How to Tune ARIMA Parameters in Python

by **Jason Brownlee** on April 3, 2017 in **Time Series**

There are many parameters to consider when configuring an ARIMA model with Statsmodels in Python.

In this tutorial, we take a look at a few key parameters (other than the order parameter) that you may be curious about.

Specifically, after completing this tutorial, you will know:

- How to suppress noisy output from the underlying mathematical libraries when fitting an ARIMA model.
- The effect of enabling or disabling a trend term in your ARIMA model.
- The influence of using different mathematical solvers to fit coefficients to your training data.

Note, if you are interested in tuning the order parameter, see the post:

- How to Grid Search ARIMA Model Hyperparameters with Python

Let's get started.

Shampoo Sales Dataset

This dataset describes the monthly number of sales of shampoo over a 3 year period.

The units are a sales count and there are 36 observations. The original dataset is credited to Makridakis, Wheelwright, and Hyndman (1998).

You can download and learn more about the dataset here.

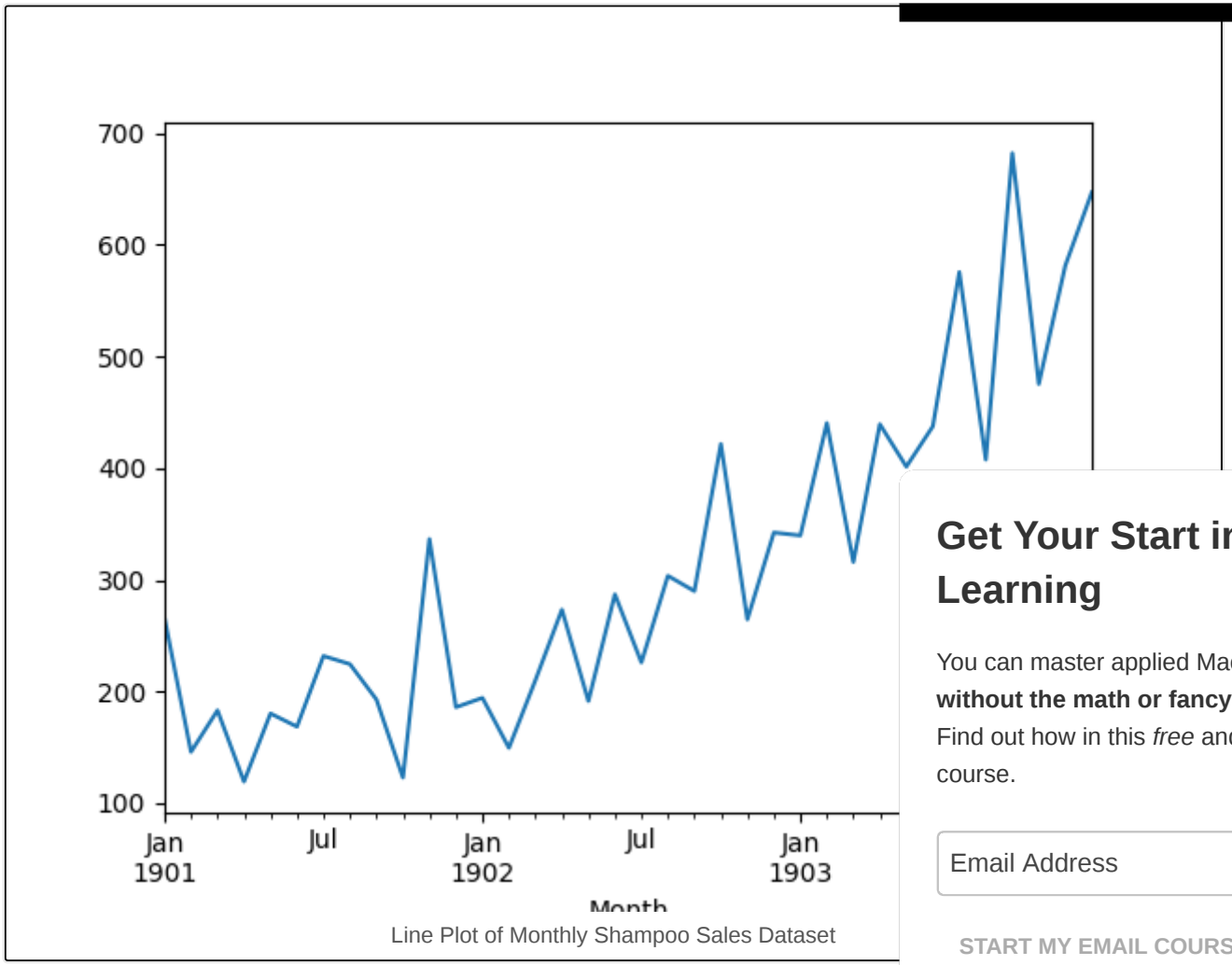
The example below loads and creates a plot of the loaded dataset.

```
# load and plot dataset
from pandas import read_csv
from pandas import datetime
from matplotlib import pyplot
# load dataset
def parser(x):
    return datetime.strptime('190'+x, '%Y-%m')
series = read_csv('shampoo-sales.csv', header=0, parse_dates=[0], index_col=0, squeeze=T
# summarize first few rows
print(series.head())
# line plot
series.plot()
pyplot.show()
```


Running the example loads the dataset as a Pandas Series and prints the first 5 rows.

```
Month
1901-01-01 266.0
1901-02-01 145.9
1901-03-01 183.1
1901-04-01 119.3
1901-05-01 180.3
Name: Sales, dtype: float64
```

A line plot of the series is then created showing a clear increasing trend.



Line Plot of Monthly Shampoo Sales Dataset

Experimental Test-Setup

It is important to evaluate time series forecasting models consistently.

In this section, we will define how we will evaluate the three forecast models in this tutorial.

First, we will hold the last one year of data back and evaluate forecasts on this data. Given the data is monthly, this means that the last 12 observations will be used as test data.

Get Your Start in Machine Learning

We will use a walk-forward validation method to evaluate model performance. This means that each time step in the test dataset will be enumerated, a model constructed on history data, and the forecast compared to the expected value. The observation will then be added to the training dataset and the process repeated.

Walk-forward validation is a realistic way to evaluate time series forecast models as one would expect models to be updated as new observations are made available.

Finally, forecasts will be evaluated using root mean squared error, or RMSE. The benefit of RMSE is that it penalizes large errors and the scores are in the same units as the forecast values (car sales per month).

An ARIMA(4,1,0) forecast model will be used as the baseline to explore the additional parameters of the model. This may not be the optimal model for the problem, but is generally skillful against some other hand tested configurations.

In summary, the test harness involves:

- The last 2 years of data used a test set.
- Walk-forward validation for model evaluation.
- Root mean squared error used to report model skill.
- An ARIMA(4,1,0) model will be used as a baseline.

The complete example is listed below.

```
from pandas import read_csv
from pandas import datetime
from matplotlib import pyplot
from statsmodels.tsa.arima_model import ARIMA
from sklearn.metrics import mean_squared_error
from math import sqrt
# load dataset
def parser(x):
    return datetime.strptime('190'+x, '%Y-%m')
series = read_csv('shampoo-sales.csv', header=0, parse_dates=[0], index_col=0, squeeze=True, date_parser=parser)
# split into train and test sets
X = series.values
train, test = X[0:-12], X[-12:]
history = [x for x in train]
predictions = list()
# walk-forward validation
for t in range(len(test)):
```

```
    # fit model
    model = ARIMA(history, order=(4,1,0))
    model_fit = model.fit()
    # one step forecast
    yhat = model_fit.forecast()[0]
    # store forecast and ob
    predictions.append(yhat)
    history.append(test[t])
# evaluate forecasts
rmse = sqrt(mean_squared_error(test, predictions))
print('Test RMSE: %.3f' % rmse)
# plot forecasts against actual outcomes
pyplot.plot(test)
pyplot.plot(predictions, color='red')
pyplot.show()
```

Running the example spews a lot of convergence information and finishes with an RMSE score of 84.832 monthly shampoo sales.

```
...
Tit   = total number of iterations
Tnf   = total number of function evaluations
Tnint = total number of segments explored during Cauchy searches
Skip  = number of BFGS updates skipped
Nact  = number of active bounds at final generalized Cauchy point
Projg = norm of the final projected gradient
F     = final function value

           * * *

   N    Tit     Tnf  Tnint  Skip  Nact     Projg        F
    5     15     20      1     0     0   8.882D-08   5.597D+00
  F =   5.5972342395324288

CONVERGENCE: REL_REDUCTION_OF_F_<=_FACTR*EPSMCH

 Cauchy                time 0.000E+00 seconds.
 Subspace minimization time 0.000E+00 seconds.
 Line search           time 0.000E+00 seconds.

 Total User time 0.000E+00 seconds.

Test RMSE: 84.832
```

A plot of the forecast vs the actual observations in the test harness is created to give some context for the model we are working with.

ARIMA Forecast for Monthly Shampoo Sales Dataset

Now let's dive into some of the other ARIMA parameters.

The "*disp*" Parameter

The first parameter we will look at is the *disp* parameter.

This is described as follows:

Get Your Start in Machine Learning

> *If True, convergence information is printed. For the default l_bfgs_b solver, disp controls the frequency of the output during the iterations. disp < 0 means no output in this case.*

By default, this parameter is set to 1, which shows output.

We are dealing with this first because it is critical in removing all of the convergence output when evaluating the ARIMA model using walk-forward validation.

Setting it to *False* turns off all of this noise.

The complete example is listed below.

```
from pandas import read_csv
from pandas import datetime
from matplotlib import pyplot
from statsmodels.tsa.arima_model import ARIMA
from sklearn.metrics import mean_squared_error
from math import sqrt
# load dataset
def parser(x):
    return datetime.strptime('190'+x, '%Y-%m')
series = read_csv('shampoo-sales.csv', header=0, parse_dates=[0], index_col=0, squeeze=T
# split into train and test sets
X = series.values
size = int(len(X) * 0.66)
train, test = X[0:size], X[size:len(X)]
history = [x for x in train]
predictions = list()
# walk-forward validation
for t in range(len(test)):
    # fit model
    model = ARIMA(history, order=(4,1,0))
    model_fit = model.fit(disp=False)
    # one step forecast
    yhat = model_fit.forecast()[0]
    # store forecast and ob
    predictions.append(yhat)
    history.append(test[t])
# evaluate forecasts
rmse = sqrt(mean_squared_error(test, predictions))
print('Test RMSE: %.3f' % rmse)
```

Running this example not only produces a cleaner output, but also is much faster to execute.

```
Test RMSE: 81.545
```

We will leave *disp=False* on all following examples.

The "*transparams*" Parameter

This parameter controls whether or not to perform a transform on AR parameters.

Specifically, it is described as:

> *Whether or not to transform the parameters to ensure stationarity. Uses the transformation su r stationarity or invertibility is done.*

By default, *transparams* is set to *True*, meaning this transform is performed.

This parameter is also used on the R version of the ARIMA implementation (see docs) and I expect

The statsmodels doco is weak on this, but you can learn more about the transform in the paper:

- Maximum Likelihood Fitting of ARMA Models to Time Series With Missing Observations

The example below demonstrates turning this parameter off.

```
from pandas import read_csv
from pandas import datetime
from matplotlib import pyplot
from statsmodels.tsa.arima_model import ARIMA
from sklearn.metrics import mean_squared_error
from math import sqrt
# load dataset
def parser(x):
    return datetime.strptime('190'+x, '%Y-%m')
series = read_csv('shampoo-sales.csv', header=0, parse_dates=[0], index_col=0, squeeze=True, date_parser=parser)
# split into train and test sets
X = series.values
size = int(len(X) * 0.66)
```

```
train, test = X[0:size], X[size:len(X)]
history = [x for x in train]
predictions = list()
# walk-forward validation
for t in range(len(test)):
    # fit model
    model = ARIMA(history, order=(4,1,0))
    model_fit = model.fit(disp=False, transparams=False)
    # one step forecast
    yhat = model_fit.forecast()[0]
    # store forecast and ob
    predictions.append(yhat)
    history.append(test[t])
# evaluate forecasts
rmse = sqrt(mean_squared_error(test, predictions))
print('Test RMSE: %.3f' % rmse)
```

Running this example results in more convergence warnings from the solver.

The RMSE of the model with *transparams* turned off also results in slightly worse results on this data

Experiment with this parameter on and off on your dataset and confirm it results in a benefit.

```
...
.../site-packages/statsmodels/base/model.py:496: ConvergenceWarning: Maximum Likelihood c                    vals
  "Check mle_retvals", ConvergenceWarning)
Test RMSE: 81.778
```

The "*trend*" Parameter

The *trend* parameter adds an additional constant term to the model. Think of it like a bias or intercep

It is described as:

> *Whether to include a constant or not. 'c' includes constant, 'nc' no constant.*

By default, a trend term is enabled with *trend* set to '*c*'.

We can see the effect clearly if we rerun the original example and print the model coefficients for each step of the walk-forward validation and compare the same with the trend term turned off.

The below example prints the coefficients each iteration with the trend constant enabled (the default).

```
from pandas import read_csv
from pandas import datetime
from matplotlib import pyplot
from statsmodels.tsa.arima_model import ARIMA
from sklearn.metrics import mean_squared_error
from math import sqrt
# load dataset
def parser(x):
    return datetime.strptime('190'+x, '%Y-%m')
series = read_csv('shampoo-sales.csv', header=0, parse_dates=[0], index_col=0, squeeze=True, date_parser=parser)
# split into train and test sets
X = series.values
size = int(len(X) * 0.66)
train, test = X[0:size], X[size:len(X)]
history = [x for x in train]
predictions = list()
# walk-forward validation
for t in range(len(test)):
    # fit model
    model = ARIMA(history, order=(4,1,0))
    model_fit = model.fit(disp=False, trend='c')
    print(model_fit.params)
    # one step forecast
    yhat = model_fit.forecast()[0]
    # store forecast and ob
    predictions.append(yhat)
    history.append(test[t])
# evaluate forecasts
rmse = sqrt(mean_squared_error(test, predictions))
print('Test RMSE: %.3f' % rmse)
```


Running the example shows the 4 AR terms specified in the order of the model plus the first term in the array, which is a trend constant.

Note that one set of parameters is printed for each model fit, one for each step of the walk-forward validation.

```
...
[ 11.42283717  -1.16087885  -0.6519841   -0.547411    -0.28820764]
[ 11.75656838  -1.11443479  -0.61607471  -0.49084722  -0.24452864]
[ 11.40486702  -1.11705478  -0.65344924  -0.50213939  -0.25677931]
```

```
Test RMSE: 81.545
```

We can repeat this experiment with the trend term disabled (*trend='nc'*), as follows.

```
from pandas import read_csv
from pandas import datetime
from matplotlib import pyplot
from statsmodels.tsa.arima_model import ARIMA
from sklearn.metrics import mean_squared_error
from math import sqrt
# load dataset
def parser(x):
    return datetime.strptime('190'+x, '%Y-%m')
series = read_csv('shampoo-sales.csv', header=0, parse_dates=[0], index_col=0, squeeze=True, date_parser=parser)
# split into train and test sets
X = series.values
size = int(len(X) * 0.66)
train, test = X[0:size], X[size:len(X)]
history = [x for x in train]
predictions = list()
# walk-forward validation
for t in range(len(test)):
    # fit model
    model = ARIMA(history, order=(4,1,0))
    model_fit = model.fit(disp=False, trend='nc')
    print(model_fit.params)
    # one step forecast
    yhat = model_fit.forecast()[0]
    # store forecast and ob
    predictions.append(yhat)
    history.append(test[t])
# evaluate forecasts
rmse = sqrt(mean_squared_error(test, predictions))
print('Test RMSE: %.3f' % rmse)
```


Running the example shows a slightly worse RMSE score on this problem, with this ARIMA configuration.

We can see that the constant term (11.xxx) removed from the array of coefficients each iteration.

```
...
[-0.90717131 -0.22332019 -0.11240858 -0.04008561]
[-0.88836083 -0.21098412 -0.09046333 -0.02121404]
[-0.89260136 -0.24120301 -0.10243393 -0.03165432]
Test RMSE: 95.061
```

Experiment on your own problem and determine whether this constant improves performance.

My own experimentation suggests that ARIMA models may be less likely to converge with the *trend* term disabled, especially when using more than zero MA terms.

The "*solver*" Parameter

The *solver* parameter specifies the numerical optimization method to fit the coefficients to the data.

There is often little reason to tune this parameter other than execution speed if you have a lot of data. The differences will likely be quite minor.

The parameter is described as follows:

> *Solver to be used. The default is 'lbfgs' (limited memory Broyden-Fletcher-Goldfarb-Shanno) Raphson), 'nm' (Nelder-Mead), 'cg' – (conjugate gradient), 'ncg' (non-conjugate gradient), ar uses m=12 to approximate the Hessian, projected gradient tolerance of 1e-8 and factr = 1e2.*

The default is the fast "*lbfgs*" method (Limited-memory BFGS).

Nevertheless, below is an experiment that compares the RMSE model skill and execution time of ea

```
from pandas import read_csv
from pandas import datetime
from matplotlib import pyplot
from statsmodels.tsa.arima_model import ARIMA
from sklearn.metrics import mean_squared_error
from math import sqrt
from time import time
# load dataset
def parser(x):
    return datetime.strptime('190'+x, '%Y-%m')
series = read_csv('shampoo-sales.csv', header=0, parse_dates=[0], index_col=0, squeeze=True, date_parser=parser)
# split into train and test sets
X = series.values
size = int(len(X) * 0.66)
train, test = X[0:size], X[size:len(X)]
# solvers
solvers = ['lbfgs', 'bfgs', 'newton', 'nm', 'cg', 'ncg', 'powell']
```

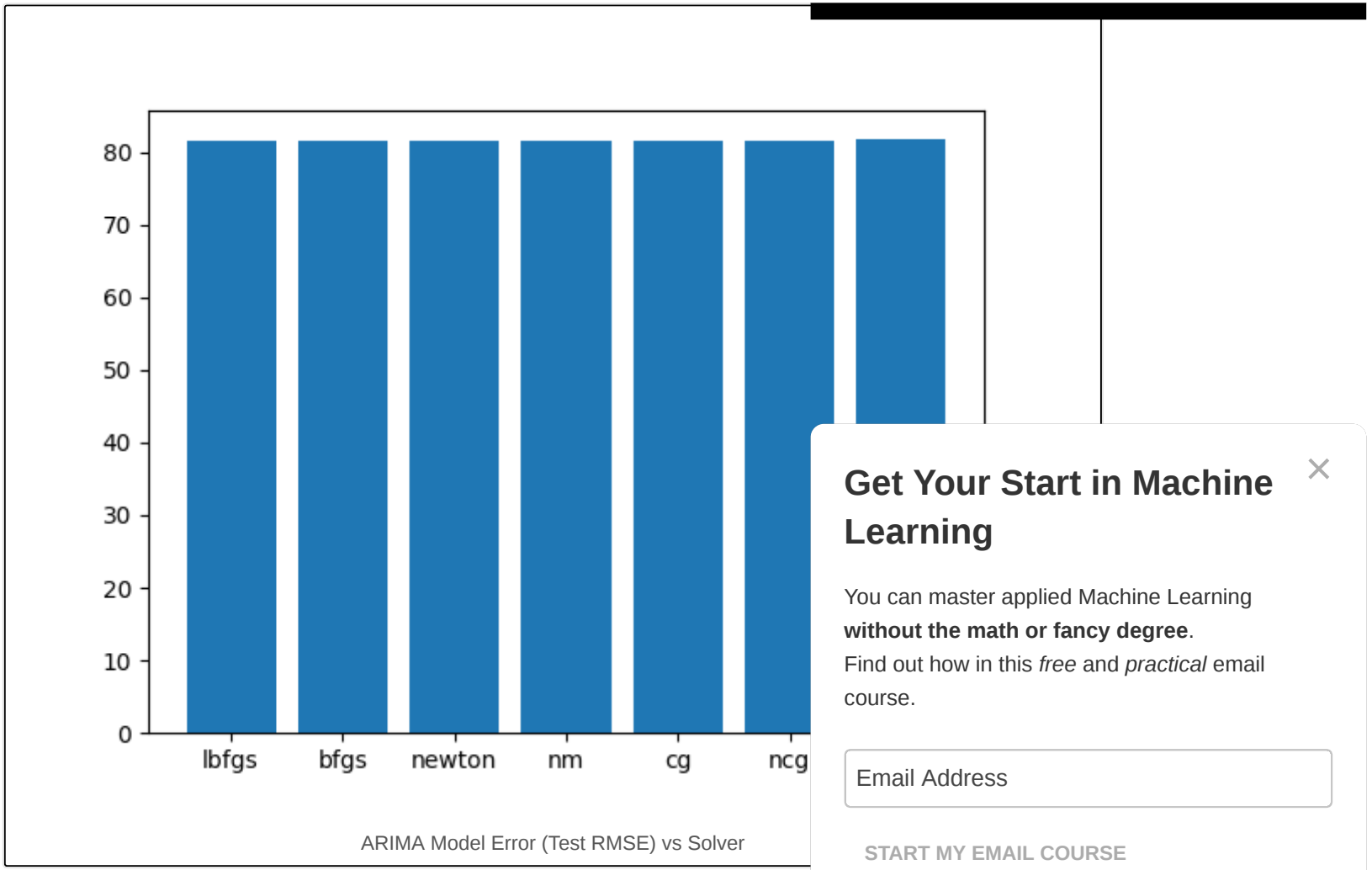
```
scores = []
times = []
for solver in solvers:
    start_time = time()
    history = [x for x in train]
    predictions = list()
    # walk-forward validation
    for t in range(len(test)):
        # fit model
        model = ARIMA(history, order=(4,1,0))
        model_fit = model.fit(disp=False, solver=solver)
        # one step forecast
        yhat = model_fit.forecast()[0]
        # store forecast and ob
        predictions.append(yhat)
        history.append(test[t])
    # evaluate forecasts
    rmse = sqrt(mean_squared_error(test, predictions))
    timing = time() - start_time
    scores.append(rmse)
    times.append(timing)
    print('Solver=%s, Test RMSE: %.3f, Time=%f' % (solver, rmse, timing))
# plot scores
ticks = [i for i in range(len(solvers))]
pyplot.bar(ticks, scores)
pyplot.xticks(ticks, solvers)
pyplot.show()
# plot times
ticks = [i for i in range(len(solvers))]
pyplot.bar(ticks, times)
pyplot.xticks(ticks, solvers)
pyplot.show()
```


Running the example prints the RMSE and time in seconds of each *solver*.

```
Solver=lbfgs, Test RMSE: 81.545, Time=1.630316
Solver=bfgs, Test RMSE: 81.545, Time=2.122630
Solver=newton, Test RMSE: 81.545, Time=2.418718
Solver=nm, Test RMSE: 81.472, Time=1.432801
Solver=cg, Test RMSE: 81.543, Time=3.474055
Solver=ncg, Test RMSE: 81.545, Time=2.643767
Solver=powell, Test RMSE: 81.704, Time=1.839257
```

A graph of *solver* vs RMSE is provided. As expected, there is little difference between the solvers on this small dataset.

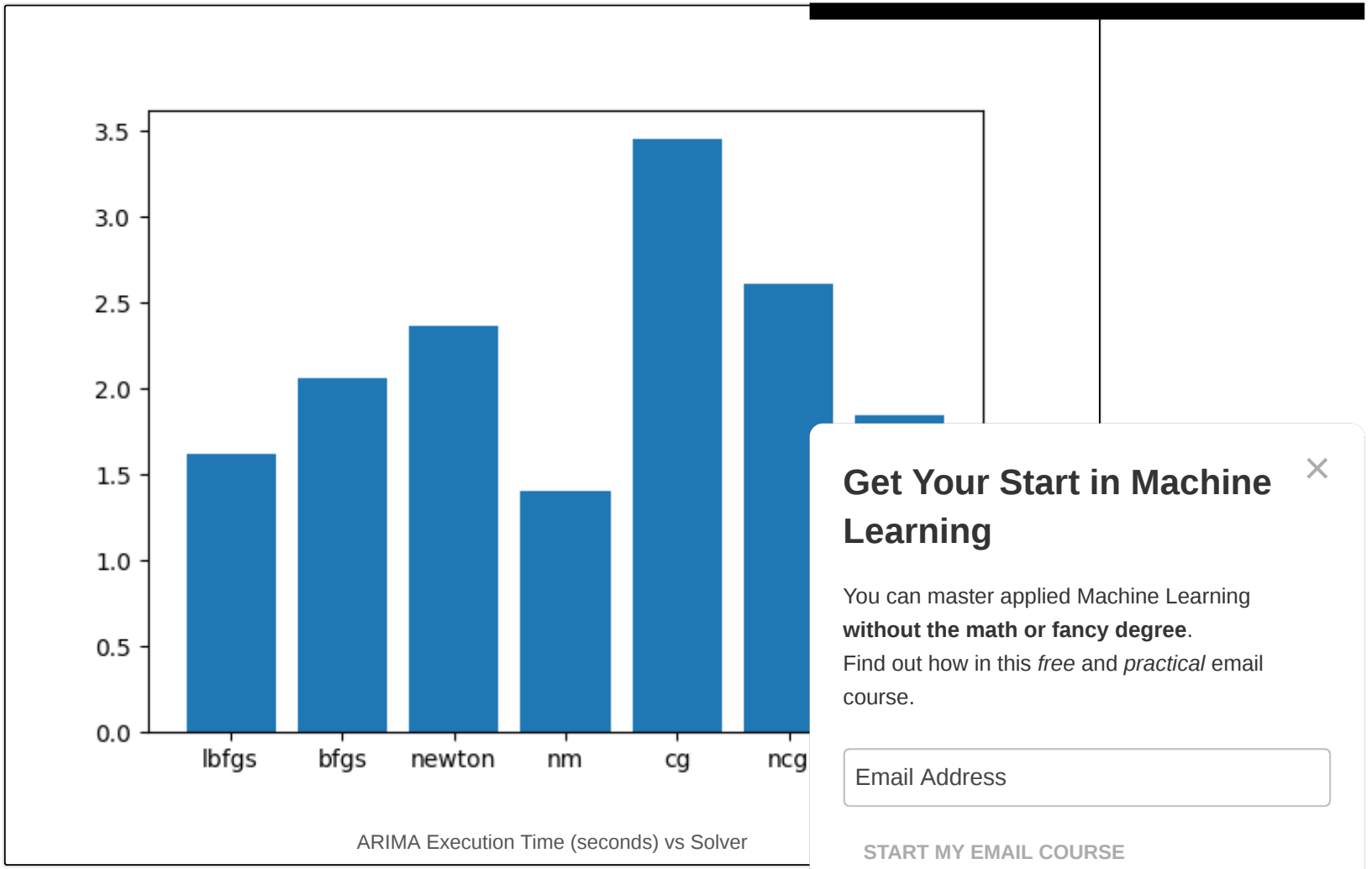
You may see different results or different stability of the solvers on your own problem.



ARIMA Model Error (Test RMSE) vs Solver

A graph of *solver* vs execution time in seconds is also created. The graph shows a marked difference between solvers.

Generally, "*lbfgs*" and "*bfgs*" provide good real-world tradeoff between speed, performance, and stability.



ARIMA Execution Time (seconds) vs Solver

If you do decide to test out solvers, you may also want to vary the "*maxiter*" that limits the number of iterations before converge, the "*tol*" parameter that defines the precision of convergence, and the "*method*" parameter that defines the cost function being optimized.

Additional Resources

This section lists some resources you may find useful alongside this tutorial.

- ARIMA Class API

- ARIMAResults Class API
- Source code for the ARIMA and ARIMAResults classes.
- How to Grid Search ARIMA Model Hyperparameters with Python

Summary

In this tutorial, you discovered some of the finer points in configuring your ARIMA model with Statsmodels in Python.

Specifically, you learned:

- How to turn off the noisy convergence output from the solver when fitting coefficients.
- How to evaluate the difference between different solvers to fit your ARIMA model.
- The effect of enabling and disabling a trend term in your ARIMA model.

Do you have any questions about fitting your ARIMA model in Python?
Ask your question in the comments below and I will do my best to answer.

About Jason Brownlee

Dr. Jason Brownlee is a husband, proud father, academic researcher, author, professional develo to helping developers get started and get good at applied machine learning. Learn more.

View all posts by Jason Brownlee →

‹ Simple Time Series Forecasting Models to Test So That You Don't Fool Yourself

Seasonal Persistence Forecasting With Python ›

Get Your Start in Machine Learning

22 Responses to *How to Tune ARIMA Parameters in Python*

Hans June 14, 2017 at 1:55 am #

REPLY

Should we use a data row which is not in test nor in train, to forecast an unseen data step with the fitted model? Or should we simply use the last prediction of test?

Jason Brownlee June 14, 2017 at 8:47 am #

Be careful to separate the concerns of making a forecast for new data and evaluating the sk

A final model should be fit on all available data before being used to make predictions. See this post
http://machinelearningmastery.com/train-final-machine-learning-model/

Hans June 15, 2017 at 3:06 am #

Thank you, but at least I need one new x (data row) which was not involved in training, to predic

Jason Brownlee June 15, 2017 at 8:48 am #

REPLY

Yes, you need one new row of input data to predict a new output.

In the case of time series, this is often lag observation, but really depends on how you have defined your model (the inputs it expects).

Hans June 15, 2017 at 5:09 am #

I have found good parameter settings resulting in a RMSE less then 2.5.
Is there a tutorial available how to finalize, train this model and predict unseen data on a fitted ARIMA model?

REPLY

Jason Brownlee June 15, 2017 at 8:52 am #

Yes, in the future please use the search feature of the blog (like I just did).
http://machinelearningmastery.com/make-sample-forecasts-arima-python/

REPLY

Hans June 15, 2017 at 12:01 pm #

The problem is, I have no seasonal data an no evenly distributed daily data.
I try to use a fitted model of the example of this site.
Right after the for loop I say:

```
start_index = len(test)
end_index = len(test)
forecast = model_fit.predict(start=start_index, end=end_index)

print(forecast)
```

This gives me a forecast of 0.664 while my data values are between 1.0 and 10.0.
What do I miss here?

REPLY

Jason Brownlee June 16, 2017 at 7:48 am #

You must adapt a given tutorial to your specific problem. No one can tell you how – you must use all of the information available to explore the best model for your problem.

Get Your Start in Machine Learning

Hans June 16, 2017 at 12:45 pm #

I try this since months with more or less success.

Everything works fine even with own data (training and testings), until it comes to real out of sample forecasts.

But I'm not talking about problems, I'm talking about coding techniques.

Is it right to use the test data set to build start and end indexes for the ARIMA prediction?

Do we have to make special preparations to such inputs, to feed this two parameters?

Jason Brownlee June 17, 2017 at 7:21 am #

The idea of test data goes away when you start making predictions on real data

See this post:
http://machinelearningmastery.com/train-final-machine-learning-model/

Hans June 19, 2017 at 5:07 am #

Then I seem to be on the right way.
I have training and predictions strictly separated in my batch environment (menu).
For predictions I always use the complete data, except of one row/x to predict unseen da
Is this right in general?

Jason Brownlee June 19, 2017 at 8:46 am #

Yes.

Get Your Start in Machine Learning

Hans June 15, 2017 at 5:41 am #

Is this a time series analysis?

REPLY

Jason Brownlee June 15, 2017 at 8:54 am #

No, a demonstration of how to use ARIMA parameters in statsmodels for forecasting.

REPLY

Rajesh July 6, 2017 at 3:33 pm #

Hi jason,
Thank you very much for your tutorials.

Here, you selected a base line model for ARIMA(4,1,0)

But, as per procedure, order need to be decided based on ACF and PACF factors
I would like to decide order based on data.
Can you suggest a procedure

Jason Brownlee July 9, 2017 at 10:24 am #

Yes, see this post:
http://machinelearningmastery.com/gentle-introduction-autocorrelation-partial-autocorrelation/

Kashif August 15, 2017 at 11:49 pm #

REPLY

Get Your Start in Machine Learning

Hi Sir
I am applying ARIMA model on my CDR dataset. I have checked that my data is non stationary (Augmented Dickey-Fuller test)
ADF Statistic: -1.569036
p-value: 0.499127
Critical Values:
1%: -3.478
10%: -2.578
5%: -2.882
Than I have plotted ACF, it showed that my first 15 lag values have autocorrelation value greater then 0.5. so I set 'p' parameter 15 (is p =15 is correct ?), and 'd' is 1 for stationarity. Could you please guide me how I will find the value of moving average MA(q) q parameter ?
Can I determine the value of 'q' parameter by visualizing ACF plot ?
Thanks

Jason Brownlee August 16, 2017 at 6:36 am #

See this post:
http://machinelearningmastery.com/gentle-introduction-autocorrelation-partial-autocorrelation/

Terry October 4, 2017 at 3:18 pm #

Hi Jason,

Do you have a similar tutorial for ARIMAX?

Jason Brownlee October 4, 2017 at 3:40 pm #

Not at this stage Terry.

REPLY

Get Your Start in Machine Learning

Terry October 5, 2017 at 11:50 am #

REPLY

Would be really keen to see a step by step guide of ARIMAX application to a real-life problem and tuning. If you add this topic to your book, it will be invaluable!

Jason Brownlee October 5, 2017 at 5:23 pm #

REPLY

Thanks Terry.

Leave a Reply

Name (required)

Email (will not be published) (required)

Website

SUBMIT COMMENT

Welcome to Machine Learning Mastery

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

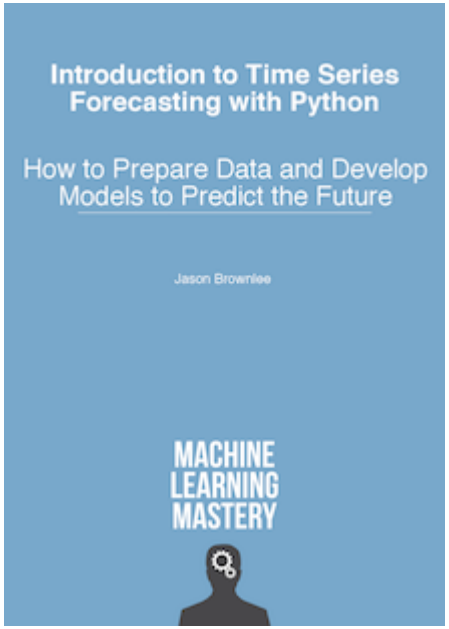
Read More

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!

Get Your Start in Machine Learning



POPULAR

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

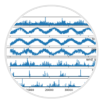
How to Setup a Python Environment for Machine Learning and Deep Learning with Anaconda
MARCH 13, 2017

Get Your Start in Machine Learning

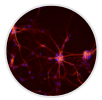
Time Series Forecasting with the Long Short-Term Memory Network in Python
APRIL 7, 2017

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

Privacy | Contact | About