Table of contents

Neptune Blog

How to Select a Model For Your Time Series Prediction Task [Guide]

Joos Korstanje

13 min

24th March, 2025

ML Model Development | Time Series

Are you working with time series data and seeking the most effective models? This guide explains how to select and evaluate time series models based on predictive performance—including classical, supervised, and deep learning-based models.

After comparing models and selecting the right one for our task, we'll build models for stock market forecasting, benchmarking each to identify the best-performing approach.

Understanding time series datasets and forecasting

Most data sets that practitioners work with are based on independent observations. For example, given a website, you could track each visitor; each data point (e.g. row in a table) would represent an individual observation about

each visitor. If we assign each visitor a “User ID,” each ID will be independent

Table of contents

User ID	Variable 1
User_1	1
User_2	2
User_3	3
User_4	4
User_5	5

=

User ID	Variable 1
User_67	1
User_12	2
User_31	3
User_42	4
User_35	5

Example of a dataset with independent observations | Source: Author

In contrast, time series data are unique because they measure one or more variables as they change over time, creating dependencies between data points. Unlike typical datasets with independent observations, each time point in a time series dataset is related to its predecessors. This impacts the choice of machine learning algorithms we should use.

Timestamp	Variable 1
1	1
2	2
3	3
4	4
5	5

≠

Timestamp	Variable 1
67	1
12	2
31	3
42	4
35	5

Example of a dataset with dependent observations taken over time | Source: Author

Related

Time Series Prediction: How Is It Different From Other Machine Learning? [ML Engineer Explains]

Read more →

Key aspects of time series modeling

Before we dive into the models themselves, let's make sure we have a good understanding of the nature of time series data and how modeling them differs

Table of contents

Univariate versus multivariate time series models

In time series data, timestamps hold intrinsic meaning. Univariate time series models use only one variable (the target variable) and its variation over time to make future predictions.

In contrast, multivariate time series models include additional variables. For instance, if you want to forecast product demand, you might consider including weather data as an influencing factor. Multivariate models extend univariate models by integrating these additional (or external) variables.

Univariate time series models	Multivariate time series models
Use only one variable	Use multiple variables
Cannot use external data	Can use external data
Based only on relationships between past and present	Based on relationships between past and present, and between variables

Suppose you want to look at the patterns (or changes over time) within your data to understand it better and make predictions. In this case, you need to understand the temporal variations you'll encounter: seasonality, trend, and noise. The next topic we'll discuss, time series decomposition, is a technique used to separate these components so you can analyze them individually.

Time series decomposition

You can decompose the time series to extract different types of variation from your dataset. This will extract three key features from your data:

- **Seasonality** is a recurring pattern based on time periods (such as seasons of the year). For example, temperatures typically rise in the summer and fall in the winter. You can use this predictable pattern to help predict future values.
- **Trends** reflect long-term increases or decreases in your data. Going back to our temperature example, you could observe a gradual upward trend due to global warming, layered on top of seasonal variations.

- **Noise** is random variability that doesn't follow seasonality or trend. It

Table of contents

Time series decomposition in Python

Here's a quick example of how to decompose a time series in Python using the CO2 dataset from the `statsmodels` library.

Before diving into the code, make sure you have the required dependencies installed. Run the following commands to set up your environment:

```
# Install the relevant libraries
!pip install numpy pandas matplotlib statsmodels scikit-l
```

Now, import the dataset:

```
# Import the CO2 dataset
import statsmodels.datasets.co2 as co2

co2_data = co2.load().data
print(co2_data)
```

The dataset includes a time index (weekly dates) and CO2 measurements, shown below.

```
                co2
Table of contents

1958-04-12  317.6
1958-04-19  317.5
1958-04-26  316.4
...           ...
2001-12-01  370.3
2001-12-08  370.8
2001-12-15  371.2
2001-12-22  371.3
2001-12-29  371.5

[2284 rows x 1 columns]
```

Example measurements and timestamps from the statsmodels CO2 dataset | Source: author

There are a few missing (NA) values. To handle these, you can use interpolation like this:

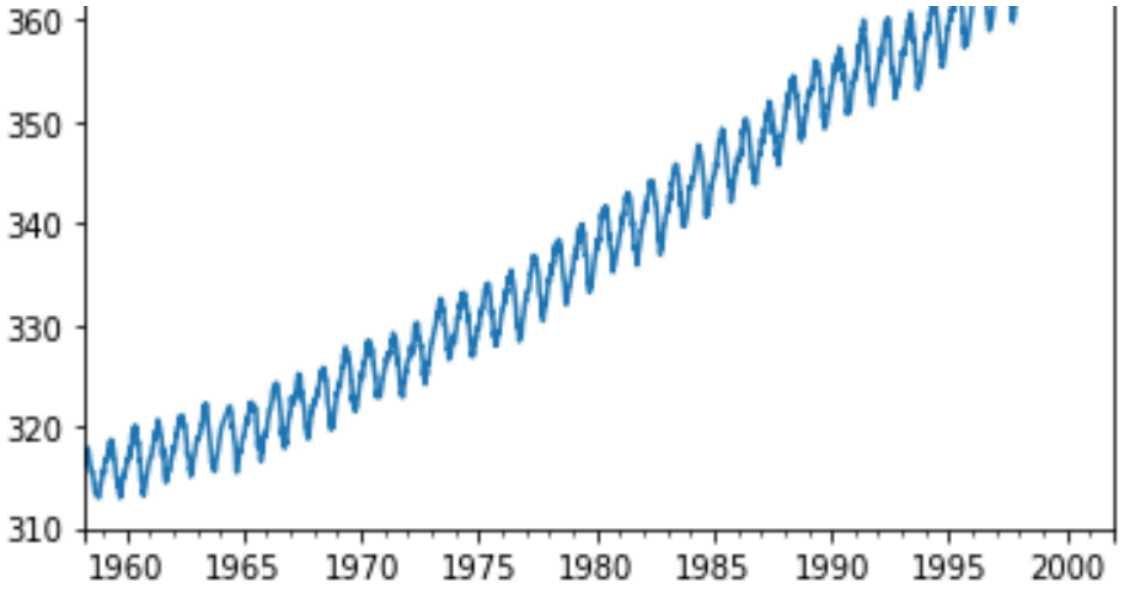
```
# Handle missing values with interpolation
co2_data = co2_data.fillna(co2_data.interpolate())
```

Next, plot the CO2 values over time to see the temporal trend:

```
# Plot CO2 values over time
co2_data.plot()
```

This will generate the following plot:

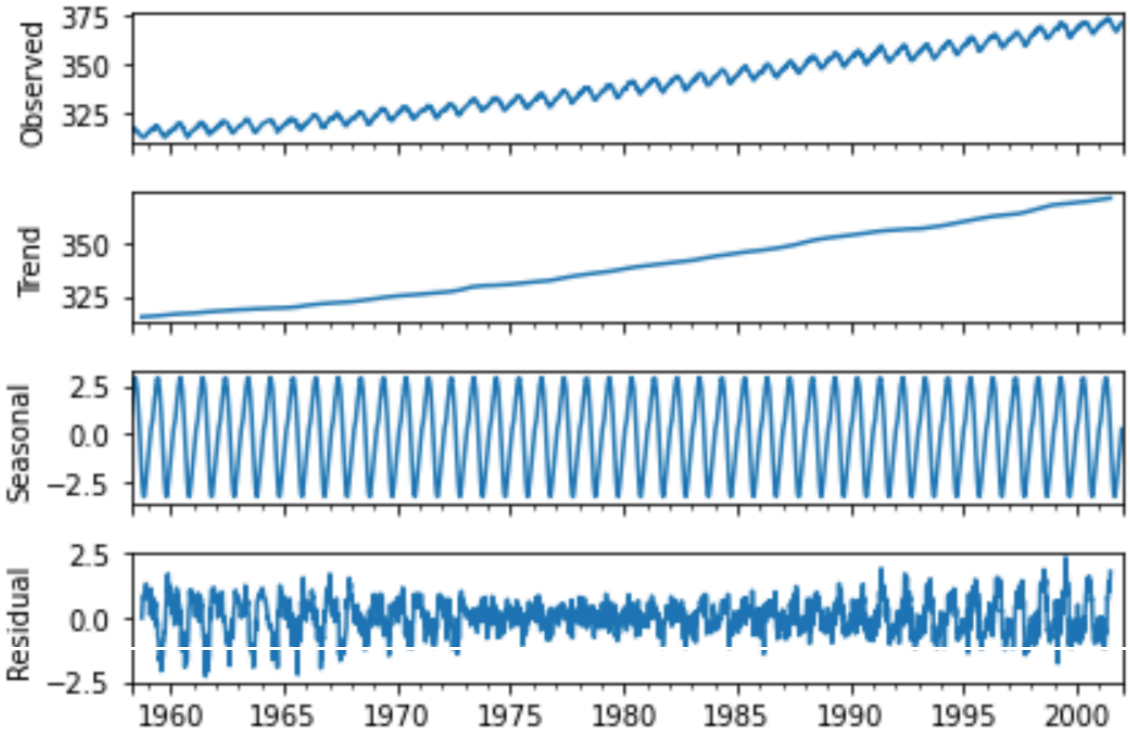
Table of contents



Plot of the CO2 time series from the statsmodels CO2 dataset | Source: Author

To decompose the time series into trend, seasonality, and noise (labeled as "residual"), use the `seasonal_decompose` function from `statsmodels`:

```
from statsmodels.tsa.seasonal import seasonal_decompose
result = seasonal_decompose(co2_data)
result.plot()
```



The CO2 time series decomposed into seasonality, trend, and residual (noise) | Source: Author

Table of contents

first plot) and strong seasonality (see the pattern in the third plot).

Autocorrelation

Autocorrelation is another key temporal feature in time series data. It measures how the current value of a time series correlates with past values, allowing for more accurate predictions based on recent trends.

Autocorrelation can be:

1. **Positive:** High values tend to be followed by other high values, and low values by low values. For example, in the stock market, a rising stock price often attracts more buyers, driving the price up further; when the price falls, many people usually sell, driving the price down.
2. **Negative:** High values are likely to be followed by low values, and vice versa. For example, a high rabbit population in the summer may deplete resources, leading to a lower population in the winter, allowing resources to recover and the population to increase again the following year.

Detecting autocorrelation

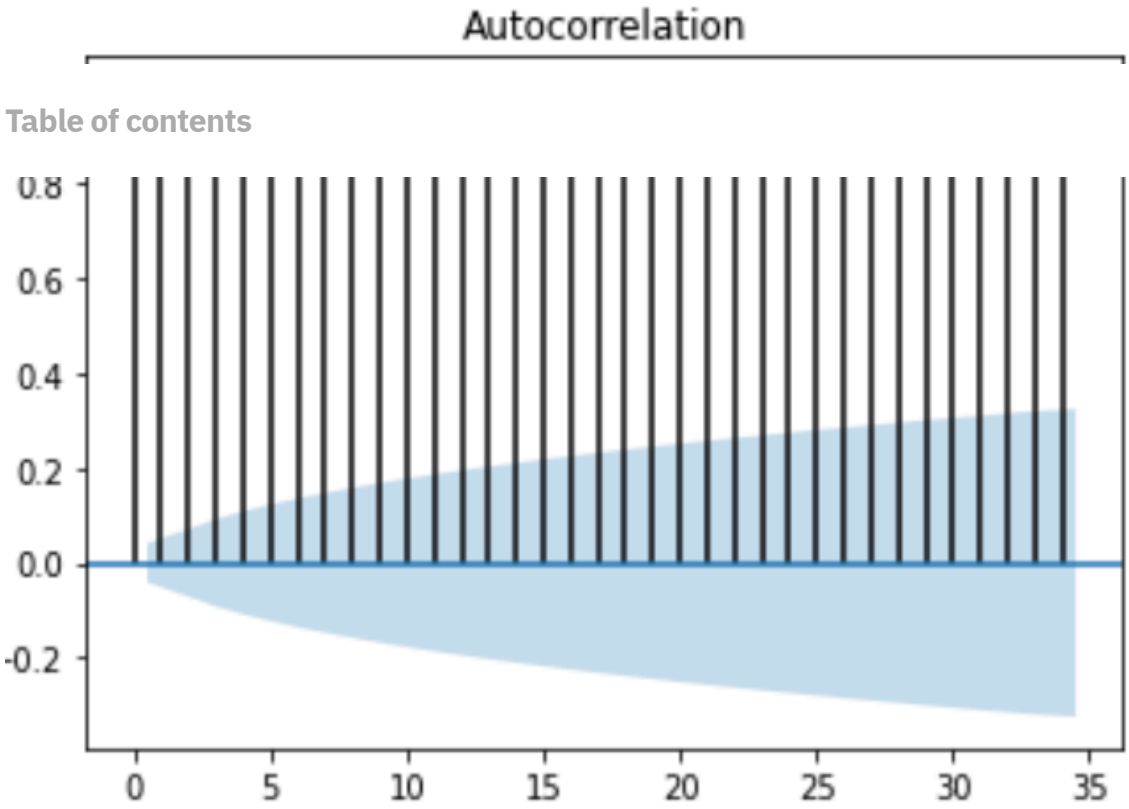
Two common tools for detecting autocorrelation are:

- The autocorrelation function (ACF) plot
- The partial autocorrelation function (PACF) plot

You can compute an ACF plot using Python as follows:

```
from statsmodels.graphics.tsaplots import plot_acf

plot_acf(co2_data)
```

For our CO2 dataset, this is what we get:



Autocorrelation plot for the CO2 dataset | Source: Author

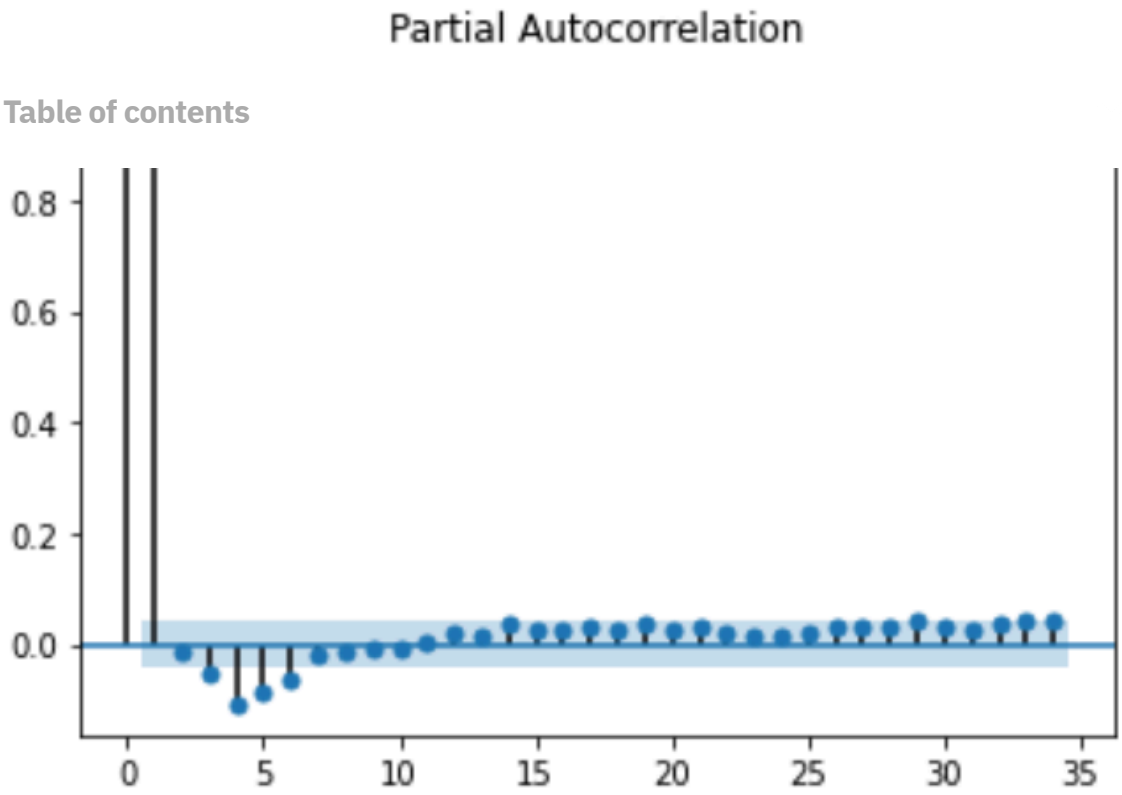
On the x-axis, you see the time steps (or “lags”) going back in time. On the y-axis, you can see the correlation of each time step with the current time. This plot clearly shows significant autocorrelation.

The PACF (partial autocorrelation function) is an alternative to the ACF. Instead of showing all autocorrelations, it shows only the *unique* correlation at each time step, filtering out indirect effects. This helps identify the true relationship between each lag and the present time.

For example, if today’s value is similar to yesterday’s and the day before, the ACF would show high correlations for both days. The PACF, however, would only show yesterday’s value as correlated, removing redundant correlations from earlier days.

You can compute a PACF plot in Python as follows:

```
from statsmodels.graphics.tsaplots import plot_pacf

plot_pacf(co2_data)
```



Partial autocorrelation plot for the CO2 dataset | Source: Author

The PACF plot provides a clearer view of the autocorrelation in the CO2 data. It shows strong positive autocorrelation at lag 1, meaning a high value now likely indicates a high value in the next time step. The PACF only displays direct correlations, avoiding duplicate effects from earlier lags. This results in a cleaner and more straightforward representation.

Stationarity

Stationarity is another important concept in time series analysis. It means a series has no trend, meaning its statistical properties, like mean and variance, remain constant over time. Many time series models require stationarity to work effectively.

To check for non-stationarity, you can use the Dickey-Fuller Test.

Dickey-Fuller test

The Dickey-Fuller test is a statistical test that detects non-stationarity in a time series. Here's how to apply it to the CO2 data in Python:

```
from statsmodels.tsa.stattools import adfuller
adf, pval, usedlag, nobs, crit_vals, icbest = adfuller(co
```

```
3
4 print('ADF test statistic:', adf)
```

Table of contents

```
8 print('Number of observations:', nobs)
9 print('Critical values:', crit_vals)
  print('Best information criterion:', icbest)
```

The result looks like this:

```
ADF test statistic: 0.03378459745826506
ADF p-values: 0.9612384528286108
ADF number of lags used: 27
ADF number of observations: 2256
ADF critical values: {'1%': -3.4332519309441296, '5%': -2.8628219967376647, '10%': -2.567452466810334}
ADF best information criterion: 2578.39090925253
```

Results of the Dickey-Fuller Test for the CO2 data in Python. The ADF test suggests that the time series is non-stationary, as the test statistic (0.0337) is greater than the critical values, and the p-value (0.9612) is much higher than 0.05, failing to reject the null hypothesis of non-stationarity.

In the ADF test:

- The **null hypothesis** assumes a unit root is present, meaning the series is non-stationary.
- The **alternative hypothesis** suggests that the time series is stationary.

If the p-value is below 0.05, you can reject the null hypothesis, suggesting that the data is stationary. We cannot reject the null hypothesis if the p-value is above 0.05 (as we see above), meaning the data is likely non-stationary. This aligns with the trend we saw above in the CO2 data.

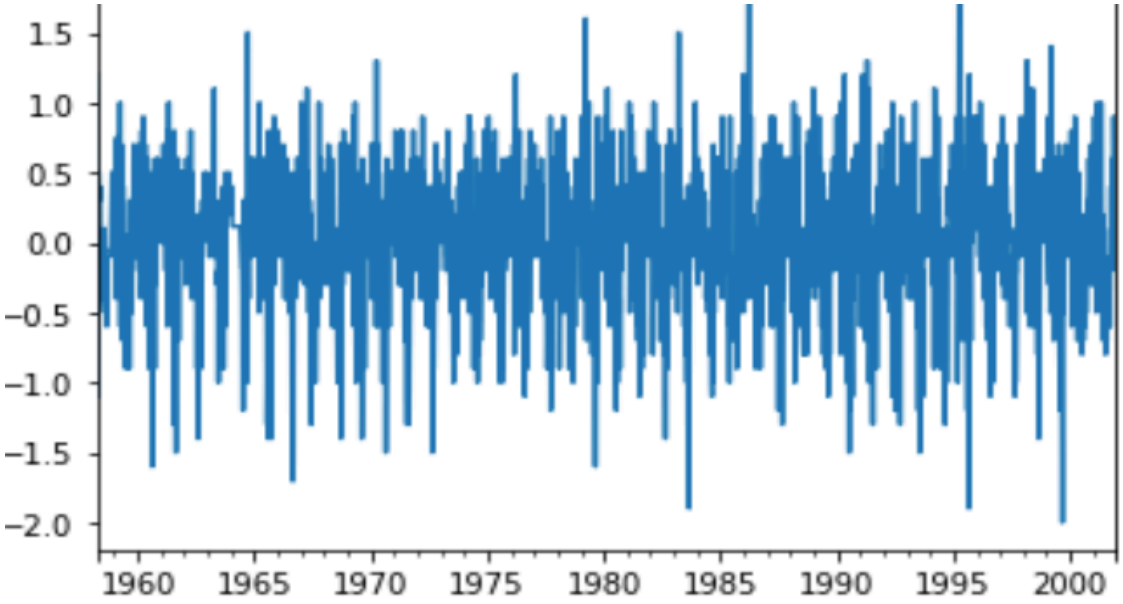
Differencing

To make a non-stationary series stationary, you can apply differencing, which removes trends and leaves only seasonal variations. This helps when using models that assume stationarity.

```
1 # Apply differencing to remove the trend
2 prev_co2_value = co2_data.co2.shift()
3 differenced_co2 = co2_data.co2 - prev_co2_value
4 differenced_co2.plot()
```

The differenced CO2 data looks like this:

Table of contents



The CO2 time series after applying differencing | Source: Author

Now, if we do the ADF test again on this differenced data, we can confirm that it has become stationary:

```
from statsmodels.tsa.stattools import adfuller
adf, pval, usedlag, nobs, crit_vals, icbest = adfuller(di

print('ADF test statistic:', adf)
print('ADF p-value:', pval)
print('ADF number of lags used:', usedlag)
print('ADF number of observations:', nobs)
print('ADF critical values:', crit_vals)
print('ADF best information criterion:', icbest)
```

```
ADF test statistic: -15.727522408375831
ADF p-values: 1.3013480157810985e-28
ADF number of lags used: 27
ADF number of observations: 2255
ADF critical values: {'1%': -3.4332532193008443, '5%': -2.862822565622804, '10%': -2.5674527697012306}
ADF best information criterion: 2556.277973363455
```

Results of the Dickey-Fuller Test after applying differencing. The ADF test suggests that the time series is stationary, as the test statistic is smaller than the critical values and the p-value is much smaller than 0.05, so we can reject the null hypothesis of non-stationarity.

Now, the p-value is very small, indicating that we can reject the null hypothesis (non-stationarity) and assume that this data is stationary.

Table of contents

One-step vs multi-step time series models

Before diving into modeling, the final concept we should cover is the difference between one-step and multi-step models.

- **One-step models** predict only the next time point in a series. To create multi-step forecasts with these models, you can repeatedly use the previous prediction as input for the next step. However, this approach can extend any existing errors over multiple steps.
- **Multi-step models** are designed to predict multiple future points simultaneously. These models are generally better for long-term forecasts and perform well for single-step forecasts.

Choosing between one-step and multi-step models depends on how many steps you need to predict for your use case.

One-step forecasts	Multi-step forecasts
Designed to forecast only one step ahead	Designed to forecast multiple steps ahead
Can be extended to multi-step by windowing	Direct multi-step capability
May be less accurate for multi-step forecasts	Ideal for multi-step forecasts

Types of time series models

Now that we've covered key aspects of time series data, let's explore the types of models used for forecasting.

Classical time series models

These traditional models, such as ARIMA and Exponential Smoothing, are based on time-based patterns in a time series. While highly effective for forecasting single-variable (univariate) series, some advanced options exist to add external variables as well.

Supervised models

Supervised models are a family of models used for many different machine learning tasks. They use clearly defined input (X) and output (Y) variables.

For time series forecasting, you can create input features from date-based elements (e.g., year, month, day), and the target to be predicted is the value of your time series at that date. You can also include lagged values to add autocorrelation effects.

Deep learning models

The rise of deep learning has enabled new forecasting methods, especially useful for complex, sequential data. Specific model architectures like LSTMs have been developed and applied for sequence-based forecasting.

Major tech companies like Facebook and Amazon have released open-source forecasting tools, offering powerful new options for practitioners. These can sometimes outperform traditional models.

Classical time series models

Now, let's dive deeper into classical time series models, starting with the ARIMA family, which combines multiple components to create a robust forecasting model.

ARIMA family

The ARIMA family of models consists of a set of smaller models that can be used on their own or combined (when all of the individual components are put together, you obtain the SARIMAX model). The main building blocks are:

1. Autoregression (AR): Uses past values to predict future ones. The order of an AR model, *p*, indicates the number of previous time steps included; the simplest model is the AR(1) model, which only uses one previous timestep to predict the current value.

2. Moving average (MA): Predicts future values based on past prediction errors

Table of contents

only the last error.

3. Autoregressive Moving Average (ARMA): Combines AR and MA, using both past values and past errors for predictions. ARMA can use different lags for either the AR or MA; for example, ARMA(1, 0) has an order of p=1 and q=0, effectively making it a regular AR(1) model. The ARMA model requires a stationary time series.

4. Autoregressive Integrated Moving Average (ARIMA): Extends ARMA by adding differencing (indicated by d) to make the series stationary, if necessary. The notation is ARIMA(p, d, q). For example, an ARMA(1, 2) model that needs to be differenced once would become an ARIMA(2, 1, 2) model. The first 2 is for the AR order, the second 1 is for the differencing, and the third 2 is for the MA order. ARIMA(1, 0, 1) would be the same as ARMA(1, 1).

5. Seasonal ARIMA (SARIMA): Adds seasonality to ARIMA, with seasonal parameters (P, D, Q) on top of the non-seasonal parameters (p, d, q). If seasonality is present in your time series, using it in your forecast is critical. The frequency m specifies the seasonal period (e.g., 12 for monthly data, or 4 for quarterly data). SARIMA notation is SARIMA(p, d, q)(P, D, Q)m.

Related

ARIMA & SARIMA: Real-World Time Series Forecasting

Read more →

6. Seasonal autoregressive integrated moving-average with exogenous regressors (SARIMAX)

6. SARIMA with Exogenous Variables (SARIMAX): Adds external variables (X) to SARIMA, allowing additional features to improve forecast accuracy. This is the most complex variant, combining AR, MA, differencing, and seasonal effects, along with the addition of external variables.

Example: using auto-ARIMA in Python on CO2 Data

Now that we've reviewed the building blocks of the ARIMA family, let's apply them to create a predictive model for CO2 data.

Choosing the right parameters for ARIMA or SARIMAX models can be challenging, as there are many combinations of (p, d, q) or (p, d, q)(P, D, Q).

While you can inspect autocorrelation graphs to make educated guesses, the pmdarima library provides an auto_arima function to automatically select

Table of contents

First, import `pmdarima` and other necessary libraries:

```
import pmdarima as pm
from pmdarima.model_selection import train_test_split
import matplotlib.pyplot as plt
```

After installation, split the data into training and testing sets (we'll go into why in more detail later on):

```
train, test = train_test_split(co2_data.co2.values, train
```

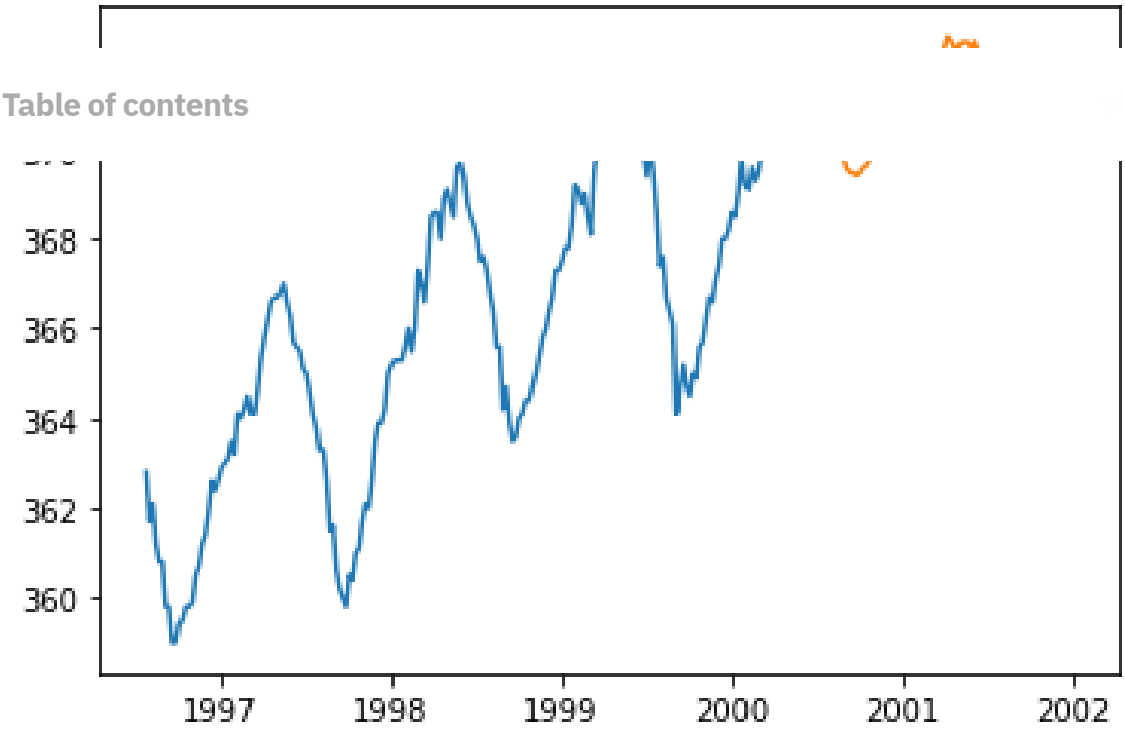
Next, fit the model using `auto_arima` on the training data with seasonal parameters, then make predictions with the best-selected model:

```
model = pm.auto_arima(train, seasonal=True, m=52)
preds = model.predict(test.shape[0])
```

Finally, visualize the actual vs. forecasted data:

```
plt.plot(co2_data.co2.values[:2200], train)
plt.plot(range(2200, 2200 + len(preds)), preds)
plt.legend()
plt.show()
```

Table of contents



In the plot, the blue line represents the actual data, and the orange line represents the forecast. | Source: Author

For more examples and details, check the pmdarina documentation.

Vector autoregression (VAR) and its variants: VARMA and VARMAX

Vector Autoregression (VAR) is a multivariate alternative to ARIMA, designed to predict multiple time series simultaneously. This is especially useful when strong relationships exist between series, as VAR models only the autoregressive component for multiple variables.

- **VARMA**: The multivariate equivalent of ARMA, adding a moving average component to VAR, allowing it to model both past values and errors across multiple series.
- **VARMAX**: Extends VARMA by adding exogenous (external) variables (X), which can improve forecasting accuracy without needing to be forecasted themselves. The `statsmodels` VARMAX implementation is a good way to get started with multivariate forecasting with external factors.

Advanced versions like Seasonal VARMAX (SVARMAX) also exist but can become highly complex, making implementation and interpretation challenging. In practice, simpler models may be preferable.

Smoothing techniques

Exponential smoothing is a statistical technique that helps to reduce short-term noise in time series data, making long-term patterns more visible. Time

series patterns often have a lot of long-term variability and short-term (noisy) variability. Smoothed time series can reveal trends more effectively for

Table of contents

1. Simple moving average: Replaces the current value with an average of the current and past values. Increasing the number of past values smooths the series further, but reduces detail. This is the simplest smoothing technique.

2. Simple exponential smoothing (SES): An adaptation of the moving average that applies weights to past values so that recent values have more influence. This approach smooths the series without losing as much detail as a simple moving average.

3. Double exponential smoothing (DES): Suitable for data with trends, DES uses two parameters—α (the data smoothing factor) and β (the trend smoothing factor)—to adjust for trends in the data. This method addresses cases where SES alone would fall short by recursively applying an exponential filter.

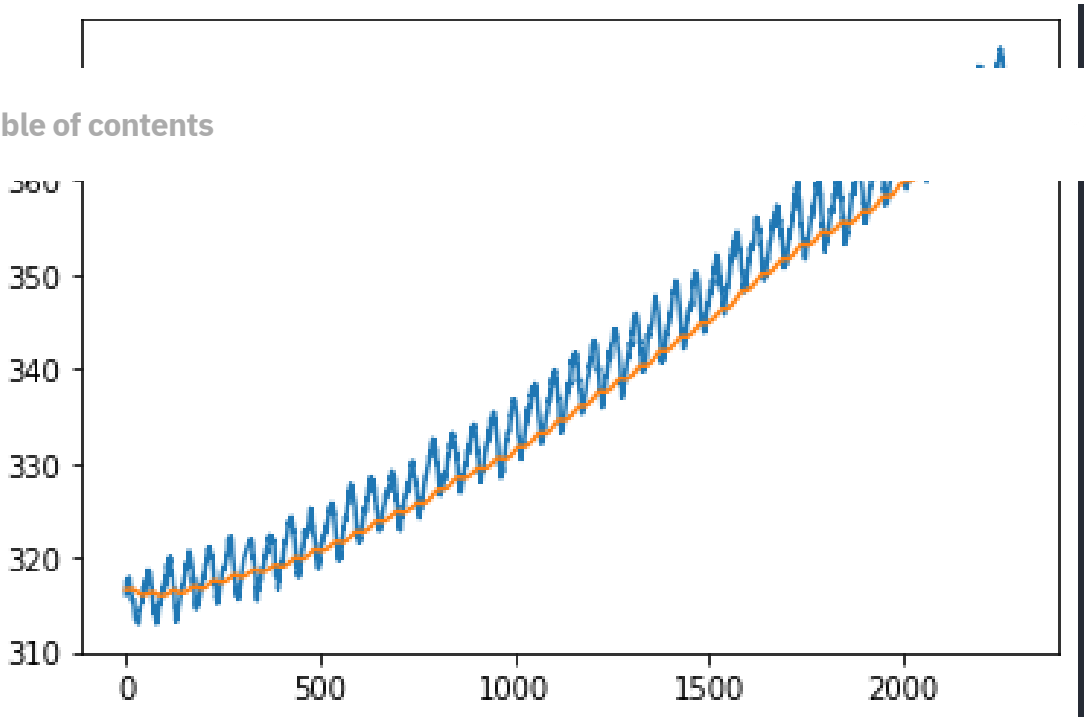
4. Holt-Winters Exponential Smoothing (HWES): Also known as Triple Exponential Smoothing, HWES is ideal for data with seasonality and trend. It adjusts for three components—trend, seasonal cycles (e.g., weekly or monthly), and noise.

Example: Exponential smoothing in Python

Here's how to apply simple exponential smoothing (SES) to our CO2 data:

```
from statsmodels.tsa.api import SimpleExpSmoothing
import matplotlib.pyplot as plt

es = SimpleExpSmoothing(co2_data.co2.values).fit(smoothin

plt.plot(co2_data.co2.values)
plt.plot(es.predict(es.params, start=0, end=None))
plt.legend()
plt.show()
```

The smoothing level indicates how smooth your curve should become. In this example, it's set very low, indicating a very smooth curve. Feel free to play around with this parameter and see what less-smooth versions look like.



The blue line shows the original data, and the orange line shows the smoothed series | Source: Author

Supervised machine learning models

Supervised machine learning models categorize variables as either dependent (target) or independent (predictor) variables. While these models aren't designed for time series, they can be adapted by treating time-based features (e.g., year, month, day) as independent variables.

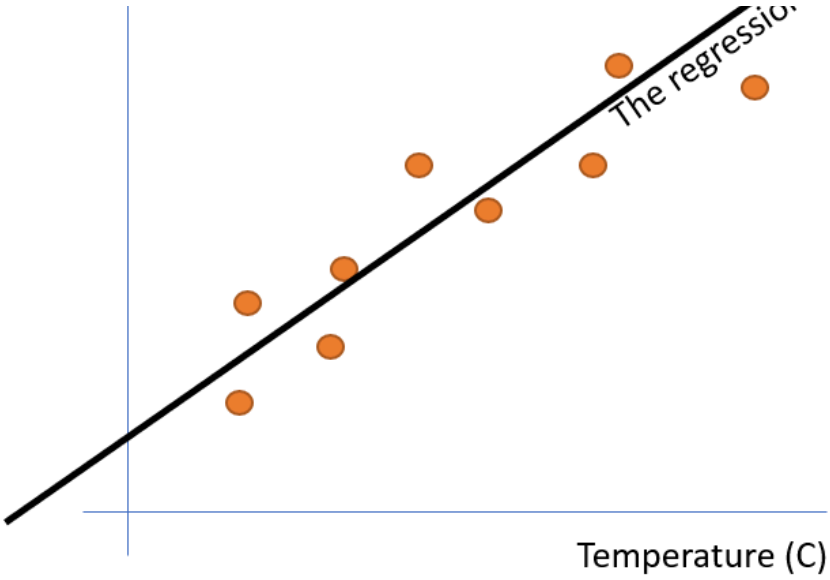
Linear regression

Linear Regression is the simplest supervised model. It estimates linear relationships: each independent variable has a coefficient that indicates how much it affects the target.

Multiple Linear Regression: Uses multiple predictors (e.g., temperature and price) to model the target variable.

Simple Linear Regression: Uses one independent variable. For example, hot chocolate sales can be modeled based on the temperature outside (pictured below).

Table of contents



An example of linear regression fit to data of hot chocolate sales according to outside temperature | Source: Author

This is not a time series dataset yet: no time variable is present. To make it a time series, we can add date-based variables such as year, month, or day instead of only using temperature and price as predictors. For example, tying this back to our CO2 dataset:

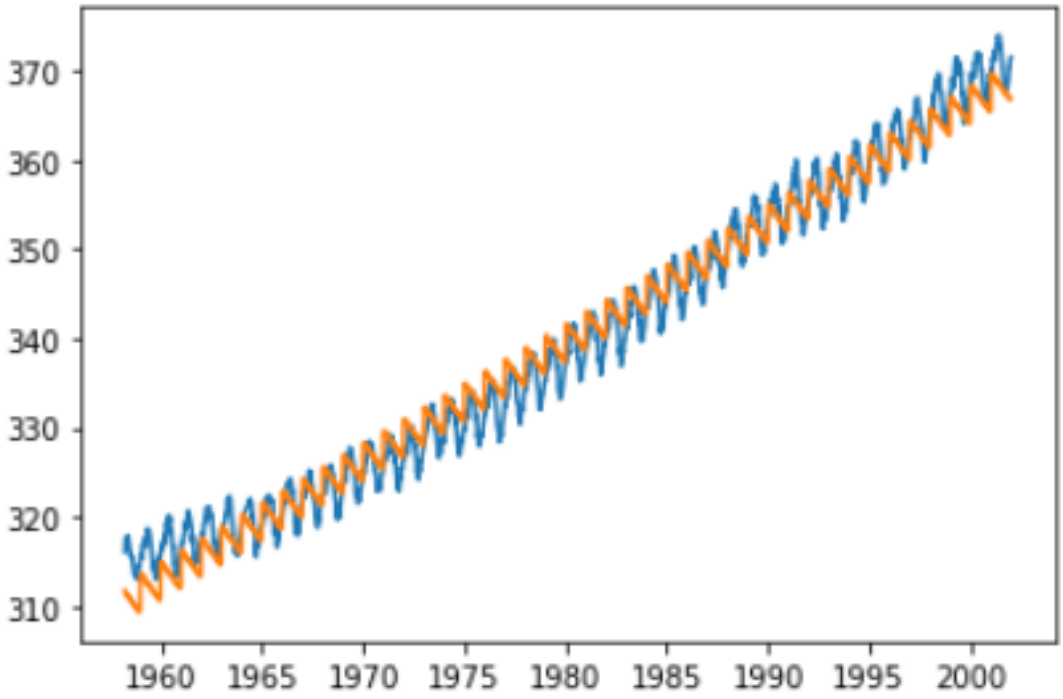
```
import numpy as np
from sklearn.linear_model import LinearRegression
import matplotlib.pyplot as plt

# Extract seasonality data
months = [x.month for x in co2_data.index]
years = [x.year for x in co2_data.index]
day = [x.day for x in co2_data.index]
X = np.array([day, months, years]).T

# Fit the Linear Regression model
my_lr = LinearRegression()
my_lr.fit(X, co2_data.co2.values)

# Make predictions
preds = my_lr.predict(X)

# Plot the results
plt.plot(co2_data.index, co2_data.co2.values, label="Actu
```

```
21  plt.plot(co2_data.index, preds, label="Predicted")
22  plt.legend()
```

Table of contents

We had to do a little bit of feature engineering to extract seasonality into variables, but the advantage is that adding external variables becomes much easier.

We used the `scikit-learn` library to build a linear regression model, fit it to our data, and make predictions. Let's see what our model learned:



The plot shows the fit of our linear regression model (in orange) to the CO2 data (presented in blue) | Source: Author

This is a pretty good fit!

Random forest

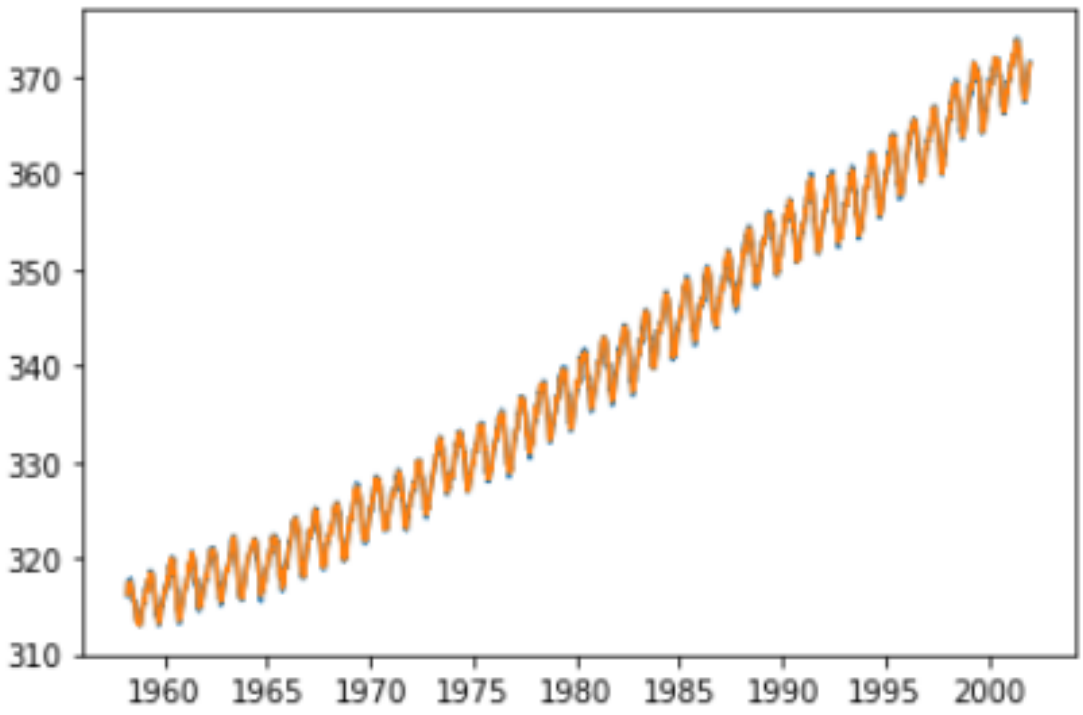
Linear Regression is limited to linear relationships. For more flexibility, Random Forest—a widely used model for nonlinear relationships—can provide a better fit.

The `scikit-learn` library has the RandomForestRegressor that you can simply use to replace the LinearRegression class in the previous code:

Table of contents

```
# Fit Random Forest
my_rf = RandomForestRegressor()
my_rf.fit(X, co2_data.co2.values)

# Make predictions
preds = my_rf.predict(X)

# Plot the results
plt.plot(co2_data.index, co2_data.co2.values, label="Actu
plt.plot(co2_data.index, preds, label="Predicted")
plt.legend()
plt.show()
```

The fit is now even better than before:



This plot demonstrates how our random forest model fits our CO2 dataset. In blue, the original data; in orange, the predicted values. As we can see, the fit is better than with linear regression | Source: Author

For now, it's enough to understand that this Random Forest model has been able to learn the training data better. Later, we'll cover more quantitative methods for model evaluation.

Related

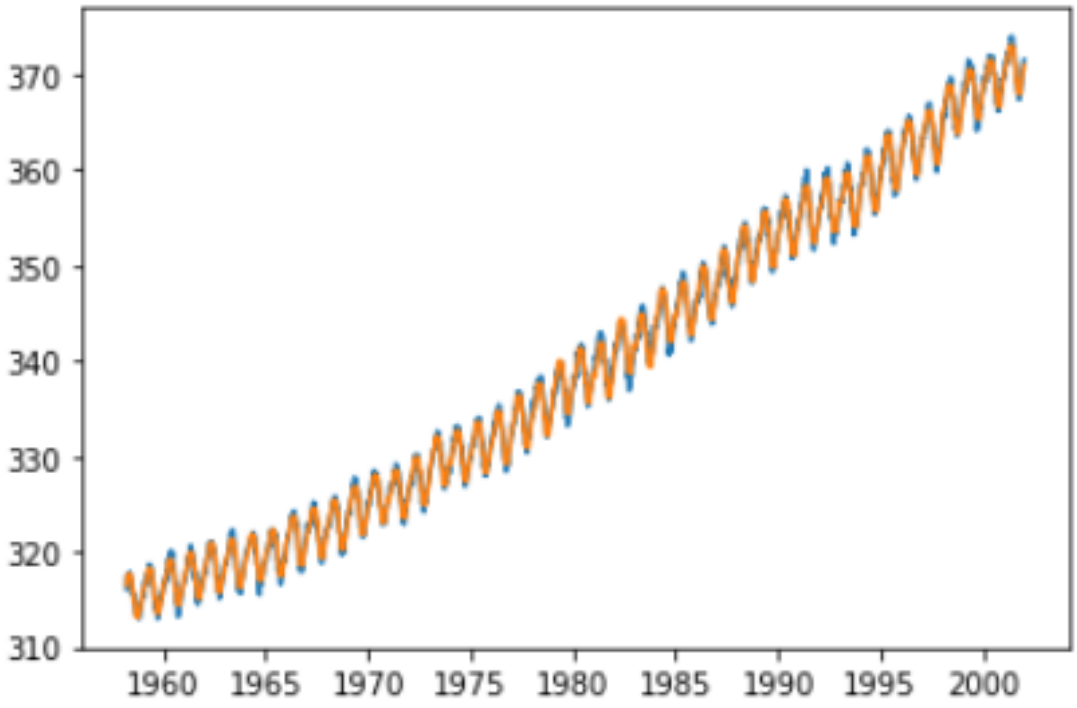
Random Forest Regression: When Does It Fail and Why?

Table of contents

XGBoost

XGBoost is another essential supervised model based on gradient boosting. It combines an ensemble of “weak learners”—like random forest—in sequence to minimize errors iteratively. It can perform parallel learning for efficiency.

```
import xgboost as xgb

# Fit XGBoost model
my_xgb = xgb.XGBRegressor()
my_xgb.fit(X, co2_data.co2.values)

# Make predictions
preds = my_xgb.predict(X)

# Plot the results
plt.plot(co2_data.index, co2_data.co2.values)
plt.plot(co2_data.index, preds)
plt.show()
```



This plot shows in orange the XGBoost’s strong fit to the data (represented in blue). | Source: Author

Advanced and specific time series

Table of contents

This section covers two advanced models for time series forecasting: **GARCH** and **TBATS**.

GARCH

GARCH (Generalized Autoregressive Conditional Heteroskedasticity) is primarily used to estimate volatility in financial markets.

Rather than predicting actual values, GARCH models the error variance in a time series, assuming an ARMA model for the variance. It's ideal for forecasting volatility rather than point values.

GARCH has several variants within its family, but it is best for predicting volatility, as it differs significantly from traditional time series models.

TBATS

TBATS stands for:

- **T**rigonometric seasonality
- **B**ox-Cox transformation
- **A**RMA errors
- **T**rend
- **S**easonal components

Introduced in 2011, TBATS is designed to handle time series with multiple seasonal cycles. This model is newer and less commonly used than ARIMA models but is effective for data with complex seasonal patterns.

A Python implementation of TBATS is available in the `sktime` package.

Deep learning-based time series models

We can now look at more advanced deep learning models after exploring

Table of contents

LSTMs (Long Short-Term Memory)

LSTM networks are a type of Recurrent Neural Network (RNN) specifically designed to handle sequential data. In LSTM models, multiple nodes pass input data through layers, each learning simple tasks that together capture complex, nonlinear relationships.

LSTMs are especially effective for time series forecasting because they can remember long-term dependencies in sequence data. Although they require substantial data and are challenging to train, they can be highly effective for complex time series patterns.

Python's Keras library is a popular starting point for building LSTM models.

Prophet

Prophet is a time series forecasting library open-sourced by Facebook. Prophet can generate forecasts with little user specification, making it easy to use, especially for non-experts in time series analysis.

However, it's essential to validate Prophet forecasts carefully, as automated model building may overlook nuances in the data. When properly validated, Prophet can be an effective forecasting tool. More resources are available on Facebook's GitHub.

Related

ARIMA vs Prophet vs LSTM for Time Series Prediction

Read more →

DeepAR

DeepAR, developed by Amazon, is another black-box model designed to simplify time series forecasting. While the underlying mechanics differ from Prophet, its user experience is automated too.

A great and easy-to-use implementation of DeepAR is available in the Gluon

Table of contents

Time series model selection

After exploring various time series models—including classical, supervised, and recent developments like LSTM, Prophet, and DeepAR—the final step is choosing the model that best suits your use case.

Model evaluation and metrics

Defining metrics

To select a model, you must first define the right metric(s) to evaluate your model.

Time series forecasting key metrics include:

- **Mean Squared Error (MSE)**: Measures squared error at each time point, then averages.
- **Root Mean Squared Error (RMSE)**: Square root of MSE, to have the error in its original units.
- **Mean Absolute Error (MAE)**: Uses absolute values of errors, making it more interpretable.
- **Mean Absolute Percentage Error (MAPE)**: Expresses absolute errors as percentages of actual values, making results easier to interpret.

Train-test split and cross-validation

When evaluating machine learning models, remember that good performance on training data doesn't guarantee good results on new, out-of-sample data. To estimate how well a model generalizes, two common approaches are **train-test split** and **cross-validation**.

Doing a **train-test split** involves holding back a portion of the data as a test set. For example, you could reserve the last 3 years of a CO2 dataset as a test set and use the remaining 40 years for training. Using a chosen evaluation metric, you'd then forecast the reserved period and compare predictions to actual values.

To benchmark multiple models, train each on the same 40-year data, forecast the test period, and select the model with the best performance.

Table of contents

performance at a single point in time. Unlike non-sequential data, where test sets can be randomly selected, time series data relies on sequence order, so it is essential to reserve the final period as the test set. However, this approach may be unreliable if the last period is atypical (e.g., due to events like COVID-19, which disrupted trends and forecasts).

Cross-validation provides a more robust approach, repeatedly splitting the data for training and testing. For example, in **3-fold cross-validation**, the data is divided into three parts. Each fold is used as a test set once, with the other two as training sets, producing three evaluation scores. Averaging these scores provides a more reliable measure of model performance.

Related

Cross-Validation in Machine Learning: How to Do It Right

Read more →

By doing this, you avoid selecting a model that performs well on the test set by chance: you now have a more reliable measure of its performance.

Time series model experiments

To guide your time series model selection, consider the following key questions before starting your experiments:

1. Which metric will you use for evaluation?
2. Which period are you aiming to forecast?
3. How will you ensure the model performs well in the future using unseen data?

Once you've answered these questions, you can begin testing different models and applying your evaluation strategy to select and refine the best-performing model.

Example use case: time series forecasting for S&P 500

In this example, we'll build a model to predict the next day's direction (up or down) for the S&P 500 index, simulating a scenario where predictions are

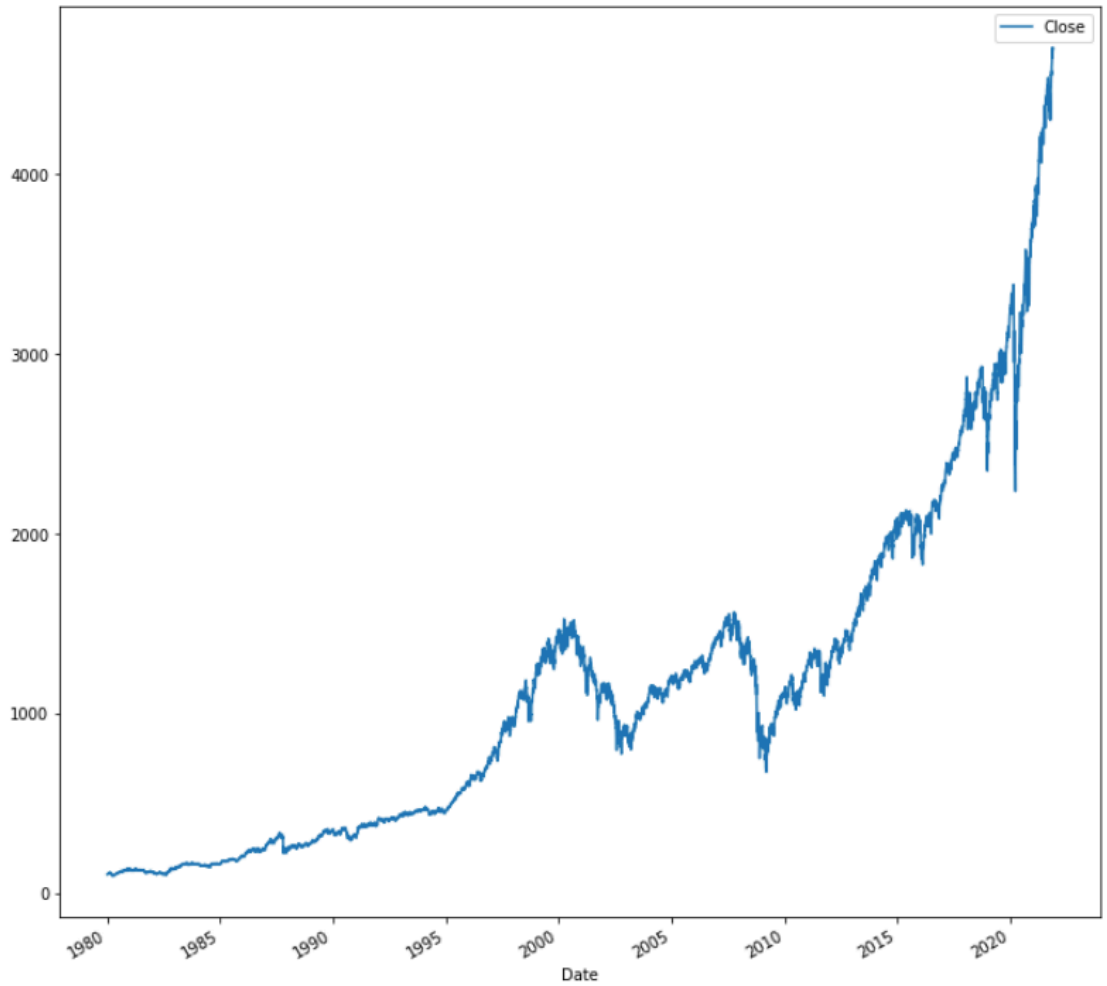
Table of contents

Stock market forecasting data

To access stock data, we can use the Yahoo Finance (`yfinance`) package in Python:

```
!pip install yfinance
import yfinance as yf

# Download S&P 500 closing prices from Yahoo Finance
sp500_data = yf.download('^GSPC', start="1980-01-01", end
sp500_data.plot(figsize=(12, 12))
```

The output:

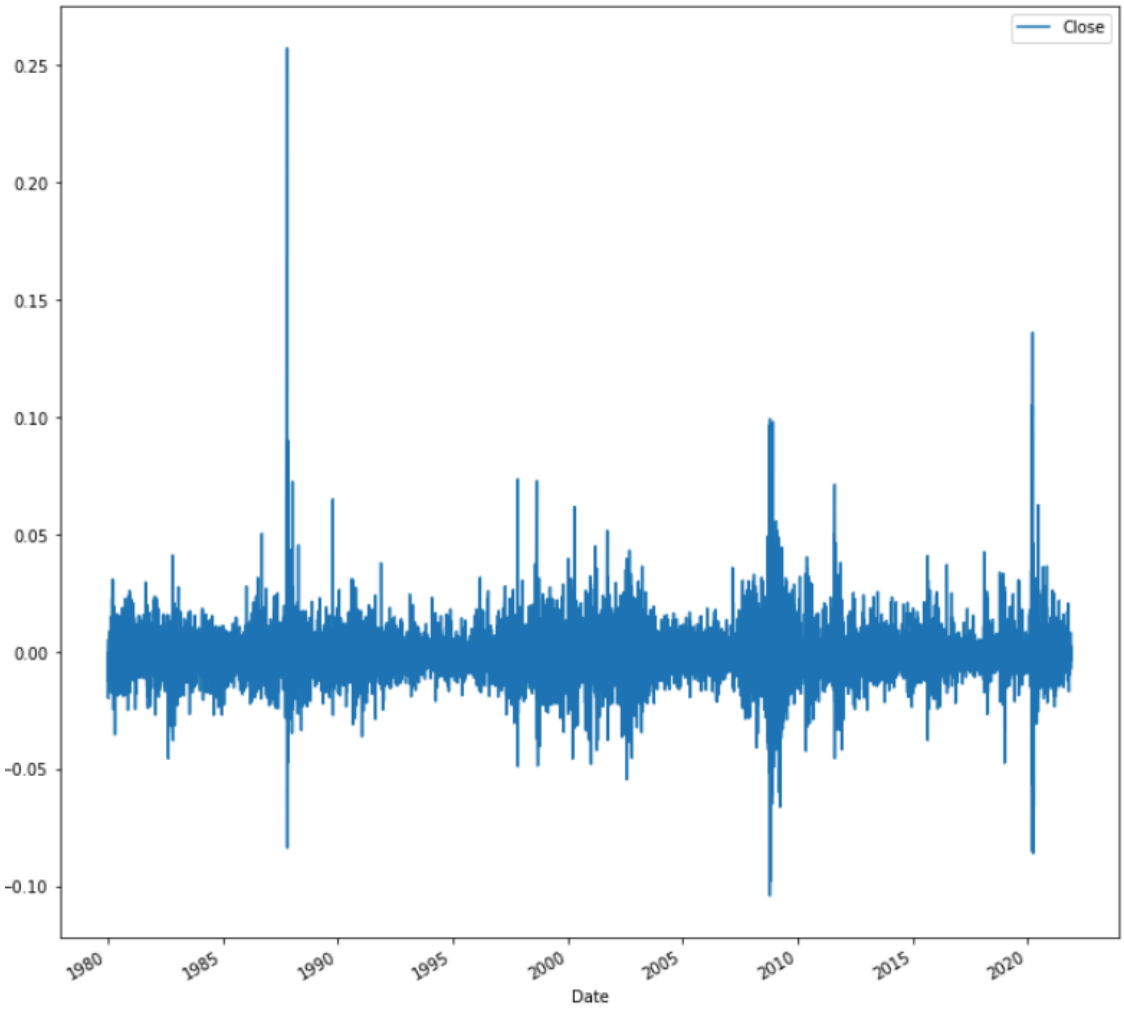


This plot shows the evolution of S&P 500 closing prices since 1980 | Source: Author

Instead of using absolute prices, traders often focus on daily percentage

Table of contents

```
difs = (sp500_data.shift() - sp500_data) / sp500_data
difs = difs.dropna()
difs.plot(figsize=(12, 12))
```



The plot now displays the percentage change in the S&P 500 over time | Source: Author

Defining the experimental approach

The model's goal is to predict the next day's percentage change accurately. Since our prediction period is just one day, the test set will be small, and multiple test splits are needed to ensure reliability.

We can set up 100 different train-test splits using the train-test split we discussed previously, with each split training on three months of data and testing on the following day. This setup allows consistent evaluation and better selection of the best-performing model.

Building a classical time series model: ARIMA

Table of contents

The code below sets up ARIMA models with orders ranging from (0, 0, 0) to (4, 4, 4). Each model is evaluated using 100 splits, where each split uses a maximum of three months for training and one day for testing.

There are a lot of training and test runs, so we will use a tracking tool, neptune.ai, for easy comparison. Before we continue, let's create our Neptune account following these steps:

1. Sign up for a free Neptune account
2. Create a new project
3. Save your credentials as environment variables

Now that we are all set up, let's start!

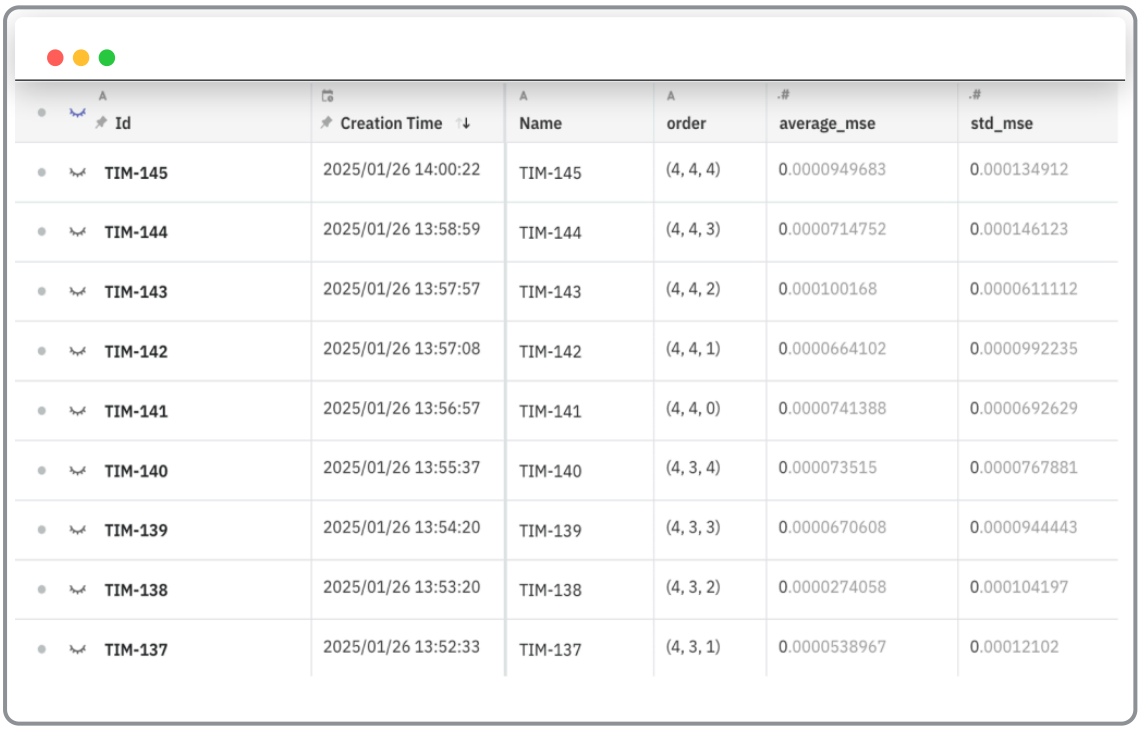
```
import numpy as np
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import TimeSeriesSplit
import neptune
from neptune.utils import stringify_unsupported
import statsmodels.api as sm

# List of ARIMA parameter combinations
param_list = [(p, d, q) for p in range(5) for d in range(

for order in param_list:
    # Initialize a Neptune run
    run = neptune.init_run(
        project="YOU/YOUR_PROJECT",
        api_token="YOUR_API_TOKEN",
    )

    run['parameters/order'] = order

    mses = []
    tscv = TimeSeriesSplit(n_splits=100, max_train_size=3

    for train_index, test_index in tscv.split(co2_data.co
        try:
            train, test = co2_data.co2.values[train_index

            # Fit ARIMA model
            model = sm.tsa.ARIMA(train, order=order)

```

```
            result = model.fit()
            prediction = result.forecast(1)[0]
```

Table of contents

```
            mse = mean_squared_error(test, prediction)
            mses.append(mse)

        except:
            # Ignore models that produce errors
            pass

    # Log results to Neptune
    run['average_mse'] = np.mean(mses) if mses else None
    run['std_mse'] = np.std(mses) if mses else None
    run.stop()
```

After running, you can view the results in a table format in the Neptune dashboard:



Id	Creation Time	Name	order	average_mse	std_mse
TIM-145	2025/01/26 14:00:22	TIM-145	(4, 4, 4)	0.0000949683	0.000134912
TIM-144	2025/01/26 13:58:59	TIM-144	(4, 4, 3)	0.0000714752	0.000146123
TIM-143	2025/01/26 13:57:57	TIM-143	(4, 4, 2)	0.000100168	0.0000611112
TIM-142	2025/01/26 13:57:08	TIM-142	(4, 4, 1)	0.0000664102	0.0000992235
TIM-141	2025/01/26 13:56:57	TIM-141	(4, 4, 0)	0.0000741388	0.0000692629
TIM-140	2025/01/26 13:55:37	TIM-140	(4, 3, 4)	0.000073515	0.0000767881
TIM-139	2025/01/26 13:54:20	TIM-139	(4, 3, 3)	0.0000670608	0.0000944443
TIM-138	2025/01/26 13:53:20	TIM-138	(4, 3, 2)	0.0000274058	0.000104197
TIM-137	2025/01/26 13:52:33	TIM-137	(4, 3, 1)	0.0000538967	0.00012102

Dashboard view of our 100 runs in the Neptune UI

The model with the lowest average MSE is ARIMA(0, 1, 3). However, its standard deviation is unexpectedly 0, which raises concerns about the stability of this result. The next best models, ARIMA(1, 0, 3) and ARIMA(1, 0, 2), have very similar performance, indicating more reliable outcomes.

Based on this, ARIMA(1, 0, 3) is the best choice, with an average MSE of 0.00000131908 and a standard deviation of 0.00000197007, suggesting both accuracy and consistency in forecasting performance.

Table of contents

and other model-building metadata.

Building a supervised machine learning model

Next, we'll explore a supervised machine learning model and see how its performance compares to a classical time series model.

As we mentioned earlier, feature engineering is important in supervised machine learning for forecasting. Supervised models need both dependent (target) and independent (predictor) variables. Sometimes, you may have additional future data (like reservation numbers to help predict a restaurant's daily customer count). However, for this stock market example, we only have past stock prices.

A supervised model can't be trained on just a target variable, so we need to create features to capture seasonality and autocorrelation effects. For this model, we'll use the stock prices from the past 30 days as input features to predict the price on the 31st day.

This approach will create a dataset where each entry contains 30 consecutive days as predictors and the 31st day as the target. By sliding this 30-day window across the S&P 500 data, we can generate a large training dataset for model development.

Now that you have the training database, you can use regular cross-validation: after all, the rows of the data set can be used independently. They are all sets of 30 training days and 1 'future' test day. Thanks to this data preparation, you can use regular KFold cross-validation.

```
import yfinance as yf
import numpy as np

# Download S&P 500 closing price data
sp500_data = yf.download('^GSPC', start="1980-01-01", end
sp500_data = sp500_data[['Close']]

# Calculate daily percentage changes
difs = (sp500_data.shift() - sp500_data) / sp500_data
difs = difs.dropna()  # Remove any NaN values from the da

```

```
12
# Extract the 'Close' values as our target variable
```

Table of contents

```
# Generate input windows of 30 days to predict the 31st d
X_data = []
y_data = []
for i in range(len(y) - 31):
    X_data.append(y[i:i+30])     # Last 30 days as input
    y_data.append(y[i+30])       # 31st day as the target

# Convert lists to numpy arrays
X_windows = np.vstack(X_data)
```

With the training dataset prepared, you can now apply regular cross-validation. Each row in this dataset is a separate sequence of 30 training days followed by a target day, allowing them to be used independently in the model.

Using cross-validation will help evaluate model performance more reliably by testing it across different splits of the data, rather than relying on a single train-test split.

The code below performs a grid search with cross-validation using XGBoost on our prepared dataset. It evaluates model performance across multiple hyperparameter combinations and logs the results to Neptune.

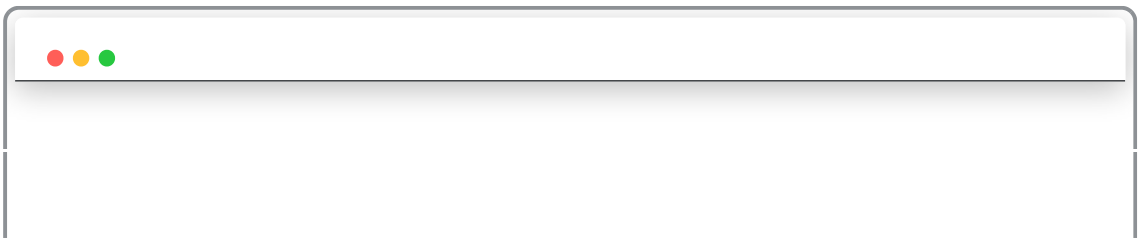
```
import numpy as np
import xgboost as xgb
from sklearn.model_selection import KFold
import neptune
from neptune.utils import stringify_unsupported
from sklearn.metrics import mean_squared_error

# Define parameter grid for hyperparameter tuning
parameters = {
    'max_depth': list(range(2, 20, 4)),
    'gamma': list(range(0, 10, 2)),
    'min_child_weight': list(range(0, 10, 2)),
    'eta': [0.01, 0.05, 0.1, 0.15, 0.2, 0.3, 0.5]
}

# Create a list of all possible parameter combinations
param_list = [(x, y, z, a) for x in parameters['max_depth
                           for y in parameters['gamma'
                           for z in parameters['min_ch
                           for a in parameters['eta']]
```

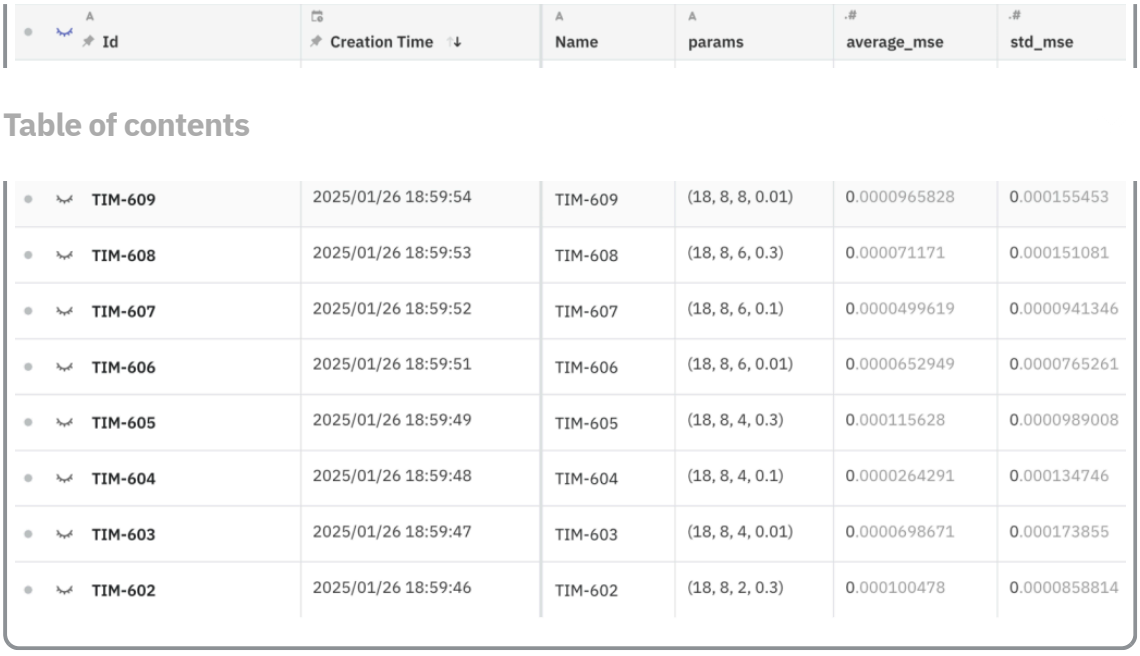
```
21
22  # Iterate over all parameter combinations
```

Table of contents

```
26      # Initialize Neptune run for logging
27      run = neptune.init_run(
28          project="YOU/YOUR_PROJECT",
29          api_token="YOUR_API_TOKEN",
30      )
31      run['params'] = params
32
33      # Set up KFold cross-validation
34      my_kfold = KFold(n_splits=10, shuffle=True, random_st
35
36      for train_index, test_index in my_kfold.split(X_windo
37          X_train, X_test = X_windows[train_index], X_windo
38          y_train, y_test = np.array(y_data)[train_index],
39
40          # Create and train the XGBoost model
41          xgb_model = xgb.XGBRegressor(
42              max_depth=params[0],
43              gamma=params[1],
44              min_child_weight=params[2],
45              eta=params[3]
46          )
47          xgb_model.fit(X_train, y_train)
48          preds = xgb_model.predict(X_test)
49
50          # Calculate and store Mean Squared Error for the
51          mses.append(mean_squared_error(y_test, preds))
52
53      # Log average MSE and standard deviation to Neptune
54      average_mse = np.mean(mses)
55      std_mse = np.std(mses)
56      run['average_mse'] = average_mse
57      run['std_mse'] = std_mse
58
59      # Stop the Neptune run
60      run.stop()
```

Some of the scores obtained using this loop are shown in the below table:





Id	Creation Time	Name	params	average_mse	std_mse
TIM-609	2025/01/26 18:59:54	TIM-609	(18, 8, 8, 0.01)	0.0000965828	0.000155453
TIM-608	2025/01/26 18:59:53	TIM-608	(18, 8, 6, 0.3)	0.000071171	0.000151081
TIM-607	2025/01/26 18:59:52	TIM-607	(18, 8, 6, 0.1)	0.0000499619	0.0000941346
TIM-606	2025/01/26 18:59:51	TIM-606	(18, 8, 6, 0.01)	0.0000652949	0.0000765261
TIM-605	2025/01/26 18:59:49	TIM-605	(18, 8, 4, 0.3)	0.000115628	0.0000989008
TIM-604	2025/01/26 18:59:48	TIM-604	(18, 8, 4, 0.1)	0.0000264291	0.000134746
TIM-603	2025/01/26 18:59:47	TIM-603	(18, 8, 4, 0.01)	0.0000698671	0.000173855
TIM-602	2025/01/26 18:59:46	TIM-602	(18, 8, 2, 0.3)	0.000100478	0.0000858814

Table view of our runs from the Neptune UI

The parameters that were tested in this grid search are reproduced below:

Parameter name	Values tested	Description
Max Depth	2, 4, 6 8, 10	Controls the tree depth. Higher values make the model more complex and increase the risk of overfitting.
Min Child Weight	0, 2, 4	Minimum sum of instance weights needed in a child node. Higher values prevent overly complex models by stopping splits that don't meet this threshold.
Eta	0.01, 0.1, 0.3	Learning rate (or step size). Low values mean slow learning, but they can improve accuracy by preventing overfitting.
Gamma	0, 2, 4	Minimum loss reduction required to split a node. Higher values make the model more conservative by reducing unnecessary splits.

For more information on XGBoost tuning, check out the official XGBoost documentation.

The best (lowest) MSE this XGBoost model achieves is 0.000129982, with several hyperparameter combinations reaching this score. However, the XGBoost model underperforms in its current setup compared to the classical time series model. To improve XGBoost's results, a different approach to organizing the data may be needed.

LSTM model for time series forecasting

As a third model for the model comparison, let's take an LSTM and see whether it can beat our ARIMA model.

The following code sets up an LSTM model using Keras. Instead of cross-

Table of contents

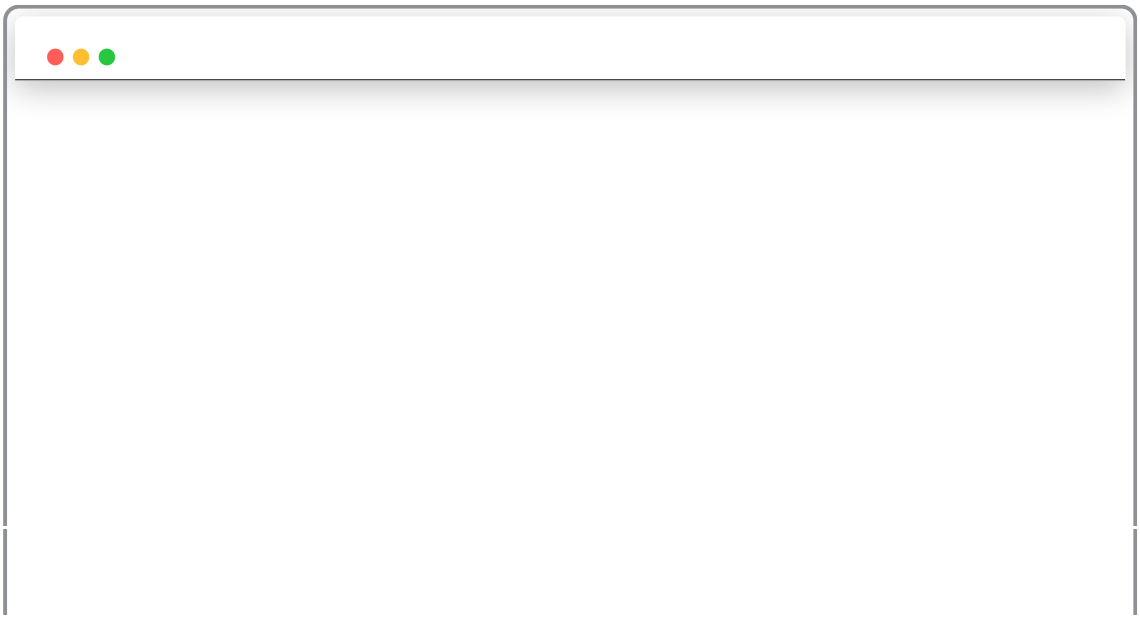
```
import tensorflow as tf
import numpy as np
import yfinance as yf
from sklearn.model_selection import train_test_split
import neptune

# Load and preprocess data
sp500_data = yf.download('^GSPC', start="1980-01-01", end
sp500_data = sp500_data[['Close']]
difs = (sp500_data.shift() - sp500_data) / sp500_data
difs = difs.dropna()
y = difs.Close.values

# Create windows
X_data = []
y_data = []
window_size = 3 * 31  # 3 months of data
for i in range(len(y) - window_size):
    X_data.append(y[i:i+window_size])
    y_data.append(y[i+window_size])

X_windows = np.array(X_data)
y_data = np.array(y_data)

# Train/test/validation split
X_train, X_test, y_train, y_test = train_test_split(X_win
X_train, X_val, y_train, y_val = train_test_split(X_train

# Reshape input data for LSTM (samples, timesteps, featur
X_train = X_train.reshape((X_train.shape[0], X_train.shap
X_val = X_val.reshape((X_val.shape[0], X_val.shape[1], 1)
X_test = X_test.reshape((X_test.shape[0], X_test.shape[1]

# Define LSTM architectures
archi_list = [
    [tf.keras.layers.LSTM(32, return_sequences=True, inpu
     tf.keras.layers.LSTM(32, return_sequences=False),
     tf.keras.layers.Dense(units=1)],
    [tf.keras.layers.LSTM(64, return_sequences=True, inpu
     tf.keras.layers.LSTM(64, return_sequences=False),
     tf.keras.layers.Dense(units=1)],
    [tf.keras.layers.LSTM(128, return_sequences=True, inp
     tf.keras.layers.LSTM(128, return_sequences=False),

```

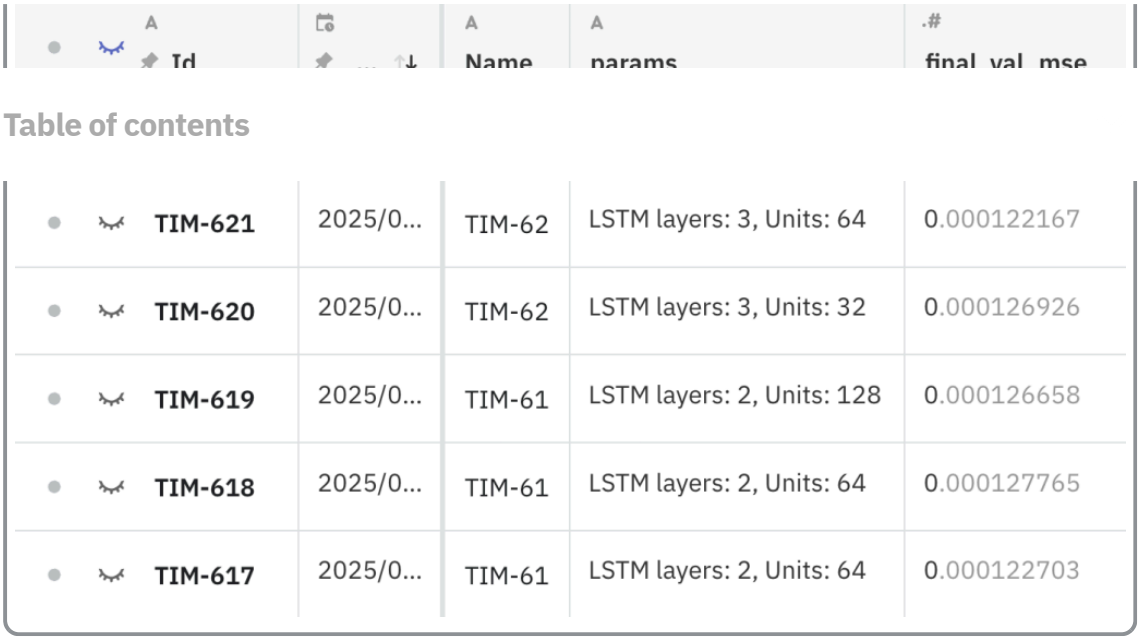
```
45      tf.keras.layers.Dense(units=1)],
46     [tf.keras.layers.LSTM(32, return_sequences=True, inpu
```

Table of contents

```
49      tf.keras.layers.Dense(units=1)],
50     [tf.keras.layers.LSTM(64, return_sequences=True, inpu
51      tf.keras.layers.LSTM(64, return_sequences=True),
52      tf.keras.layers.LSTM(64, return_sequences=False),
53      tf.keras.layers.Dense(units=1)],
54 ]
55
56 # Loop through architectures and log results
57 for archi in archi_list:
58     run = neptune.init_run(
59         project="YOU/YOUR_PROJECT",
60         api_token="YOUR_API_TOKEN",
61     )
62
63     run['params'] = f'LSTM Layers: {len(archi) - 1}, Unit
64     run['Tags'] = 'lstm_model_comparison'
65
66     # Build, compile, and train model
67     lstm_model = tf.keras.models.Sequential(archi)
68     lstm_model.compile(loss=tf.losses.MeanSquaredError(),
69                        optimizer=tf.optimizers.Adam(),
70                        metrics=[tf.metrics.MeanSquaredErr
71     history = lstm_model.fit(X_train, y_train, epochs=10,
72
73     # Log final validation MSE to Neptune
74     run['final_val_mse'] = history.history['val_mean_squa
       run.stop()
```

Here, we can see the output for the 10 epochs:





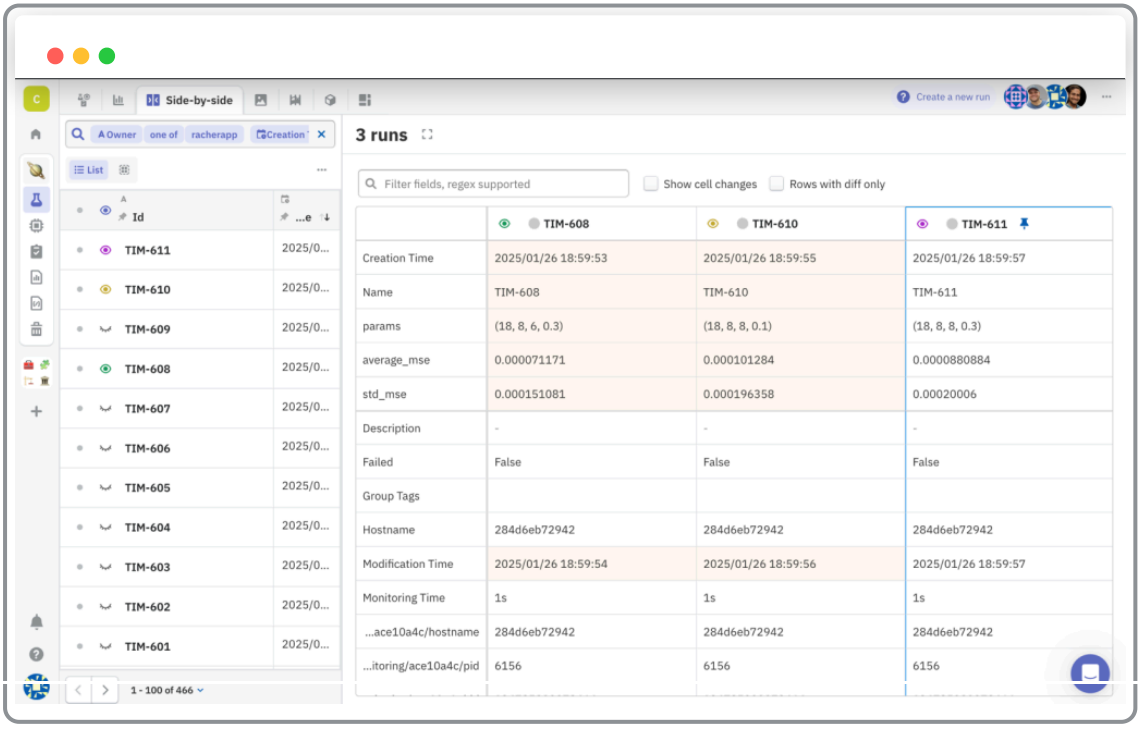
Id		Name	params	final_val_mse
TIM-621	2025/0...	TIM-62	LSTM layers: 3, Units: 64	0.000122167
TIM-620	2025/0...	TIM-62	LSTM layers: 3, Units: 32	0.000126926
TIM-619	2025/0...	TIM-61	LSTM layers: 2, Units: 128	0.000126658
TIM-618	2025/0...	TIM-61	LSTM layers: 2, Units: 64	0.000127765
TIM-617	2025/0...	TIM-61	LSTM layers: 2, Units: 64	0.000122703

Output values for our LSTM viewed in the Neptune UI

The LSTM performed similarly to the XGBoost model. To improve the results, you could experiment with different training period lengths or adjust data standardization methods, which often impact neural network performance.

Selecting the best model

Out of the three models we tested, the ARIMA model (highlighted in the blue box below) showed the best performance based on a three-month training period and a one-day forecast, with the lowest mean squared error value (the row “average”, with MSE 0.423, compared to 0.46 and 1.07 for the other two models).



Comparison view of our three models in the Neptune UI

Table of contents

Next steps

To further improve this model, you could experiment with different training period lengths or add more data, such as seasonal indicators (day of the week, month, etc.) or additional predictors like market sentiment. If adding external variables, consider using a SARIMAX model.

Now you have a solid overview of time series model selection, including model types and tools like windowing and time series splits!

Was the article useful?

Yes No

Suggest changes

More about How to Select a Model For Your Time Series Prediction Task [Guide]

Check out our **product resources** and **related articles** below:

Related article

How to Build an LLM Agent With AutoGen: Step-by-Step Guide

Read more →

Related article

Bayesian Deep Learning is Needed in the Age of Large-Scale AI [Paper Reflection]

Read more →

Related article

Introduction to State Space Models as Natural Language Models

Read more →

Related article

Ethical Considerations and Best Practices in LLM Development

Read more →

Explore more content topics:

Table of contents

MLOps | Natural Language Processing | Paper Reflections | Product Updates | Reinforcement Learning | Tabular Data | Time Series

Newsletter

Top articles, case studies, events (and more) in your inbox every month.

Your e-mail

Get Newsletter

PRODUCT

Table of contents

COMPARE

COMMUNITY

COMPANY