Time series models are very important in business applications, they have been used over the last 30 years to great success. Lots of business rely on time series model to project earnings and capital expenditure and to prepare budgets for the next economic period. There are various methods for forecasting values of a time series from past observations. Given a time series of spaced observations Yt, t = 1,2, ..., n on a quantity Y, we require to forecast Yf+h, h= 1, 2, 3, There are a good number of techniques, with varying levels of complexity, that can deliver reasonable accurate results.

### Stationarity

### A static procedure includes the property that the mean, variance and autocorrelation structure usually do not change over time.

### Random Walk Theory

The random walk theory implies that fluctuations in stock prices have exactly the same distribution and are independent of one another, therefore, the last movement or tendency of a stock market or price cannot be employed to predict its future movement. In short, this really is the idea that stocks take a random and unpredictable path

### Autocorrelation

[Autocorrelation](http://en.wikipedia.org/wiki/Autocorrelation) is a measure of the internal correlation within a time series. It is a way of measuring and explaining internal association between observations in a time series. For example, for the measles time series you might ask: “*take an arbitrary spot in time, on the average what does the time series look like in four weeks time, compared to now?* ” In other words, you are asking a question about how strong is the internal association within the series at a period of four weeks? That association could be very strong and positive (i.e. the series in four weeks is similar to now), or be very strong and negative (i.e. the series in four weeks is very dissimilar to now) or there could be a weak or no relationship (i.e. there is no identifiable association). Also, you might also be interested in time periods other than four weeks. For example, you might be interested in periods of one week (and for measles you might expect that over one week the series does not change very much, so the association would be strong over periods of one week) or periods of a 52 weeks, a year (and again, you might assume that the association might be strong since measles epidemics have an annual component).

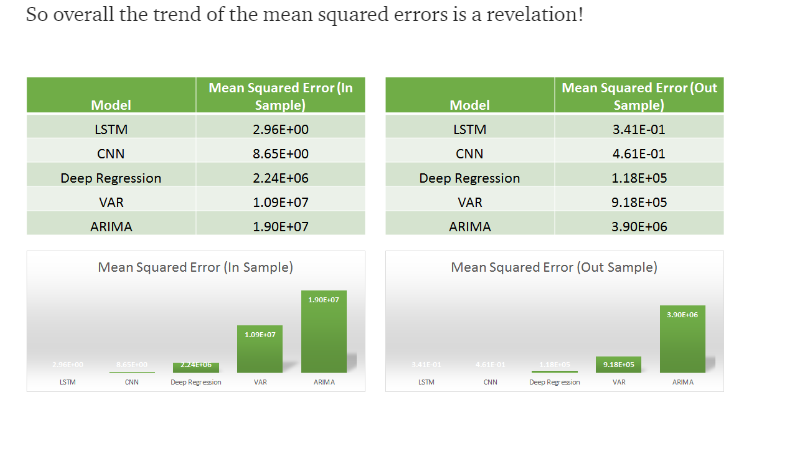
Autocorrelation quantifies the internal association: assigning a value of +1 to strong positive association, -1 to strong negative association and 0 to no association (note: autocorrelation measures linear association, in the same way that ordinary [correlation](http://en.wikipedia.org/wiki/Correlation) between two variables does).

### Dickey Fuller Test of Stationarity

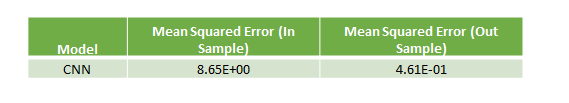
### ARIMA

### AR Model

### MA Mode

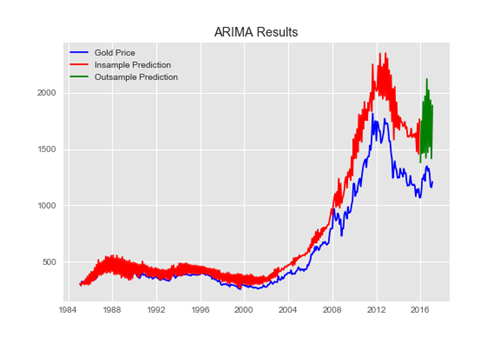


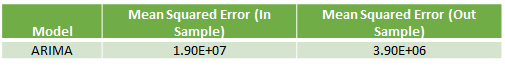
For each model



[**ARIMA**](https://en.wikipedia.org/wiki/Autoregressive_integrated_moving_average)

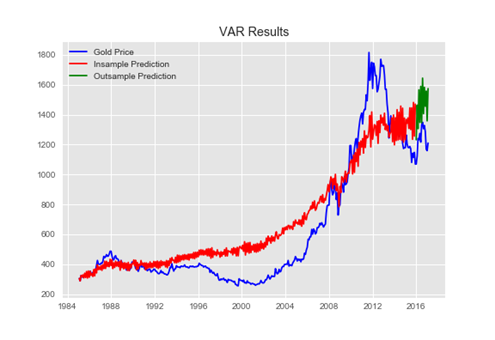
Using the Autoregressive Integrated moving Average model, which tries to predict a stationary time series keeping the seasonal component in place we get a result

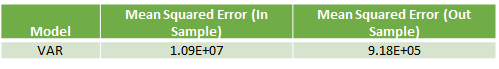




[**VAR**](https://en.wikipedia.org/wiki/Vector_autoregression)

If we add related predictor variables to our auto-regressive model and move to a Vector Auto Regressive model, we get these results —





Predicting steps ahead

However by changing the number of neurons in LSTM neural network prediction code, it was possible to improve MSE and get predictions much closer to actual data. Below are screenshots of plots of prediction and actual data for LSTM with 5, 60 and 1000 neurons. Note that the plot is showing only testing data. Training data is not shown on the plots. Also you can find below the data for MSE and number of neurons. As the number neorons in LSTM layer is changing there were 2 minimums (one around 100 neurons and another around 1000 neurons)

**LSTM Data Preparation**

Before we can fit an LSTM model to the dataset, we must transform the data.

This section is broken down into three steps:

1. Transform the time series into a supervised learning problem
2. Transform the time series data so that it is stationary.
3. Transform the observations to have a specific scale.

**Transform Time Series to Supervised Learning**

The LSTM model in Keras assumes that your data is divided into input (X) and output (y) components.

For a time series problem, we can achieve this by using the observation from the last time step (t-1) as the input and the observation at the current time step (t) as the output.

We can achieve this using the [shift()](http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.shift.html) function in Pandas that will push all values in a series down by a specified number places. We require a shift of 1 place, which will become the input variables. The time series as it stands will be the output variables.

We can then concatenate these two series together to create a DataFrame ready for supervised learning. The pushed-down series will have a new position at the top with no value. A NaN (not a number) value will be used in this position. We will replace these NaN values with 0 values, which the LSTM model will have to learn as “the start of the series” or “I have no data here,” as a month with zero sales on this dataset has not been observed.

The code below defines a helper function to do this called *timeseries\_to\_supervised()*. It takes a NumPy array of the raw time series data and a lag or number of shifted series to create and use as inputs.

We can test this function with our loaded Shampoo Sales dataset and convert it into a supervised learning problem.

### Transform Time Series to Stationary

The Shampoo Sales dataset is not stationary.

This means that there is a structure in the data that is dependent on the time. Specifically, there is an increasing trend in the data.

Stationary data is easier to model and will very likely result in more skillful forecasts.

The trend can be removed from the observations, then added back to forecasts later to return the prediction to the original scale and calculate a comparable error score.

A standard way to remove a trend is by differencing the data. That is the observation from the previous time step (t-1) is subtracted from the current observation (t). This removes the trend and we are left with a difference series, or the changes to the observations from one time step to the next.

We can achieve this automatically using the [diff() function](http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.diff.html) in pandas. Alternatively, we can get finer grained control and write our own function to do this, which is preferred for its flexibility in this case.

Below is a function called difference() that calculates a differenced series. Note that the first observation in the series is skipped as there is no prior observation with which to calculate a differenced value.

We also need to invert this process in order to take forecasts made on the differenced series back into their original scale.

The function below, called *inverse\_difference()*, inverts this operation.

We can test out these functions by differencing the whole series, then returning it to the original scale, as follows:

Running the example prints the first 5 rows of the loaded data, then the first 5 rows of the differenced series, then finally the first 5 rows with the difference operation inverted.

Note that the first observation in the original dataset was removed from the inverted difference data. Besides that, the last set of data matches the first as expected.

### Transform Time Series to Scale

Like other neural networks, LSTMs expect data to be within the scale of the activation function used by the network.

The default activation function for LSTMs is the hyperbolic tangent (tanh), which outputs values between -1 and 1. This is the preferred range for the time series data.

To make the experiment fair, the scaling coefficients (min and max) values must be calculated on the training dataset and applied to scale the test dataset and any forecasts. This is to avoid contaminating the experiment with knowledge from the test dataset, which might give the model a small edge.

We can transform the dataset to the range [-1, 1] using the [MinMaxScaler class](http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html). Like other scikit-learn transform classes, it requires data provided in a matrix format with rows and columns. Therefore, we must reshape our NumPy arrays before transforming.

For example:

Again, we must invert the scale on forecasts to return the values back to the original scale so that the results can be interpreted and a comparable error score can be calculated.

Putting all of this together, the example below transforms the scale of the Shampoo Sales data.

Now that we know how to prepare data for the LSTM network, we can start developing our model.

## LSTM Model Development

The Long Short-Term Memory network (LSTM) is a type of Recurrent Neural Network (RNN).

A benefit of this type of network is that it can learn and remember over long sequences and does not rely on a pre-specified window lagged observation as input.

In Keras, this is referred to as stateful, and involves setting the “stateful” argument to “True” when defining an LSTM layer.

By default, an LSTM layer in Keras maintains state between data within one batch. A batch of data is a fixed-sized number of rows from the training dataset that defines how many patterns to process before updating the weights of the network. State in the LSTM layer between batches is cleared by default, therefore we must make the LSTM stateful. This gives us fine-grained control over when state of the LSTM layer is cleared, by calling the reset\_states()function.

The LSTM layer expects input to be in a matrix with the dimensions: [samples, time steps, features].

* **Samples**: These are independent observations from the domain, typically rows of data.
* **Time steps**: These are separate time steps of a given variable for a given observation.
* **Features**: These are separate measures observed at the time of observation.

We have some flexibility in how the Shampoo Sales dataset is framed for the network. We will keep it simple and frame the problem as each time step in the original sequence is one separate sample, with one timestep and one feature.

Given that the training dataset is defined as X inputs and y outputs, it must be reshaped into the Samples/TimeSteps/Features format, for example:

The shape of the input data must be specified in the LSTM layer using the “batch\_input\_shape” argument as a tuple that specifies the expected number of observations to read each batch, the number of time steps, and the number of features.

The batch size is often much smaller than the total number of samples. It, along with the number of epochs, defines how quickly the network learns the data (how often the weights are updated).

The final import parameter in defining the LSTM layer is the number of neurons, also called the number of memory units or blocks. This is a reasonably simple problem and a number between 1 and 5 should be sufficient.

The line below creates a single LSTM hidden layer that also specifies the expectations of the input layer via the “batch\_input\_shape” argument.

The network requires a single neuron in the output layer with a linear activation to predict the number of shampoo sales at the next time step.

Once the network is specified, it must be compiled into an efficient symbolic representation using a backend mathematical library, such as TensorFlow or Theano.

In compiling the network, we must specify a loss function and optimization algorithm. We will use “mean\_squared\_error” as the loss function as it closely matches RMSE that we will are interested in, and the efficient ADAM optimization algorithm.

Using the Sequential Keras API to define the network, the below snippet creates and compiles the network.

Once compiled, it can be fit to the training data. Because the network is stateful, we must control when the internal state is reset. Therefore, we must manually manage the training process one epoch at a time across the desired number of epochs.

By default, the samples within an epoch are shuffled prior to being exposed to the network. Again, this is undesirable for the LSTM because we want the network to build up state as it learns across the sequence of observations. We can disable the shuffling of samples by setting “shuffle” to “False“.

Also by default, the network reports a lot of debug information about the learning progress and skill of the model at the end of each epoch. We can disable this by setting the “verbose” argument to the level of “0“.

We can then reset the internal state at the end of the training epoch, ready for the next training iteration.

Below is a loop that manually fits the network to the training data.

Putting this all together, we can define a function called fit\_lstm() that trains and returns an LSTM model. As arguments, it takes the training dataset in a supervised learning format, a batch size, a number of epochs, and a number of neurons.

The batch\_size must be set to 1. This is because it must be a factor of the size of the training and test datasets.

The predict() function on the model is also constrained by the batch size; there it must be set to 1 because we are interested in making one-step forecasts on the test data.

We will not tune the network parameters in this tutorial; instead we will use the following configuration, found with a little trial and error:

* Batch Size: 1
* Epochs: 3000
* Neurons: 4

As an extension to this tutorial, you might like to explore different model parameters and see if you can improve performance.

* **Update**: Consider trying 1500 epochs and 1 neuron, the performance may be better!

Next, we will look at how we can use a fit LSTM model to make a one-step forecast.

## LSTM Forecast

Once the LSTM model is fit to the training data, it can be used to make forecasts.

Again, we have some flexibility. We can decide to fit the model once on all of the training data, then predict each new time step one at a time from the test data (we’ll call this the fixed approach), or we can re-fit the model or update the model each time step of the test data as new observations from the test data are made available (we’ll call this the dynamic approach).

In this tutorial, we will go with the fixed approach for its simplicity, although, we would expect the dynamic approach to result in better model skill.

To make a forecast, we can call the predict() function on the model. This requires a 3D NumPy array input as an argument. In this case, it will be an array of one value, the observation at the previous time step.

The predict() function returns an array of predictions, one for each input row provided. Because we are providing a single input, the output will be a 2D NumPy array with one value.

We can capture this behavior in a function named forecast() listed below. Given a fit model, a batch-size used when fitting the model (e.g. 1), and a row from the test data, the function will separate out the input data from the test row, reshape it, and return the prediction as a single floating point value.

During training, the internal state is reset after each epoch. While forecasting, we will not want to reset the internal state between forecasts. In fact, we would like the model to build up state as we forecast each time step in the test dataset.

This raises the question as to what would be a good initial state for the network prior to forecasting the test dataset.

In this tutorial, we will seed the state by making a prediction on all samples in the training dataset. In theory, the internal state should be set up ready to forecast the next time step.

We now have all of the pieces to fit an LSTM Network model for the Shampoo Sales dataset and evaluate its performance.

In the next section, we will put all of these pieces together.

## Complete LSTM Example

In this section, we will fit an LSTM to the Shampoo Sales dataset and evaluate the model.

This will involve drawing together all of the elements from the prior sections. There are a lot of them, so let’s review:

1. Load the dataset from CSV file.
2. Transform the dataset to make it suitable for the LSTM model, including:
   1. Transforming the data to a supervised learning problem.
   2. Transforming the data to be stationary.
   3. Transforming the data so that it has the scale -1 to 1.
3. Fitting a stateful LSTM network model to the training data.
4. Evaluating the static LSTM model on the test data.
5. Report the performance of the forecasts.

Some things to note about the example:

* The scaling and inverse scaling behaviors have been moved to the functions scale() and invert\_scale() for brevity.
* The test data is scaled using the fit of the scaler on the training data, as is required to ensure the min/max values of the test data do not influence the model.
* The order of data transforms was adjusted for convenience to first make the data stationary, then a supervised learning problem, then scaled.
* Differencing was performed on the entire dataset prior to splitting into train and test sets for convenience. We could just as easily collect observations during the walk-forward validation and difference them as we go. I decided against it for readability

Running the example prints the expected and predicted values for each of the 12 months in the test dataset.

The example also prints the RMSE of all forecasts. The model shows an RMSE of 71.721 monthly shampoo sales, which is better than the persistence model that achieved an RMSE of 136.761 shampoo sales.

Random numbers are used in seeding the LSTM, and as a result, you may have a different result from a single run of the model. We cover this further in the next section

## Develop a Robust Result

A difficulty with neural networks is that they give different results with different starting conditions.

One approach might be to fix the random number seed used by Keras to ensure the results are reproducible. Another approach would be to control for the random initial conditions using a different experimental setup.

For more on randomness in machine learning, see the post:

* [Embrace Randomness in Machine Learning](http://machinelearningmastery.com/randomness-in-machine-learning/)

We can repeat the experiment from the previous section multiple times, then take the average RMSE as an indication of how well the configuration would be expected to perform on unseen data on average.

This is often called multiple repeats or multiple restarts.

We can wrap the model fitting and walk-forward validation in a loop of fixed number of repeats. Each iteration the RMSE of the run can be recorded. We can then summarize the distribution of RMSE scores.

The data preparation would be the same as before.

We will use 30 repeats as that is sufficient to provide a good distribution of RMSE scores.

The complete example is listed below.

Running the example prints the RMSE score each repeat. The end of the run provides summary statistics of the collected RMSE scores.

We can see that the mean and standard deviation RMSE scores are 138.491905 and 46.313783 monthly shampoo sales respectively.

This is a very useful result as it suggests the result reported above was probably a statistical fluke. The experiment suggests that the model is probably about as good as the persistence model on average (136.761), if not slightly worse.

This indicates that, at the very least, further model tuning is required.