**Time Series Forecasting**

**Difference Between Forecasting and Regression**

Time series are datasets indexed by time, usually sequentially. We will often want to find some temporal pattern in the data. They are basically Stochastic Difference Equations, or Stochastic Differential Equations. So time series forecasting is a little different than regression. Say we have a sequence of data points xn = x0, x1, x2, …, xn. A regression curve through the data points would attempt to model the series as:



where fn is the regression curve formula, and ΔWnis white noise fluctuations. So fn is essentially the average of all realizations of xn, as seen from:



Such an equation is useful, but time series forecasting looks to extract a little more information from the series. It attempts to calculate, not E(xn) = <xn>, but E(xn|xj<n), which is to say, the expectation of xn, given knowledge of all the previous xj in the series. If successive points fluctuations about their mean values are uncorrelated, then this will just reduce to <xn>. For instance, consider our regression curve’s prediction for xn and xn-1.



E(xn|xn-1) = E(xn) since the two variables are not correlated, since:



In other words, xn-1 being above or below the regression line (its mean value fn-1) has no bearing on whether xn will be above or below the regression line (its mean value fn). So if the variables are uncorrelated, meaning their fluctuations about the regression line are uncorrelated, then knowing the regression curve fn = <xn> (for all n) is knowing everything. Not so in other cases. Consider random walk.



The solution to this equation is:



And the average is:



So fn = x0. But the points in this series are correlated.



So if we know xj<n, we can make a better prediction than just E(xn) = x0. In fact, returning to the original difference equation, we see that given knowledge of xn-1, we specify completely the probability distribution of xn (assuming for the sake of discussion that ΔWn-1 is a normally distributed random variable with mean 0 and variance DΔt). So we’d have:



which is to say xn is normally distributed with a mean of xn-1 and variance DΔt. And so specifically, taking just the average, we have:



So our prediction would not be that xn = x0, which is what it averages out to in general, over many different instantiations of this series, but that it will be on average what it was previously, in this particular series. Or consider an AR(1):



The solution, average, and correlation are:



We can see the average value of <xn>, and if we had no information about xn besides this, then that would be our best estimate for it. But we also see that the covariance is non-zero; so the fluctuations of the points about their average values are correlated. If φ > 1, then if xm is above average, xn will likely be too. And if φ < -1, then points will be alternately correlated. By looking at the difference equation itself, we see:



And more relevantly,



**Training/Testing a Regression Model**

Despite the emphasis up above on fitting a stochastic model, sometimes a regression model is truly the best fit. For instance, if you’re trying to predict the position of a particle ostensibly moving with constant acceleration, then you should fit the parabola: x = x0 + v0t + (1/2)at2 through the data points, because x(t) is really only correlated with t, not with, x(t-1), say. One often heuristically separates a time series into three parts: trend, seasonality, and noise.



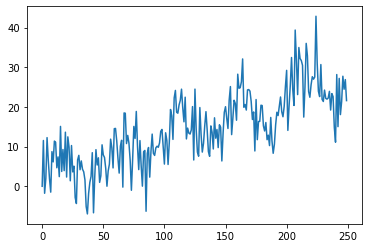
The first part is the so-called ‘trend’. The trend might be T(n) = an2 + bn + c, or φn + β, etc. The seasonality could be S(n) = Asin(ωn + φ0) or whatever. The noise is generally assumed to be white, but could be more complicated. To get a clearer picture of the trend and/or seasonality, i.e., to eliminate the white noise, we can do a rolling average. This could be defined as:



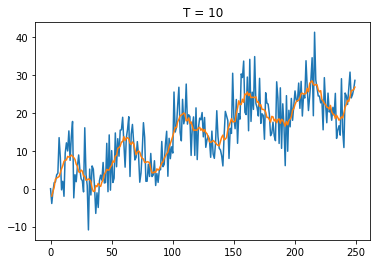
The window, T, would be a user defined parameter. Can’t exactly use this for forecasting, but it might give an idea as to what shape we should be looking for. For instance, consider this series:



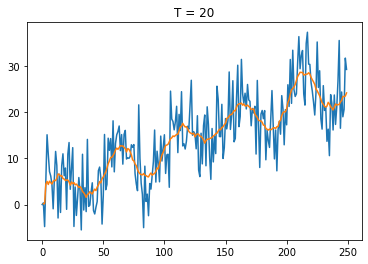
where ΔWn is white noise, which looks like this,



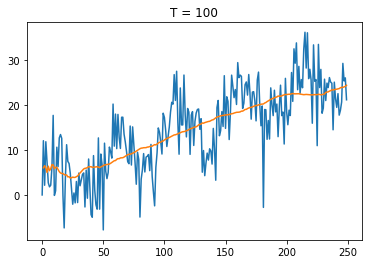
If we do a rolling average, we get this:



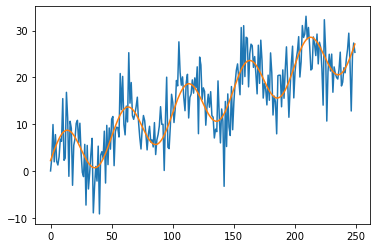
This makes the sin wave a little more apparent. A larger window size obviously smooths it out more. But we’ll note that the rolling average will miss the peaks and troughs by more, the larger the window (T) is.



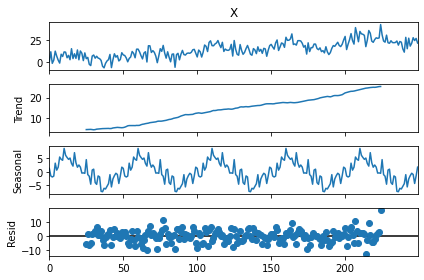
A *very* large window makes the linear trend more apparent.



Once the general shape is known, we could attempt to find a best fit function of that form; we might guess f(n) = an + b + csin(dn) or so. We can use *lmfit* for this. You do have to give it parameters that are fairly close to the actual ones. And you can easily turn it into a forecaster, since it gives you the optimized parameters. The fit looks like this:

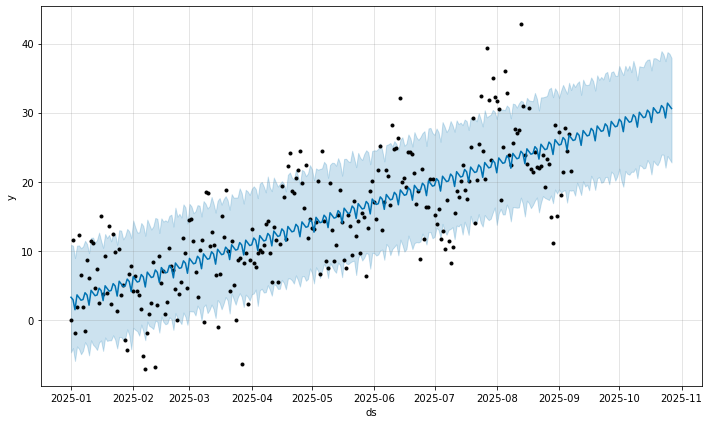


which is pretty good! Statsmodels has a seasonal\_decomposition function to formally separate out the trend, seasonality, and noise. If we plug our sequence into it, we get:



The top box is trend + seasonality (noise is taken out). This is a decent result. Alas, statsmodels doesn’t do forecasting (predicting future points) from the seasonal decomposition.

Prophet is kind of like Statsmodels’ seasonal\_decomposition, but does have the ability to forecast. When I plug this data into Prophet, just to get a fit, I get:



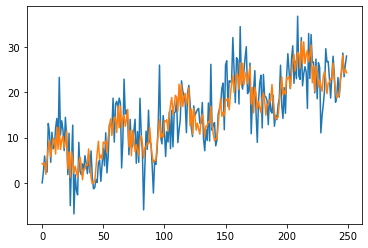
Hmmm. Another option is Statsmodels’ Exponential Smoothing. The simplest single parameter exponential smoothing formula looks like this:



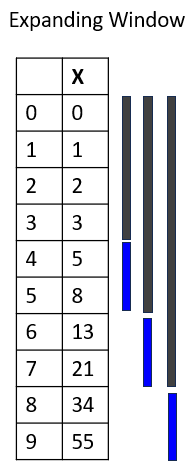
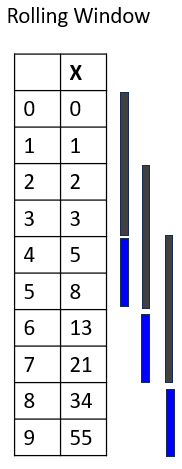
sn is the smoothed series, xn is the actual series, and α is the smoothing parameter between 0 and 1. We solve this series in one of the solutions files. The result is, translated to present terminology,



If α = 0, then looks like sn = x0 forever. If α = 1, then sn = xn, so there’d be no smoothing. Looks like the smaller α is, the more smoothing we have. There is also double exponential smoothing and triple exponential smoothing. These work better with trends and seasonality respectively and can be implemented using the Statsmodels module in Python. Below, triple exponential smoothing (orange) is applied to the data. Still looks kind of noisy. The Exponential Smoothing function does support forecasting too.



Maybe parameters can be optimized a little. So how do we train and test a regression model? I think we would use an expanding window approach, most likely. This is pictured below left. The grey rectangles encompass the training/fitting data, and the blue rectangles encompass the testing data.

So say we wanted to predict two steps ahead, in a non-overlapping way, from x4 to x9. Then we’d fit the model to x[0:4], and forecast x[4:6]. Then fit to x[0:6] and forecast x[6:8]. Finally fit to x[0:8], and forecast x[8:10]. I’d presume we’d fit by minimizing the mean square error (mse). And we’d find the mse of all the forecasted points too. The best model would be the one that has the smallest overall testing mse.

On the right is pictured a rolling window approach, where we keep the fitting size the same. I think this would be less appropriate for a regression model, but might be justified on the grounds that we aren’t sure if our regression model is universally applicable. For instance, we might be approximating a complicated curve locally with a polynomial, as kind of a Taylor series approximation that can’t be trusted too far.

**Training/Testing a Stochastic Model**

Now let’s look at stochastic models. The sequence explored above has no correlations, and so there wouldn’t really be a point in exploring it using a stochastic model, FWIW. The first thing to do, I suppose, when trying to apply a stochastic model is to check whether its stationary or not. If it’s stationary, then its mean, variance, and all higher moments, as well as correlation functions, would be time-independent. We could look at a rolling average, or variance, etc., to get an idea of whether the sequence is stationary. More rigorously, we can use the AD Fuller or KPSS tests in Statsmodels to check for such time-independence. If it is time-independent, then next thing to do is to look for correlations between terms.

***Correlation Functions***

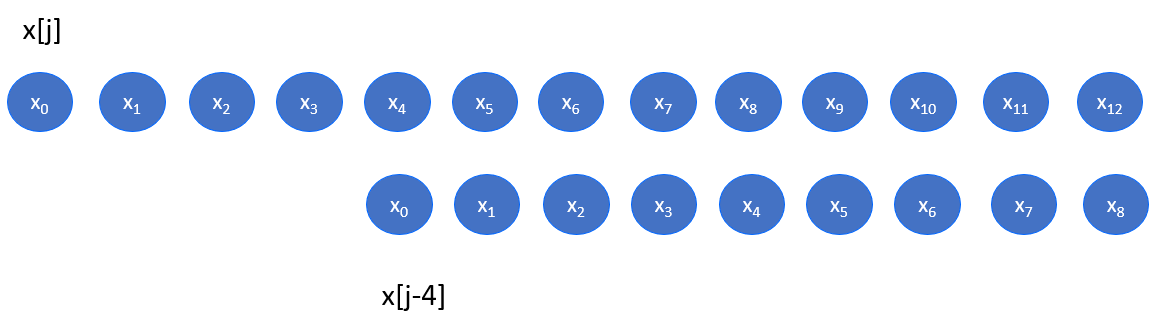
A useful construct to this end is the ACF. The autocorrelation function with lag k is,



This is just the typical correlation function between any two random variables. *If* the series is stationary, an unbiased estimate of ACF(k) is:

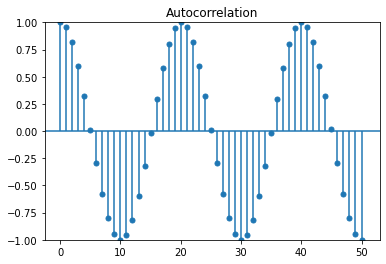


[I think Statsmodels default divides by N in the numerator instead of N-k, for some reason, but can prevent this by setting **adjusted = True**] For instance, consider a series x[j], and match it up with its time lag x[j-k] (k = 4 in this case)



The autocorrelation function would quantify how much terms on top of each other are correlated. I left off the j subscript in AFC(k) because this formula only applies if the series is stationary, and in that case, ACFj(k) will also not depend on j. If the series is *not* stationary, then this expression carries no meaning basically, as then we cannot replace noise averages with time averages. ACF(k) has a max of 1 and min of -1. Generally, two terms with lag k in a series will have a high positive correlation (near 1) if they vary in the same direction – if one is above the mean, the other is above the mean, and if one is below, the other is also below. If they have a high negative correlation (near -1), then they nearly perfectly vary in the opposite direction – when one is above the mean, the other is below the mean. If the two terms sometimes vary jointly and sometimes anti-jointly, then it’s possible we’ll get an autocorrelation of 0. Or if the terms are completely random, we’ll get an autocorrelation of 0, in which case we might infer that we can just do with a regression analysis instead.

We can use Statsmodels to produce ACF plots. Below, we have the ACF for xn = 5sin(2πn/20). Note we set **adjusted = True** so that it divides by 1/(N-k), and not N. Obviously when k = 0, we get a correlation of 1, as xn and xn-0 are perfectly correlated. When k = 20 we get a correlation of 1, since xn and xn-20 are a wavelength apart and so go up and down in synchrony. When k = 10 we get -1 since xn and xn-10, being ½ a wavelength apart move in completely opposite directions. When k = 5, say, we get 0, since xn and xn-5, being a quarter wavelength apart, half the time move in the same direction from the mean (0) and half the time in opposite directions.



There is also a partial autocorrelation function. I’m not sure how its defined.



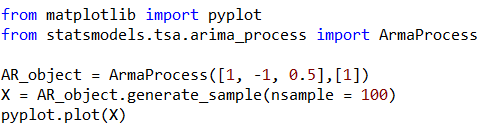
But, it’s supposed to remove the effects of intermediate correlations. Consider for instance:



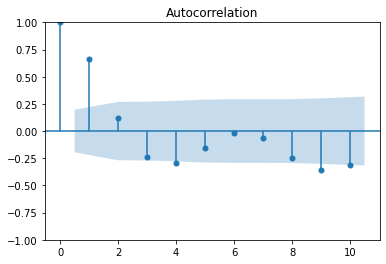
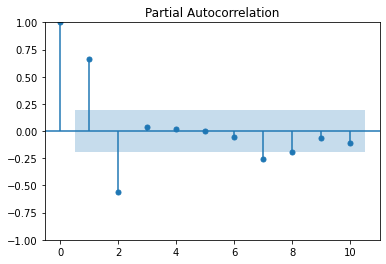
Then we also have:



So for instance, X[j] and X[j-1] are correlated with each other. And so are X[j] and X[j-2]. But as we can see, do to these correlations, X[j] and all the others preceding it are also correlated. So the ACF would show all of these correlations. If the series is stationary, then these other correlations should get smaller I think, in a power law fashion. And so it is possible to separate out the lags which are directly correlated, i.e. k = 1 and k = 2, from the lags which are indirectly correlated, i.e., k > 2. The partial autocorrelation function (PACF) does this. For instance, I plotted an AR(2) series in statsmodels. φ1,2 were 1, -0.5 respectively (have to input the values with extra negative sign in statsmodels).



Then I plotted the ACF(k) and PACF(k) guys. It is easier to tell from the PACF that there are only direct correlations up to k = 2.

***ARMA Process***

If we have an estimate for the maximum lag that’s correlated, then we can fit an ARMA model to the data. An ARMA model looks like this:



where,



Explicitly,



It’s a linear model, and so can be solved exactly, in principle. The lags p, and q, can be estimated as the max lag in the PACF. But we’d probably do grid search on p and q to find the values which produce the best fit. If the ACF shows a seasonality to the data, then we might try a SARIMA model we’ll talk about later.



So how is a best fit model determined? That is, how do we determine the best fit parameters φ1, φ2, βΔt, D (the variance of ΔWn), θ1, and θ2. Well, given x0 and x1, we can write the joint probability distribution of variables x2, x3, …, xn as (recall the basic formula P(a, b) = P(a)P(b|a)):



Now consider the first three terms, with ΔW1, ΔW0 set to 0, as is I think customary.



We’ll observe that with x0, x1 fixed, x2 is just a function of ΔW2. And with x0, x1, x2 fixed, x3 is just a function of ΔW3. And with x0, x1, x2, x3 fixed, x4 is just a function of ΔW4, etc. So x2, x3, and x4 will be normally distributed variables with mean μn = φ1xn-1 + φ2xn-2 + βΔt + θ1ΔWn-1 + θ2ΔWn-2, and variance σn2 = DΔt. Note ΔWn = xn - <xn>. More explicitly,



Will note that μn depends on all xj<n through its implicit dependence on all the prior means. So we can write:



where DΔt is the variance of ΔWn. Further, since this is a product of normal distributions, it will be itself a normal distribution. And we can write:



where μn is the column matrix of averages, and σ2mn is the covariance matrix. These can be explicitly calculated – see the solved examples file for a lot of examples. Now let’s say we have data points (first two are fixed by initial conditions, say) x0,\* x1\*, x2\*, x3\*, …, xn\*. We’ll typically then define the log-likelihood as the ln of the likelihood that the model predicts these datapoints.



And so to determine the best fit, we’d differentiate w/r to all the parameters: φ1, φ2, D, β, θ1, θ2, and find the values which maximize the probability, and accordingly maximize ln(L). Different measures of loss associated with this are the Aikaike Information Criterion:



(also seen this formula stated with no (k+1)/(n-k) term) and the Bayesian Information Criterion,



where k is the number of free parameters in the model, and n is the number of datapoints. So we can frame determining the fit as equivalently minimizing the loss. All of these formulas can be adapted to any other ARMA equation, or ARIMA, SARIMA, etc. In general finding a best fit would be a laborious process, given how complicated the equation is. I think finding the proper parameters numerically depends on ARMA being stationary; otherwise we run into numerical matrix invertibility problems?

So once the parameters are found, how does Statsmodels create a fit curve with the data points x0\*, x1\*, x2\*, x3\*, …, xn\*. Apparently, it plots the expected value of the conditional probability distribution for that point, i.e., E(xn|xj<n), given the points preceding it (and recall that since preceding points are known, preceding noises ΔWn-1\* are known, assuming the first couple noises that start off the series are zero). So the first couple forecasts would be:



So that’s how the fit works. It’s a rolling forecast, basically using the best fit difference equation. But how does the forecasting beyond this work? Say our data stops at xn\*. Then I think we kind of boot strap our prediction; we use the forecasted data points to get the next ones. So the next forecasted points would be:



Note that ΔWn is the difference between xn\* and n. Since we have actual data points for xj≤n, we will have actual values for ΔWj≤n = xj\* - <xj>. Since we will be using forecasted points as data points for j > n, we’ll have Δj = xj\* - j = j – j = 0 for j > n. Gradually, our knowledge of the noise will attenuate. As can see, more than two predictions out, all of our noises will be zero. All in all, given the prior x0\*, x1\*, …, xn\*, we have a deterministic formula to forecast as far out as we want to go.

It seems, however, that instead of doing *this* stochastic algorithm, we should just calculate n+3, say, by calculating the marginal distribution of P(xn+3) and finding the average from *that*:



Don’t forget that ΔWj, in this perspective, is known for j ≤ n, but not after. Maybe this prediction works out to the same thing? It doesn’t seem to. For instance, let’s take the simpler case of an AR(1): xn+1 = φxn + ΔWn. And let’s use the rule (see Prob/Stat Event Composition notes).



Then we construct the probability distribution of the forecast xn+2, given only knowledge of up till xn. We’ll do this by integrating over the joint distribution of P(xn+2, xn+1|xn) w/r to xn+1.



Now it would seem to only follow that P(xn+2|xn) = P(xn+2|n+1), and hence that n+2 = φn+1 only if P(xn+1|xn) = δ(xn+1 – n+1), i.e., it’s a super narrow Gaussian. So while it might be a good approximation to use future predictions to predict even futurer predictions, it does *seem* like just an approximation. And one that all the models use.

***ARIMA Process***

So if sequence isn’t stationary, but seems to a power law trend, then we can try to model it with an ARIMA model. The power law trend can be in any of the moments – either mean, variance, or whatever. The ARIMA model looks like this:



where,



So xn would basically be the integral of an ARMA process. I’d imagine d would be either d = 1, 2, or nothing, in practice. In principle, d would be somewhat apparent from a rolling average kind of plot. Even better would be to apply the difference operator Dd to the series, and then recheck for stationarity. The smallest d that does this is probably the one we’d want, ‘cause if you apply D to a sequence that is already stationary, you just get another stationary sequence. Once we have a stationary Ddxn, we could look at the PACF to estimate p, q. Then we’d try to fit our ARIMA model to the data. For instance, consider the random walk for example:



The solution is:



So this series is not stationary because of the linear growth in the average and the noise. Consequently, it is not able to be modeled by ARMA because it isn’t stationary. But if we take the difference we get:



So this is just white noise plus a constant, and *is* stationary. So we could model our series with an ARIMA with d = 1. If our series were:



say, then this would also not be able to be modeled by ARMA. But two differences would give us:



So we can model it with an ARIMA with d = 2. In practice we’ll probably want to use grid search to find the best parameters: d, p, q. Fitting and Forecasting would be done just like with ARMA.

***SARIMA Process***

If the sequence seems to have a sinusoidal behavior and/or power law behavior, we can try to model it with a SARIMA model. It looks like this:



where,



Helps to know the general behavior of the sequence as it relates to the p,d,q,s,P,D,Q parameters. I would say that Dd captures a sequence whose mean or variance is going as nk≤d. DTD captures a sequence whose mean or variance is going as [n/T]k≤Dcos(2πn/(T/j)), where j = 1, 2, …., T, and [ ] is the ‘floor’ function. If it’s a sum or product of these two forms, then do both. Now consider the Φp(B) term. If p = 1, then it adds terms that are exponentially diminishing or increasing as eφn. If p > 1, then we have exponential damping/increasing of sinusoidal terms like e[Re(φ)+iIm(φ)]∙n or something. The Φ´P(BT) behaves similarly. If P = 1, then it adds terms that are exponentially diminishing/increasing times the T-periodic functions as eφn/Tcos(2πn/(T/j)). If P > 1, then we have exponential damping/increasing of sinusoidal terms multiplied by the T-periodic functions, like eRe(φ)∙n/T+iIm(φ)∙n/Tcos(2πn/(T/j)).

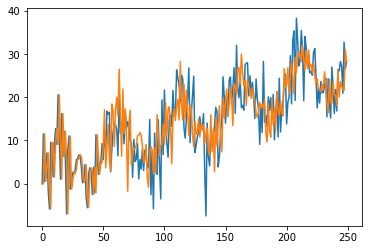
So in practice, how might we determine d and s, D? d could be done roughly by visual inspection, and would be the power that removes any power law trend in the moments. s is just the seasonality, i.e., the period. Not sure how we’d guess D. But I guess D, along with d, would be the smallest integers that give us a stationary sequence. Once that’s done we could estimate determine p,q and P,Q by looking at the PACF of DdDsDxn. Just for an example, consider a series that looks like:



This is time-dependent. But if we apply some difference operators, we get:



And this is a stationary series. So we’d want d = 1, D = 2. We can observe from the remainder that we’d then need q = 1, Q = 2. Fitting and Forecasting would be done just like with ARMA. I plotted the fit below:



***ARIMAX and SARIMAX processes***

For what it’s worth ARIMA and SARIMA also allow the possibility of including exogeneous sequences.



And,



Fitting and Forecasting would be done just like with ARMA.

***Exponential Processes***

Could be that the trends are exponential, rather than polynomial. In that case, differencing won’t help to make it stationary. Consider the general AR(1).



The solution has exponential behavior, not power law:



Is Dxn stationary? Well, we have:



So we can see no amount of differencing will turn this into a stationary process, just as no amount of derivatives will turn an exponential function into a constant. In this case, we can do the following. Consider the sequence:



I’ll give up trying to capture the MA parts of the series, for simplicity, and just focus on:

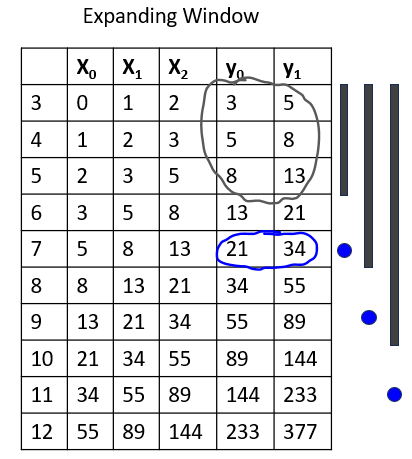
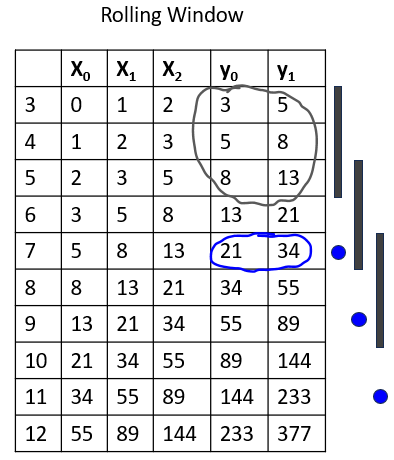


But then we could do a linear regression on xn vs. xn-1 and xn-2. The regression coefficients would give us φ1,2, and the intercept would be βΔt. I’ll call this a DE\_Forecaster thing.

As we saw before though, xn is correlated with not just xn-1, and xn-2, but xn-k for k > 2 as well. How do we know to just set k = 2? I think we’d have to see which results in the best testing mean square error. We’d forecast, like up above:



How would we train and test? We can use an expanding and rolling window approach as above. But it’d have to be formulated a little differently. Unlike with the previous stochastic models, we need to train on more than just a single set of data to get the best fit regression parameters. If we want to predict two steps ahead, in a non-overlapping fashion, and presume three terms in our difference equation, then we’d organize our sequence like below:

Our training data would be the grey rectangles and the testing data would be the blue dot. Note the gap between training sets and testing set. This is so that none of the training predictions overlap with the testing predictions. Might try DES\_Forecaster too, if have a *specific* model in mind.

***Detrending***

If the trend is not power law, but some crazy exponential or whatever thing, one thing we can do is attempt a regression to basically find the mean, subtract that off from the sequence, and then model the remainder with an ARMA model. But even though the mean of the adjusted sequence should be stationary (0), the variance, and other higher moments may still grow with time, and so we may still have problems. But it’s worth a shot, as a method to improve a regression model.

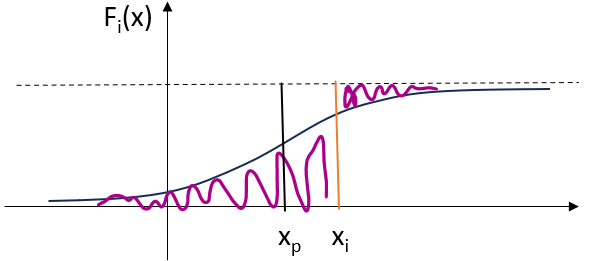
**Training/Testing a Machine Learning Model**

Besides Stochastic models, we can also try to model the series with a machines learning algorithm, like RNN or LSTM neural networks. These will basically try to output what E(xn|xj<n) is. You will have to enter a standard number of precursor xj<n terms, but whether they are all necessary, the algorithm will decide for itself. The PACF plots might help out some in this regard, if applicable. The ML models should be just as versatile as the Stochastic models, in principle. If we review the stochastic model forecasting discussion up above, we see that they simply output a linear function of the input data points. ML models can do the same, and better, since they can conjure of non-linear models. Further, they shouldn’t have to be stationary because they are fit kind of like the exponential stochastic model mentioned up above. Won’t get into the details of the ML models here – we’ll save that for later when we cover neural networks.

One advantage stochastic models have is that they can output a confidence interval for the prediction. We can mimic that functionality with ML models. One option is that instead of trying to output just E(xn|xj<n), we output E(xn|xj<n) and σ2(xn|xj<n). So we’re kind of outputing a probability distribution. In that case, a better loss function is:



where Fi(x) is the cumulative distribution function of the predicted probability distribution function Pi(x) of the ith data point. xpis roughly the prediction/average value of Pi(x), and xi is the actual value of the ith data point. Graphically, CRPS is kind of the purple area shown below, summed over each data point.



Can kind of see that if xi is lined up with xp, then CRPS is minimized (for comparison, consider what CRPS would be if xi were much greater or less than xpfor instance; then can see CRPS would be much larger).

The training/testing sets for a ML model would look like they did for the Exponential Stochastic model. We can use an expanding window approach, or rolling window approach. Note the gap between training sets and testing set. This is so that none of the training predictions overlap with the testing predictions.

