How to Make Manual Predictions for ARIMA Models with Python

by **Jason Brownlee** on February 8, 2017 in **Time Series**

The autoregression integrated moving average model or ARIMA model can seem intimidating to beginners.

A good way to pull back the curtain in the method is to to use a trained model to make predictions manually. This demonstrates that ARIMA is a linear regression model at its core.

Making manual predictions with a fit ARIMA models may also be a requirement in your project, meaning that you can save the coefficients from the fit model and use them as configuration in your own code to make predictions without the need for heavy Python libraries in a production environment.

In this tutorial, you will discover how to make manual predictions with a trained ARIMA model in Python.

Specifically, you will learn:

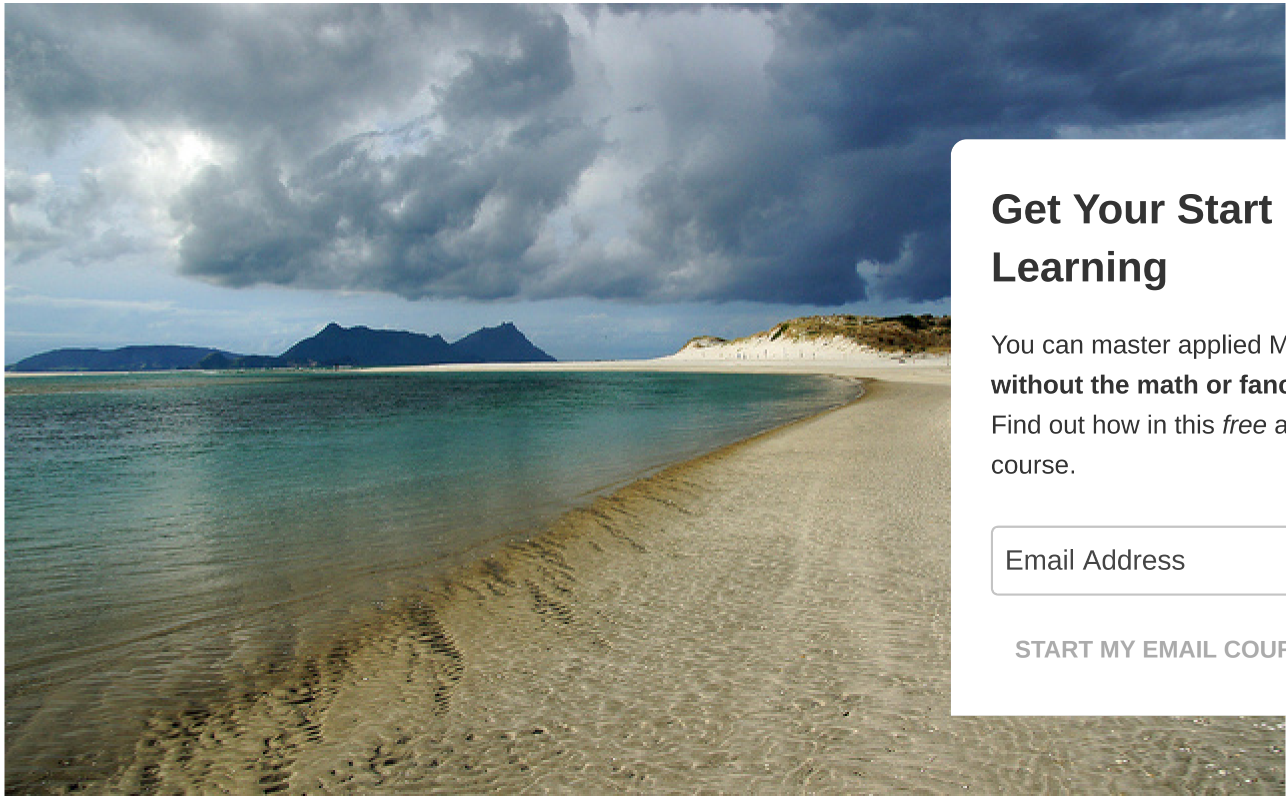
- How to make manual predictions with an autoregressive model.

- How to make manual predictions with a moving average model.
- How to make predictions with an autoregression integrated moving average model.

Let’s dive in.

Update: You may find this post useful:

- How to Make Out-of-Sample Forecasts with ARIMA in Python



How to Make Manual Predictions for ARIMA Models with Python
Photo by Bernard Spragg. NZ, some rights reserved.

Minimum Daily Temperatures Dataset

This dataset describes the minimum daily temperatures over 10 years (1981-1990) in the city Melbo

The units are in degrees Celsius and there are 3,650 observations. The source of the data is credite[redacted]

You can learn more and download the dataset from the Data Market website.

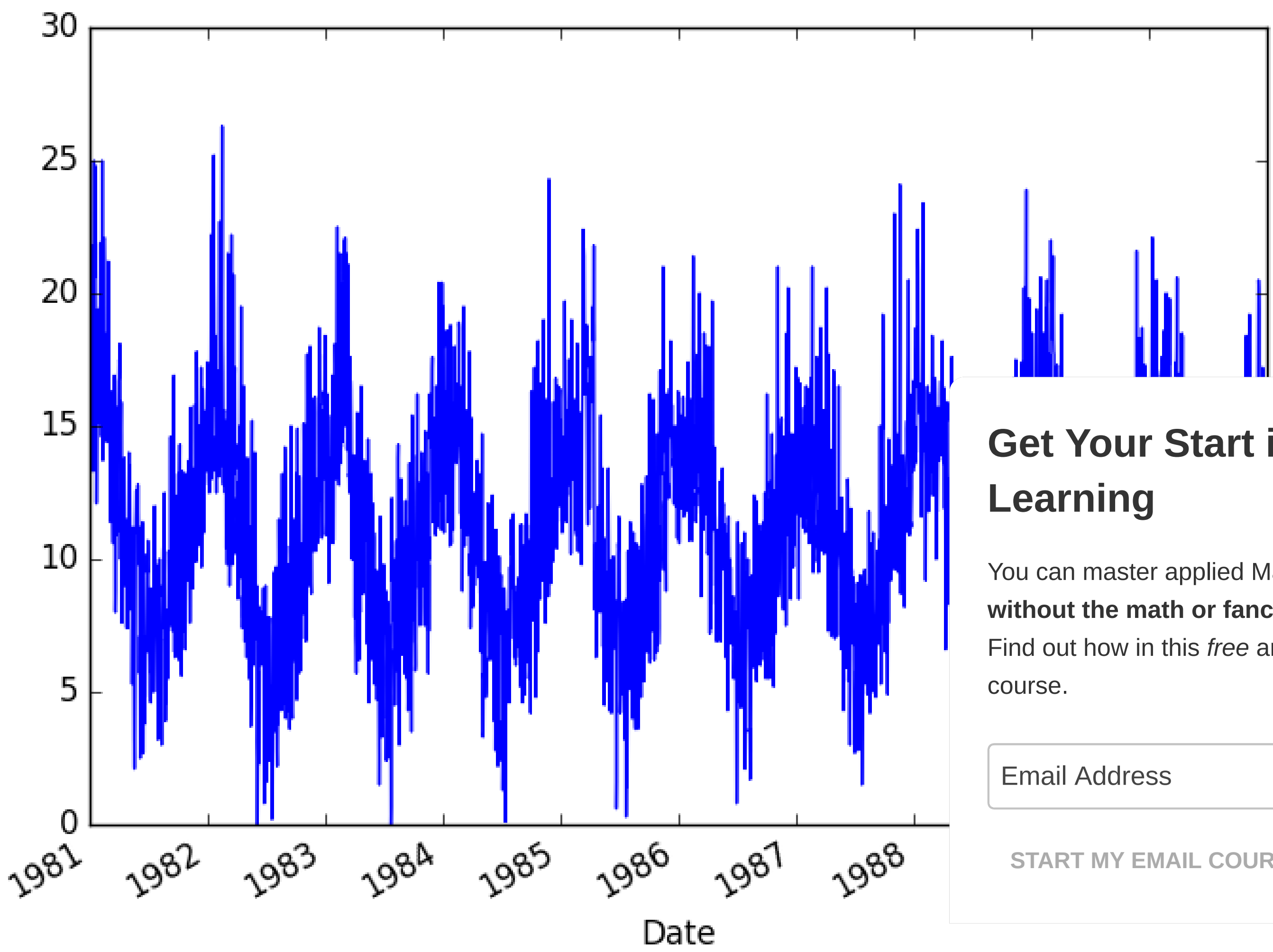
Download the dataset and place it into your current working directory with the filename "*daily-minimum-temperatures.csv*".

Note: The downloaded file contains some question mark ("?") characters that must be removed before you can use the dataset. Open the file in a text editor and remove the "?" characters. Also remove any footer information in the file.

The example below demonstrates how to load the dataset as a Pandas Series and graph the loaded dataset.

```
from pandas import Series
from matplotlib import pyplot
series = Series.from_csv('daily-minimum-temperatures.csv', header=0)
series.plot()
pyplot.show()
```

Running the example creates a line plot of the time series.



Minimum Daily Temperatures Dataset Plot

ARIMA Test Setup

We will use a consistent test harness to fit ARIMA models and evaluate their predictions.

First, the loaded dataset is split into a train and test dataset. The majority of the dataset is used to fit
are held back as the test dataset to evaluate the fit model.

A walk-forward validation, or rolling forecast, method is used as follows:

1. Each time step in the test dataset is iterated.
2. Within each iteration, a new ARIMA model is trained on all available historical data.
3. The model is used to make a prediction for the next day.
4. The prediction is stored and the "real" observation is retrieved from the test set and added to the
5. The performance of the model is summarized at the end by calculating the root mean squared e
expected values in the test dataset.

Simple AR, MA, ARMA and ARMA models are developed. They are unoptimized and are used for demonstration purposes. You will surely be able to achieve better performance with a little tuning.

The ARIMA implementation from the statsmodels Python library is used and AR and MA coefficients are extracted from the ARIMAResults object returned from fitting the model.

The ARIMA model supports forecasts via the predict() and the forecast() functions.

Nevertheless, we will make manual predictions in this tutorial using the learned coefficients.

This is useful as it demonstrates that all that is required from a trained ARIMA model is the coefficients.

The coefficients in the statsmodels implementation of the ARIMA model do not use intercept terms. This means we can calculate the output values by taking the dot product of the learned coefficients and lag values (in the case of an AR model) and lag residuals (in the case of an MA model). For example:

```
y = dot_product(ar_coefficients, lags) + dot_product(ma_coefficients, residuals)
```

The coefficients of a learned ARIMA model can be accessed from aARIMAResults object as follows:

- **AR Coefficients**: model_fit.arparams
- **MA Coefficients**: model_fit.maparams

We can use these retrieved coefficients to make predictions using the following manual *predict()* fun

```
def predict(coef, history):
    yhat = 0.0
    for i in range(1, len(coef)+1):
        yhat += coef[i-1] * history[-i]
    return yhat
```


For reference, you may find the following resources useful:

- ARIMA API Documentation
- ARIMAResults API Documentation
- ARIMA statsmodels Source Code

Let’s look at some simple but specific models and how to make manual predictions with this test setup.

Autoregression Model

The autoregression model, or AR, is a linear regression model on the lag observations.

An AR model with a lag of *k* can be specified in the ARIMA model as follows:

```
model = ARIMA(history, order=(k,0,0))
```

In this example, we will use a simple AR(1) for demonstration purposes.

Making a prediction requires that we retrieve the AR coefficients from the fit model and use them with the lag of observed values and call the custom *predict()* function defined above.

The complete example is listed below.

```
from pandas import Series
from matplotlib import pyplot
from statsmodels.tsa.arima_model import ARIMA
from sklearn.metrics import mean_squared_error
from math import sqrt

def predict(coef, history):
    yhat = 0.0
    for i in range(1, len(coef)+1):
        yhat += coef[i-1] * history[-i]
    return yhat

series = Series.from_csv('daily-minimum-temperatures.csv', header=0)
X = series.values
size = len(X) - 7
train, test = X[0:size], X[size:]
history = [x for x in train]
predictions = list()
for t in range(len(test)):
    model = ARIMA(history, order=(1,0,0))
    model_fit = model.fit(trend='nc', disp=False)
    ar_coef = model_fit.arparams
    yhat = predict(ar_coef, history)
    predictions.append(yhat)
    obs = test[t]
    history.append(obs)
    print('>predicted=%.3f, expected=%.3f' % (yhat, obs))
rmse = sqrt(mean_squared_error(test, predictions))
print('Test RMSE: %.3f' % rmse)
```


Note that the ARIMA implementation will automatically model a trend in the time series. This adds a constant to the regression equation that we do not need for demonstration purposes. We turn this convenience off by setting the 'trend' argument in the *fit()* function to the value '*nc*' for '*no constant*'.

The fit() function also outputs a lot of verbose messages that we can turn off by setting the '*disp*' arg

Running the example prints the prediction and expected value each iteration for 7 days. The final R[redacted] 1.9 degrees Celsius for this simple model.

```
>predicted=9.738, expected=12.900
>predicted=12.563, expected=14.600
>predicted=14.219, expected=14.000
>predicted=13.635, expected=13.600
>predicted=13.245, expected=13.500
>predicted=13.148, expected=15.700
>predicted=15.292, expected=13.000
Test RMSE: 1.928
```

Experiment with AR models with different orders, such as 2 or more.

Moving Average Model

The moving average model, or MA, is a linear regression model of the lag residual errors.

An MA model with a lag of k can be specified in the ARIMA model as follows:

```
model = ARIMA(history, order=(0,0,k))
```

In this example, we will use a simple MA(1) for demonstration purposes.

Much like above, making a prediction requires that we retrieve the MA coefficients from the fit model and call the custom *predict()* function defined above.

The residual errors during training are stored in the ARIMA model under the '*resid*' parameter of the

```
model_fit.resid
```

The complete example is listed below.

```
from pandas import Series
from matplotlib import pyplot
from statsmodels.tsa.arima_model import ARIMA
from sklearn.metrics import mean_squared_error
from math import sqrt

def predict(coef, history):
```

```
    yhat = 0.0
    for i in range(1, len(coef)+1):
        yhat += coef[i-1] * history[-i]
    return yhat

series = Series.from_csv('daily-minimum-temperatures.csv', header=0)
X = series.values
size = len(X) - 7
train, test = X[0:size], X[size:]
history = [x for x in train]
predictions = list()
for t in range(len(test)):
    model = ARIMA(history, order=(0,0,1))
    model_fit = model.fit(trend='nc', disp=False)
    ma_coef = model_fit.maparams
    resid = model_fit.resid
    yhat = predict(ma_coef, resid)
    predictions.append(yhat)
    obs = test[t]
    history.append(obs)
    print('>predicted=%.3f, expected=%.3f' % (yhat, obs))
rmse = sqrt(mean_squared_error(test, predictions))
print('Test RMSE: %.3f' % rmse)
```

Running the example prints the predictions and expected values each iteration for 7 days and ends l

The skill of the model is not great and you can use this as an opportunity to explore MA models with predictions.

```
>predicted=4.610, expected=12.900
>predicted=7.085, expected=14.600
>predicted=6.423, expected=14.000
>predicted=6.476, expected=13.600
>predicted=6.089, expected=13.500
>predicted=6.335, expected=15.700
>predicted=8.006, expected=13.000
Test RMSE: 7.568
```


You can see how it would be straightforward to keep track of the residual errors manually outside of the *ARIMAResults* object as new observations are made available. For example:

```
residuals = list()
...
error = expected - predicted
```

```
residuals.append(error)
```

Next, let's put the AR and MA models together and see how we can perform manual predictions.

Autoregression Moving Average Model

We have now seen how we can make manual predictions for a fit AR and MA model.

These approaches can be put directly together to make manual predictions for a fuller ARMA model.

In this example, we will fit an ARMA(1,1) model that can be configured in an ARIMA model as ARIMA(1,0,1) with no differencing.

The complete example is listed below.

```
from pandas import Series
from matplotlib import pyplot
from statsmodels.tsa.arima_model import ARIMA
from sklearn.metrics import mean_squared_error
from math import sqrt

def predict(coef, history):
    yhat = 0.0
    for i in range(1, len(coef)+1):
        yhat += coef[i-1] * history[-i]
    return yhat

series = Series.from_csv('daily-minimum-temperatures.csv', header=0)
X = series.values
size = len(X) - 7
train, test = X[0:size], X[size:]
history = [x for x in train]
predictions = list()
for t in range(len(test)):
    model = ARIMA(history, order=(1,0,1))
    model_fit = model.fit(trend='nc', disp=False)
    ar_coef, ma_coef = model_fit.arparams, model_fit.maparams
    resid = model_fit.resid
    yhat = predict(ar_coef, history) + predict(ma_coef, resid)
    predictions.append(yhat)
    obs = test[t]
    history.append(obs)
    print('>predicted=%.3f, expected=%.3f' % (yhat, obs))
rmse = sqrt(mean_squared_error(test, predictions))
```

```
print('Test RMSE: %.3f' % rmse)
```

You can see that the prediction (*yhat*) is the sum of the dot product of the AR coefficients and lag observations and the MA coefficients and lag residual errors.

```
yhat = predict(ar_coef, history) + predict(ma_coef, resid)
```

Again, running the example prints the predictions and expected values each iteration and the summary RMSE for all predictions made.

```
>predicted=11.920, expected=12.900
>predicted=12.309, expected=14.600
>predicted=13.293, expected=14.000
>predicted=13.549, expected=13.600
>predicted=13.504, expected=13.500
>predicted=13.434, expected=15.700
>predicted=14.401, expected=13.000
Test RMSE: 1.405
```

We can now add differencing and show how to make predictions for a complete ARIMA model.

Autoregression Integrated Moving Average Model

The I in ARIMA stands for integrated and refers to the differencing performed on the time series obs regression model.

When making manual predictions, we must perform this differencing of the dataset prior to calling the implements differencing of the entire dataset.


```
def difference(dataset):
    diff = list()
    for i in range(1, len(dataset)):
        value = dataset[i] - dataset[i - 1]
        diff.append(value)
    return numpy.array(diff)
```

A simplification would be to keep track of the observation at the oldest required lag value and use that to calculate the differenced series prior to prediction as needed.

This difference function can be called once for each difference required of the ARIMA model.

In this example, we will use a difference level of 1, and combine it with the ARMA example in the pre[redacted]

The complete example is listed below.

```
from pandas import Series
from matplotlib import pyplot
from statsmodels.tsa.arima_model import ARIMA
from sklearn.metrics import mean_squared_error
from math import sqrt
import numpy

def predict(coef, history):
    yhat = 0.0
    for i in range(1, len(coef)+1):
        yhat += coef[i-1] * history[-i]
    return yhat

def difference(dataset):
    diff = list()
    for i in range(1, len(dataset)):
        value = dataset[i] - dataset[i - 1]
        diff.append(value)
    return numpy.array(diff)

series = Series.from_csv('daily-minimum-temperatures.csv', header=0)
X = series.values
size = len(X) - 7
train, test = X[0:size], X[size:]
history = [x for x in train]
predictions = list()
for t in range(len(test)):
    model = ARIMA(history, order=(1,1,1))
    model_fit = model.fit(trend='nc', disp=False)
    ar_coef, ma_coef = model_fit.arparams, model_fit.maparams
    resid = model_fit.resid
    diff = difference(history)
    yhat = history[-1] + predict(ar_coef, diff) + predict(ma_coef, resid)
    predictions.append(yhat)
    obs = test[t]
    history.append(obs)
    print('>predicted=%.3f, expected=%.3f' % (yhat, obs))
rmse = sqrt(mean_squared_error(test, predictions))
print('Test RMSE: %.3f' % rmse)
```

You can see that the lag observations are differenced prior to their use in the call to the *predict()* fun[redacted] will also be calculated with regard to these differenced input values.

Running the example prints the prediction and expected value each iteration and summarizes the performance of all predictions made.

```
>predicted=11.837, expected=12.900
>predicted=13.265, expected=14.600
>predicted=14.159, expected=14.000
>predicted=13.868, expected=13.600
>predicted=13.662, expected=13.500
>predicted=13.603, expected=15.700
>predicted=14.788, expected=13.000
Test RMSE: 1.232
```

Summary

In this tutorial, you discovered how to make manual predictions for an ARIMA model with Python.

Specifically, you learned:

- How to make manual predictions for an AR model.
- How to make manual predictions for an MA model.
- How to make manual predictions for an ARMA and ARIMA model.

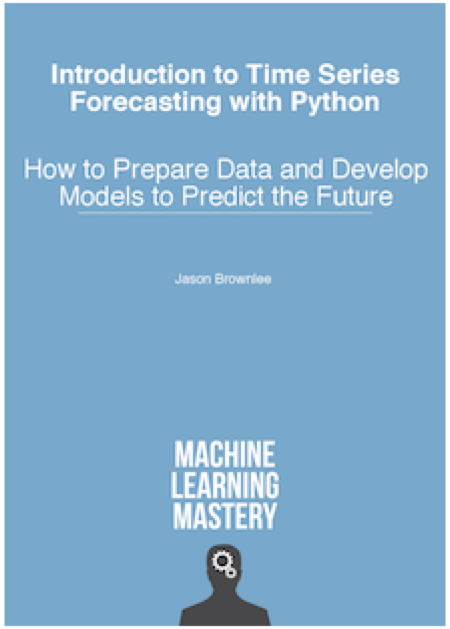
Do you have any questions about making manual predictions?
Ask your questions in the comments below and I will do my best to answer.

Want to Develop Time Series Forecasts with Python?

Develop Your Own Forecasts in Minutes

...with just a few lines of python code

Discover how in my new Ebook:
Introduction to Time Series Forecasting With Python



It covers **self-study tutorials** and **end-to-**[redacted]
Loading data, visualization, modeling, algorithm tuning, and much more...

Finally Bring Time Series Forecasting to Your Own Projects

Skip the Academics. Just Results.

Click to learn more.

About Jason Brownlee

Dr. Jason Brownlee is a husband, proud father, academic researcher, author, professional devel
to helping developers get started and get good at applied machine learning. Learn more.
View all posts by Jason Brownlee →

< A Gentle Introduction to Autocorrelation and Partial Autocorrelation

Understand Time Series Forecast Uncertainty Using Confidence Intervals with Python >

22 Responses to *How to Make Manual Predictions for ARIMA Models with Python*

MIC February 8, 2017 at 4:50 pm # REPLY

Hi Jason,

Thanks for this tutorial.
Now the error occure as follow, Please advise me about it.
I think also that the code do not have an error.

—> 23 model_fit = model.fit(trend='nc', disp=False)

ValueError: could not convert string to float: ?0.2

thanks

Jason Brownlee February 9, 2017 at 7:22 am #

Open the downloaded data file and delete all instances of the "?" character.

MIC February 10, 2017 at 11:56 am #

It worked fine.
I did not think that it was caused by converting the file to CSV.

Thank you, Jason.

Jason Brownlee February 11, 2017 at 4:53 am #

Glad to hear it!

Luca May 5, 2017 at 2:42 am #

REPLY

Hi Jason

Really appreciate for this article, I've got one question: in this example, why do we use an iterative way to determine the ARIMA parameters, can we fix the model parameters before the loop in test dataset and then doing the validation process?

Thanks a lot for some more insights on it.

Jason Brownlee May 5, 2017 at 7:33 am #

I'm not sure what you mean by iterative? Can you please elaborate?

Luca May 6, 2017 at 12:20 am #

Thanks for the feedback, Jason, what I meant before is:
…
for t in range(len(test)):
model = ARIMA(history, order=(1,1,1))
…
we see that in each loop, we train the model again and get a new set of parameters. Why not train the model just based on the training set and all parameters in the model is fixed, then loop for all test set and validate the error for each of them.

Thanks again for your reply.

Jason Brownlee May 6, 2017 at 7:46 am #

You can, but if we have new data (e.g. it is the next month and a new observation i[redacted]

That is what we are simulating here. It is called walk forward validation:
http://machinelearningmastery.com/backtest-machine-learning-models-time-series-forecasting/

Luca May 8, 2017 at 7:16 pm #

Thanks a lot Jason.
Nice example from that link. I find that there're plenty of quite useful and interesting information from your posts, I will go through others and post questions (if I have).
Again, thanks for the work, great job. 🙂

Jason Brownlee May 9, 2017 at 7:40 am #

Thanks Luca!

Hans June 15, 2017 at 11:37 pm #

Let's say I have two data points in a week recorded over several years, for example data from N
Would this be relevant for the difference function?

Jason Brownlee June 16, 2017 at 8:00 am #

REPLY

It may, it depends on the problem.

Get Your Start in Machine Learning

Hans June 15, 2017 at 11:38 pm # REPLY

I have several ARIMA tutorials read on this site. Some use the difference function some not- like the parameter tunings. When do I need the difference function.

Jason Brownlee June 16, 2017 at 8:01 am # REPLY

When you have a trend, please read this:
http://machinelearningmastery.com/difference-time-series-dataset-python/

Luca June 20, 2017 at 6:28 pm #

Hi Jason

Since the dataset seems to have strong seasonality, in this case, do we need to first decompose the dat
applying models such as ARIMA?

Thanks

Jason Brownlee June 21, 2017 at 8:09 am #

It would be a good idea to seasonally adjust the dataset first:
http://machinelearningmastery.com/time-series-seasonality-with-python/

David Ravnsborg July 6, 2017 at 2:28 pm # REPLY

Hi Jason,

Thanks for all the ARIMA tutorials! I'm preparing to analyze some gait data from a biomechanics lab I w[redacted]
hang of the model.

Jason Brownlee July 9, 2017 at 10:22 am # REPLY

Consider a neural net model?

Srini July 15, 2017 at 4:53 am # REPLY

Hi Jason,

Imagine a scenario where we are using the fitted ARIMA model i.e. the coefficients on a new dataset (ou
previous data of length 'p' (history). Whereas for the MA part, one needs the residuals for past 'q' values
the residuals calculated for the new data. Do we use the residuals found in the training data? This is use
coefficients).

Jason Brownlee July 15, 2017 at 9:46 am #

Good question. The ARIMA model will make these available in "model_fit.resid" I believe.

buffy July 18, 2017 at 4:19 pm # REPLY

thanks for post, it is helpful.
But I have question, why the history data need to append test data for each loop:
model = ARIMA(history, order=(1,0,0))
obs = test[t]
history.append(obs)

If I do't know the test data value, for example just predict 6 month or 1 years days(365 days range) value[redacted] learning, used the train set to train model, and predict new data.

Jason Brownlee July 18, 2017 at 5:03 pm # REPLY

Because we are evaluating the model using walk-forward validation:
http://machinelearningmastery.com/backtest-machine-learning-models-time-series-forecasting/

Leave a Reply

Name (required)

Email (will not be published) (required)

Website

SUBMIT COMMENT

Welcome to Machine Learning Mastery

[redacted]

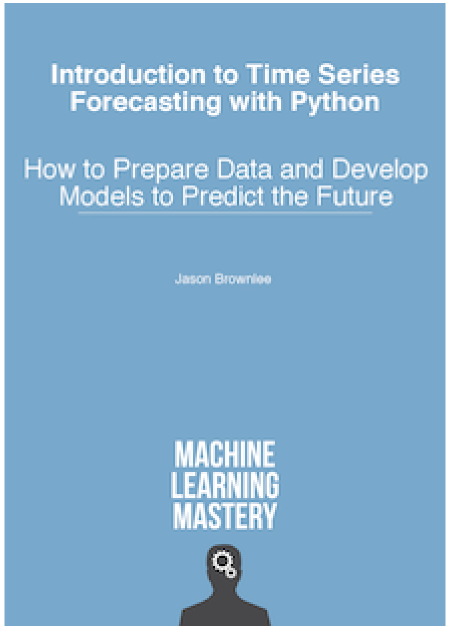
Hi, I'm Dr. Jason Brownlee.
My goal is to make practitioners like YOU awesome at applied machine learning.

Read More

Get Good at Time Series Forecasting

Need visualizations and forecast models?
Looking for step-by-step tutorials?
Want end-to-end projects?

Get Started with Time Series Forecasting in Python!



POPULAR

Get Your Start in Machine Learning


Time Series Prediction with LSTM Recurrent Neural Networks in Python with Keras
JULY 21, 2016


Your First Machine Learning Project in Python Step-By-Step
JUNE 10, 2016


Develop Your First Neural Network in Python With Keras Step-By-Step
MAY 24, 2016

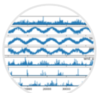

Sequence Classification with LSTM Recurrent Neural Networks in Python with Keras
JULY 26, 2016

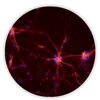

How to Setup a Python Environment for Machine Learning and Deep Learning with Anaconda
MARCH 13, 2017


Time Series Forecasting with the Long Short-Term Memory Network in Python
APRIL 7, 2017


Multi-Class Classification Tutorial with the Keras Deep Learning Library
JUNE 2, 2016


Regression Tutorial with the Keras Deep Learning Library in Python
JUNE 9, 2016


Multivariate Time Series Forecasting with LSTMs in Keras
AUGUST 14, 2017


How to Implement the Backpropagation Algorithm From Scratch In Python
NOVEMBER 7, 2016

Get Your Start in Machine Learning

Privacy | Contact | About