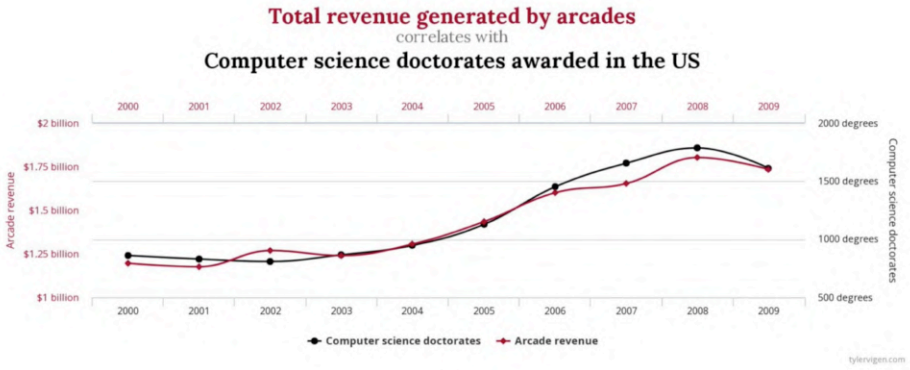
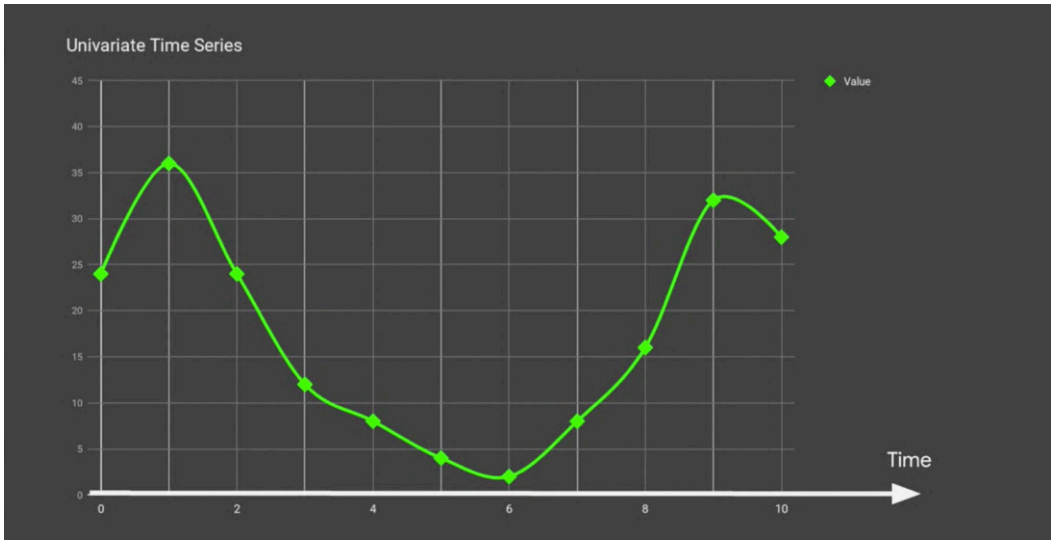
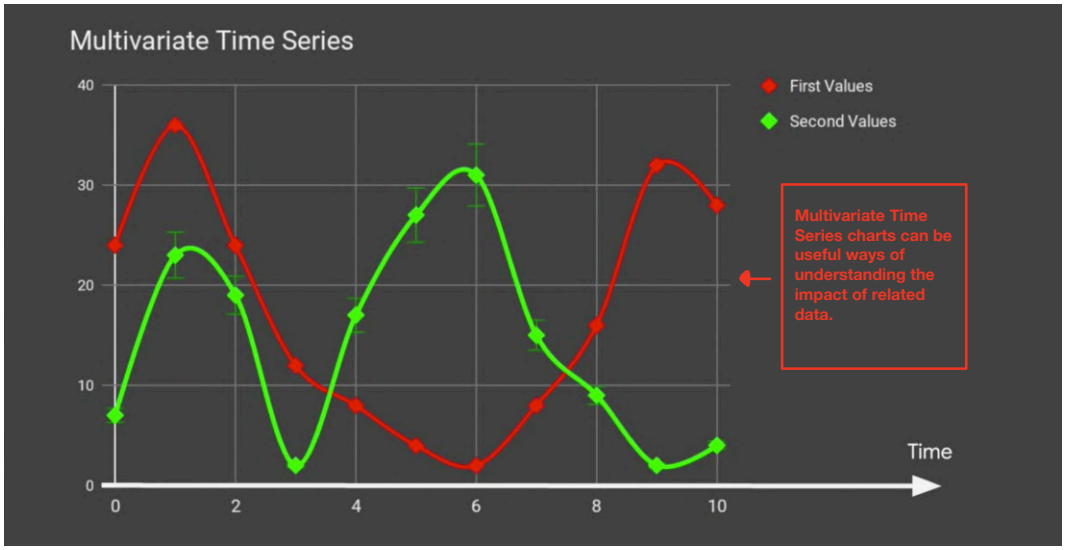
Chart From Google Search

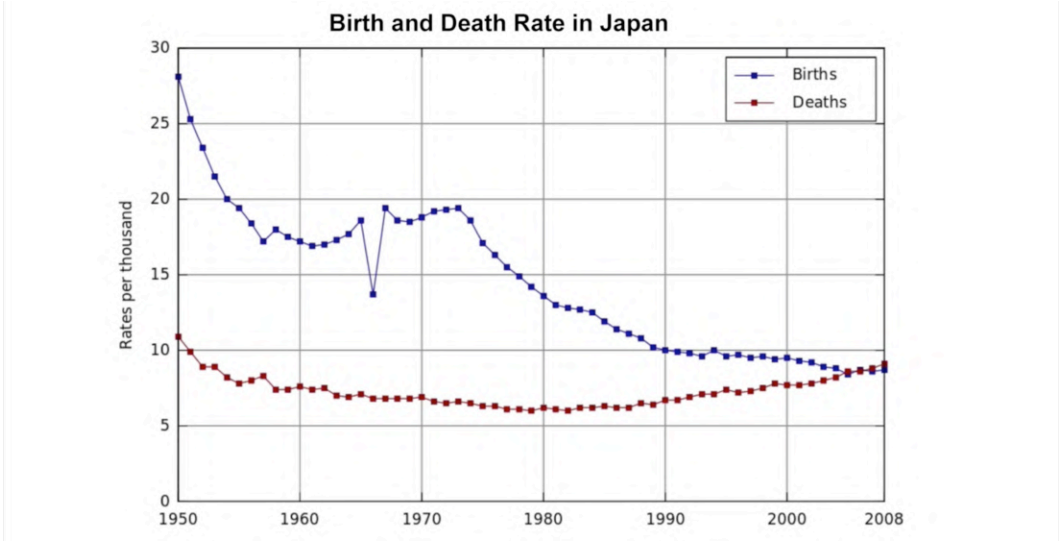
Chart Created by lmoroney@

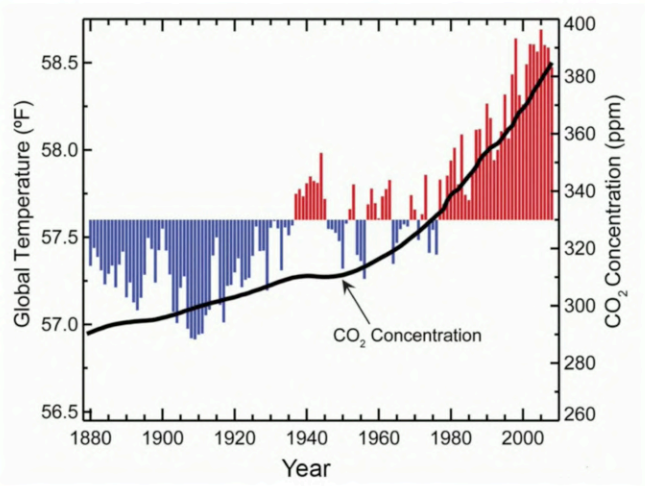


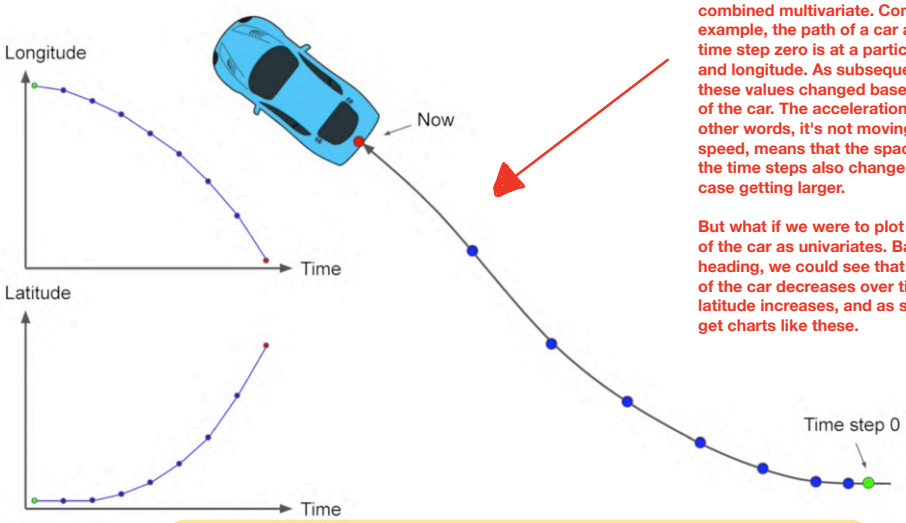
CC By 4.0 - TylerVigen.com


Univariate Time Series
Value
45
40
35
30
25
20
15
10
5
0
0
2
4
6
8
10
Time


Multivariate Time Series
First Values
Second Values
40
30
20
10
0
0
2
4
6
8
10
Time
Multivariate Time Series charts can be useful ways of understanding the impact of related data.


Birth and Death Rate in Japan
Births
Deaths
Rates per thousand
30
25
20
15
10
5
0
1950
1960
1970
1980
1990
2000
2008

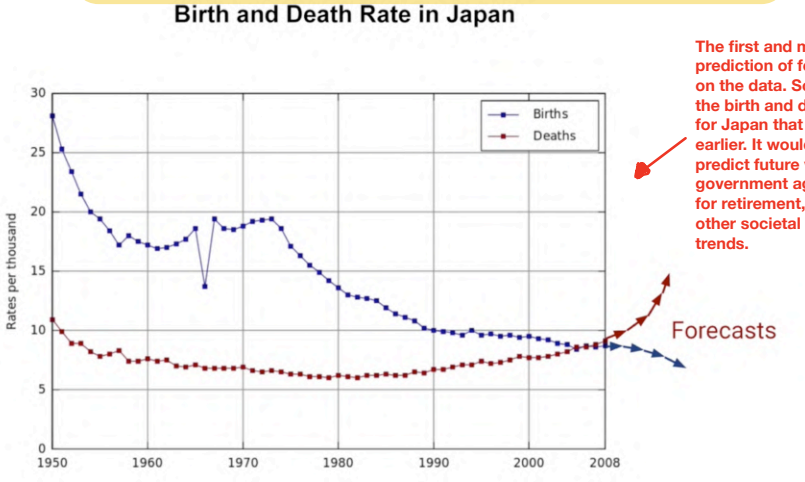




Movement of a body can also be plotted as a series of univariates or as a combined multivariate. Consider, for example, the path of a car as it travels. A time step zero is at a particular latitude and longitude. As subsequent time steps, these values changed based on the path of the car. The acceleration of the car, in other words, it's not moving at a constant speed, means that the spaces between the time steps also change in size, in this case getting larger.

But what if we were to plot the direction of the car as univariates. Based on its heading, we could see that the longitude of the car decreases over time, but its latitude increases, and as such you will get charts like these.

what types of things can we do with machine learning over time series?



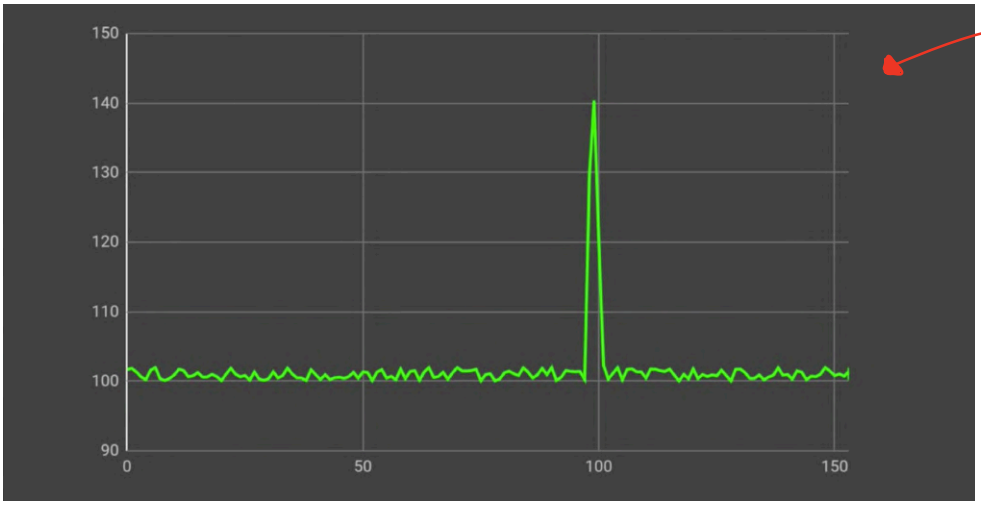
The first and most obvious is prediction of forecasting based on the data. So for example with the birth and death rate chart for Japan that we showed earlier. It would be very useful to predict future values so that government agencies can plan for retirement, immigration and other societal impacts of these trends.

Chart Created by lmoroney@

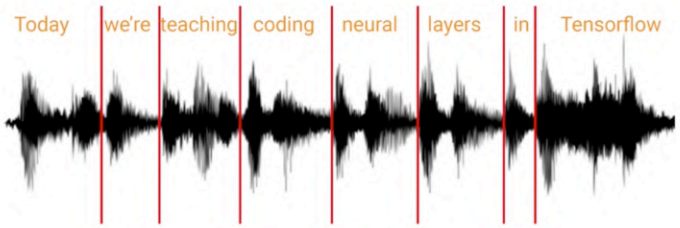
In some cases, you might also want to project back into the past to see how we got to where we are now.

This process is called imputation. Now maybe you want to get an idea for what the data would have been like had you been able to collect it before the data you already have.

Chart Created by lmoroney@



Additionally, time series prediction can be used to detect anomalies. For example, in website logs so that you could see potential denial of service attacks showing up as a spike on the time series like this.

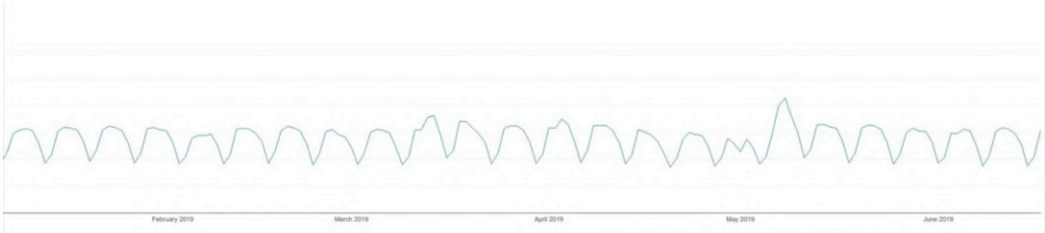


The other option is to analyze the time series to spot patterns in them that determine what generated the series itself. A classic example of this is to analyze sound waves to spot words in them which can be used as a neural network for speech recognition. Here for example, you can see how a sound wave is split into words. Using machine learning, it becomes possible to train a neural network based on the time series to recognize words or sub words.

Trend is where time series have a specific direction that they're moving in.

As you can see from the Moore's Law example we showed earlier, this is an upwards facing trend.

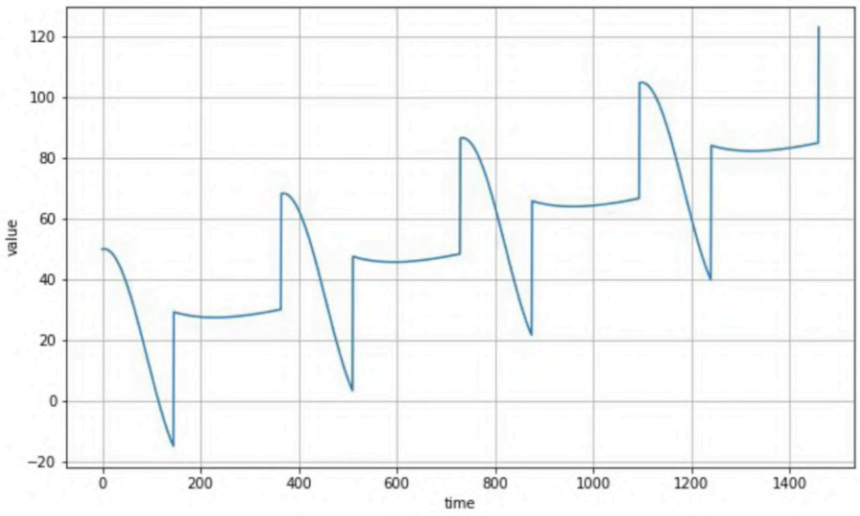
Chart Created by lmoroney@



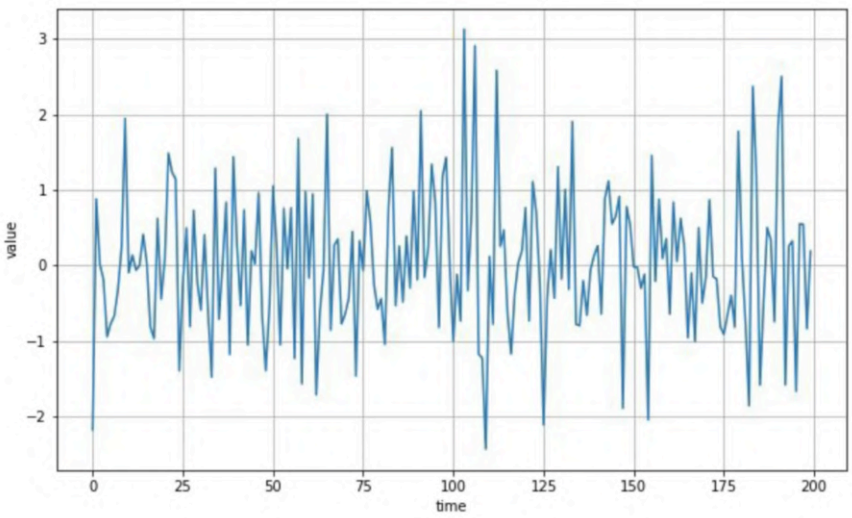
Seasonality is seen when patterns repeat at predictable intervals.

For example, take a look at this chart showing active users at a website for software developers. It follows a very distinct pattern of regular dips. Can you guess what they are? Well, what if I told you if it was up for five units and then down for two? Then you could tell that it very clearly dips on the weekends when less people are working and thus it shows seasonality.

Other seasonal series could be shopping sites that peak on weekends or sport sites that peak at various times throughout the year, like the draft or opening day, the All-Star day playoffs and maybe the championship game.



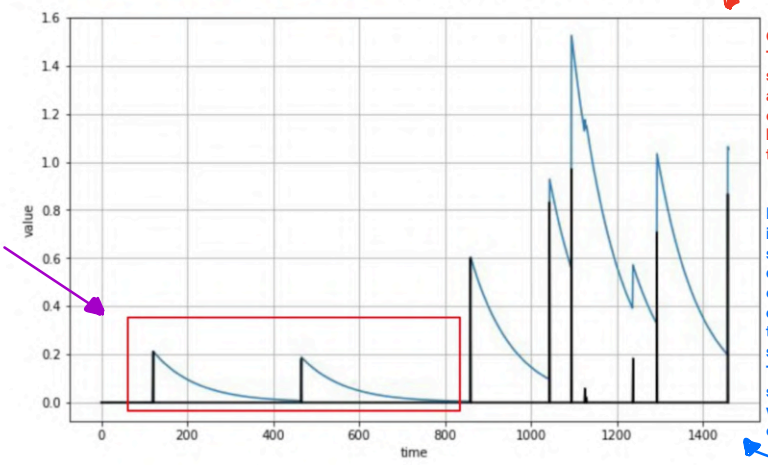
Some time series can have a combination of both trend and seasonality as this chart shows. There's an overall upwards trend but there are local peaks and troughs.



There are also some that are probably not predictable at all and just a complete set of random values producing what's typically called white noise. There's not a whole lot you can do with this type of data.

Autocorrelation

$v(t) = 0.99 \times v(t-1)$ + occasional spike

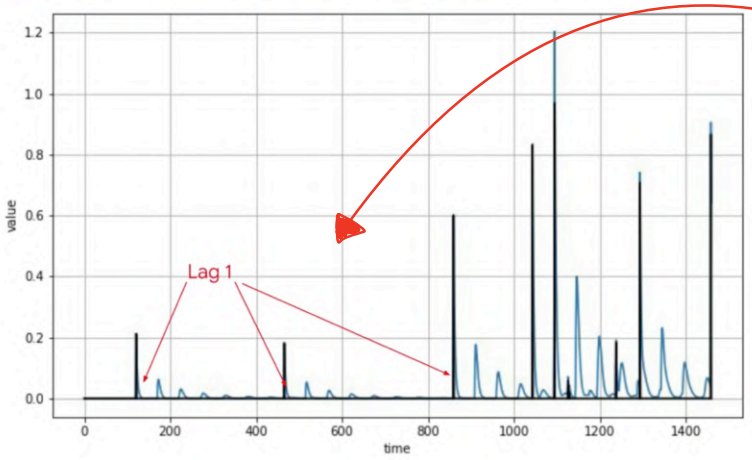


This example you can see at lag one there's a strong autocorrelation. Often a time series like this is described as having memory as steps are dependent on previous ones. The spikes which are unpredictable are often called Innovations. In other words, they cannot be predicted based on past values.

Consider this time series. There's no trend and there's no seasonality. The spikes appear at random timestamps. You can't predict when that will happen next or how strong they will be.

But clearly, the entire series isn't random. Between the spikes there's a very deterministic type of decay. We can see here that the value of each time step is 99 percent of the value of the previous time step plus an occasional spike. This is an auto correlated time series. Namely it correlates with a delayed copy of itself often called a lag.

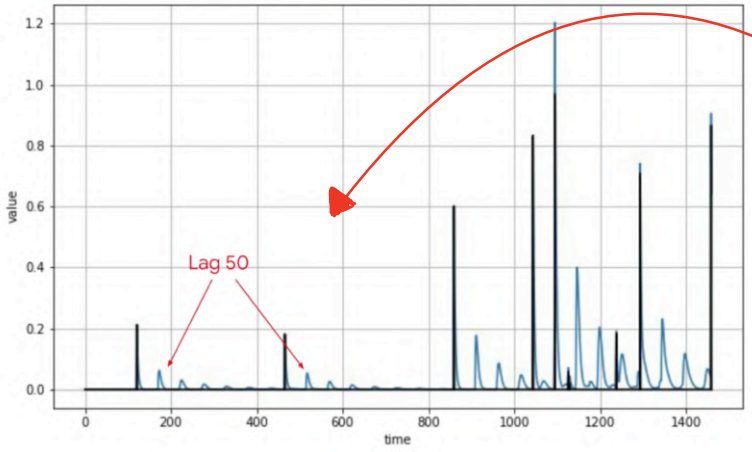
v(t) = 0.7 × v(t–1) + 0.2 × v(t–50) + occasional spike



Another example is here where there are multiple autocorrelations, in this case, at time steps one and 50.

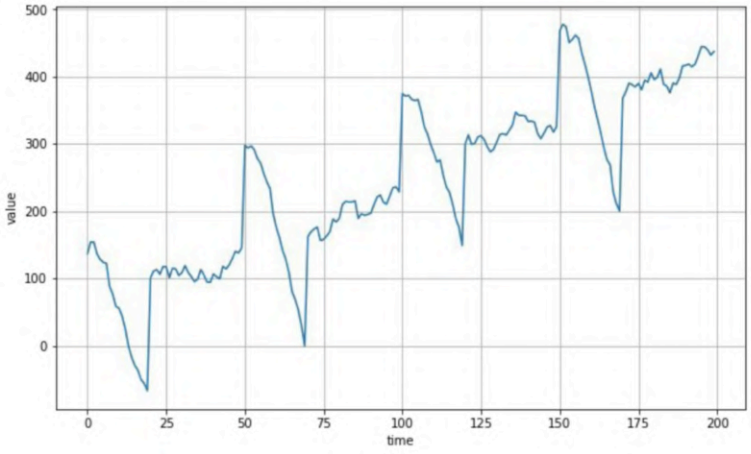
The lag one autocorrelation gives these very quick short-term exponential delays

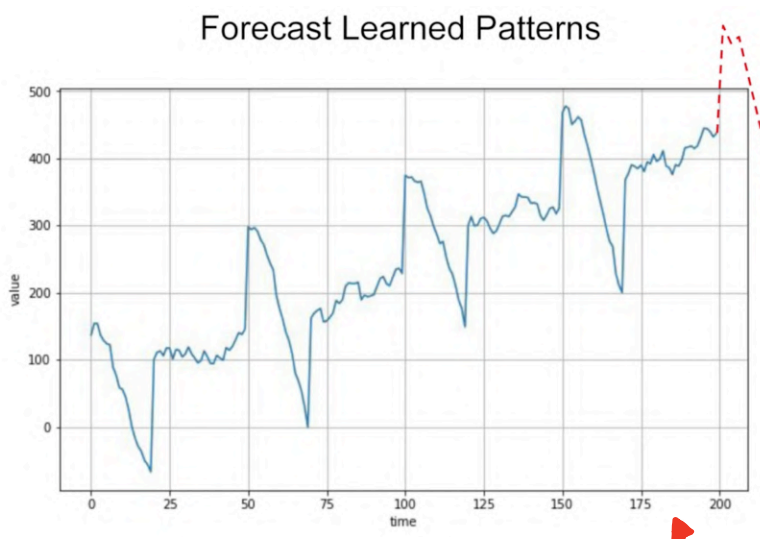
v(t) = 0.7 × v(t–1) + 0.2 × v(t–50) + occasional spike



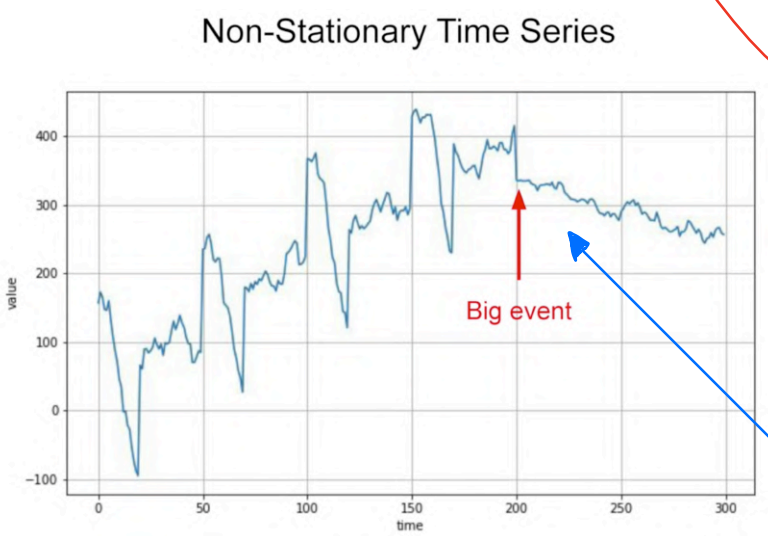
The lag 50 gives the small balance after each spike

Trend + Seasonality + Autocorrelation + Noise





Time series you'll encounter in real life probably have a bit of each of these features: trend, seasonality, autocorrelation, and noise. As we've learned a machine-learning model is designed to spot patterns, and when we spot patterns we can make predictions. For the most part this can also work with time series except for the noise which is unpredictable. But we should recognize that this assumes that patterns that existed in the past will of course continue on into the future.



Of course, real life time series are not always that simple. Their behavior can change drastically over time. For example, this time series had a positive trend and a clear seasonality up to time step 200. But then something happened to change its behavior completely. If this were stock, price then maybe it was a big financial crisis or a big scandal or perhaps a disruptive technological breakthrough causing a massive change. After that the time series started to trend downward without any clear seasonality. We'll typically call this a non-stationary time series.

To predict on this we could just train for limited period of time.

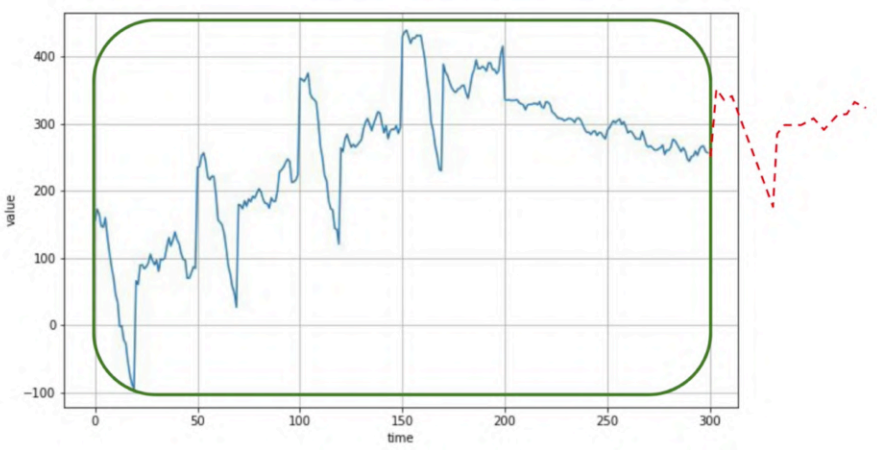
For example, here where I take just the last 100 steps. You'll probably get a better performance than if you had trained on the entire time series.

But that's breaking the mold for typical machine, learning where we always assume that more data is better. But for time series forecasting it really depends on the time series.

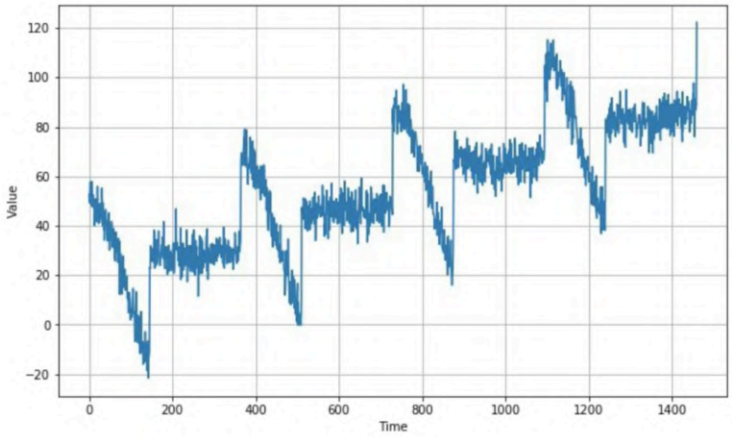
If it's stationary, meaning its behavior does not change over time, then great. The more data you have the better.

If it's not stationary then the optimal time window that you should use for training will vary

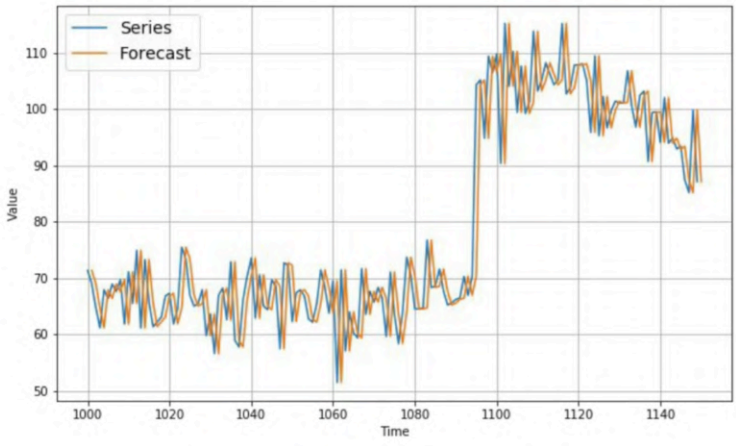
Non-Stationary Time Series



Trend + Seasonality + Noise



Naive Forecasting

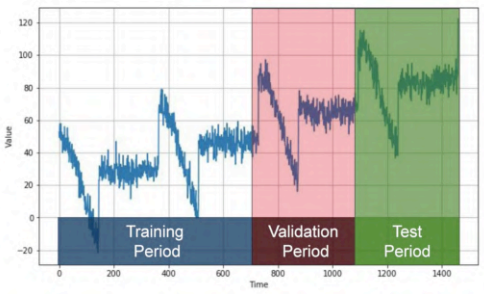


We could, for example, take the last value and assume that the next value will be the same one, and this is called naive forecasting.

We can do that to get a baseline at the very least, and believe it or not, that baseline can be pretty good.

But how do you measure performance?

Fixed Partitioning



To measure the performance of our forecasting model,. We typically want to split the time series into a training period, a validation period and a test period. This is called fixed partitioning.

If the time series has some seasonality, you generally want to ensure that each period contains a whole number of seasons. For example, one year, or two years, or three years, if the time series has a yearly seasonality. You generally don't want one year and a half, or else some months will be represented more than others.

While this might appear a little different from the training validation test, that you might be familiar with from non-time series data sets. Where you just picked random values out of the corpus to make all three, you should see that the impact is effectively the same.

Next you'll train your model on the training period, and you'll evaluate it on the validation period. Here's where you can experiment to find the right architecture for training. And work on it and your hyper parameters, until you get the desired performance, measured using the validation set.

Often, once you've done that, you can retrain using both the training and validation data. And then test on the test period to see if your model will perform just as well. And if it does, then you could take the unusual step of retraining again, using also the test data.

But why would you do that? Well, it's because the test data is the closest data you have to the current point in time. And as such it's often the strongest signal in determining future values. If your model is not trained using that data, too, then it may not be optimal.

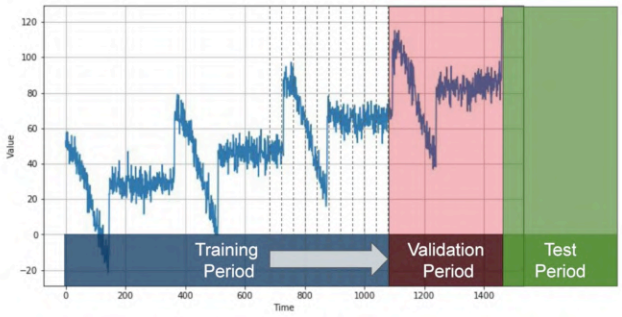
Due to this, it's actually quite common to forgo a test set all together. And just train, using a training period and a validation period, and the test set is in the future. Fixed partitioning like this is very simple and very intuitive, but there's also another way.

We start with a short training period, and we gradually increase it, say by one day at a time, or by one week at a time. At each iteration, we train the model on a training period. And we use it to forecast the following day, or the following week, in the validation period. And this is called roll-forward partitioning.

You could see it as doing fixed partitioning a number of times, and then continually refining the model as such.

For the purposes of learning time series prediction in this course, will learn the overall code for doing series prediction. Which you could then apply yourself to a roll-forward scenario, but our focus is going to be on fixed partitioning.

Roll-Forward Partitioning



Metrics

```
errors = forecasts - actual

mse = np.square(errors).mean()

rmse = np.sqrt(mse)

mae = np.abs(errors).mean()

mape = np.abs(errors / x_valid).mean()
```

Metrics

```
errors = forecasts - actual

mse = np.square(errors).mean()

rmse = np.sqrt(mse)

mae = np.abs(errors).mean()

mape = np.abs(errors / x_valid).mean()
```

The most common metric to evaluate the forecasting performance of a model is the mean squared error or mse where we square the errors and then calculate their mean. Well, why would we square it?

Well, the reason for this is to get rid of negative values. So, for example, if our error was two above the value, then it will be two, but if it were two below the value, then it will be minus two. These errors could then effectively cancel each other out, which will be wrong because we have two errors and not none. But if we square the error of value before analyzing, then both of these errors would square to four, not canceling each other out and effectively being equal.

Metrics

```
errors = forecasts - actual

mse = np.square(errors).mean()

rmse = np.sqrt(mse)

mae = np.abs(errors).mean()

mape = np.abs(errors / x_valid).mean()
```

If we want the mean of our errors' calculation to be of the same scale as the original errors, then we just get its square root, giving us a root means squared error or rmse.

Metrics

```
errors = forecasts - actual

mse = np.square(errors).mean()

rmse = np.sqrt(mse)

mae = np.abs(errors).mean()

mape = np.abs(errors / x_valid).mean()
```

Another common metric and one of my favorites is the mean absolute error or mae, and it's also called the main absolute deviation or mad. And in this case, instead of squaring to get rid of negatives, it just uses their absolute value. This does not penalize large errors as much as the mse does. Depending on your task, you may prefer the mae or the mse. For example, if large errors are potentially dangerous and they cost you much more than smaller errors, then you may prefer the mse. But if your gain or your loss is just proportional to the size of the error, then the mae may be better.

Metrics

```
errors = forecasts - actual

mse = np.square(errors).mean()

rmse = np.sqrt(mse)

mae = np.abs(errors).mean()

mape = np.abs(errors / x_valid).mean()
```

Also, you can measure the mean absolute percentage error or mape, this is the mean ratio between the absolute error and the absolute value, this gives an idea of the size of the errors compared to the values.

Naive Forecast MAE

```
keras.metrics.mean_absolute_error(x_valid, naive_forecast).numpy()
```

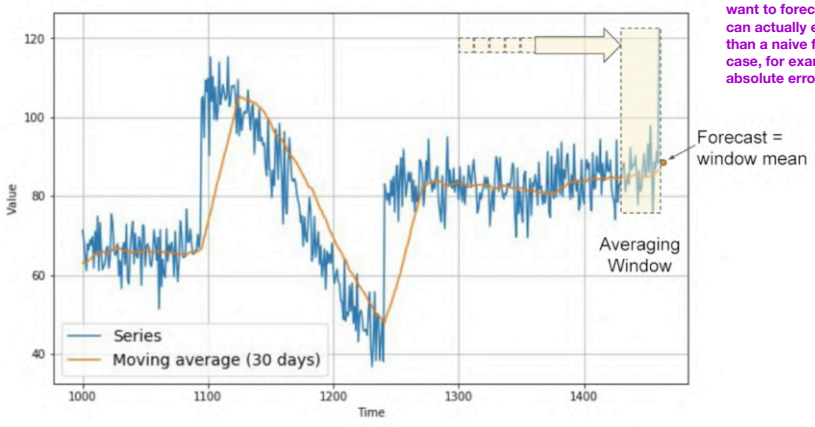
```
5.937908515321673
```

A common and very simple forecasting method is to calculate a moving average.

The idea here is that the yellow line is a plot of the average of the blue values over a fixed period called an averaging window, for example, 30 days. Now this nicely eliminates a lot of the noise and it gives us a curve roughly emulating the original series, but it does not anticipate trend or seasonality.

Depending on the current time i.e. the period after which you want to forecast for the future, it can actually end up being worse than a naive forecast. In this case, for example, I got a mean absolute error of about 7.14.

Moving Average

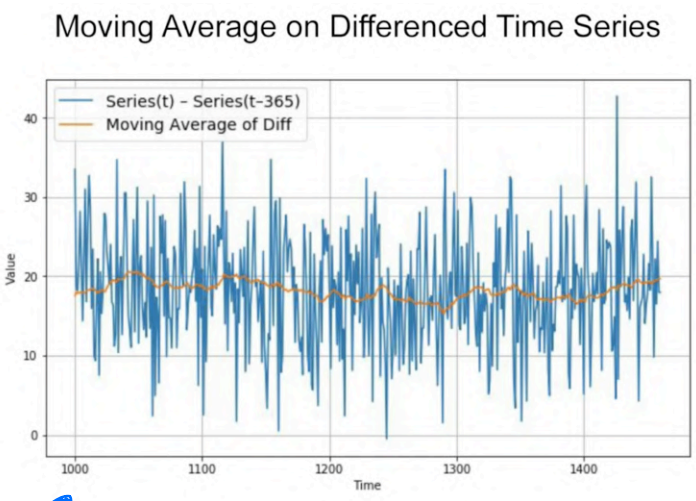


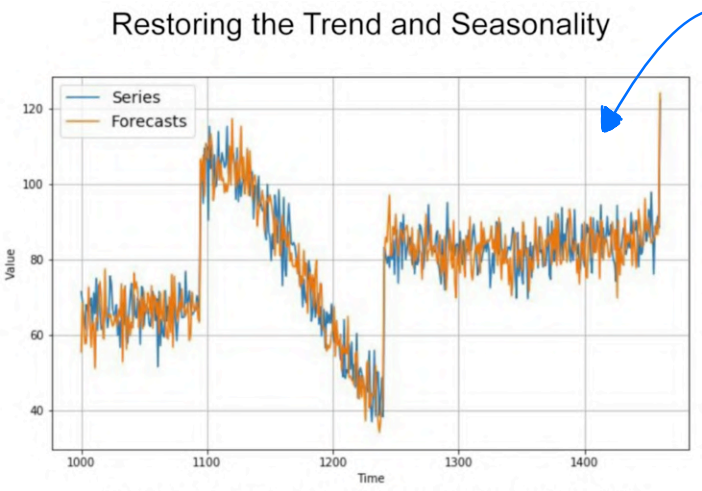
One method to avoid this is to remove the trend and seasonality from the time series with a technique called differencing.

So instead of studying the time series itself, we study the difference between the value at time T and the value at an earlier period.

Depending on the time of your data, that period might be a year, a day, a month or whatever. Let's look at a year earlier. So for this data, at time T minus 365, we'll get this difference time series which has no trend and no seasonality.

We can then use a moving average to forecast this time series which gives us these forecasts. But these are just forecasts for the difference time series, not the original time series. To get the final forecasts for the original time series, we just need to add back the value at time T minus 365





Forecasts = moving average of differenced series + series(t – 365)

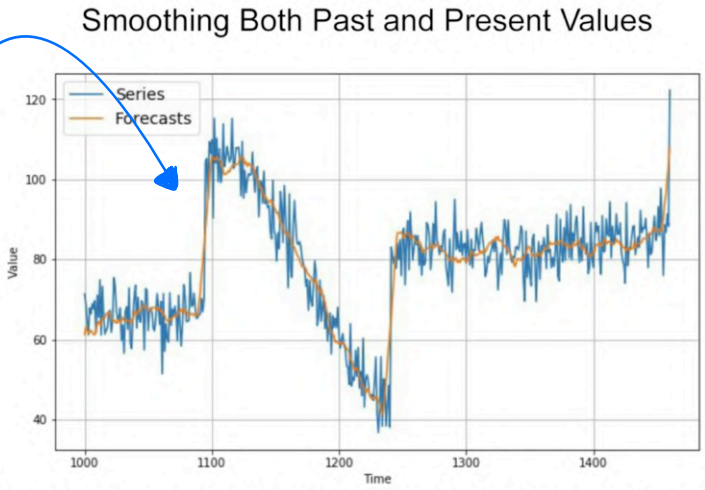
They look much better, don't they? If we measure the mean absolute error on the validation period, we get about 5.8. So it's slightly better than naive forecasting but not tremendously better.

You may have noticed that our moving average removed a lot of noise but our final forecasts are still pretty noisy. Where does that noise come from?

Well, that's coming from the past values that we added back into our forecasts. So we can improve these forecasts by also removing the past noise using a moving average on that.

If we do that, we get much smoother forecasts. In fact, this gives us a mean squared error over the validation period of just about 4.5. Now that's much better than all of the previous methods.

In fact, since the series is generated, we can compute that a perfect model will give a mean absolute error of about four due to the noise. Apparently, with this approach, we're not too far from the optimal.



Forecasts = trailing moving average of differenced series + centered moving average of past series (t – 365)

Note that when we use the trailing window when computing the moving average of present values from t minus 32, t minus one. But when we use a centered window to compute the moving average of past values from one year ago, that's t minus one year minus five days, to t minus one year plus five days.

Then moving averages using centered windows can be more accurate than using trailing windows. But we can't use centered windows to smooth present values since we don't know future values. However, to smooth past values we can afford to use centered windows.