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

Sep 9, 2024 · 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`.

```
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
```


Load the data

We then read the data into our local environment.

```
data = pd.read_csv("future-gc00-daily-prices.csv", index_col="Date")
```


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, if we were forecasting the

opening value of a stock on a particular exchange, we would have to consider that the stock market is not open on weekends.

Check for stationarity

Keeping things stationary makes the modeling task a lot easier, helps to improve our model accuracy and in return provides us with more reliable predictions. While ARIMA models can deal with non-stationarity up to a point, they cannot effectively account for time-varying variance. Here we can use the Augmented Dickey-Fuller test to tell us if our data has a constant mean and variance.

```
result = adfuller(data["Price"])
print(f"ADF Statistic: {result[0]}")
print(f"p-value: {result[1]}")
```

Explain code

Handle missing values

As part of data preprocessing, we also have to consider how to handle missing values using an imputation method like forward filling or mean replacement.

```
data.fillna(method='ffill', inplace=True)  # fill missing values
```


Perform differencing

To analyze our time series data for stationarity, we should first calculate the differences in our data. If the data is not stationary, apply the differencing technique to transform it into a stationary series. The steps to take in performing differencing are:

- Subtract each observation from the next to give us a new time series of first differences. This creates a new time series that is one element shorter than the original.
- Test if the differenced series is now stationary. If not, then we can take the second difference by differencing the original series again.
- Continue differencing the series until it is stationary. The order of differencing required is the minimum number of differences needed to get a series with no autocorrelation.

```
if result[1] > 0.05:
data["Price"] = data["Price"].diff().dropna()
result = adfuller(data["Price"])
stationarity_interpretation = "Stationary" if result[1] < 0.05 else "Non-Stationa

print(f"ADF Statistic after differencing: {result[0]}")
print(f"p-value after differencing: {result[1]}")
print(f"Interpretation: The series is {stationarity_interpretation}.")
```



```
ADF Statistic: -11.498371141896145
p-value: 4.5550962204394835e-21
Interpretation: The series is Stationary.
```


Model identification

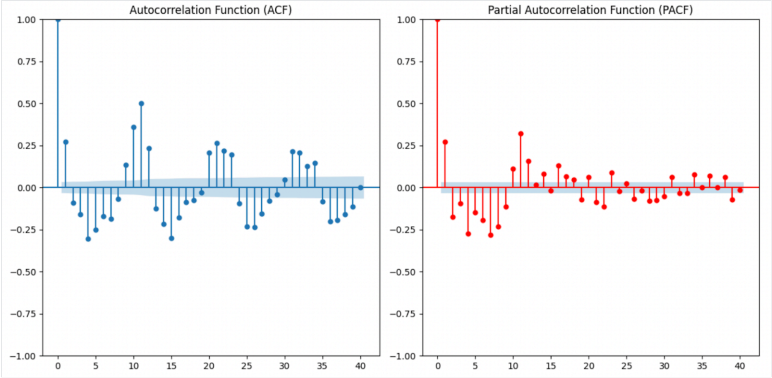
When we build an ARIMA model, we have to consider the *p*, *d*, and *q* terms that go into our ARIMA model.

- 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. Differencing simply means finding the differences between consecutive timesteps. It is a way to make our data stationary, which means removing the trends and seasonality. *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.

```
plot_acf(data["Price"], lags=40)
plot_pacf(data["Price"], lags=40)
```

ACF and PACF plots used to determine ARIMA terms. Image by Author

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).

Model fitting

```
# Fit the ARIMA model
# Initial ARIMA Model parameters
p, d, q = 1, 0, 1
model = ARIMA(data["Price"], order=(p, d, q))
model_fit = model.fit()
```

```
model_summary = model_fit.summary()
model_summary
```

POWERED BY datalab

Variable:	Price	No. Observations:	3703
Model:	ARIMA(1, 0, 1)	Log Likelihood	-26037.699
Time:	16:10:32	AIC	52083.399
Sample:	0	BIC	52108.266
	-3703	HQIC	52092.248
Covariance Type:	opg		

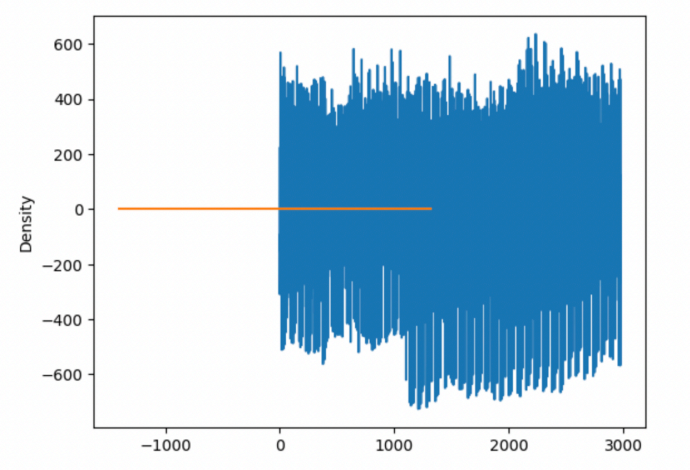
Model statistics and model diagnostics

We now check the residuals and make sure they act like white noise, which means they should have no patterns or trends. One option is to use our ACF and PACF plots again, but this time applied to the residuals. If there are no large spikes in these graphs lag outside of the band, then it means our residuals appear to be white noise. We can also check the residuals of the overall model to make sure there are no obvious patterns, as we are doing here:

Residual plot

```
# plot residual errors
residuals = model_fit.resid
residuals.plot()
residuals.plot(kind='kde')
plt.show()
```

POWERED BY datalab



ARIMA model residuals. Image by Author

AIC and BIC

We check out the model statistics relevant to model selection. Lower values mean the model fits better, but we also might compare the results with the results from simpler models to avoid overfitting.

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

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

Explain code

POWERED BY datalab

```
AIC: 41919.18902176751
BIC: 41937.18705062565
```

POWERED BY datalab

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. Our Model Validation in Python course is a great resource to learn the ins and outs of model validation.

1. Use a train/test workflow

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

```
data = data[“Price”]train_size = int(len(data) * 0.8)train, test = data[:tra si

# Fit the model to training data. Replace p, d, q with our ARIMA parameters
model = ARIMA(train_data["Price"], order=(p, d, q))

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

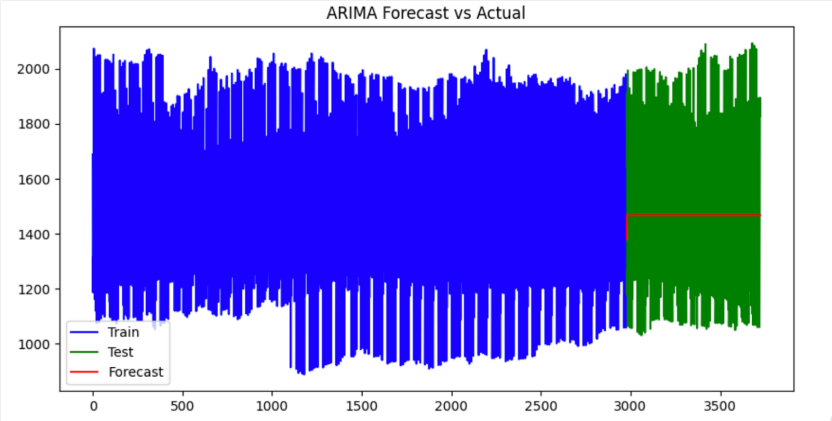
POWERED BY datalab

2. Visualize our time series

Our next step is to visually inspect our time series forecast.

```
# Plotting
plt.figure(figsize=(10, 5))
plt.plot(data.index[:train_size], train, label='Train', color='blue')
plt.plot(data.index[train_size:], test, label='Test', color='green')
plt.plot(data.index[train_size:], forecast, label='Forecast', color='red')
plt.legend()
plt.title('ARIMA Forecast vs Actual')
plt.show()
```

POWERED BY datalab



ARIMA forecast actual vs. predicted values. Image by Author

3. Evaluate model statistics

We evaluate model statistics, particularly the mean squared error, to assess our model's fit. A lower RMSE indicates a better ARIMA model, reflecting smaller differences between actual and predicted values.

```
# Evaluate model performance on the test set
rmse = mean_squared_error(test_data["Price"], predictions, squared=False)
print(f"RMSE: {rmse}")
```

POWERED BY datalab

```
“RMSE”: 135.87678712210163
```

POWERED BY datalab