ARIMA Model

In previous blog posts, we explored popular machine learning models like regressions, decision trees, and ensemble methods. Today, let's dive into a powerful statistical model specifically designed for time series forecasting: the **ARIMA model**. ARIMA stands for AutoRegressive Integrated Moving Average, and it's uniquely tailored to analyze a single sequence of historical data points to uncover patterns and predict future values.

Unlike typical machine learning models that use multiple predictors, ARIMA learns solely from the history of the series itself. This makes it especially valuable for forecasting tasks where past behavior provides the best clues to what comes next. To illustrate, we'll use ARIMA to attempt forecasting future closing prices of Apple Inc. stock, based on daily historical data of 2024.

What is ARIMA?

ARIMA is popular because it effectively models time series data, making it flexible to utilize across industries, such as finance and meteorology. At its heart, it combines three core techniques for modeling and forecasting time series data:

1. The AutoRegressive (AR) component predicts the current value based on a weighted sum of previous values, capturing the idea that "history matters."
2. The Integration (I) component, or differencing, removes trends and seasonality by subtracting yesterday's value from today's. This process makes the series **stationary**—its statistical properties remain consistent over time—so the data becomes more stable and easier to model.
3. The Moving Average (MA) component models the current value using weighted past forecast errors, correcting past mistakes to improve accuracy.

Together, these components form a flexible recipe that handles a variety of real-world time series patterns. The ARIMA model is typically denoted as ARIMA(p, d, q), where each parameter corresponds to:

1. p = how many past days/ values used in the AR component (lag observations)
2. d = how many times you difference the series to make it stationary (differencing steps)
3. q = how many past errors you take into consideration (lagged forecast errors)

Understanding these parameters in the context of our Apple stock example, p captures how many previous days influence today's price, d how many times we difference the series to remove trends, and q how past prediction errors affect the current prediction.

Before we build the model, however, it's important to recognize that ARIMA is a statistical model with a mathematical foundation. Its equations only work correctly when our data behaves in a certain way. If its assumptions are violated, the predictions can become misleading. Let's review these assumptions.

Assumptions of ARIMA

In this section, we'll walk through some of the important assumptions of ARIMA that must be met, or at least approximated, and explain important terminology:

1. **Stationarity**

The most critical assumption is that the data should be stationary: its properties, like mean, variance, and **autocorrelation**, don't change over time. Before we proceed, let's define a new term: autocorrelation measures how related a variable is to its lagged versions. For example:

- High autocorrelation at lag 1 means today's value depends heavily on yesterday's.
- High autocorrelation at lag 7 might reflect a weekly cycle.
- Low autocorrelation suggests past values don't predict the current well.

If the series exhibits trends or seasonality, differencing (the "I" in ARIMA) removes these systematic effects, making the series more predictable.

2. **Linearity**

ARIMA assumes the future value of the series is a linear combination of past values and past errors. If your series has highly nonlinear relationships, ARIMA may not perform well, and you may consider other models, such as nonlinear regressions.

3. **No Seasonality**

ARIMA doesn't naturally handle strong seasonal effects, which are repetitive patterns at fixed frequencies. An example of seasonality would be the forecast of toy sales at FAO Schwarz. Typically, around Christmas, this store sees a massive spike in sales, rendering this pattern seasonal. If seasonality is this strong, you'd preprocess the data or consider **SARIMA** - Seasonal ARIMA - which explicitly models seasonal components.

4. **Univariate Data**

Classical ARIMA is designed for a single time series variable, rather than multiple variables.

5. **No autocorrelation in residuals**

After fitting, the residuals should ideally have no autocorrelation. If residuals show patterns, it indicates the model hasn't captured all structure, and parameters *p, d,* and *q* need re-tuning.

With these foundations laid, we're ready to explore the workflow of the ARIMA model known as the Box–Jenkins method. If ARIMA's assumptions are the rules of the game, Box–Jenkins is the strategic plan on how to play effectively.

Box-Jenkins

How does one actually build a robust forecasting model from real-time series data? This is where the Box–Jenkins methodology comes into play. Created by George Box and Gwilym Jenkins in the 1970s, this influential approach provides a practical, iterative framework for identifying, estimating, and validating ARIMA models.

The process typically follows three main steps, which are also outlined in my code file:

Step 1: Identification – Determine values of *p, d,* and *q* using visualization and statistical tests.
Step 2: Estimation – Fit the ARIMA model and estimate parameters.
Step 3: Validation – Run diagnostics to ensure the model explains the data without overfitting.

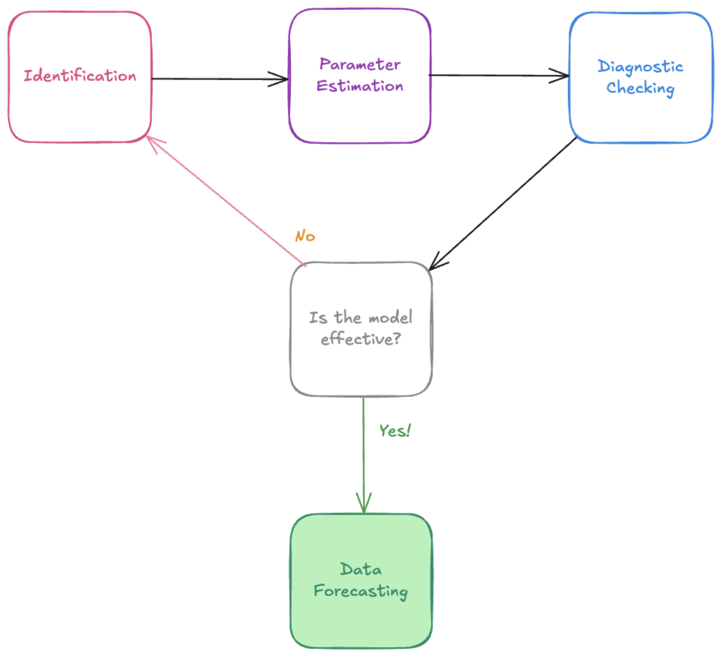


Figure 1

Step 1: Identification

The first task is choosing values of *p, d,* and *q* for our ARIMA model, guided by a combination of visualization and the following statistical tests:

a) ACF - Autocorrelation Function

The ACF measures how strongly today's value is correlated with values from *k* days ago. If correlations remain high across many lags, it signals a non-stationary, trending series. The autocorrelation (ρ - "rho") at lag *k* is given by:

$$\rho_k = \frac{Cov(Y_t, Y_{t,-k})}{Var(Y_t)}$$

Where Y_t is the value of the series at time *t,* and $Cov(Y_t, Y_{t,-k})$ is the covariance between Y_t and its *k*-lagged value. In plain terms, the autocorrelation coefficient tells us how much today's value resembles past values at lag *k*. Very high ACF values across many lags signal non-stationarity or strong trends.

Now, let's understand ACF through the lens of the Apple stock prices.

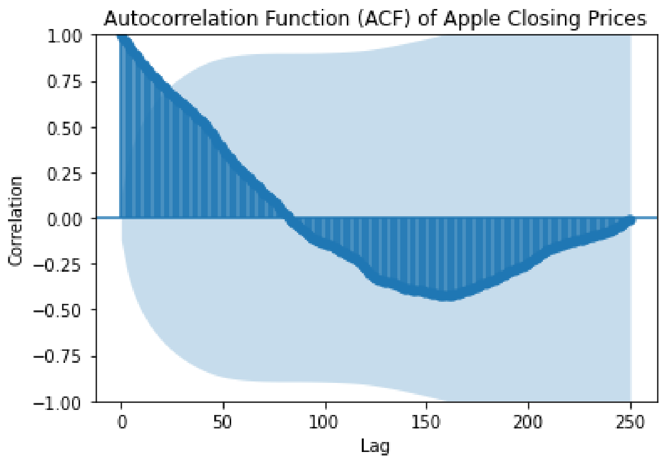


Figure 2

This ACF plot clearly shows a strong positive correlation at lag 1 and a gradual decay across higher lags, indicating the Apple series is non-stationary and heavily influenced by its own history. The persistence of autocorrelation confirms the presence of a trend, meaning the series must be differenced before applying ARIMA.

b) PACF - Partial Autocorrelation Function

PACF highlights the direct correlation between Y_t and $Y_{t,-k}$, stripping away the effects of intermediary lags. This is analogous to checking how much additional predictive power each lag adds, after accounting for others. In an autoregressive AR(k) model, the equation is:

$$Y_t = \phi_{k1}Y_{t-1} + \phi_{k2}Y_{t-2} + \dots + \phi_{kj}Y_{t-k} + \varepsilon_t$$

where each ϕ_{kj} is an autoregressive coefficient found via the Yule–Walker equations.

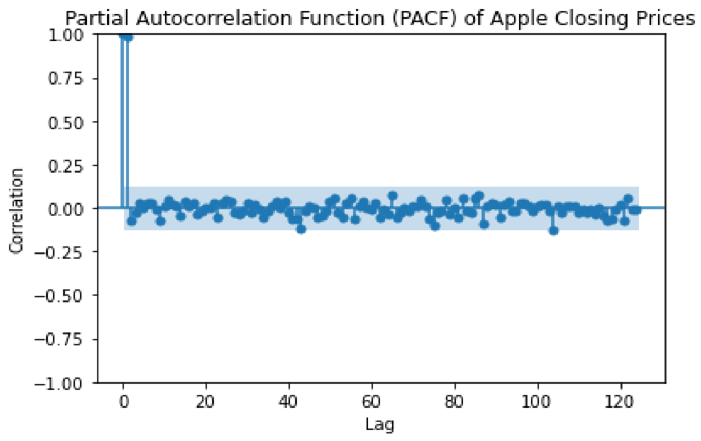


Figure 3

This PACF plot reveals sharp spikes at the first two lags, indicating that today's Apple closing price is strongly influenced by both yesterday's and the day before's values. This pattern suggests an AR(2) process, where the last two observations are key drivers of the current price.

c) ADF test - Augmented Dickey–Fuller test

Visual inspection is helpful, but we also want a statistical test. The ADF test checks for stationarity by testing for a **unit root**[1] — a feature that indicates non-stationarity. At its core, the test asks a simple question:

- Null hypothesis H_0: the series has a unit root → it is *non-stationary*
- Alternative H_1: the series has no unit root → it is *stationary*

Like any hypothesis test, the ADF produces a test statistic and a p-value. If the p-value is below 0.05, we reject H_0 and conclude the series is stationary.

The ADF test is just a regression problem; the model looks like this:

$$\Delta Y_t = \alpha + \beta t + \gamma Y_{t-1} + \sum_{i=1}^{p} \delta_i \Delta Y_{t-i} + \varepsilon_t$$

This might look like a Greek sentence, but I promise it gets easier when you treat it like a multiple regression. The variables here denote:

$\Delta Y_t = Y_t - Y_{t-1}$ or the first difference of the series

α = the constant term (depending on test type)

βt = trend term (depending on test type)

γ = coefficient on Y_{t-1}: the key term — tells us if there's a unit root

δ_i = extra lags to soak up correlation so γ is properly estimated

The entire test boils down to the coefficient γ:

- If $\gamma = 0$, we have a unit root → the series is non-stationary.
- If $\gamma < 0$, the series is pulled back toward a mean → it's stationary.

Look at the closing price of Apple stock. A clear upward trend is visible in the raw data, so we are already suspicious of a trend.

[1] characteristic of a time series that indicates it is non-stationary. Resource

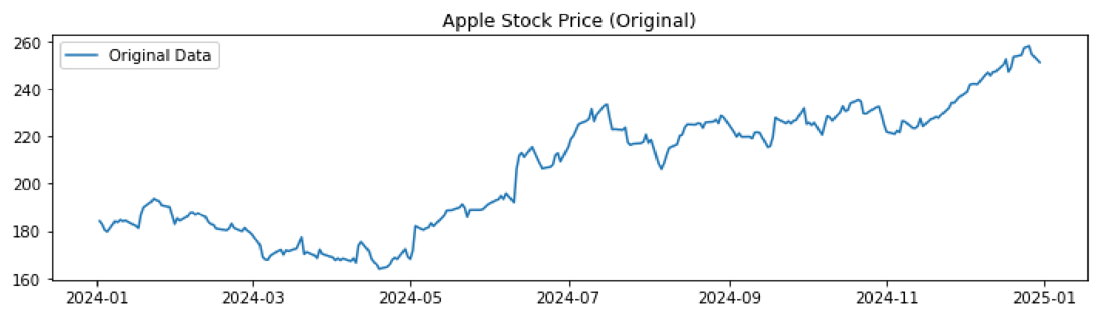


Figure 4

Let's back our hypothesis up by running the ADF test in Python:

```
ADF Statistic: 1.7638146742369016
p-value: 0.9982744832859146
Critical Values: {'1%': -3.4320936126640174,
'5%': -2.8623104319229618, '10%': -2.5671801115022364}
Fail to reject the null hypothesis → Series is non-stationary
```

The ADF statistic (1.76) is far above all critical values, and the p-value is much higher than the significance level (0.05). Therefore, we fail to reject the null hypothesis of the ADF test, concluding that our Apple stock time series data is non-stationary and needs to be differenced.

Step 2: Estimation

The next step in the Box–Jenkins framework is to apply differencing to remove the trend and stabilize the mean. We do so by subtracting each value from the one before it:

$$Y'_t = Y_t - Y_{t-1}$$

The plot below shows the Apple stock prices differenced. The series now appears to fluctuate around a stable mean rather than trending upwards, as seen in Figure 4.

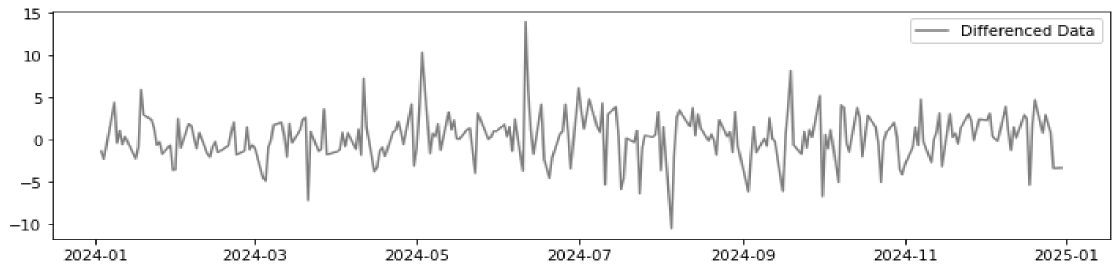
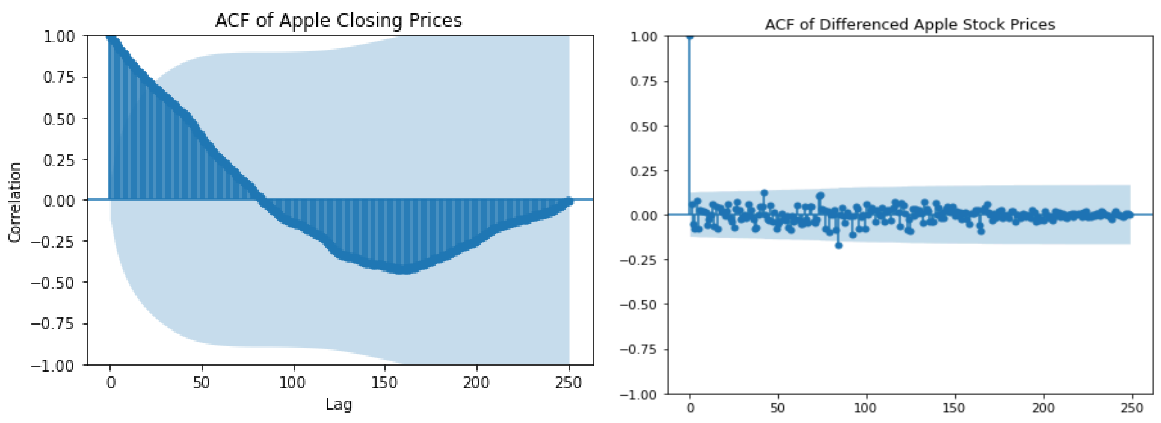


Figure 5

Now, we'll re-check the ACF:

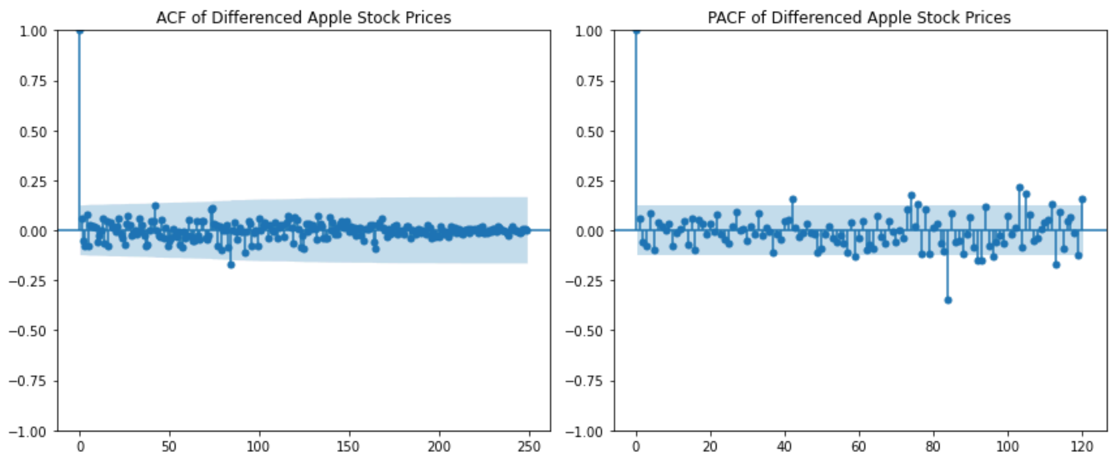


Figures 2 & 6

After differencing the series, the ACF looks very different. Autocorrelations now decay much faster, a sign that the series is closer to being stationary. Re-running the ADF test (in the code file) confirms it: the p-value drops below 0.05, so we can reject the null and treat the differenced series as stationary.

Step 3: Model Selection

Once the series is stationary, the ACF and PACF plots guide our choice of *p* and *q*:



Figures 6 & 7

Notice how the ACF has a strong spike at lag 1 and then quickly falls within the confidence bands. This is a classic signal of a potential MA(1) process. Similarly, the PACF also shows a clear spike at lag 1 before tapering off, which suggests an AR(1) component.

Taken together, these patterns imply that the differenced Apple stock series may be best modeled by an ARIMA(1,1,1) process.

Of course, ACF and PACF plots are more of a guide than a definitive answer. In the Box–Jenkins approach, we use them to generate a handful of candidate models (e.g., ARIMA(1,1,0), ARIMA(0,1,1), ARIMA(1,1,1)). The next step is to compare these candidates more formally using statistical criteria such as the **Akaike Information Criterion (AIC)** and the **Bayesian Information Criterion (BIC)**.

AIC and BIC are statistical tools that help us balance goodness of fit against model complexity. In both cases, lower values indicate a better model.

a) AIC: Rewards goodness of fit but penalizes adding too many parameters.

$$\text{AIC} = 2k - 2ln(L)$$

b) BIC: Similar to AIC, but applies a stronger penalty for model complexity. This means BIC tends to favor simpler models compared to AIC.

$$\text{BIC} = k \times ln(n) - 2ln(L)$$

Where k = he number of parameters,
n = the sample size,
L = likelihood of the model, i.e., the probability of observing the data given the parameters (in practice, we use the log-likelihood)

AIC is especially useful when the goal is predictive accuracy. It tends to be more forgiving about adding extra parameters, making it helpful when the true data-generating process is unknown. BIC, on the other hand, favors parsimony. By imposing a stronger penalty for complexity, it often selects simpler models—particularly in large datasets, where overfitting is a bigger concern.

Using these criteria, we compared several candidate ARIMA models:

```
ARIMA(1, 1, 0) - AIC: 1239.92, BIC: 1246.97
ARIMA(0, 1, 1) - AIC: 1239.83, BIC: 1246.87
ARIMA(1, 1, 1) - AIC: 1241.76, BIC: 1252.33
ARIMA(2, 1, 1) - AIC: 1243.09, BIC: 1257.17
```

Among these, ARIMA(0,1,1) has the lowest AIC and BIC values, making it the best candidate for modeling Apple stock prices. This will serve as the foundation for building our final ARIMA model.

Can We Build It? Yes, We Can!

With Python, initializing and training an ARIMA model is remarkably straightforward – it takes just two lines of code:

```
model = ARIMA(data, order=(p, d, q))
model_fit = model.fit()
```

The first line initializes the model with your chosen parameters, and the second line fits it to the data. That's it! But as with most things in data science, the simplicity hides a lot of important math under the hood. Statsmodels makes ARIMA modeling feel like a "black box," but to really trust it, we need to peek inside and understand what's going on.

Let's take a closer look at what our ARIMA arguments mean.

p = 0: no autoregressive term. The differenced series doesn't directly depend on its past values.

d = 1: one level of differencing

q = 1: one moving average term, so the differenced price depends on yesterday's error:

$$Y'_t = \theta_1 \times \varepsilon_{t-1} + \varepsilon_t$$

Where $\theta_1 \varepsilon_{t-1}$ is yesterday's error term, and ε_t is white noise, or the unpredictable component of price movements. Why does this matter? For financial time series, this setup makes sense: today's change in price is mostly driven by unexpected shocks (earnings surprises, news, market moves), not a predictable continuation of yesterday's trend.

When we call .fit() in statsmodels, Python doesn't just plug values into this formula. Instead, it estimates the parameter theta-1 and the error variance (sigma-squared) using Maximum Likelihood Estimation (MLE)[2]. Here's the general process:

1. Differencing: Since $d = 1$, the raw series is differenced once
2. Likelihood function: Assuming the residuals are normally distributed, the model defines a likelihood — the probability of observing the data given candidate values of theta-1 and sigma-squared. The log-likelihood is:

$$\ln L = -\frac{n}{2}\ln(2\pi\sigma^2) - \frac{1}{2\sigma^2}\sum_{t=1}^{n}(Y_t' - \theta_1 \times \varepsilon_{t-1})^2$$

3. Optimization: The algorithm searches for parameter values that maximize this likelihood. Statsmodels typically uses quasi-Newton optimization methods (like BFGS).
4. Output: Once the log-likelihood is maximized (a.k.a. convergence), you get:
   - parameter estimates
   - standard errors & significance levels
   - fit diagnostics like AIC, BIC, and the final log-likelihood

So, behind the two lines of code is an optimization engine finding the most likely ARIMA model given your data.

Prediction

Now that we've peeked inside the black box, let's see what all this math actually gives us in practice. Once the ARIMA model is trained, forecasting is just as simple. Python provides .forecast() or .predict() to generate expected future values. For example, forecasting Apple's next 30 closing prices:

```
forecast = model_fit.forecast(steps=30)
```

As seen below, the model in the code file tells us the predicted price for the next 30 days is $251.08:

[2] Scroll to MLE Section

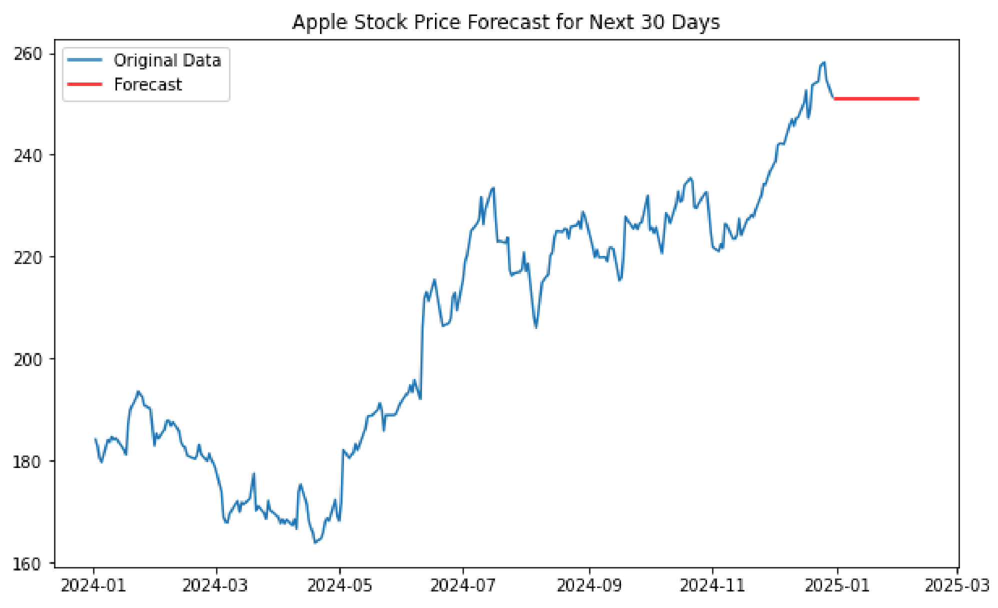


Figure 9

Notice something strange? The forecast is a flat line. That's not a bug — it's how ARIMA(0,1,1) works. Since the model is built on differences, it assumes future changes are essentially random (white noise). When we add those differences back up, the forecast just sticks to the most recent observed level. This is the statistical honesty of ARIMA: it won't try to predict if Apple is going to skyrocket or crash — only that it's equally likely to go either way from here.

Evaluation and Interpretation

Let's take a look at the summary table from our Apple stock price model:

```
                               SARIMAX Results
==============================================================================
Dep. Variable:                   AAPL   No. Observations:                  251
Model:                 ARIMA(0, 1, 1)   Log Likelihood                -617.914
Date:                Wed, 10 Sep 2025   AIC                           1239.829
Time:                        15:16:28   BIC                           1246.872
Sample:                             0   HQIC                          1242.663
                                - 251
Covariance Type:                  opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ma.L1          0.0713      0.057      1.250      0.211      -0.040       0.183
sigma2         8.2105      0.482     17.021      0.000       7.265       9.156
===================================================================================
Ljung-Box (L1) (Q):                   0.03   Jarque-Bera (JB):                94.36
Prob(Q):                              0.87   Prob(JB):                         0.00
Heteroskedasticity (H):               1.47   Skew:                             0.20
Prob(H) (two-sided):                  0.08   Kurtosis:                         5.98
===================================================================================
```

At first glance, the header says SARIMAX Results. Don't worry, we didn't sneak in seasonality or exogenous variables. Statsmodels just uses a general SARIMAX engine to fit ARIMA models. Since our order is (0,1,1), this is still a plain ARIMA model. The dataset contains 251 daily observations of Apple's closing price, labeled as AAPL.

Our model reports a log-likelihood of –617.9. You can think of log-likelihood as a kind of "surprise score": the higher (closer to zero), the less shocked the model is by the data. Since it's fairly negative, that tells us Apple's daily price changes are still hard to predict, which is no surprise in finance.

We also get three common information criteria: AIC = 1239.8, BIC = 1246.9, and HQIC = 1242.7, which stands for Hannan–Quinn Information Criterion, a score similar to AIC and BIC. All three balance model fit against complexity, with lower values being better. We used these earlier to compare candidate models and found ARIMA(0,1,1) to be most efficient.

Moving on to the parameter estimates: ma.L1 (theta-1, moving average coefficient) is 0.07. Its p-value shows it's *not statistically significant*. In plain English, Apple's daily moves aren't strongly tied to yesterday's "shock" (error). Parameter sigma2 is the variance of the random shocks, or the standard deviation (std dev) of residual dollars squared. Taking the square root gives a residual standard deviation of ≈ $2.87. Compare that to the raw Apple stock data, which had a much larger daily standard deviation of ≈ $25.5. That's a big win — ARIMA has explained most of the trend and short-term autocorrelation, leaving behind a much smaller, cleaner noise term.

Next come the diagnostic checks. Here's where we test whether the leftover residuals behave like random white noise:

- Ljung-Box test (p = 0.87): No leftover autocorrelation. That's good! The model isn't missing major short-term dependencies.
- Jarque-Bera test (p = 0.00): Residuals aren't normally distributed. Instead, they have fat tails, meaning big moves happen more often than a bell curve would predict.

- Skew = 0.20, Kurtosis = 5.98: Confirms the story: residuals are slightly right-skewed and far more "peaked" than a normal bell curve. Big swings are more common than ARIMA assumes.

This causes a problem; you'll recall that we assumed the residuals of the differenced series are normally distributed, allowing us to use MLE for coefficient estimation. The good news is ARIMA can still capture the main structure of the series, but the bad news is the confidence intervals may be too optimistic. In practice, analysts often combine ARIMA with volatility models like GARCH or use fatter-tailed error distributions to reflect better the real world, where big surprises happen more often than a bell curve would predict.

- Heteroskedasticity test ($p = 0.08$): Borderline evidence that variance may not be constant over time. This hints at **volatility clustering** (calm periods followed by stormy ones), a well-known feature of financial markets.

Beyond statistical diagnostics, it's also important to assess how well the model's forecasts line up with the actual stock prices. For that, we can turn to familiar regression error metrics:

- Mean Absolute Error (MAE): 10.43
- Mean Squared Error (MSE): 176.02
- Root Mean Squared Error (RMSE): 13.27

MAE tells us that, on average, the model's predictions are about \$10 away from the true Apple closing price. RMSE, being in the same units as the stock price, gives a sense of the "typical" prediction error, which is around \$13. Both values are reasonable when compared to Apple's average price level of about \$206, but they still highlight that daily fluctuations are not being captured perfectly. MSE, which penalizes larger errors more heavily, underscores that some days see bigger deviations than the model can account for, consistent with the fat tails we observed in the residual diagnostics.

Taken together, these metrics confirm what the residual analysis already hinted at: ARIMA can handle short-term dependencies and reduce noise, but it doesn't fully capture the wild swings that characterize financial markets. This makes it a useful, but limited, tool for stock forecasting.

Conclusion

The ARIMA model offers a practical framework for forecasting Apple's stock prices. By differencing once, we successfully removed the strong upward trend in the raw data and achieved stationarity. The model also reduced the daily residual volatility from over $25 to about $2.87, showing that ARIMA can effectively strip away trend and short-term autocorrelation.

At the same time, our diagnostics highlighted the limits of ARIMA when applied to financial time series. The moving average parameter was not statistically significant, suggesting that yesterday's shocks do not strongly explain today's price changes. More importantly, the residuals displayed fat tails and signs of heteroskedasticity, meaning that Apple's returns experience more frequent extreme moves than a normal distribution would predict. These features are common in financial markets and fall outside ARIMA's assumptions.

In short, ARIMA models are valuable tools for forecasts. They clarify how much of the data can be explained by simple autoregressive and moving average structures, and they provide a disciplined statistical framework for testing stationarity, model fit, and predictive power. For financial data in particular, ARIMA is best complemented with models that explicitly handle volatility dynamics or with more flexible approaches like machine learning models.

In this post, we have learned:

- What ARIMA models are
- How to use ACF, PACF plots as part of Box-Jenkins
- How to compare candidate ARIMA models with AIC, BIC
- How to build an ARIMA model in Python, and predict future values
- How to interpret ARIMA output, including coefficients, variance, and log-likelihood
- The limitations of ARIMA in financial time series
- Bonus: How to plot time series data with forecasts using matplotlib

That's a wrap on ARIMA models! Stay tuned to learn what's inside the black box of the k-nearest neighbors model — a machine learning algorithm.