patterns and lead times.

- **Demand Forecasting**: ARIMA has the ability to predict the future demand of products and plan production schedules or stock levels.
- **Inventory Management:** ARIMA ensures that the right levels of stock are maintained on items so as to not have too much capital tied up in inventory investments and to reduce costs related to over- or understocking.
- **Supply Chain Optimization**: ARIMA can forecast supply chain disruptions by analyzing the interactions between multiple variables including, for example, transport delays or demand fluctuations.

Healthcare

In healthcare, ARIMA models are particularly valuable because they can predict patient admissions and other important trends.

- **Disease Outbreak Prediction**: ARIMA models prove utility by predicting the propagation of infectious diseases, which then will pave the way for pre-emptive public health interventions.
- **Hospital Admission Forecasting**: ARIMA predicts hospital admission rates and helps optimize resources and staff schedules.
- **Patient Monitoring**: ARIMA is a useful tool for professionals who want to examine medical data to warn of early signs of health issues and tailor-fit treatment strategies.

Things to Consider for Better ARIMA Forecasting

Here are some common mistakes to avoid while working on building ARIMA models:

Overfitting and underfitting

If we choose incorrect *p*, *d*, and *q* values, it can lead to overfitting or underfitting. We overfit when our model is too complex and it gloms onto the noise in our data, so that it doesn't generalize well to new observations. On the other extreme, underfitting simply means that our model is less complex and cannot capture all of the underlying patterns.

To prevent overfitting, an approach could be to use fewer lag terms and also possibly fewer differencing terms. Underfitting can be fixed by increasing the number of autoregressive terms, if appropriate. One must strike a balance between complexity and simplicity. Techniques such as validation/cross-validation can help.

Stationarity

Stationarity is a statistical assumption that deals with time-dependencies of data. Unreliable forecasts and spurious relationships can result from non-stationary data. Differencing or transformations such as log transformations or seasonal adjustments can be used to make non-stationary data stationary.

Seasonality

The presence of seasonality is another vital component to take into account while dealing with time-series analysis. Daily, weekly, and yearly are some of the fixed intervals over which many real-world datasets exhibit repeated patterns. Disregarding these seasonal patterns can result in improper forecasting. In the context of seasonality, we need seasonal differences and seasonal AR and MA terms in addition to *p* and *q* values. Keep in mind that a series can have more than one kind of seasonality.

Residual analysis

One of the most important steps in ARIMA modeling is to check if the residual series that is generated is stationary. Residuals are the difference between observed values and those produced by a model. By looking at the residuals, we can check if our model is able to find and work with the dynamics in the data. The residuals should show a random scatter without indicating any trends or correlations.

When the residuals show patterns or correlations, it means that there is information somewhere that the model has not completely captured. Statistical tests and visual diagnostics, including the Ljung-Box test, as well as histograms and other diagnostic plots can be used to verify that the model is adequate.

Next Steps with ARIMA and Related Models

In many cases, ARIMA is not the final step. Just as ARIMA is an evolution of autoregressive or moving average models, newer ideas have been developed as well. For one thing, ARIMA models themselves can handle both linear and non-linear patterns in a time series. If you want a seasonal forecast, consider SARIMA models, which can handle multi-period/periodic patterns in our time series. SARIMA models are especially useful in areas where data has a recurring pattern or cyclic behavior like sales forecasting and weather predictions. ARIMAX models are another popular options. ARIMAX models are ARIMA models that take an external variable or exogenous regressor. They can help a great deal in the performance and accuracy of our forecasts.

Furthermore, moving into the realm of machine learning, we can think about diving deeper in time series analysis using tools such **recurrent neural networks (RNN)** and **LSTM** for predicting complex temporal dependencies. As a final thought, the field of Bayesian time series analysis and understanding how such an approach can provide benefits in forecasting and decision-making.

Conclusion / Final Thoughts

As we have seen, ARIMA is a common statistical model that assesses the time series, and predicts future values by taking into account both autoregressive and moving average elements. It allows us to generate a forecast of the historical data even though the features of a particular dataset might be very different from the features of another dataset. Its adaptability is what makes it a common and widely used forecasting method.

Hands-on experience is important to master the basics of ARIMA. DataCamp offers complete courses based on your learning needs to improve and master the subject of ARIMA modeling. In these top online tutorials, learn the fundamentals of ARIMA modeling as well as the most practical tools and techniques for implementing analytical solutions that solve hard real-world problems with far less effort (and in less time) than you ever thought possible. By the end, you should feel comfortable applying ARIMA modeling in your future data science work. Check out the ARIMA modeling courses available on DataCamp and achieve your maximum potential in time series analysis: **Forecasting in R**, **Time Series with R**, **ARIMA Models in Python**, **ARIMA Models in R**.