ARIMA Models with Turing.jl

Using the Probabilistic Programming Language (PPL) Turing.jl for Time Series Analysis and Prediction. Julia implementation can be found here (Jupyter notebook) or here (.jl file).

Saumya Shah Aug 27, 2019 · 6 min read ★

Time Series Models often have useful applications in the field of finance; Photo by Markus Spiske on Unsplash

Hello!

This article is a part of my work done in Julia Season of Contributions (JSoC) 2019. It describes the Julia implementation of ARIMA models using the Probabilistic Programming Language (PPL) Turing.jl, which provides great ease in defining probabilistic models. This aspect of Turing will become more obvious when we look at model definitions later on in the article. Furthermore, Turing supports the use of custom distributions for specifying models.

Okay, so let's get straight onto it!

Importing Libraries

Loading and Visualising the Dataset

We will use a dataset containing S&P 500 Adjusted Closing Values 1995–2015 with a monthly frequency. The dataset can be downloaded from here (MIT License).

The following plot is obtained after running the code above. This is essentially how our data looks like, plotting the value at each time index:

Complete Data

We split the data into training and test sets, taking 95% of the data as training set:

Train Data

Check for Stationarity

We can see from the plots that the mean of the series rises towards the end. So, the series is not stationary. This is reinforced by the Augmented Dickey-Fuller (ADF)Test for stationarity:

ADF test before differencing

Thus, we difference the time series in an attempt to remove the stationarity:

Plot of the differenced time series

This series seems to have a roughly constant mean, though this does not necessarily mean that the series is stationary. We use the ADF test once again:

ADF test after differencing

Success! We can safely assume this series to be stationary since the p-value is so low. We can now move on to selecting the AR and MA terms for our differenced time series with the help of ACF and PACF plots.

Selecting MA and AR terms with the help of ACF and PACF plots

The ACF and PACF plots obtained for our training data are as shown below:

These plots can be construed in the following ways:

- We can have a first-order autoregressive term since there is positive autocorrelation at lag 1.
- The PACF plot decays gradually and the ACF plot cuts off sharply after a few lags. This suggests the presence of a moving average term. The order this term should be one since all the lags greater than 1 do not have a significant correlation.

Having both AR and MA terms in an ARIMA model is not very common. So, we will not consider this case. From the two points above, it seems that the model is more likely to have a Moving Average term. Nevertheless, we will consider two plausible cases for our ARIMA model:

- *ARIMA(1,1,0)*
- *ARIMA(0,1,1)*

The notation for an *ARIMA(p, d, q)* model is defined as follows:

- p: The number of autoregressive terms
- q: The number of moving average terms
- d: The order of differencing

We implement both of these cases and compare the models using the Akaike information criterion (AIC). This webpage[1] is used as a reference for defining the *ARIMA(1, 1, 0)* and *ARIMA(0, 1, 1)* models below.

ARIMA(1,1,0)

The *ARIMA(1,1,0)* model is defined as follows:

```
@model ARIMA110(x) = begin
    T = length(x)
    μ ~ Uniform(-10, 10)
    ϕ ~ Uniform(-1, 1)
    for t in 3:T
        val = μ +                       # Drift term.
              x[t-1] +                  # ARIMA(0,1,0) portion.
              ϕ * (x[t-1] - x[t-2]) # ARIMA(1,0,0) portion.
        x[t] ~ Normal(val, 1)
    end
end
```

Here, x is the original time series as we have accounted for differencing in the model definition itself. Note that we will have one autoregressive term since $p = 1$.

ARIMA(0,1,1)

The *ARIMA(1,1,0)* model is defined as follows:

```
@model ARIMA011(x) = begin
    T = length(x)

    # Set up error vector.
    ϵ = Vector(undef, T)
    x_hat = Vector(undef, T)

    θ ~ Uniform(-5, 5)

    # Treat the first x_hat as a parameter to estimate.
    x_hat[1] ~ Normal(0, 1)
    ϵ[1] = x[1] - x_hat[1]

    for t in 2:T
        # Predicted value for x.
        x_hat[t] = x[t-1] - θ * ϵ[t-1]
        # Calculate observed error.
        ϵ[t] = x[t] - x_hat[t]
        # Observe likelihood.
        x[t] ~ Normal(x_hat[t], 1)
    end
end
```

As in the previous model definition, x is the original time series. Note that we will have one Moving Average term since $q = 1$.

A point to be observed here is how the code written in Turing is essentially the same as it would be written on a piece of paper. This is evident from the model definitions above where one can understand these model definitions simply by looking at the code.

Sampling

The chain is sampled using the NUTS sampler. You can check out the docs to know more about NUTS and several other samplers that Turing supports. The code for sampling is as follows:

To get the visualisations and the summary statistics for the parameters, you can have a look at them in the code here (Jupyter notebook) or here (.jl file).

Comparing AIC Values

The Akaike information criterion (AIC) measures the relative "goodness" of different statistical models. Thus, it can be used for the purpose of model comparison. The lower the value of AIC, the better the model is. Also, one must remember that the absolute value of AIC does not have much meaning, the relative values are what matter. Mathematically, AIC is given by:

Formula for AIC

Using this formula, we can calculate the AIC values for our two models. This PDF[2] has been used as a reference for calculating the AIC values of the two models.

- *ARIMA(1,1,0)*

```
function calculate_aic_ARIMA110(β::Float64, μ:: Float64, σ::Float64,
s::Array{Float64, 1})
```

```
    T = length(s)
    ϵ = Vector(undef, T)
    s_pred = Vector(undef, T)

    s_pred[1], s_pred[2] = s[1], s[2]
    ϵ[1], ϵ[2] = 0.0, 0.0
    for t in 3:T
        s_pred[t] = μ +
              s_pred[t-1] +
              β * (s_pred[t-1] - s_pred[t-2])
        ϵ[t] = s_pred[t] - s[t]
    end
    log_likelihood = (-(T - 1)/2)*2*π*σ^2 - (1/σ^2)*sum(ϵ.^2)
    - π*σ^2/(1 - β^2) - ((s[1] - μ/(1 - β))^2)/(2*σ^2/(1 - β^2))
    aic = -2*log_likelihood + 2
    return aic
end
```

Using this function, we get the value of AIC for ARIMA(1,1,0) as approximately -299655.26

- *ARIMA(0,1,1)*

```
function calculate_aic_ARIMA011(β::Float64, σ::Float64,
s::Array{Float64, 1})
    T = length(s)
    ϵ = Vector(undef, T)
    s_pred = Vector(undef, T)

    s_pred[1] = s[1]
    ϵ[1] = 0.0
    for t in 2:T
        s_pred[t] = s[t-1] - β * ϵ[t-1]
        ϵ[t] = s[t] - s_pred[t]
    end
    log_likelihood = -(T/2)*log(2π) - (T/2)*log(σ^2) +
    (1/2*σ^2)*sum(ϵ.^2)
    aic = -2*log_likelihood + 1
    return aic
end
```

This brings to an end my article on ARIMA Models with Turing.jl. I hope you found it interesting. If you have any questions or doubts regarding this article, feel free to contact me at sshah@iitk.ac.in or you can tag me on the Julia slack with @Saumya Shah.

References

[1] Introduction to ARIMA models. (2019). Retrieved 26 August 2019, from https://people.duke.edu/~rnau/411arim.htm

[2] Thomas, S. (2009). *Estimating AR/MA models* [Ebook]. Retrieved from http://www.igidr.ac.in/faculty/susant/TEACHING/TSA/print06.pdf

Thanks to Cameron Pfiffer.

Using this function, we get the value of AIC for ARIMA(1,1,0) as approximately 6.22 x

Julialang Bayesian Machine Learning Towards Data Science Timeseries

Probabilistic Programming